Software Security

Ethical Hacking

Alessandro Brighente
Eleonora Losiouk

Master Degree on Cybersecurity

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

Outline

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Format string
- Access optional arguments
- How printf() works
- Format string attack
- How to exploit the vulnerability
- Countermeasures

Format String

- `printf()` - To print out a string according to a format

```
int printf(const char *format, ...);
```

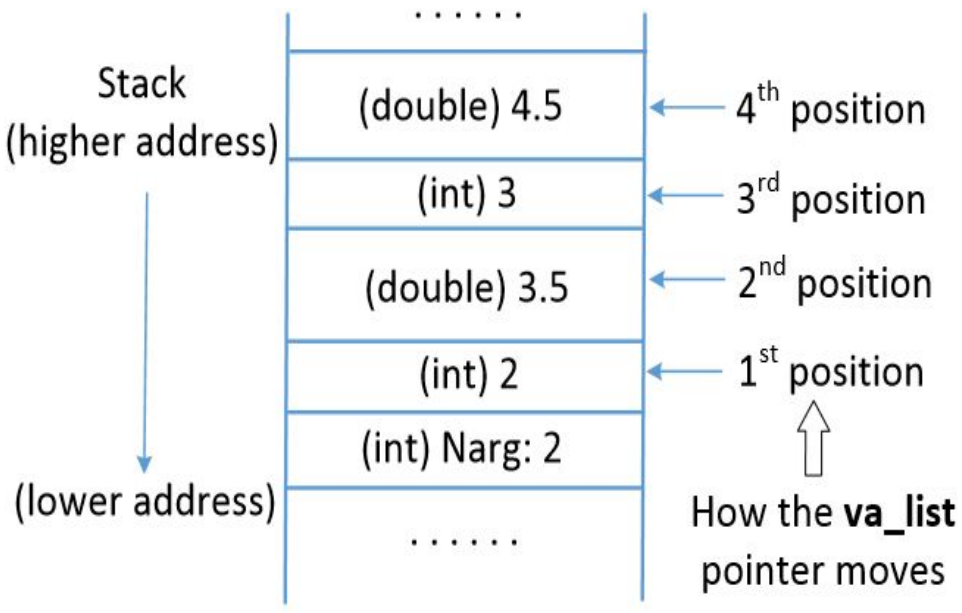
- The argument list of printf() consists of :
  - One concrete argument format
  - Zero or more optional arguments
- Hence, compilers don't complain if less arguments are passed to printf() during invocation

Access Optional Arguments


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

int myprint(int Narg, ... )
{
  int i;
  va_list ap;                                          ①

  va_start(ap, Narg);                                  ②
  for(i=0; i<Narg; i++) {
    printf("%d  ", va_arg(ap, int));                   ③
    printf("%f\n", va_arg(ap, double));                ④
  }
  va_end(ap);                                          ⑤
}

int main() {
  myprint(1, 2, 3.5);                                  ⑥
  myprint(2, 2, 3.5, 3, 4.5);                          ⑦
  return 1;
}
```

- `myprint()` shows how `printf()` actually works
- Consider `myprint()` is invoked in line 7
- `va_list` pointer (line 1) accesses the optional arguments
- `va_start()` macro (line 2) calculates the initial position of `va_list` based on the second argument Narg (last argument before the optional arguments begin)

Access Optional Arguments



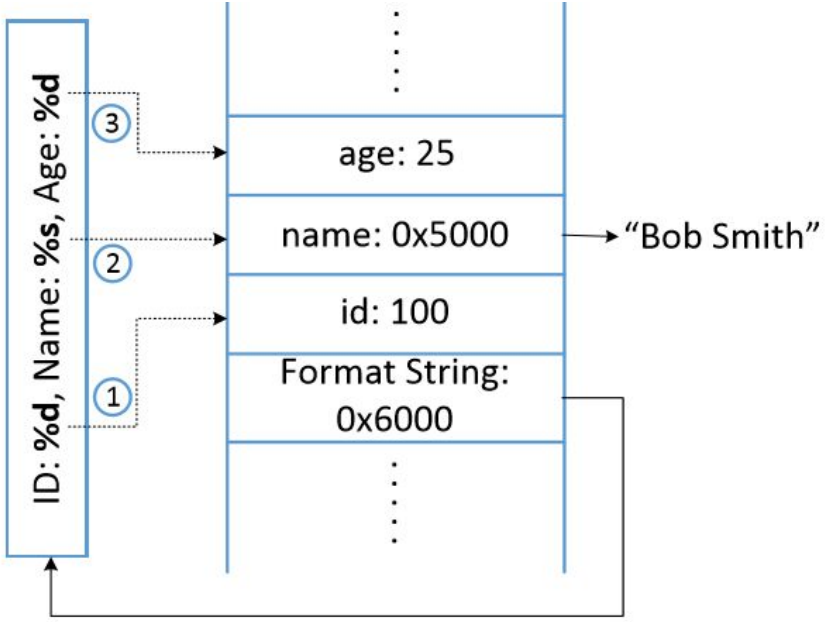
- `va_start()` macro gets the start address of Narg, finds the size based on the data type and sets the value for `va_list` pointer
- `va_list` pointer advances using `va_arg()` macro
- `va_arg(ap, int)`: Moves the ap pointer (`va_list`) up by 4 bytes
- When all the optional arguments are accessed, `va_end()` is called

printf() Access Optional Arguments


```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- Here, `printf()` has three optional arguments. Elements starting with "%" are called format specifiers
- `printf()` scans the format string and prints out each character until "%" is encountered
- `printf()` calls `va_arg()`, which returns the optional argument pointed by `va_list` and advances it to the next argument

`printf()` Access Optional Arguments

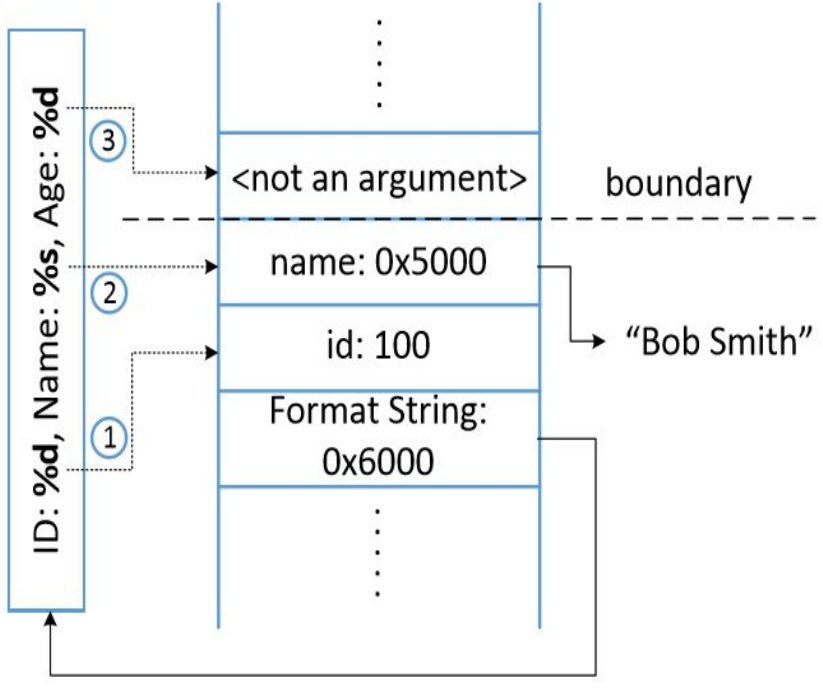


- When `printf()` is invoked, the arguments are pushed onto the stack in reverse order
- When it scans and prints the format string, `printf()` replaces %d with the value from the first optional argument and prints out the value
- `va_list` is then moved to the position 2

Missing Optional Arguments


```
#include <stdio.h>

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

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

- `va_arg()` macro doesn't understand if it reached the end of the optional argument list
- It continues fetching data from the stack and advancing `va_list` pointer



Format String Vulnerability


```
printf(user_input);
```

```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

```
sprintf(format, "%s %s", getenv("PWD"), ": %d");
printf(format, program_data);
```

- In these three examples, user's input (user_input) becomes part of a format string
- What will happen if **user_input** contains format specifiers?

What Can We Achieve?

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- Attack 1 : Crash program
- Attack 2 : Print out data on the stack
- Attack 3 : Change the program's data in the memory
- Attack 4 : Change the program's data to specific value

Hints

- %s
  - For each %s, printf() fetches a value where va_list points to and advances va_list to the next position
  - As we give %s, printf() treats the value as address and fetches data from that address
- %x
  - printf() prints out the integer value pointed by va_list pointer and advances it by 4 bytes
- %n
  - writes the number of characters printed out so far into memory
  - %n treats the value pointed by the va_list pointer as a memory address and writes into that location

Hints

- Assuming the address of var is 0xbffff304 (can be obtained using gdb)

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
```

- The address of var is given in the beginning of the input so that it is stored on the stack
- $(command): Command substitution. Allows the output of the command to replace the command itself
- Width modifier
  - `%.10000000x`

Countermeasures: Developer

- Avoid using untrusted user inputs for format strings in functions like `printf, sprintf, fprintf, vprintf, scanf, vfscanf`

```
// Vulnerable version (user inputs become part of the format string):
        sprintf(format, "%s %s", user_input, ": %d");
        printf(format, program_data);


// Safe version (user inputs are not part of the format string):
        strcpy(format, "%s: %d");
        printf(format, user_input, program_data);
```

Countermeasures: Compiler

Compilers can detect potential format string vulnerabilities

```
#include <stdio.h>

int main()
{
   char *format = "Hello  %x%x%x\n";

   printf("Hello %x%x%x\n", 5, 4);     ①
   printf(format, 5, 4);               ②

   return 0;
}
```

- Use two compilers to compile the program: gcc and clang.
- We can see that there is a mismatch in the format string.

Countermeasures: Compiler


```
$ gcc test_compiler.c
test_compiler.c: In function main:
test_compiler.c:7:4: warning: format %x expects a matching unsigned
    int argument [-Wformat]

$ clang test_compiler.c
test_compiler.c:7:23: warning: more '%' conversions than data
    arguments
      [-Wformat]
   printf("Hello %x%x%x\n", 5, 4);
                      ~^
1 warning generated.
```

- With default settings, both compilers gave warning for the first printf().
- No warning was given out for the second one.

Countermeasures: Compiler


```
$ gcc -Wformat=2 test_compiler.c
test_compiler.c:7:4: ... (omitted, same as before)
test_compiler.c:8:4: warning: format not a string literal, argument
    types not checked
[-Wformat-nonliteral]

$ clang -Wformat=2 test_compiler.c
test_compiler.c:7:23: ... (omitted, same as before)
test_compiler.c:8:11: warning: format string is not a string literal
      [-Wformat-nonliteral]
   printf(format, 5, 4);
          ^~~~~~
2 warnings generated.
```

- On giving an option -wformat=2, both compilers give warnings for both printf statements stating that the format string is not a string literal
- These warnings just act as reminders to the developers that there is a potential problem but nevertheless compile the programs

Countermeasures

- **Address randomization**: Makes it difficult for the attackers to guess the address of the target memory (return address, address of the malicious code)
- **Non-executable Stack/Heap**: This will not work. Attackers can use the return-to-libc technique to defeat the countermeasure.
- **StackGuard**: This will not work. Unlike buffer overflow, using format string vulnerabilities, we can ensure that only the target memory is modified; no other memory is affected.