Security Testing

Edoardo Righi

December 30, 2019

Contents

1 Attack taxonomy

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

1.1 Attacks

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

SQL Injection

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

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

The query could become:

```
SELECT * FROM customers WHERE id = 1 OR 2>1
SELECT * FROM customers WHERE id = 1; UPDATE accout...
SELECT * FROM customers WHERE id = 1; DROP accout ...
```

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

Cross Site Scripting (XSS)

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

1. The attacker identifies a web site with XSS vulnerabilities
2. The attacker creates a URL that submits malicious input (e.g., including malicious links or JS code) to the attacked web site
3. The attacker tries to induce the victim to click on the URL (e.g., by including the link in an email)
4. The victim clicks the URL, hence submitting malicious input to the attacked web site
5. The web site response page includes malicious links or malicious JS code (executed on the victim's browser)

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

2 Laboratory 1

2.1 OWASP Zap

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

2.2 WebGoat

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

2.3 Homework 1

See the linked PDF for more informations about it.

2.3.1 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*.

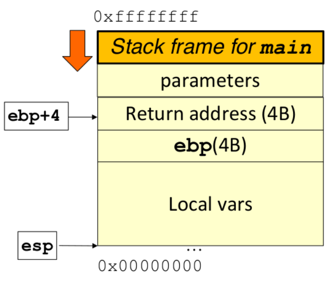
3 Attack taxonomy (part 2)

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

3.1 Call stack

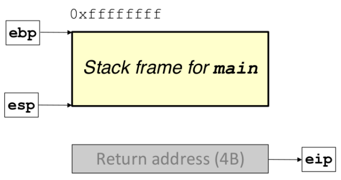
The call stack is a stack data structure that stores information about the active subroutines of a computer program. The call stack grows to lower memory addresses.

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



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

1. The 3 params are saved to the stack
2. Return address (**eip** of **ebp**+ 4) is saved to the stack
3. **ebp** of previous frame is saved to the stack
4. Local variables are pushed to the stack



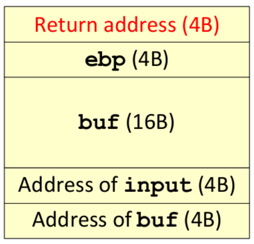
`return: f(x1, x2, x3);`

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

Buffer overflow

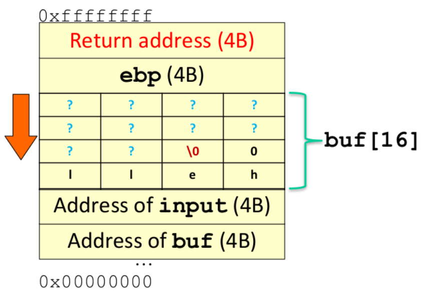
It stands for the attempt to write more data to a fixed length memory block.

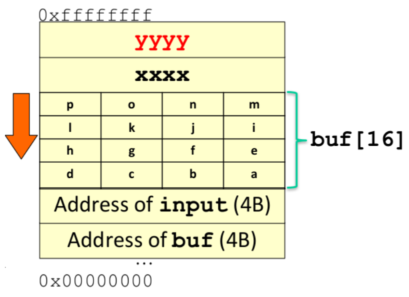
```
#include <stdio.h>
void f(char* input) {
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}
int main(int argc, char* argv[]) {
    f(argv[1]);
    return 0;
}
```



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

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





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

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

Instead of "abcd...", the attacker can input executable code in HEX, called shellcode.
To sum it up, the problem is that user data and control flow information (e.g., function pointer tables, return addresses) are mixed together on the stack and on the heap, hence user data exceeding a buffer may corrupt control flow information. How to spot it? The use of unsafe string manipulation functions (e.g., strcpy) is a wake-up call of it. To fix it:

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

More examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Affected languages

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

Format Strings

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

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

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

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

```
> a.out “hello”
hello
```

But with:

```
> a.out “%x %x”
12ffc0a0 4011e5a1
```

The system POP two values from the call stack and print them (in hex format).

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

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

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

To fix it:

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

More examples

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

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

```
fprintf(STDOUT, "%s", err_msg)
```

Affected languages

- **C**, **C++**, **Perl**: languages supporting (1) format strings, that can be provided externally, and (2) variable number of arguments, which are obtained from the call stack without any check
- High level languages that use C implementations of their string formatting functions

Integer overflow

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

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

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

```
int MAX = 32767000;
int main(int argc, char* argv[]) {
    short len = MAX;
    char s[len+2000];
    strncpy(s, argv[1], 32769000);
}
```

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

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

To fix integer overflow:

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

More examples

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

y is positive (y = 65535).

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

y is negative (y = -1).

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

```
        short z2 = 32000;
        short z = z1 + z2;  // z == -1536
        z = z1 * z2;        // z == 0
    }

    int main(int argc, char* argv[]) {
        short len = strlen(argv[1]);
        char* s;
        if (len < 0)
            len = -len;
        s = malloc(len);
        strncpy(s, argv[1], len);
    }
```

Crashes if length of argv[1] = 32768 (max short +1). This is caused by the fact that the **len** value is a short integer, so it is converted to -32768, it enters the if clause and gets converted again to +32768; but again this number can not be stored in a short and becomes -32768. Crash!

Affected languages

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

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

4 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)
```

4.1 Homework 2

See the linked PDF for more informations about it.

5 Attack taxonomy (part 3)

Command Injection

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

Error handling

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

Network traffic

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

Hidden form fields and magic URLs

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

Improper use of SSL and TLS

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

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

Weak passwords

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

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

6 Laboratory 3

6.1 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*.

6.2 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.

6.3 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.

6.4 Homework 3

See the linked PDF for more informations about it.

6.4.1 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)")
```

7 Laboratory 4

7.1 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).

7.2 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).

7.3 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.

7.4 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
```

7.5 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.

8 Attack taxonomy (part 4)

Data storage

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

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

Information leakage

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

File access

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

Network name resolution

The application relies on a DNS for network name resolution.

Race conditions

The application is crashed by concurrent code accessing unprotected data.

Unauthenticated keys

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

Random numbers

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

Usability

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

9 Laboratory 5

9.1 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"
  }
  ```

9.2 Homework 5

See the linked PDF for more informations about it.

9.2.1 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.

10 Flow Analysis

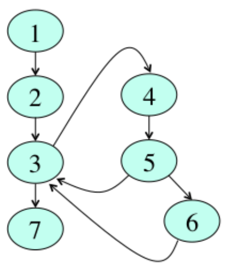
Flow analysis is a general static analysis framework that can be instantiated in several specific code analyses, among which taint analysis. In turn, flow analysis instantiates an even more general static analysis framework, **abstract interpretation**.

10.1 Control flow graph

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

- Node set N: statements of the program P.
- 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).
- Node $n_e \in N$: entry node of P.
- Node $n_x \in N$: exit node of P.

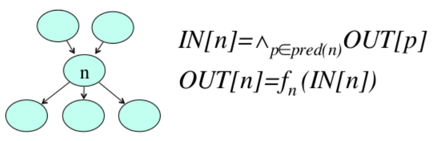
```
<?php
1 $xx = explode(";", $_POST["x"]);
2 $s = $_POST["y"];
3 foreach ($xx as $x) {
4   $s = " " . $s;
5   if (htmlentities($x) == $x)
6       $s .= $x;
 }
7 echo $s;
?>
```



10.2 Flow Analysis Framework

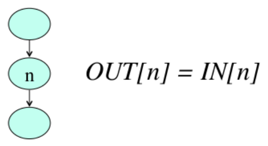
Flow Analysis Framework is a general procedure that can be used to determine properties that hold for a program by propagating proper flow information inside the control flow graph of the program. Flow information is altered during propagation according to the computation performed by each program statement.

1. **Flow information** set V: Flow information that is propagated in the CFG, 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.
$$OUT[n] = f_n(IN[n])$$
3. **Confluence (meet) operator** $\wedge$: To join flow values coming from the OUT of the predecessors (or successors) of current node n:
$$IN[n] = \wedge_{p \in pred(n)} OUT[p]$$
4. **Direction of propagation**: Forward (meet takes output from predecessors) or backward (meet uses successors)



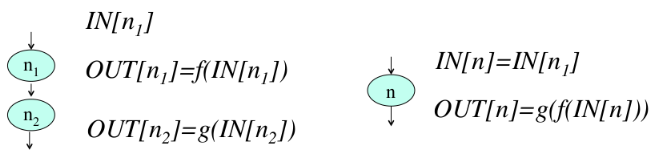
10.2.1 Assumptions

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



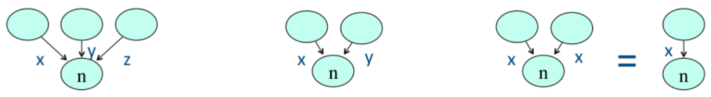
- Transfer functions can be obtained by composition:

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



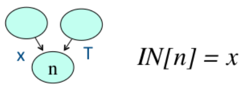
- $\wedge$ is associative, commutative and idempotent:

$$\begin{aligned} x \wedge (y \wedge z) &= (x \wedge y) \wedge z \\ x \wedge y &= y \wedge x \\ x \wedge x &= x1 \end{aligned}$$



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

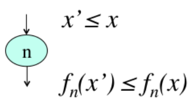
$$T \wedge x = x$$



- 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 $x \leq T \forall x \in V$ by definition.

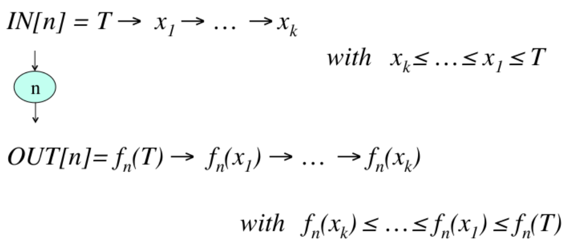


10.2.2 Flow analysis algorithm

```
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.          $IN[n] = \wedge p \in pred(n) OUT[p]$
8.          $OUT[n] = f_n(IN[n])$
9.      end for
10. end while
```

10.2.3 Convergence

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

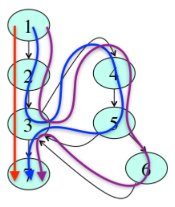

$IN[n] = T \rightarrow x_1 \rightarrow \ldots \rightarrow x_k$
with $x_k \leq \ldots \leq x_1 \leq T$
n
$OUT[n]= f_n(T) \rightarrow f_n(x_1) \rightarrow \ldots \rightarrow f_n(x_k)$
with $f_n(x_k) \leq \ldots \leq f_n(x_1) \leq f_n(T)$

10.2.4 Meet over path solution

Meet over path (MOP) solution:

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

EXACT solution (meet over feasible paths):



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

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

10.2.5 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:

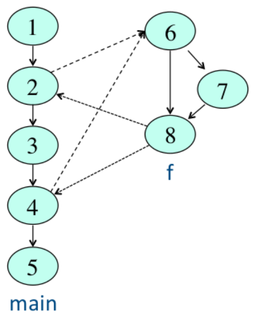
$$MOP[n] = FA[n] \text{ if } f_n(x \wedge y) = f_n(x) \wedge f_n(y)$$

11 Call string

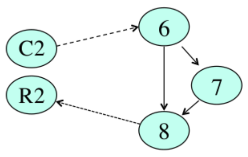
11.1 Interprocedural flow analysis

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

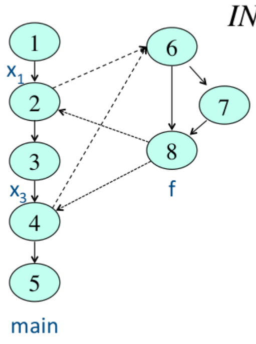


<1, C2, 6, 7, 8, R2, 3, C4, 6, 8, R4, 5> is realizable;
<1, C2, 6, 7, 8, R4, 5> is not realizable;



11.2 Call string method

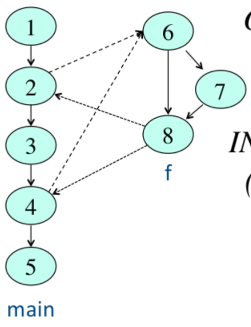
- Flow information x is propagated together with the associated call string: (x, CS). The call string is k-bounded in the presence of recursion.



$IN[6] = (x_1, \text{“C2”}), (x_3, \text{“C4”})$

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

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

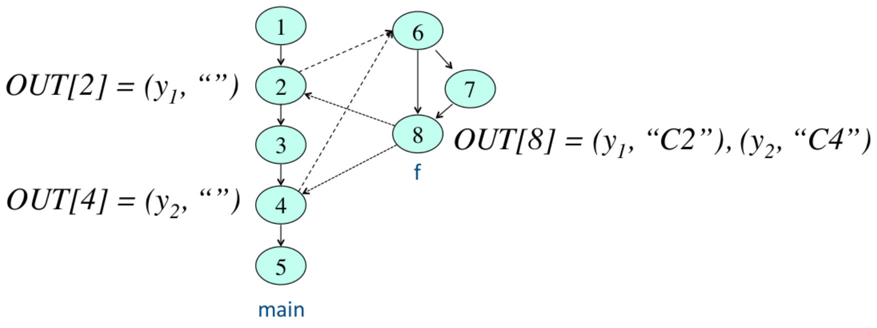


$OUT[6] = (y_1, \text{“C2”}), (y_2, \text{“C4”})$

$OUT[7] = (z_1, \text{“C2”}), (z_2, \text{“C4”})$

$IN[8] = OUT[6] \wedge OUT[7] =$
$(y_1 \wedge z_1, \text{“C2”}), (y_2 \wedge z_2, \text{“C4”})$

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



+ examples in videos 'Lecture_5_Flow_analysis' (at the end) and 'Lecture_6_Call_String'.

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");
```

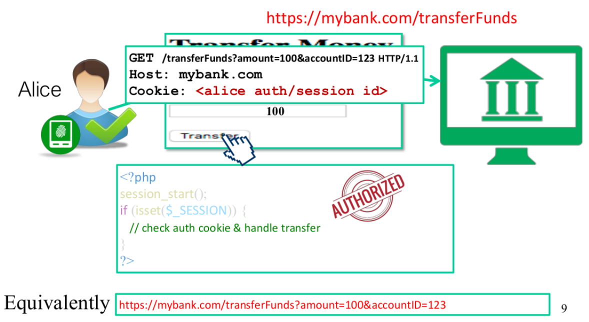
13 Attack taxonomy (part 5)

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:

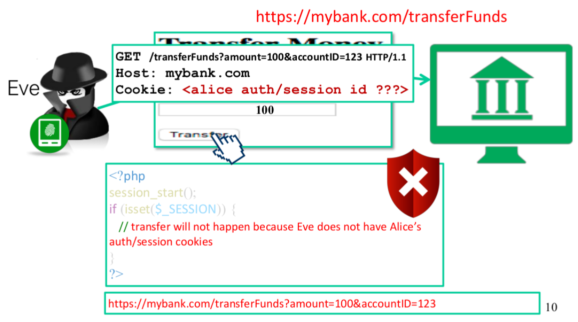


```
GET /transferFunds?amount=100&accountID=123 HTTP/1.1
Host: mybank.com
Cookie: <alice auth/session id>
```

```
<?php
session_start();
if (isset($_SESSION)) {
  // check auth cookie & handle transfer
}
?>
```

```
https://mybank.com/transferFunds?amount=100&accountID=123
```

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

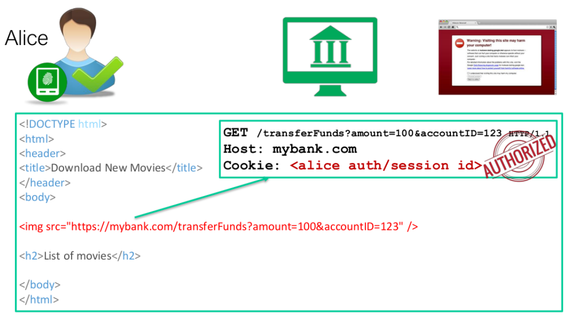


```
GET /transferFunds?amount=100&accountID=123 HTTP/1.1
Host: mybank.com
Cookie: <alice auth/session id ???>
```

```
<?php
session_start();
if (isset($_SESSION)) {
  // transfer will not happen because Eve does not have Alice's
auth/session cookies
}
?>
```

```
https://mybank.com/transferFunds?amount=100&accountID=123
```

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:



```
<!DOCTYPE html>
<html>
<header>
<title>Download New Movies</title>
</header>
<body>

<img src="https://mybank.com/transferFunds?amount=100&accountID=123" />

<h2>List of movies</h2>

</body>
</html>
```

```
GET /transferFunds?amount=100&accountID=123 HTTP/1.1
Host: mybank.com
Cookie: <alice auth/session id>
```

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

14 Laboratory 7

14.1 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>
```

14.2 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.

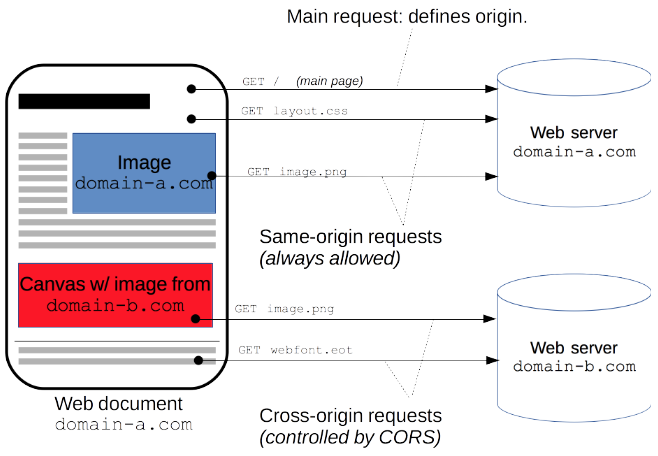
An example of a cross-origin request: the front-end JavaScript code served from

`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.



14.3 Homework 7

See the linked PDF for more informations about it.

15 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}
```

Meet operator

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

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

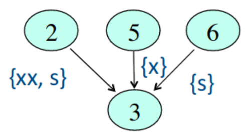
As a consequence, the meet operator is union: the taint set at node n is the union of the taint sets of its predecessors.

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 ($x \wedge \bot = \bot \forall x$)
- 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;
?>
```



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

15.1 Transfer function

The transfer function for taint analysis has the form:

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

with:

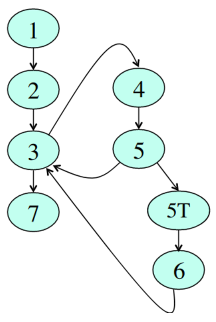
- $GEN[n] = \{x | x \text{ is assigned an input value at statement n}\} \cup \{x | \exists y : x \text{ is assigned a value obtained from } y \wedge y \rightarrow T\}$
- $KILL[n] = \{x | x \text{ is sanitized by statement n}\} \cup \{x | \forall y : x \text{ is assigned a value obtained from } y \wedge y \rightarrow F\}$

Since transfer functions are monotonic, convergence is ensured.

15.2 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);
```

16 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.

17 Dynamic Taint Analysis

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

becomes:

```php
<?php
  $xx = explode(";", $_POST["x"]);
  makeTainted("xx"); // $Tainted["xx"] = true;
  $s = $_POST["y"];
  makeTainted("s");
  foreach ($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;
?>
```

17.1 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).

17.2 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).

17.3 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.

17.4 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);
```

17.5 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

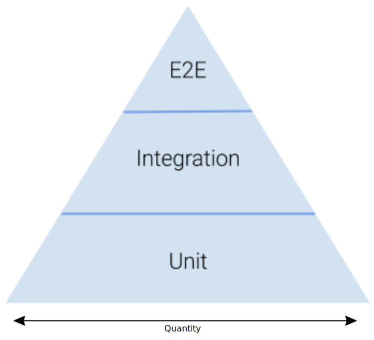
17.6 Example

Watch the video.

18 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:

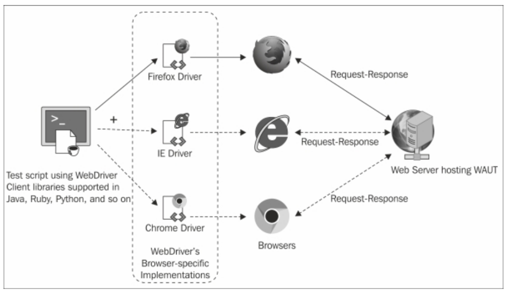
- Sequence of user events from the initial page (state) to the target page (state)
- Sequence of Inputs (e.g., to fill in a form)
- Expected result (oracle)

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).


Firefox Driver
IE Driver
Chrome Driver
+
Test script using WebDriver Client libraries supported in Java, Ruby, Python, and so on
WebDriver's Browser-specific Implementations
Browsers
Request-Response
Request-Response
Request-Response
Web Server hosting WAUT

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

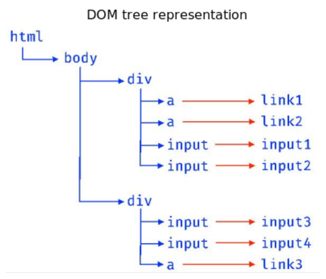
  ```
  /html/body/div/input => input{1,2,3,4}
  /html/body/div[1]/input[2] => input2
  ```
- **Relative XPath.** It begins with double slash "//' which represents search in the entire web page

  ```
  //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:

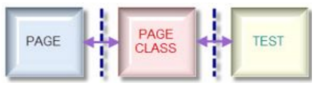
```
//1: go to web page
driver.get(URL);

//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);
```

19 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);
```

20 Laboratory 11-12

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

21 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);
```

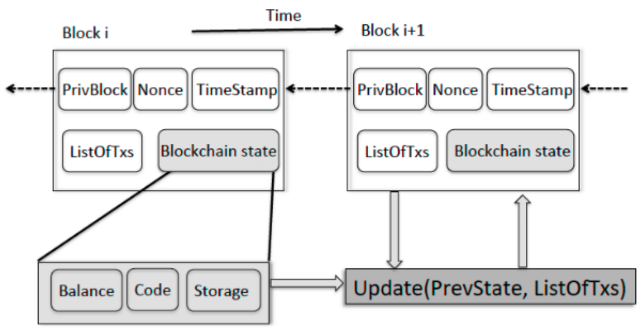
`executeScript` is a powerful tool: it enables to edit manually elements and send XHR requests.

22 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).

22.1 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).

22.2 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.

22.2.1 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.

22.3 Security bugs

There are 4 types of bugs:

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

22.3.1 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.

```
contract MarketPlace{
  uint public price;
  uint public stock;
  /.../
  function updatePrice(uint _price){
    if (msg.sender == owner)
      price = _price;
  }
  function buy (uint quant) returns (uint){
    if (msg.value < quant * price || quant > stock)
      throw;
    stock -= quant;
    /.../
  }}
```

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.

22.3.2 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.

22.3.3 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.

22.3.4 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$.

22.4 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.

22.4.1 Guarded transactions

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

22.4.2 Deterministic timestamp

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

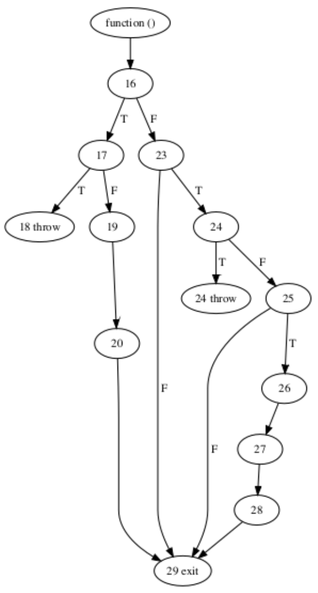
- The block index always increase by 1;
- no flexibility at attacker side.

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

22.4.3 Better exception handling

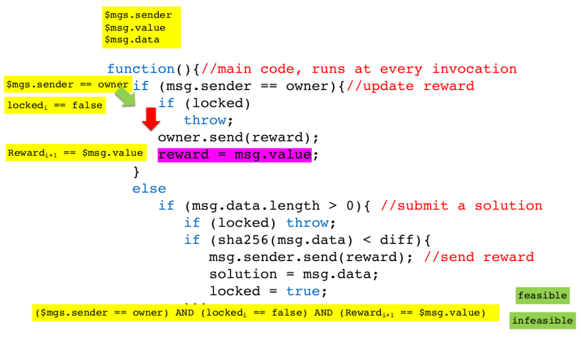
- Automatically propagate exceptions (at EVM level) from callee to caller.
- Adding explicit **throw** and **catch** statements.

22.5 Static Analysis



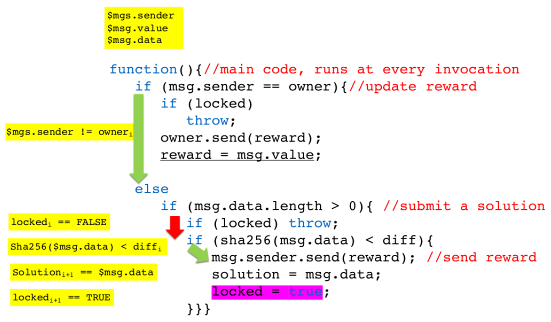
```
15 function(){
16   if (msg.sender == owner){
17      if (locked)
18         throw;
19      owner.send(reward);
20      reward = msg.value;
21   }
22   else
23      if (msg.data.length > 0){
24         if (locked) throw;
25         if (sha256(msg.data) < diff){
26            msg.sender.send(reward);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

Symbolic execution



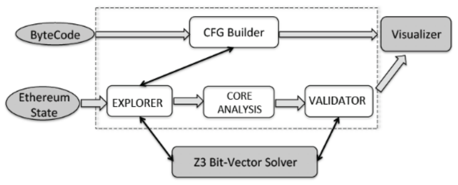
```
function(){//main code, runs at every invocation
   if (msg.sender == owner){//update reward
      if (locked)
         throw;
      owner.send(reward);
      reward = msg.value;
   }
   else
      if (msg.data.length > 0){ //submit a solution
         if (locked) throw;
         if (sha256(msg.data)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

$mgs.sender
$msg.value
$msg.data
$mgs.sender == owner
locked$_i$ == false
Reward$_{i+1}$ == $msg.value
($mgs.sender == owner) AND (locked$_i$ == false) AND (Reward$_{i+1}$ == $msg.value)
feasible
infeasible



```
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;
         }}}
```

22.5.1 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

22.6 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

22.7 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