

CS4459.001

Cyber Attacks & Defense Lab

Arbitrary Read / Write and Format String Vulnerability

Apr 2, 2024

Unit-6

- Challenges in the /home/labs/unit6 directory
 - Hosted in ctf-vm2
- Serial Read, Arbitrary Read (AR), Arbitrary Write (AW)
 - Level 0 ~ 3
- Format String Vulnerability (FSV) - 32 / 64 bits
 - Level 4/5, 6/7, 8/9, a/b, c/d
- PIE-enabled binary
 - Level e (40 point)
 - Level f (bonus 50 point)

Unit-6

Serial Read (sr), Arbitrary Read (ar), Arbitrary Write (aw)

- 0-sr-1, 1-ar-2: 15 pts
- 2-aw-1, 20 pt, 3-aw-2: 20 pts

Format string 32|64

- 4-fs-read-1-32, 5-fs-read-1-64, 20 pts
- 6-fs-read-2-32, 7-fs-read-2-64, 20 pts
- 8-fs-arbt-read-32, 9-fs-arbt-read-64,
25 pts

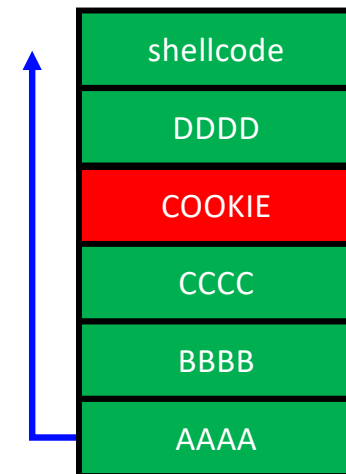
- a-fs-arbt-write-32, b-fs-arbt-write-64,
30 pts
- c-fs-code-exec-32, d-fs-code-exec-64,
30 pts

PIE challenges (bonuses)

- e-fs-code-exec-pie-64, 40 pt
- f-fs-no-binary-pie-64, 50 pt

Buffer Overflow (BoF) and Control Flow Hijacking (CFH)

- **BoF** has been our primary attack vector
 - To hijack control flow and deviate
- **BoF**: fill the buffer more than its size
 - Overwriting return address (saved %eip)
 - Overwriting frame pointer (saved %ebp)
- Control Flow Hijacking
 - Return to shellcode: Run your own code
 - Return to some other functions: Code re-use attack
 - Return to ROP gadgets: Code re-use attack with ROP gadgets



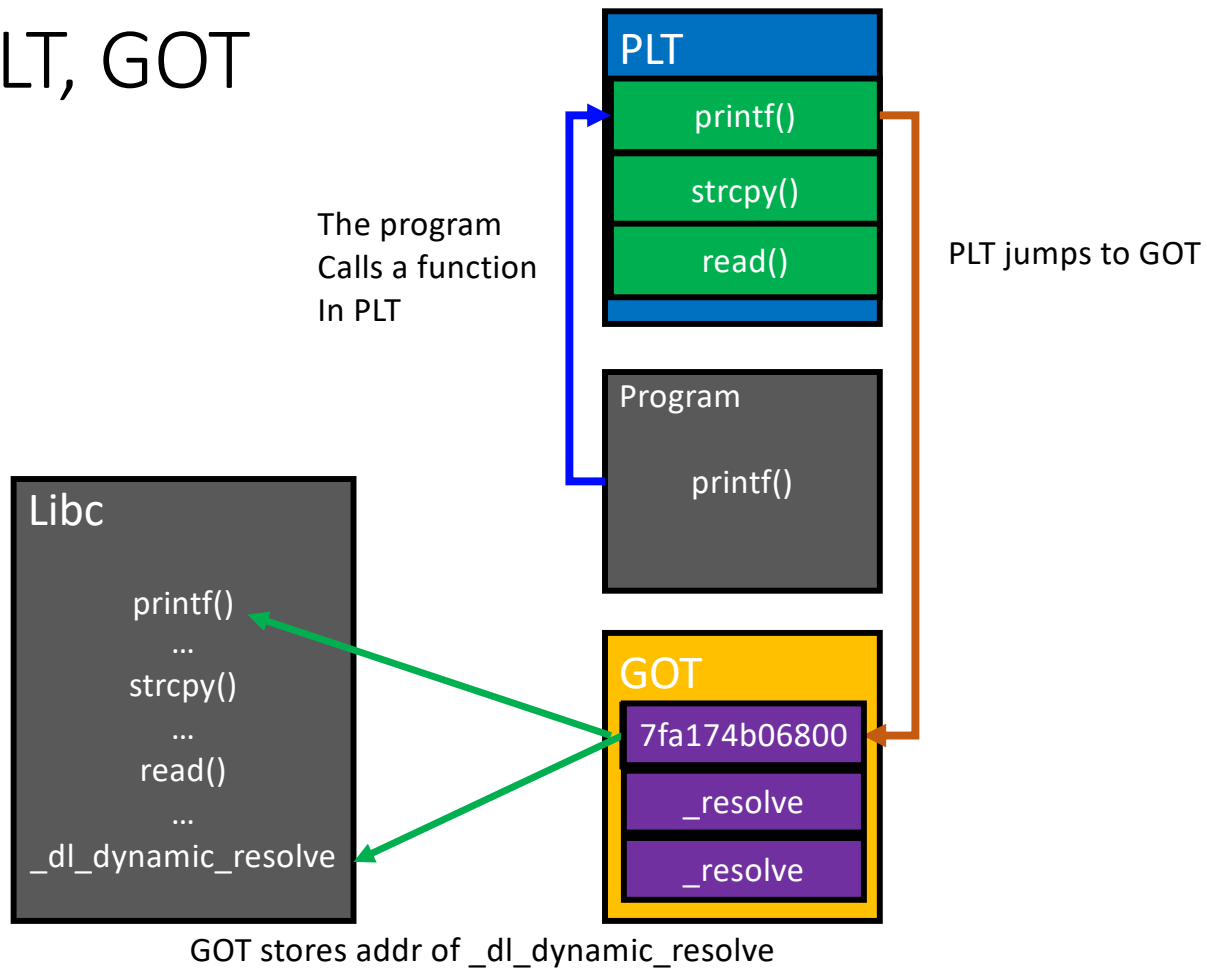
Control Flow Hijacking (CFH)

- Entry points to CFH
 - **RET / JMP / CALL** to shellcode
 - **RET / JMP / CALL** to some other functions
 - **RET / JMP / CALL** to ROP gadgets
- Targets to overwrite
 - Return address
 - Global offset table (GOT)
 - **Function pointers**
 - **Jump table entry**



Eventually we seek to gain control over %EIP / %RIP

ELF, PLT, GOT



What is the Requirement for CFH?

- Is *buffer overflow* the only way to achieve *Control Flow Hijacking*?
 - Overwriting return address (saved %eip)
 - Overwriting frame pointer (saved %ebp)
- What we need is the capability to overwrite certain address
 - Arbitrary write!
- Today we will learn **arbitrary read/write**



Attack Primitives




- Arbitrary Read

- Read from any address A , any number N of bytes
- The attacker must know the address A

- Example:

- Read 100 bytes from `0xffffd100` (somewhere in the stack)
- Read 100 bytes from `0x8048500` (somewhere in the code section)

- Can this break:

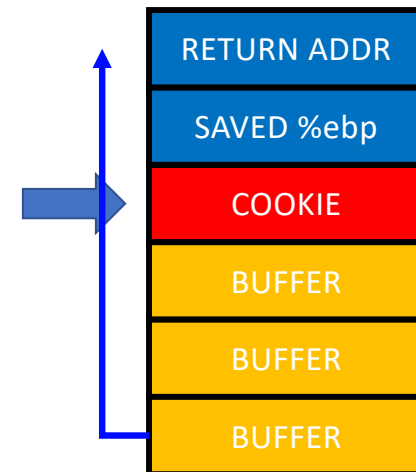
- Stack-Cookie? 
- ASLR? 
- DEP? 

Breaking Stack-Cookie

- The cookie value is on the stack

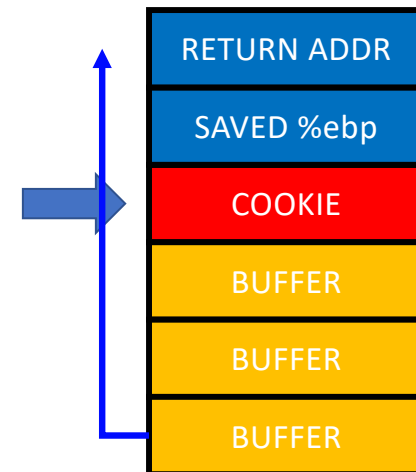
```
printf("How many bytes of your name do you want to print?\n");  
scanf("%d", &i);  
printf("Hello ");  
write(1, buf, i);  
printf("!\n");
```

- Sequential read can break **stack-cookie** easily



Breaking Stack-Cookie

- The *cookie* value is on the stack
- Arbitrary Read
 - Read where?
 - The address that the cookie is stored!
- Then launch buffer overflow attack
- You should know the address of the stack..



Breaking ASLR

- Arbitrary Read
 - Read anywhere if you know the address...
- Read where?
 - Non-PIE: Code section is fixed `0x8048000 ~ 0x804a000`
 - `0x401000 ~` and `0x601000` in 64 bit
 - Stack – random
 - Library – random
 - Heap – random
- Code/data sections are the only fixed locations

Breaking ASLR + DEP

- GOT stores address of libc functions
 - printf(), puts(), read(), etc.

```
Relocation section '.rela.plt' at offset 0x560 contains 11 entries:
  Offset          Info           Type           Sym. Value      Sym. Name + Addend
000000601018     000100000007  R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000601020     000200000007  R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000601028     000300000007  R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000601030     000400000007  R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000601038     000500000007  R_X86_64_JUMP_SLO 0000000000000000 fgets@GLIBC_2.2.5 + 0
000000601040     000700000007  R_X86_64_JUMP_SLO 0000000000000000 prctl@GLIBC_2.2.5 + 0
000000601048     000800000007  R_X86_64_JUMP_SLO 0000000000000000 fflush@GLIBC_2.2.5 + 0
000000601050     000900000007  R_X86_64_JUMP_SLO 0000000000000000 __isoc99_sscanf@GLIBC_2.2.5 + 0
000000601058     000a00000007  R_X86_64_JUMP_SLO 0000000000000000 getegid@GLIBC_2.2.5 + 0
000000601060     000b00000007  R_X86_64_JUMP_SLO 0000000000000000 setregid@GLIBC_2.2.5 + 0
000000601068     000c00000007  R_X86_64_JUMP_SLO 0000000000000000 fwrite@GLIBC_2.2.5 + 0
```

Breaking ASLR + DEP

- Read a value from GOT
 - Can get the address of puts()
- Can you calculate the addresses of execve() or system() from puts?
- Yes
 - Load *libc* and find offsets, diff, and calculate

```
kjee@ctf-vm2.utdallas.edu:/home/kjee/unit6/0-sr-1 $ ldd sr-1
linux-vdso.so.1 => (0x00007ffd297ca000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1812f73000)
/lib64/ld-linux-x86-64.so.2 (0x00007f181333d000)
```

Attacking Global Offset Table (GOT)

- Read the address in GOT
- Leak the address of puts()
- Calculate the address of `execve()` or `system()` from the offset

Challenges: Unit6

- **sr-1**

- The program will perform a *sequential read* from the stack for you
- Read addresses from the leak and exploit the vulnerability!

- **ar-2**

- You may perform *arbitrary read*
 - Can specify where to read and how many bytes to read
- Read GOT of some functions to get the address of that function in libc
- Call `system()`, `execve()`, whatever you want!

Attack Primitives

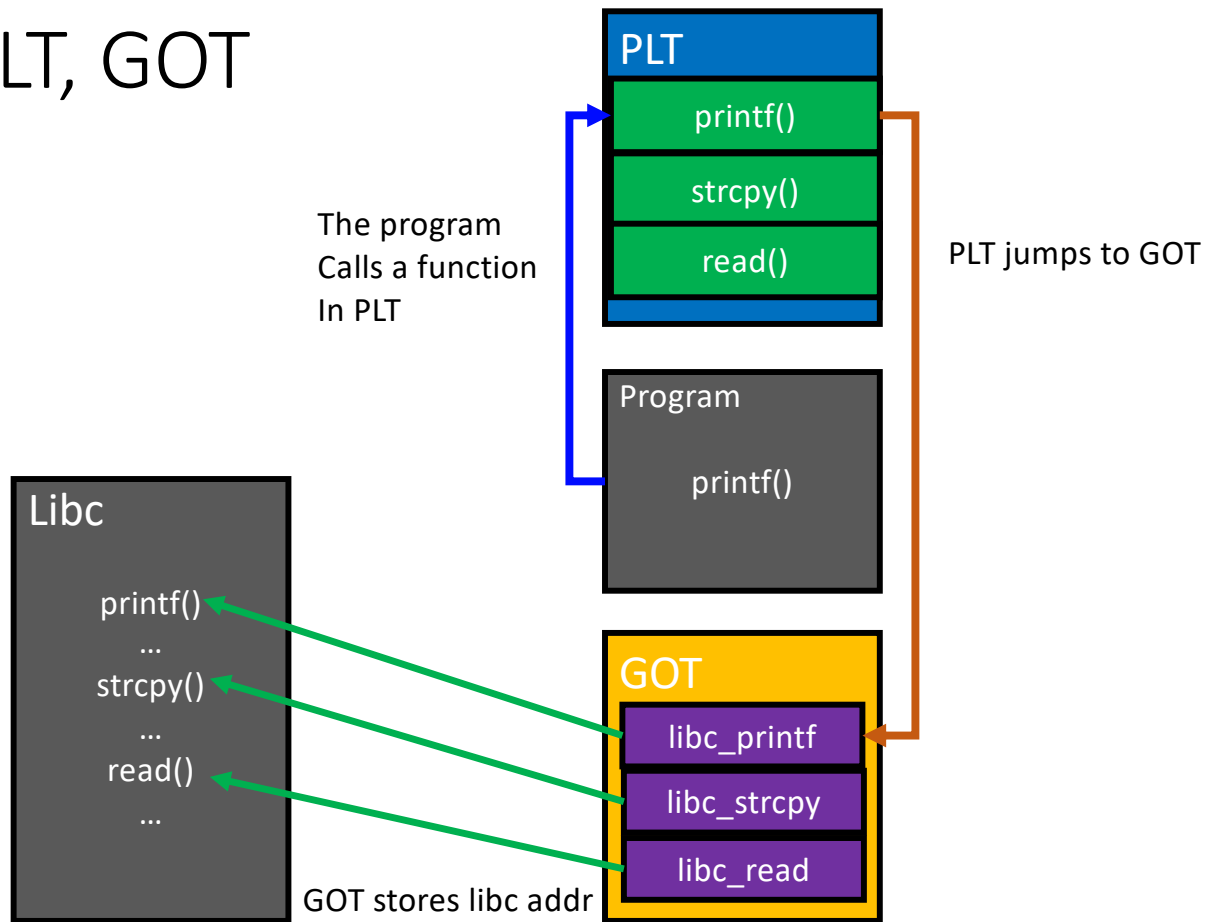
- Arbitrary Write (AR)
 - Write on any address, any number of bytes
 - The attacker must know the address
- Example:
 - Write 100 bytes of data to `0xffffd100` (somewhere in the stack)
 - Write 100 bytes of data to `0x8048500` (somewhere in the code section)
- Can achieve Control Flow Hijacking
 - Overwrite return address
 - Overwrite frame pointer
 - *Overwriting GOT*

Attacking Global Offset Table (GOT)

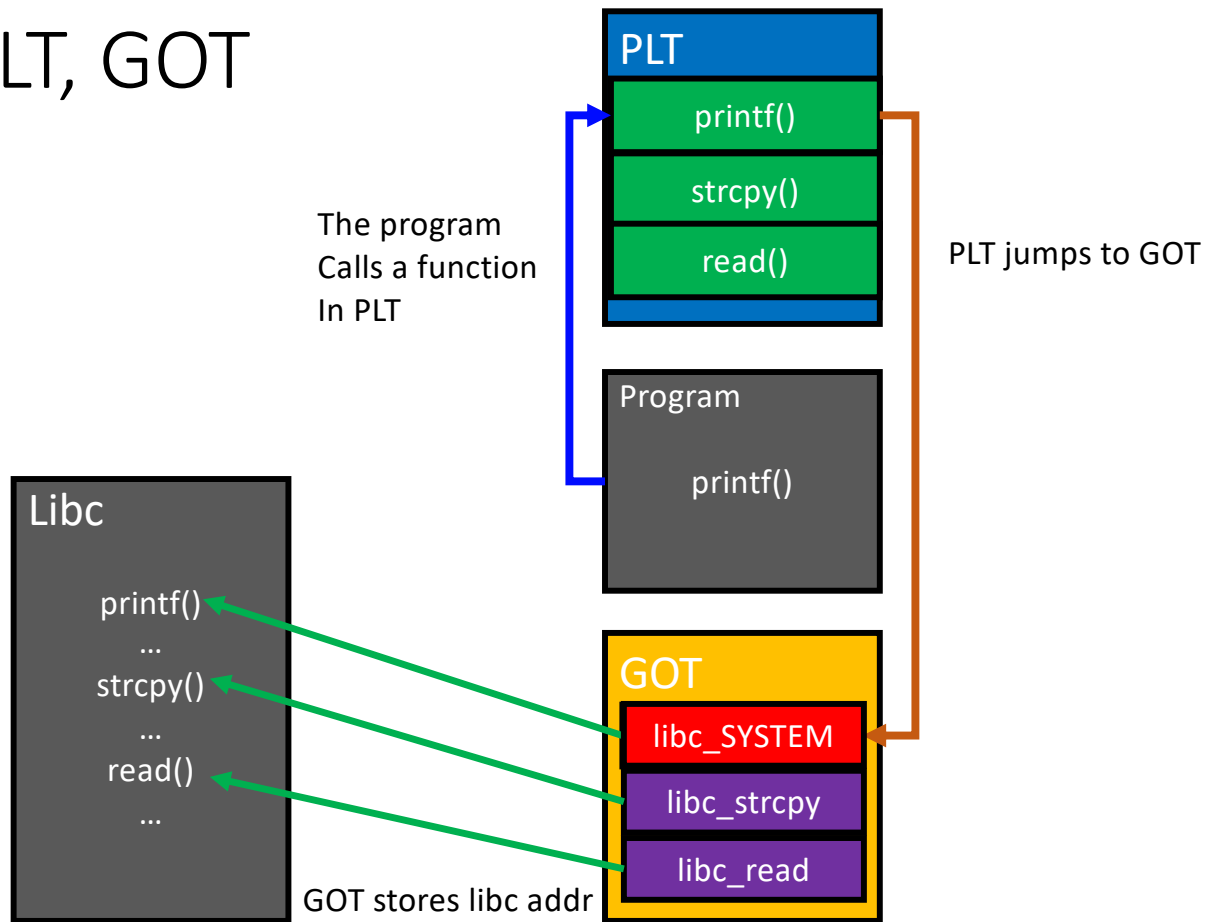
- Attack Outline

1. Read the address in GOT
2. Leak the address of `printf()`
3. Calculate the address of `system()` from the offset
4. Write that address `system()` to `printf()`'s GOT
 - What will happen a program calls `printf()` ?

ELF, PLT, GOT



ELF, PLT, GOT



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"); // ??
```

Arbitrary-Write-1 (AR-1)

- Change the GOT of `printf()` to to `'please_execute_me()'` via arbitrary write capability!
- `printf()` after the BoF vulnerability will run that function for you.

Arbitrary-Write-2 (AR-2)

- Program contains both arbitrary-read (ar) and arbitrary-write (aw) vulnerabilities.
- Exploit arbitrary-read (ar) to get the address of libc function
 - Then calculate the address of `system()`
- Exploit arbitrary write to overwrite the GOT of `printf()` to `system()`

```
printf("Writing %lu bytes to %p\n");  
system("Writing %lu bytes to %p\n");
```

ONE GADGET

ONE GADGET

- Or you may take a simpler approach

```
$ldd aw-2
    linux-vdso.so.1 => (0x00007ffffb6df4000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f270b792000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f270bb5c000)
$
$one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x45226 execve("/bin/sh", rsp+0x30, environ)
constraints:
    rax == NULL

0x4527a execve("/bin/sh", rsp+0x30, environ)
constraints:
    [rsp+0x30] == NULL

0xf0364 execve("/bin/sh", rsp+0x50, environ)
constraints:
    [rsp+0x50] == NULL

0xf1207 execve("/bin/sh", rsp+0x70, environ)
constraints:
    [rsp+0x70] == NULL
```

Why one_gadget exists?

```
system("command")
  1. fork() // parent waits
  2. execve("/bin/sh", ["sh", "-c", command,
    NULL], environ) // child
```

- When we jump to the start of `system()`
 - Expects 1st arg is the command (either `%rdi` or 1st arg on the stack)
 - Run
`execve("/bin/sh", ["sh", "-c", command, NULL], environ);`
- When we jump in the middle of `system()`
 - Skip the part that set up arguments
 - Call
`execve("/bin/sh", rsp + 0x30, environ);`
 - What if `%rsp + 0x30 == 0`?

Why one_gadget exists?

- When we jump in the middle of `system()`
 - Skip the part that set up arguments
 - Call `execve("/bin/sh", rsp+0x30, environ);`
 - What if `%rsp+0x30 == 0`?
 - What if `%rax == 0`?
 - What if `%rsp+0x50 == 0`?
 - What if `%rsp+0x70 == 0`?
- Use it with care...

```
$ ldd aw-2
linux-vdso.so.1 => (0x00007ffffb6df4000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f270b792000)
/lib64/ld-linux-x86-64.so.2 (0x00007f270bb5c000)
$
$one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x45226 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4527a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf0364 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1207 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

one_gadget

```
$ldd aw-2
linux-vdso.so.1 => (0x00007fffb6df4000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f270b792000)
/lib64/ld-linux-x86-64.so.2 (0x00007f270bb5c000)
$
$one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x45226 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL
0x4527a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL
0xf0364 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL
0xf1207 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

$$(0x45281 + 7) + 0x147B8F = 0x18ce17$$

```
$strings -tx /lib/x86_64-linux-gnu/libc.so.6 |grep bin/sh
18ce17 /bin/sh
```

```
4527a: 48 8b 05 37 ec 37 00  mov    0x37ec37(%rip),%rax    # 3c3eb8 <_IO_file_jumps@GLIBC_2.2.5+0x7d8>
45281: 48 8d 3d 8f 7b 14 00  lea   0x147b8f(%rip),%rdi    # 18ce17 <_libc_intl_domainname@GLIBC_2.2.5+0x197>
45288: 48 8d 74 24 30        lea   0x30(%rsp),%rsi
4528d: c7 05 09 12 38 00 00  movl  $0x0,0x381209(%rip)    # 3c64a0 <__abort_msg@GLIBC_PRIVATE+0x8c0>
45294: 00 00 00
45297: c7 05 03 12 38 00 00  movl  $0x0,0x381203(%rip)    # 3c64a4 <__abort_msg@GLIBC_PRIVATE+0x8c4>
4529e: 00 00 00
452a1: 48 8b 10              mov   (%rax),%rdx
452a4: e8 47 75 08 00      callq cc7f0 <execve@GLIBC_2.2.5>
```

Format String Vulnerability

In ASLR-2

```
int check_function(void) {  
    printf("Your buffer? I don't wanna let you know my address!\nDoes these leak some?: ");  
    printf("%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p\n");  
    return input_func();  
}
```

- What kind of information this will print???
- 15 values of %p (print hexadecimal number as 0x????????, an addr)
- No arguments...

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 `decimal value`
 - Justify its length to `length (08)`
 - Get the value from `n`-th (1) 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
```

fs-read-1-(32 | 64)

- Exploit a format string vulnerability to leak pointers from stack
- Guess the random value correctly to get the shell!

```
kjee@ctf-vm2.utdallas.edu:/home/kjee/unit6/6-fs-read-2-32 $ ./fs-read-2-32
Please type your name first:
%x
Hello ffc6d418

Can you guess the random?
asdf
Wrong, your random was 0xbe0b189b but you typed 0x0000000a
```

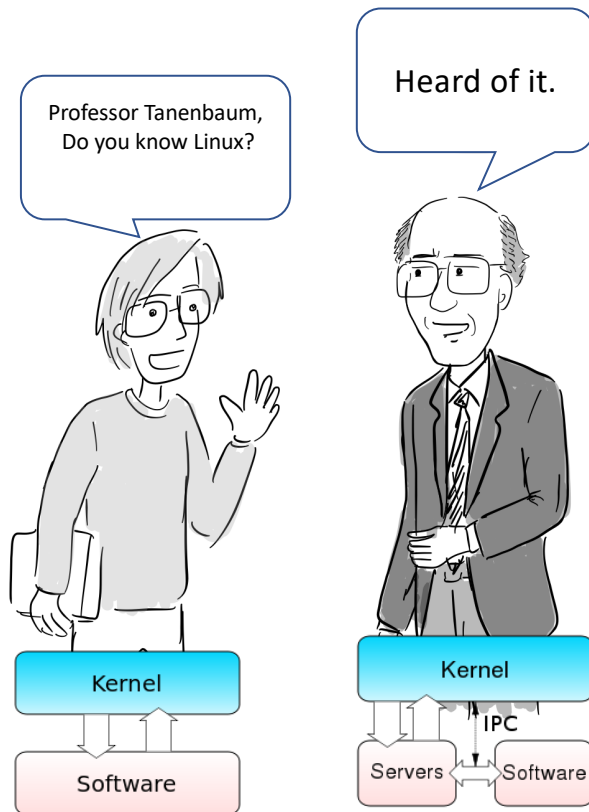
fs-read-2-(32 | 64)

- Exploit a format string vulnerability to leak *pointers* from stack
- You have limited size for your format string as your input
- Use directives like **%100\$x**
 - To skip some uninterested parts of the stack

```
34  /*
35  * On the stack, these variables will be placed in backward:
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 information
48  */
49  };
```

Backup

Tanenbaum–Torvalds Debate



Open Sources: Voices from the Open Source Revolution

1st Edition January 1999
 1-56592-582-3, Order Number: 5823
 280 pages, \$24.95

Appendix A

The Tanenbaum-Torvalds Debate

What follows in this appendix are what are known in the community as the "Tanenbaum/Linus 'Linux is obsolete'" debates. Andrew Tanenbaum is a well-respected researcher who has made a very good living thinking about operating systems and OS design. In early 1992, noticing the way that the Linux discussion had taken over the discussion in comp.os.minix, he decided it was time to comment on Linux.

Although Andrew Tanenbaum has been derided for his heavy hand and misjudgements of the Linux kernel, such a reaction to Tanenbaum is unfair. When Linus himself heard that we were including this, he wanted to make sure that the world understood that he holds no animus towards Tanenbaum and in fact would not have sanctioned its inclusion if we had not been able to convince him that it would show the way the world was thinking about OS design at the time.

We felt the inclusion of this appendix would give a good perspective on how things were when Linus was under pressure because he abandoned the idea of microkernels in academia. The first third of Linus' essay discusses this further.

Electronic copies of this debate are available on the Web and are easily found through any search service. It's fun to read this and note who joined into the discussion; you see user-hacker Ken Thompson (one of the founders of Unix) and David Miller (who is a major Linux kernel hacker now), as well as many others.

To put this discussion into perspective, when it occurred in 1992, the 386 was the dominating chip and the 486 had not come out on the market. Microsoft was still a small company selling DOS and Word for DOS. Lotus 123 ruled the spreadsheet space and WordPerfect the word processing market. DBASE was the dominant database vendor and many companies that are household names today—Netscape, Yahoo, Excite—simply did not exist.

From: astfcs.vu.nl (Andy Tanenbaum)
 Newsgroup: comp.os.minix
 Subject: LINUX is obsolete
 Date: 29 Jan 92 12:12:50 GMT

I was in the U.S. for a couple of weeks, so I haven't commented much on LINUX (not that I would have said much had I been around), but for what it is worth, I have a couple of comments now.

As most of you know, for me MINIX is a hobby, something that I do in the evening when I get bored writing books and there are no major wars, revolutions, or senate hearings being televised live on CNN. My real job is a professor and researcher in the area of operating systems.

As a result of my occupation, I think I know a bit about where operating are going in the next decade or so. Two aspects stand out:

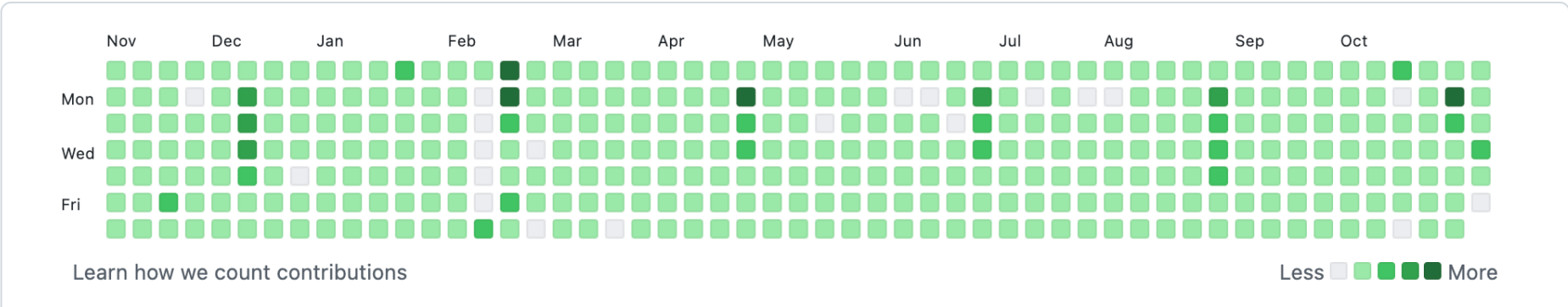
1. MICROKERNEL VS MONOLITHIC SYSTEM
 Most older operating systems are monolithic, that is, the whole operating system is a single a.out file that runs in "kernel mode." This binary contains the process management, memory management, file system and the rest. Examples of such systems are UNIX, MS-DOS, VMS, MVS, OS/360, MZLICS, and many more.

The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel. They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O. Examples of this design are the MCK600, Amosha, Chorus, Mach, and the not-yet-released Windows/NT.

Who is Linus Torvalds?

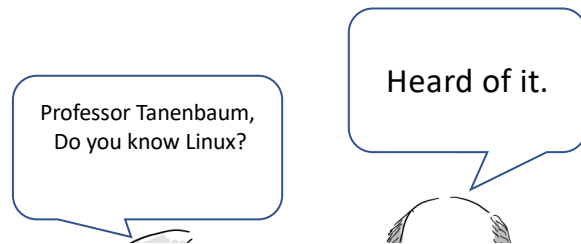
- Do I have to tell you?
- Two major contributions to the world
 - Which one do you think is bigger?

2,557 contributions in the last year



From <https://github.com/torvalds>

Tanenbaum–Torvalds debate



- Micro-kernel vs. Monolithic kernel
- Portability

Stop flaming, MINIX and Linux are two different systems with different purposes. One is a teaching tool (and a good one I think), the other is real UNIX for real hackers.

Anyone who says you can have a lot of widely dispersed people hack away on a complicated piece of code and avoid total anarchy has never managed a software project.

