Security (Breaches)

Security

- Injection
  - SQL Injection
  - XXE (XML External Entity) Injection (attacker is able to cause Denial of Service (DoS) and access local or remote files and services, by abusing a widely available, rarely used feature in XML parsers.
  - Command Injection (execution of arbitrary commands on the host operating system via a vulnerable application)
- Broken Authentication And Session Management
  - Use of Insufficiently Random Values (When software generates predictable values in a context requiring unpredictability.
  - Session Fixation (hijack a valid user session)
- Cross Site Scripting
  - Reflected XSS
  - Persistent (Stored) XSS
  - DOM XSS
- Directory Object References

  Directory Traversal or ../ (dot dot slash) attack (unauthorized access to the server file system)

Security

- 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
- Cross Site Scripting
  - Reflected XSS
  - Persistent (Stored) XSS

Going to Showcase

- Injection
  - SQL Injection
- Security Misconfiguration
  - Leftover Debug Code
- Cross Site Scripting
  - Reflected XSS
  - Persistent (Stored) XSS

SQL Injection

SQL Injection is one of the most dangerous vulnerabilities a web application can be prone to. When a user's input is being passed unvalidated and unsanitized as part of an SQL query that means that the user can manipulate the query itself and force it to return different data than what it was supposed to return.

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

Login application for user authentication

Follow Interest

Log In

test

•••

Log In

New User Click here to Register

Code Behind

```
def authenticate(request):
    email = request.POST['email']
    password = request.POST['password']
    sql = "select * from users where (email ='" + email + "' and password ='" +      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 Attack 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 InternalServerError`**

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

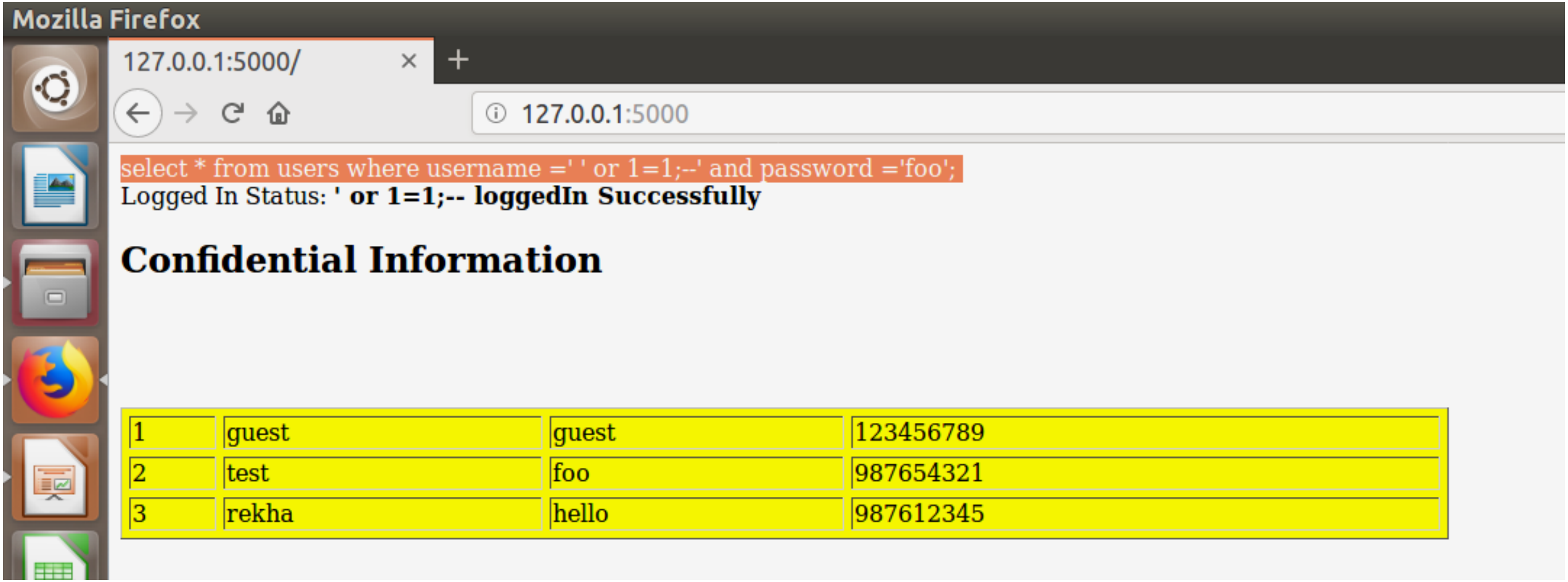
Hacker got the clue

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: `alice@bank.com`
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, including the extra `'` and **`)`** character.

Below is the raw query sent to the DB:

select * from users where username =' ' or 1=1;--' and password ='foo';



Mitigation

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

  Prepared statements are used to abstract SQL statement syntax from input parameters. Statement templates are first defined at the application layer, and the parameters are then passed to them.

  Aside from a better security posture against SQL injection attacks, prepared statements offer improved code quality from a legibility and maintainability perspective due to separation of the SQL logic from its inputs.

Going to Showcase

- Injection
  - SQL Injection
- Security Misconfiguration
  - Leftover Debug Code
- Cross Site Scripting
  - Reflected XSS
  - Persistent (Stored) XSS

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,append ?debug=1 flag within the URL 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

```
def authenticate(request):
    con = sql.connect("database.db")
    isDebug = request.form.get('debug', False)
    isDebug = isDebug and isDebug == "1"

# bypass the authentication code in debugging mode
    if isDebug:
        loggedIn = "admin loggedIn Successfully"
        sqlQuery = "Running in debug Mode"
    Else:
# actual authentication code follows
        ....

    return isDebug, "<h1>"+ sqlQuery + "\r\n<br> <u>Logged In Status:</u><br> <strong>" + loggedIn +
"</strong></h1>"
```

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,append ?debug=1 flag within the URL to access the application without authentication. -->

      <!--input type="hidden" name="debug" id="debug" value="1" / -->
        <input type="submit" class="button" value="Log In"/>
</form>
```

Fix

- 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
- Cross Site Scripting
  - Reflected XSS
  - Persistent (Stored) XSS

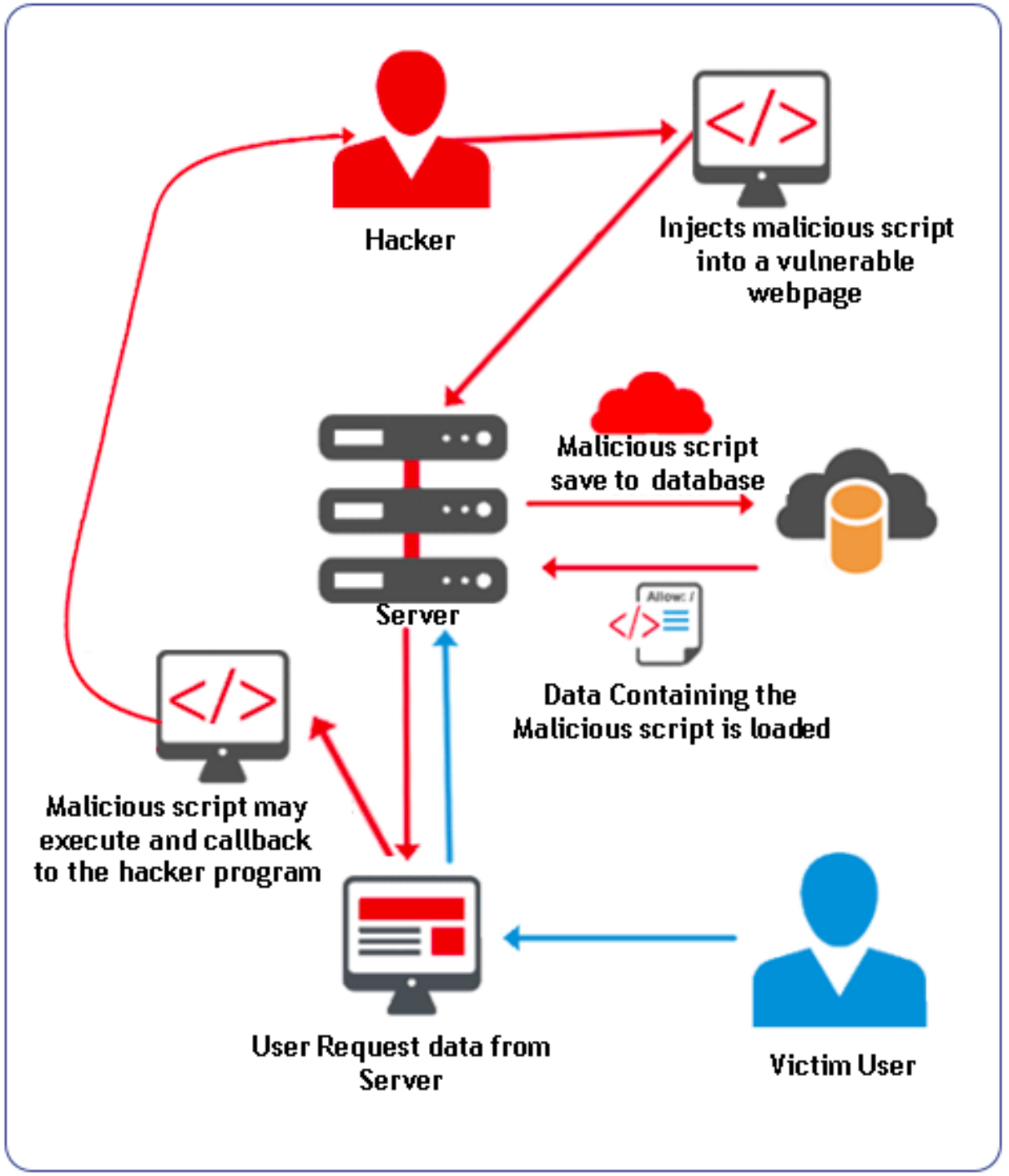
Cross-Site Scripting (XSS)

- XSS enables attackers to inject client-side script into Web pages viewed by other users.
- XSS vulnerability is introduced when a web site accepts input from the client without first validating that the data is “clean” in the expected format and also when the web site outputs data to the web client without sanitizing it, i.e. removing or masking any potentially dangerous code.
- This lets the hacker’s script execute on a victim's browser and show arbitrary content, that the victim believes is real and from the application, and so he provides his information.
- This means that XSS attacks potentially harm the users of the application (victims) but not the application itself.

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is a vulnerability that permits an attacker to inject code (typically HTML or JavaScript) into contents of a website not under the attacker's control.

When a victim views such a page, the injected code executes in the victim's browser. the malicious code has the privileges of the user and can take control of the session between the browser and a website, and access sensitive data or redirect the browser to a malicious web site.



Cross-Site Scripting (XSS)

- When an attacker discovers a server that is not validating or sanitizing data, it will send the server malicious data,

  e.g. by filling out a form with input that is a script instead of simply text.

- Since the server is not doing input validation,

  it either incorrectly stores the script in its database where it gets executed by the next user accessing the database

  or it gets reflected back to the browser.

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

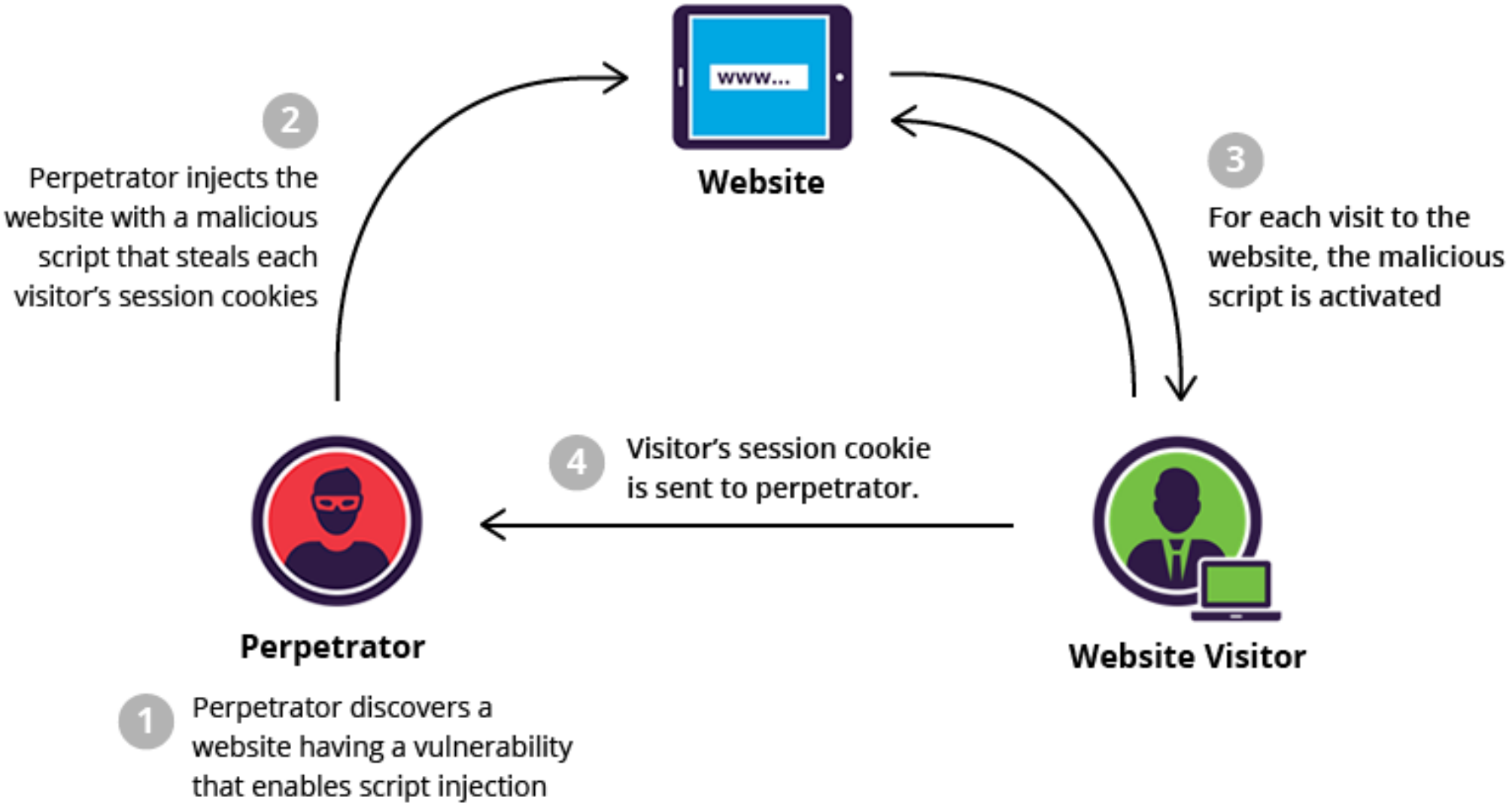
Reflected XSS

reflected XSS attack is usually a link that contains malicious code. When someone clicks on that link, they are taken to a vulnerable website and that malicious code is ‘reflected’ back into their browser to perform some malicious action.

https://www.youtube.com/watch?v=dFci82qwXA0

Cross-Site Scripting

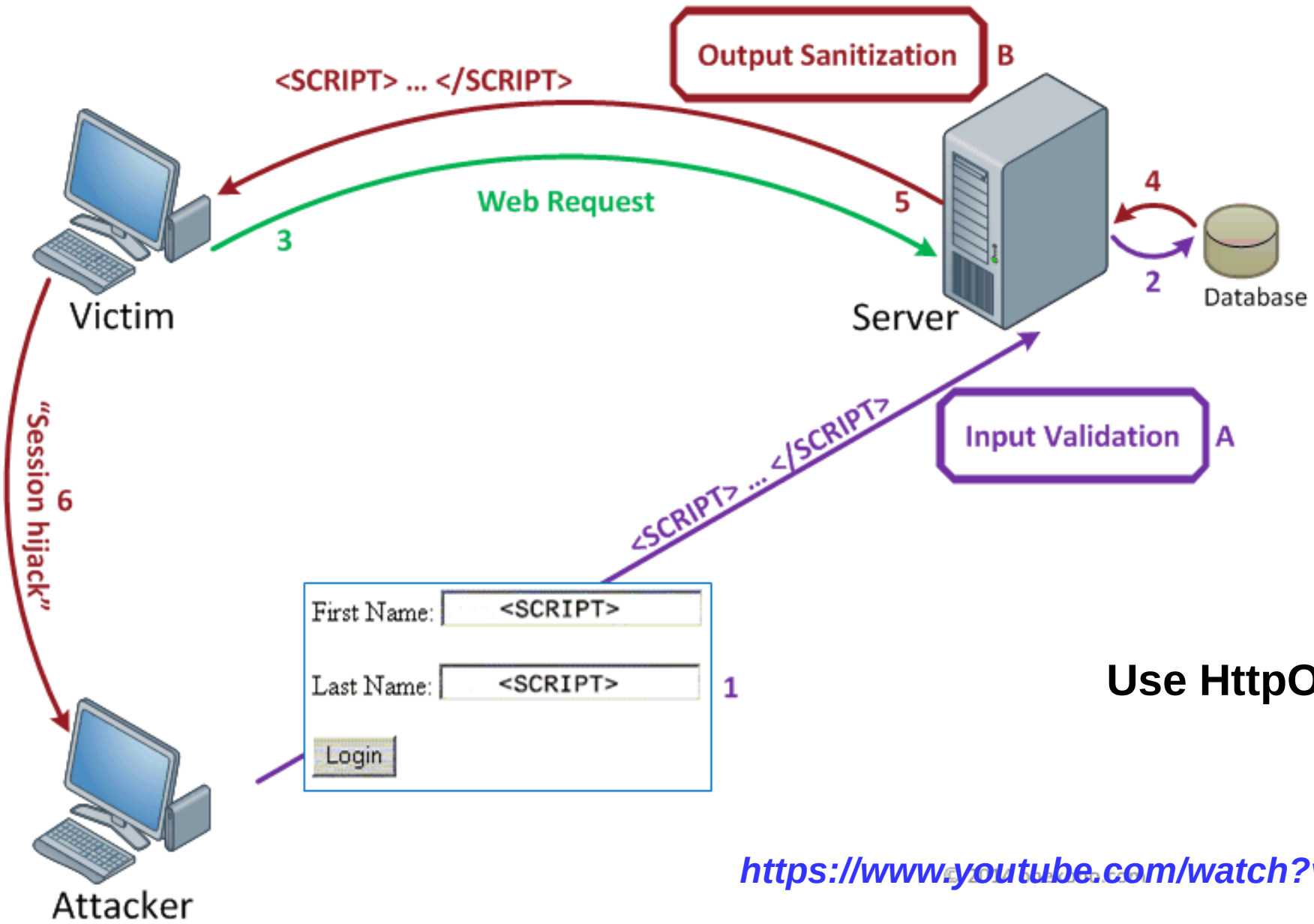
(stealing session cookie)



Cross-Site Scripting (stored XSS)

- Stored XSS Attacks – are those where the code to exploit the vulnerability is permanently stored on the target servers (typically in a database) and can be retrieved to create pages sent to any user visiting the site. The victim retrieves the injected script from the server when it requests the stored information.
- Forums and blogs that allow the use of HTML and that store information in databases are potentially vulnerable to this form of exploit.
- Protecting against stored XSS exploits involves input validation and output sanitization, i.e. translating the elements such that it will not be interpreted by the browser as code.

Stored Cross-Side Scripting



Use HttpOnly flag

https://www.youtube.com/watch?v=F3e1zGh3sro

3 Ways to Prevent XSS

Escaping: Converting a control character to its escape sequence. For example, a < symbol may be converted to < so that the characters following the < are not interpreted as an XML tag instead of XML content. Escape data as much as possible on output to avoid XSS and malformed HTML.

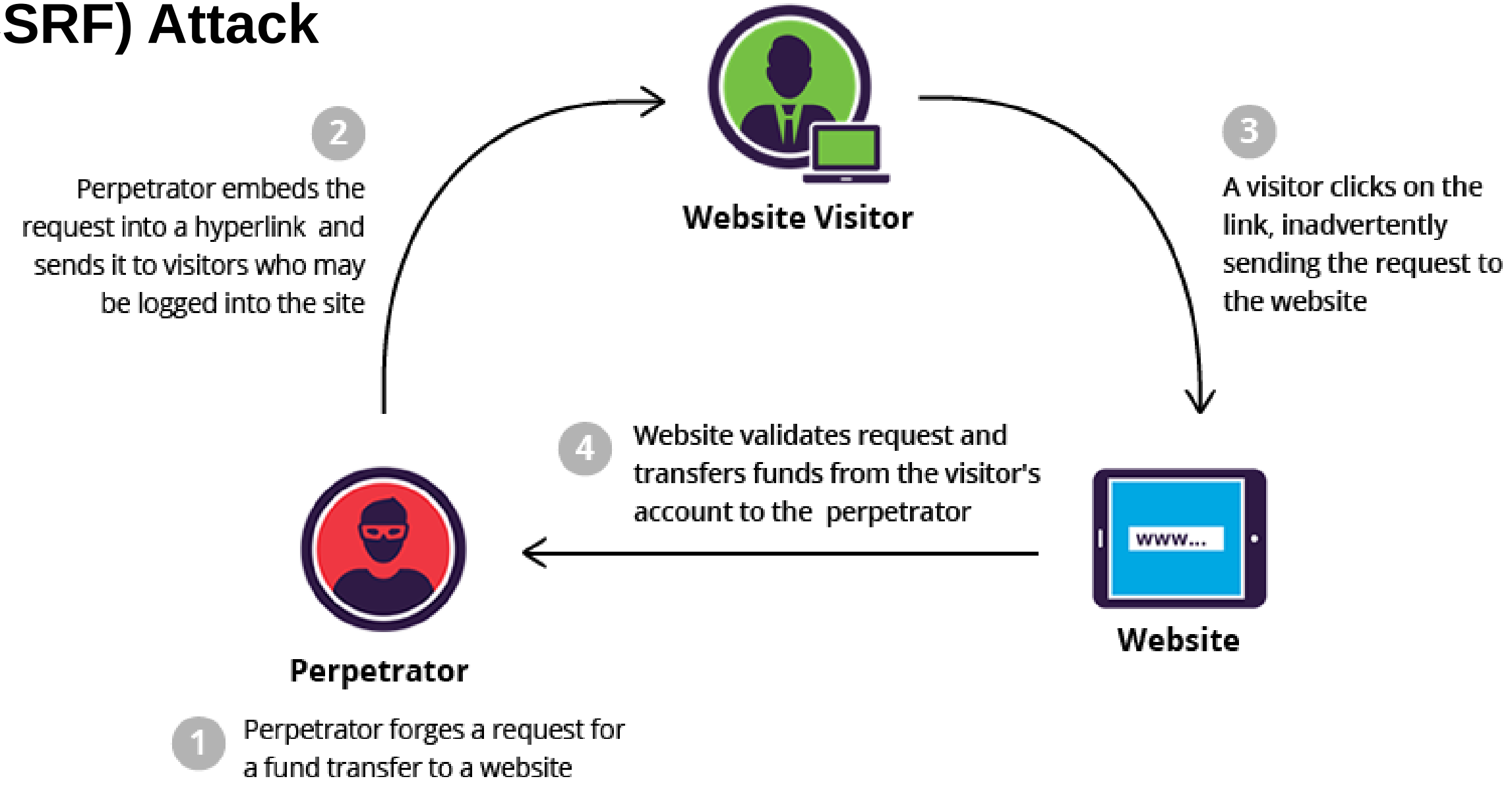
< --> < > --> >

Sanitized: Like escaping, but instead of replacing the control character, it is simply removed. Scrubbing the data clean of potentially harmful markup, changing unacceptable user input to an acceptable format.

Validated: Comparison of an input against a white list or regular expression to detect control characters or other character sequences that would trigger an unauthorized behavior. For example, an account number entered by a user might be validated against a list of account numbers known to be tied to the user.

A combination of escaping, Sanitizing, and validation that ensures that an input to a system function does not trigger an unexpected and unauthorized behavior.

Cross Site Request Forgery (CSRF) Attack



XSS Prevention

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

Escaping: HTML tags, attribute values, javascript data values

Use HTTPOnly cookie flag

Auto-Escaping Template System