

Low level attacks

Format string vulnerabilities

(part 2)

Mario Alviano

University of Calabria, Italy

A.Y. 2019/2020

Many many questions!

- How to read specific memory locations?
- How to write specific memory locations?
- How to leak sensitive memory addresses?
- How to exploit format strings?

We are going to answer all these questions!

- We can use %s
- The address must be in the stack
- Let's reach the format string!

```
<address><stackpop><read-code>
```

- address: the address we want to read
- stackpop: format parameters such as %u or %8x to reach <address>
- read-code: format parameter %s

- Try `printf_s.c`
- Find the right stackpop with
`AAAABBBB|%8x...%8x|%08x|`
(see `printf_s_stackpop.py`)
- You have the right stackpop when the printed string ends with `|41414141|`
- Possibly, prepend 1, 2 or 3 characters to align memory
- Use `gdb` to find the address of `unlinked`
- Replace `AAAA` with the address
- Replace `|%08x|` with `|%s|`
(see `printf_s_build.py`)

- We can use `%n`
- The address must be in the stack
- Let's reach the format string!

```
(<junk><address>)^4<stackpop><write-code>
```

- `junk`: four dummy bytes (eg. JUNK)
- `address`: the address we want to write
- `stackpop`: format parameters such as `%8x` to reach the format string (`%u` is problematic; why?)
- `write-code`: increase counter with `%nx` (where $n \geq 8$) and write with `%n`

- Try `printf_write_to_address.c`
- Find the right `stackpop` with
`AAAABBBBJUNKCCCCJUNKCCCCJUNKCCCC | %8x...%8x | %08x |`
(see `printf_write_stackpop.py`)
- Use `gdb` to find the address of `target`
- Replace the first part of the format string
 - The four addresses point to the four bytes of `target`
- Replace `| %08x |` with the write-code
 - See `printf_write_build.py`
 - Note that the padding function has been improved
(we are going to print hexadecimal numbers)

Rewrite the return address

- Try `printf_retaddr.c`
- Use `printf_retaddr_stackpop.py` to find the right stackpop
- Use `gdb` to find the address of the `unlinked` function
- Use `gdb` to find the address of the return address
- Use `printf_retaddr_build.py` to inject the address

```
run "$ (./printf_retaddr_build.py) "
```

Note

- The address of the return address will be different if you run the program normally (out of `gdb`)
- The address has to be brute forced

- Can we find the address of the format string?
- This would allow to compute the address of the return address
- Try `printf_retaddr_find.py`
 - 1 When you see the format string printed back, you have the right address! (it should be at 7 in the example)
 - 2 You also know the relative position of the format string (where you see `|41414141|`; it should be at 11 in the example)
 - 3 And you know the relative position of the return address (it should be at 6 in the example)
 - Either use `gdb` to find it, or
 - Recognize it in the output of `printf_retaddr_find.py`
- Let's try $1 - 4 * (2 - 3)$
 - `(gdb) p/x 0xADDR1 - 4 * (OFFSET2 - OFFSET3)`

Use `printf_retaddr_build.py` without `gdb`

```
$ python -c "import os; os.system(''a.out ``./printf_retaddr_build.py`` '')"
```

```
This is noncompliant function:  
JUNK[0x00]JUNK[0x00]JUNK[0x00]JUNK[0x00]|f7f  
  
4b4e554a  
  
4b4e554a|.....  
.....  
.....  
.....  
.....  
  
This is unlinked function.
```

How a format string exploit works

- Essentially, a combination of what we have seen
- We have the address of the return address
- We have the address of the format string
- We can inject a shellcode
 - The shellcode will be at the end of the format string
 - The return address will jump to the shellcode
- We can cast a return-to-libc attack
 - The address of `system` will replace the return address
 - The string `/bin/sh` will be at the end of the format string
 - The address of this string will follow the return address

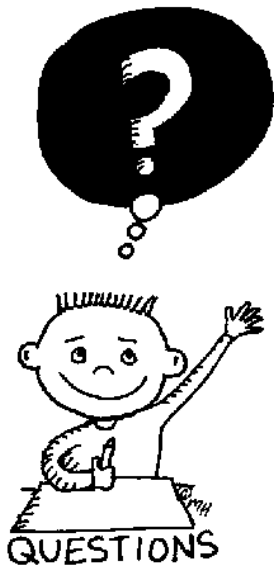
Let's see the first attack; the second is similar.

Shellcode injection

- Let's try `printf_shellcode_find.py`
- First step is to find the right addresses
- Note that the shellcode has been added with a proper nop sled
- Now open `printf_shellcode_build.py`
- We have to rewrite the return address to hit the nop sled
 - We put it at the end of the format string
 - In the write-code we specify the address of the nop sled; ie. address of the format string + size of the format string up to the write-code + something

Run the exploit

- Change the owner of the executable to `root`
- Set the SUID bit
- Give in input our casted format string



END OF THE
LECTURE