Security Testing

Edoardo Righi

October 15, 2019

1 Introduction

Bla bla bla

2 Attack taxonomy

- **Vulnerability**: the state of being open to attack or damage
- **Exploit**: take advantage of a weakness (vulnerability)

2.1 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!

SQL Injection

It is a type of attack that exploit the possibility to add SQL code into a user input; the user provided data is used to form a SQL query that the server executes. An example with PHP:

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

The query could become:

```
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 ...
```

A fix can be made by limiting the types of characters accepted and/or the length of the input. Otherwise a lot of languages have included functions that fix this.

Cross Site Scripting (XSS)

It is a vulnerability that enables to insert or execute in a form input (client side) an attack: execute malware, steal data or cookies. The problem is caused by the direct displaying in an output web page, without any sanitization. Typical attack:

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)

As with the SQL Injection, this attack can be avoided by checking the input, manually or using the functions that languages have.

3 Laboratory 1

3.1 OWASP Zap

Zap is a security tool to search vulnerabilities in web applications.

3.2 WebGoat

WebGoat is a localhost server that contains exercises to train with security. In this case, it was used to study SQL injection.

3.3 Homework 1

See the linked PDF for more informations about it.

3.3.1 After the correction

The exercise would have been faster with the use of **Fuzzer**, a Zap tool that makes possible to repeat HTTP requests, without writing again all the unchanged data and by altering selected parts of it with *rules*. For example, by selecting the starting position in substring you can make it change of value in an interval (from 1 to 23 in this case) while also editing the letter to be checked (mantain separeted more conditions if the objective is to concatenate them). To use it just search in the requests history for the desired request, right-click and select *Attack->Fuz*.

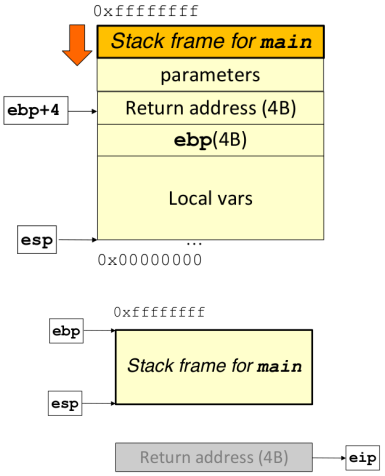
4 Attack taxonomy (part 2)

Before going on with *Buffer overflow*, a quick review of the call stack.

4.1 Call stack

The call stack is a stack data structure that stores information about the active subroutines of a computer program. The call stack grows to lower memory addresses.

- **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

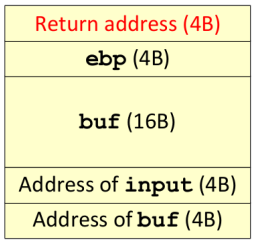
`return: f(x1, x2, x3);`

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

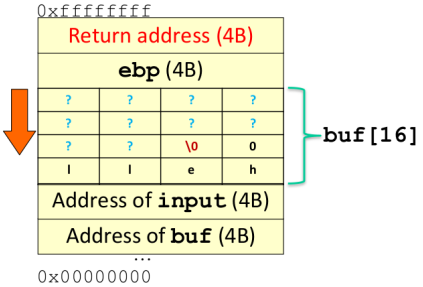
Buffer overflow

It stands for the attempt to write more data to a fixed length memory block.

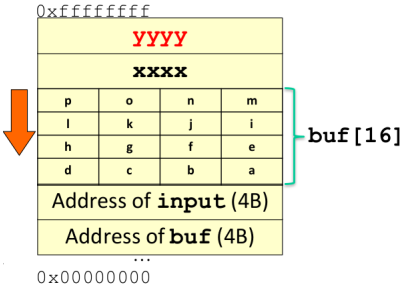
```
#include <stdio.h>
void f(char* input) {
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
    f(argv[1]);
    return 0;
}
```



`> a.out "hello"`



`> a.out "abcdefghijklmnopxxxxyyyy"`



With as input `> a.out "HELLO"`, there is no problem as the character occupy 6 chars (ending char included) of the 16 available. But if the input is `> a.out "abcdefghijklmnopxxxxyyyy"`, things are different. The string is 24 characters long, so `xxxxyyyy` goes out of the buffer:

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

Instead of "abcd...", the attacker can input executable code in HEX, called shellcode.
To sum it up, the problem is that 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 to spot it? The use of unsafe string manipulation functions (e.g., strcpy) is a wake-up call of it. To fix it:

- 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);`.

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

Since there is a prefix, it should be: `sizeof(buf)-7`.

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

Use **snprintf** instead of **sprintf**.

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

Should be the size of **buf** (20).

```
char src[10];
char dest[10];
char* base_url = "www.fbk.eu";
strncpy(src, base_url, 10);
strcpy(dest, src);
```

The string **base_url** is 11 chars long because of the '\0' at the end of the string, so **src** will not be null terminated. We will have buffer overflow because **strcpy** doesn't know when to stop.

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

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

```
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;
    }
}
```

With count coming from user input.
`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.

```
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++**

Format Strings

String arguments for Format Functions like printf contain Format String parameters like %d, %s.

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

To add the value of *name* to *%s* the system executes a POP from a certain memory area. Prendendo come esempio:

```
int main(int argc, char* argv[]) {
    if (argc > 1)
        printf(argv[1]);
    return 0;
}
```

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

But with:

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

The system 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 10^{th} integer

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.

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

To fix it:

- 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)
```

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

Integer overflow

This vulnerability is based on arithmetical specifications of calculators. In the C language there are different types of integers, defined by variables with specific bits. For example on a 32bit machine:

- **int** is an integer with 32 bits
- **short** is an integer with 16 bits

A 16-bit integer can store 65,536 distinct values. In an unsigned representation, these values are the integers between 0 and 65,535; using two's complement, possible values range from -32,768 to 32,767. An implicit or explicit integer type conversion can produce unexpected results due to truncation or bit extension; integer operations overflow, producing unexpected results.

```
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]
```

To fix integer overflow:

- Use large enough integer types
- Use unsigned integers if possible
- Do not mix signed and unsigned integers in operations
- Check explicitly that expected boundaries are not exceeded
- Use **size_t** for data structure and array size (guaranteed to be able to hold the size of any data object that the particular C implementation can create)

More examples

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

y is positive (y = 65535).

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

y is negative (y = -1).

```
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
    }

    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). This is caused by the fact that the **len** value is a short integer, so it is converted to -32768, it enters the if clause and gets converted again to +32768; but again this number can not be stored in a short and becomes -32768. Crash!

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)

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

5 Laboratory 2

Really simple:

- non-prepared statements **bad**
- prepared statements **good**

Queries without prepared statements are subject to SQL injection.

```
name_query = "SELECT * FROM names WHERE first_name = " + name + ";"
cursor = conn.execute(name_query)
```

To use prepared statements, replace the query parameters with `?`, create a tuple of parameters (in the correct order) and execute the query with it:

```
query = "SELECT * FROM names WHERE first_name = ?;"
params = (name, )   #tuple
c.execute(query, params)
```

5.1 Homework 2

See the linked PDF for more informations about it.

6 Attack taxonomy (part 3)

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.

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).

Network traffic

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.

Hidden form fields and magic URLs

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

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

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

- 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

7 Laboratory 3

7.1 Postman

Application to make GET requests with parameters. For example in *www.bing.com* you can add a key, which is the name of the parameter, called *q*, and the value *xss*: the request sent is: *https://www.bing.com/search?q=xss*.

7.2 XSS Reflected scripting

The XSS reflected scripting does not alterate the server data, it just show the changes based on the request. The XSS scripts can be avoided by sanitizing the text in input. For example in a Python code this is done by using the *escape()* function.

7.3 WebGoat - XSS

In the search field:

```
<script>console.log(webgoat.customjs.phoneHome() ) </script>
```

or:

```
<script>webgoat.customjs.phoneHome()</script>
```

because the function makes the alert itself.

7.4 Homework 3

See the linked PDF for more informations about it.

7.4.1 After the correction

From the javascript log in the browser it is possible to get the cookie rapidly. The exercise 3 could have been done also by pressing a button to change page or just to fetch the result. In this last case just use:

```
onclick("fetch('secret?cookies='.concat(x)")
```

8 Laboratory 4

8.1 Stored XSS

It is possible to add a new station with a link in the name:

```
/add?id=543&location=<a href="www.bing.com">Torino</a>
```

The content is going to be stored in the database (it is permanent). It is not possible to inject in the id field because there are some checks made. A check on the client side is not going to be really efficient, so it is better to add the sanitization to the server side (same as with the reflected XSS).

8.2 Client-side filtering/tampering

NEVER filter sensitive data client-side: the user can see all the data stored in the client!

```
var users = fetch('localhost:5000/users')
for(const u of users) {
    if (u.role != 'ceo') {
        document.append('<li>' + u.name + u.salary + '</li>')
    }
}
```

NEVER trust input sent by client: the user can manipulate the requests and pass client limitation. For example, if the service lets you choose, with a dropdown menu, between three possibilities but it lets you know that there is a fourth for specific users, it is possible to edit the request to select it (via ZAP).

8.3 Direct Object Reference

Given for example the URL, with the ending number corresponding to the user ID:

```
See all my personal detail at:
https://some.company.tld/app/user/23398
```

If authorization is not well implemented, it may be easy to access private detail just by changing it with a different ID. Direct Object Reference should be avoided or at least made safe.

8.4 WebGoat - Client Side

Bypass front-end restrictions

Users have a great degree of control over the front-end of the web application. They can alter HTML code, sometimes also scripts. This is why apps that require certain format of input should also validate on server-side.
For the first exercise, go in request editor (right->open/resend with Request Editor..) of ZAP, edit the request to:

```
select=option3&radio=option3&checkbox=org&shortInput=123456
```

For the second one, as before:

```
field1=AB&field2=WE&field3=^^^&field4=ven&field5=we&
    field6=aaaaa&field7=aaa&error=0
```

Both exercises can also be made by adding a breakpoint (left of the HUD) and edit the requests before sending them.

Html tampering

Browsers generally offer many options of editing the displayed content. Developers therefore must be aware that the values sent by the user may have been tampered with.
For the exercise, similar to before just edit the request like this:

```
QTY=2&Total=0.99
```

8.5 Homework 4

See the linked PDF for more informations about it. An alternative: check network tab in ispector, find the request made to:

`http://localhost:8881/WebGoat/clientSideFiltering/challenge-store/coupons/123`

remove '123' and get the URL to the JSON file with the coupons.

9 Attack taxonomy (part 4)

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

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

File access

Attackers delete or replace files, provide device names, or traverse directories.

Network name resolution

The application relies on a DNS for network name resolution.

Race conditions

The application is crashed by concurrent code accessing unprotected data.

Unauthenticated keys

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

Random numbers

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

Usability

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

10 Laboratory 5

10.1 WebGoat - Access Control Flaws

The access control flaw covered is the Direct Object Reference. Direct Object References are when an application uses client-provided input to access data and objects.

`https://some.company.tld/dor?id=12345`

These are considered insecure when the reference is not properly handled and allows for authorization bypasses or disclose private data that could be used to perform operations or access data that the user should not be able to perform or access.
In exercise 3 analyze the page and in the network tab search for the profile file. It is easy to recover the infos hidden from the page:

```
"role" : 3,
"color" : "yellow",
"size" : "small",
"name" : "Tom Cat",
"userId" : "2342384"
```

In exercise 4 just access the page with:

`WebGoat/IDOR/profile/2342384`

In exercise 5 try to change the id number, until you get to:

`http://localhost:8881/WebGoat/IDOR/profile/2342388`

`"{role=3, color=brown, size=large, name=Buffalo Bill, userId=2342388}"`

Editing can be done with or without ZAP:

- **With ZAP**: modify the request, making it a PUT and adding the parameters in the body
- **Without ZAP**: open postman and do a GET with the above URL. The session cookies from the current session ('`document.cookie`' in the console of firefox) have to be added to localhost (with path '/WebGoat'), so that the logins are already executed. Then change the request to a PUT and in the **body** tab change to **raw** and switch from **text** to **JSON**, then write:

```
{
    "role" : 1,
    "color" : "red",
    "size" : "large",
    "name" : "Buffalo Bill",
    "userId" : "2342388"
}
```

10.2 Homework 5

See the linked PDF for more informations about it.