Securing Web Applications from Top Ten Security Threats

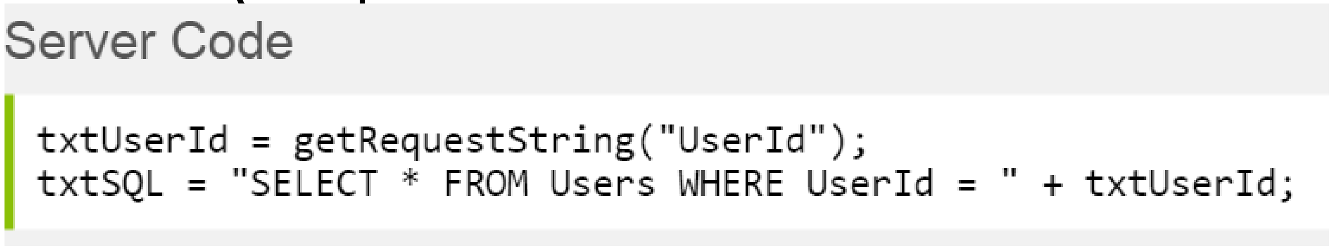
OWASP Top Ten (2017 Edition)
(Open Web Application Security Project)

1. *Injection*
2. Broker Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. *Broken Access Control*
6. *Security Misconfiguration*
7. Cross-Site Script (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging and Monitoring

More details @
https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

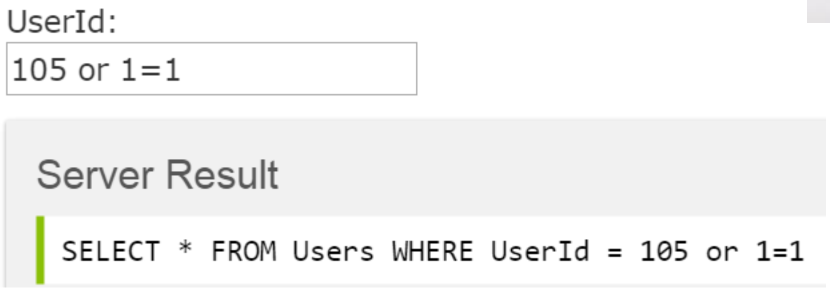
1. Injection

- Executing code provided (injected) by attacker
  - SQL injection



Server Code

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```



UserId:

105 or 1=1

Server Result

```
SELECT * FROM Users WHERE UserId = 105 or 1=1
```

- Solutions:
  - validate user input
  - escape values (use escape functions)
  - use parameterized queries (SQL)
  - enforce least privilege when accessing a DB, OS etc.

`'` -> `\'`

2. Broken Authentication

Hacker uses flaws in the authentication or session management functions to **impersonate** other users:

- **Scenario #1:** Session timeout isn’t set properly. Instead of selecting “logout” the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.
- **Scenario #2:** Session hijacking by stealing session id (e.g., using eavesdropping if not https) then hacker uses this authenticated session id to connect to application without needing to enter user name and password.
- **Scenario #3:** Hacker gains access to the system's password database. User passwords are not properly hashed, exposing every users' password to the attacker.
- **Scenario #4:** Brute force password guessing, more likely using tools that generate random passwords.

2. Broken Authentication - Solutions

- User authentication credentials should be **protected when stored** using hashing or encryption.
- Session IDs should not be exposed in the URL, **use cookies for storing session ID**
- Session IDs should **timeout**. User sessions or authentication tokens should get properly invalidated during logout.
- Session IDs should be regenerated after successful login.
- Passwords, session IDs, and other credentials should not be sent over unencrypted connections (**use https** for login)
- Enforce **minimum passwords length and complexity** makes a password difficult to guess
- Enforce **account disabling** after an established number of invalid login attempts (e.g., three attempts is common).

3. Sensitive Data Exposure

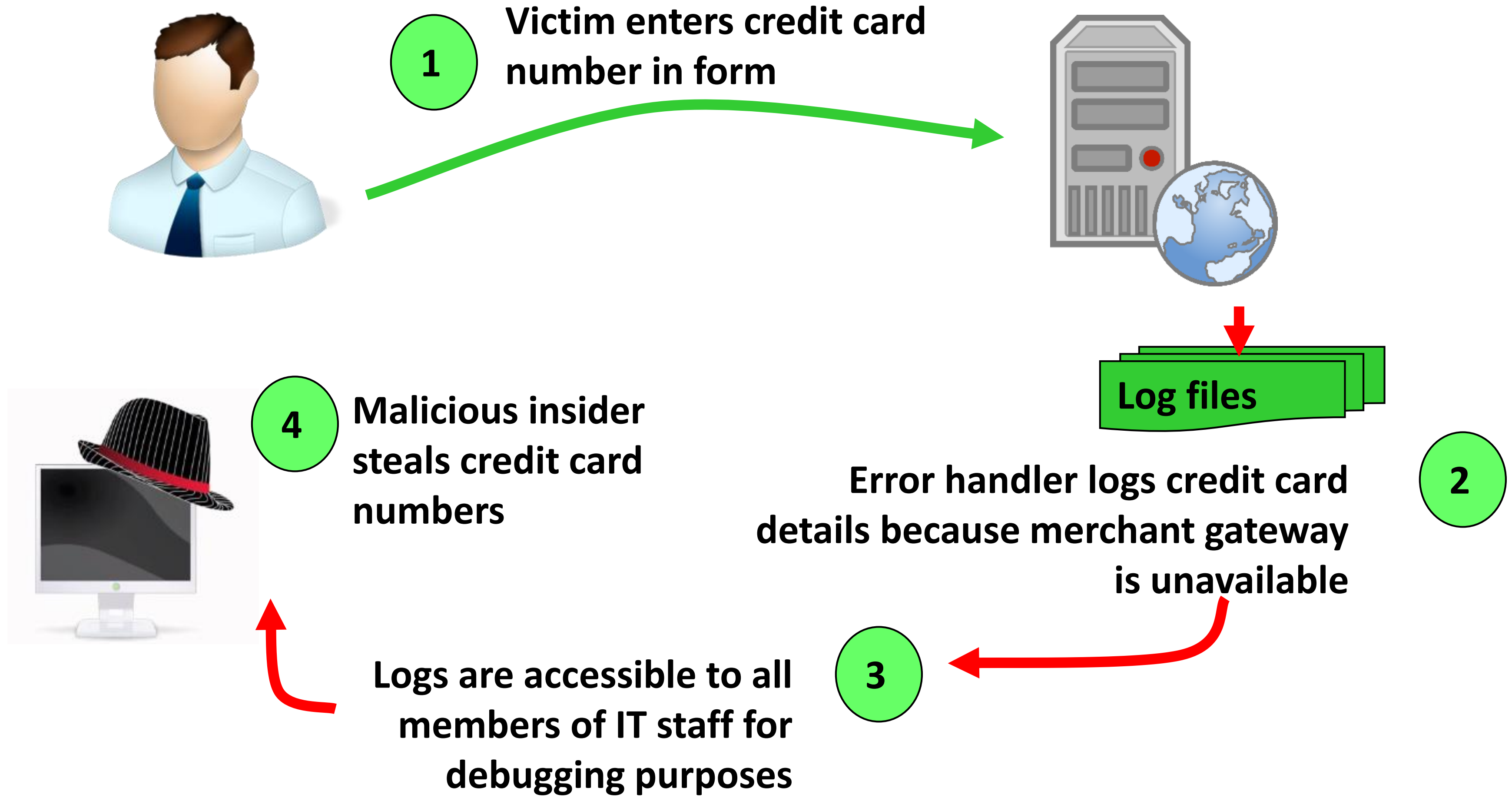
Storing and transmitting sensitive data insecurely

- Failure to identify all sensitive data
- Failure to identify all the places that this sensitive data gets stored
  - Databases, files, directories, log files, backups, etc.
- Failure to identify all the places that this sensitive data is sent
  - On the web, to backend databases, to business partners, internal communications
- Failure to properly protect this data in every location

Typical Impact

- Attackers access or modify confidential or private information
  - e.g, credit cards, health care records, financial data (yours or your customers)
- Attackers extract secrets to use in additional attacks
- Company embarrassment, customer dissatisfaction, and loss of trust
- Expense of cleaning up the incident, such as forensics, sending apology letters, reissuing thousands of credit cards, providing identity theft insurance
- Business gets sued and/or fined

3. Sensitive Data Exposure – Example



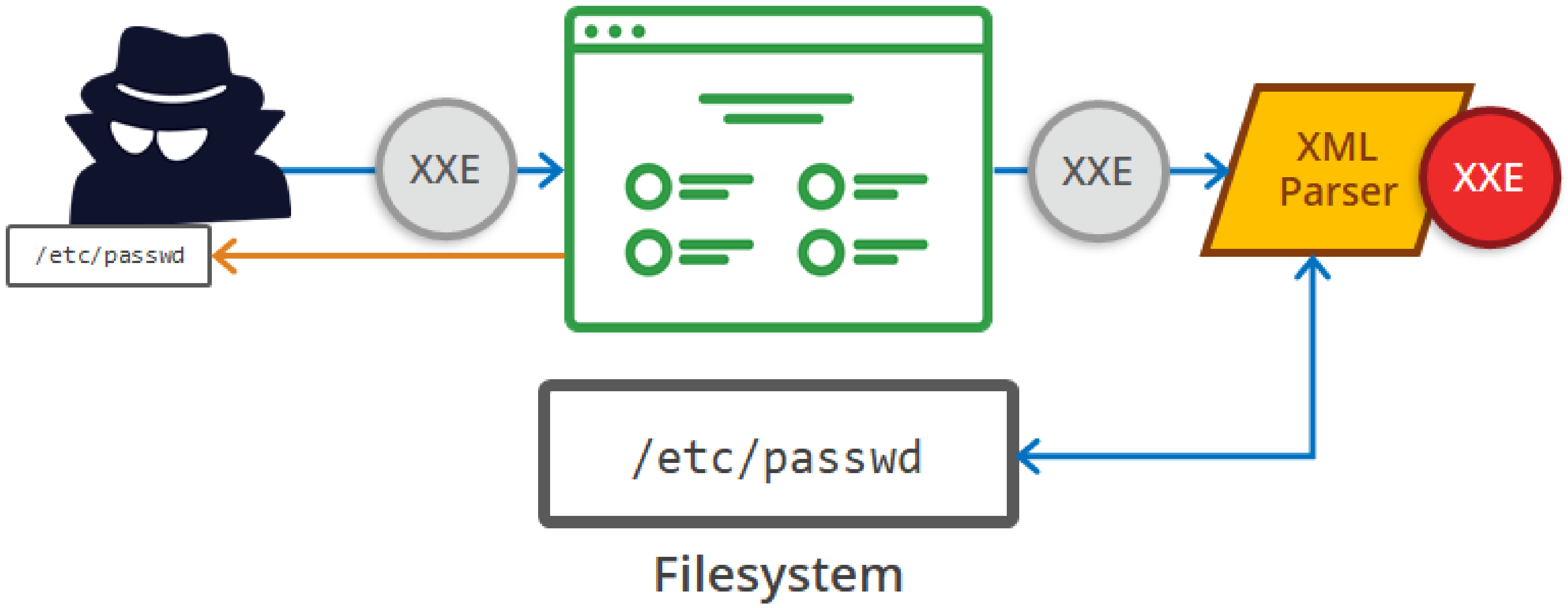
1 Victim enters credit card number in form

2 Error handler logs credit card details because merchant gateway is unavailable

3 Logs are accessible to all members of IT staff for debugging purposes

4 Malicious insider steals credit card numbers

4. XML External Entities (XXE)

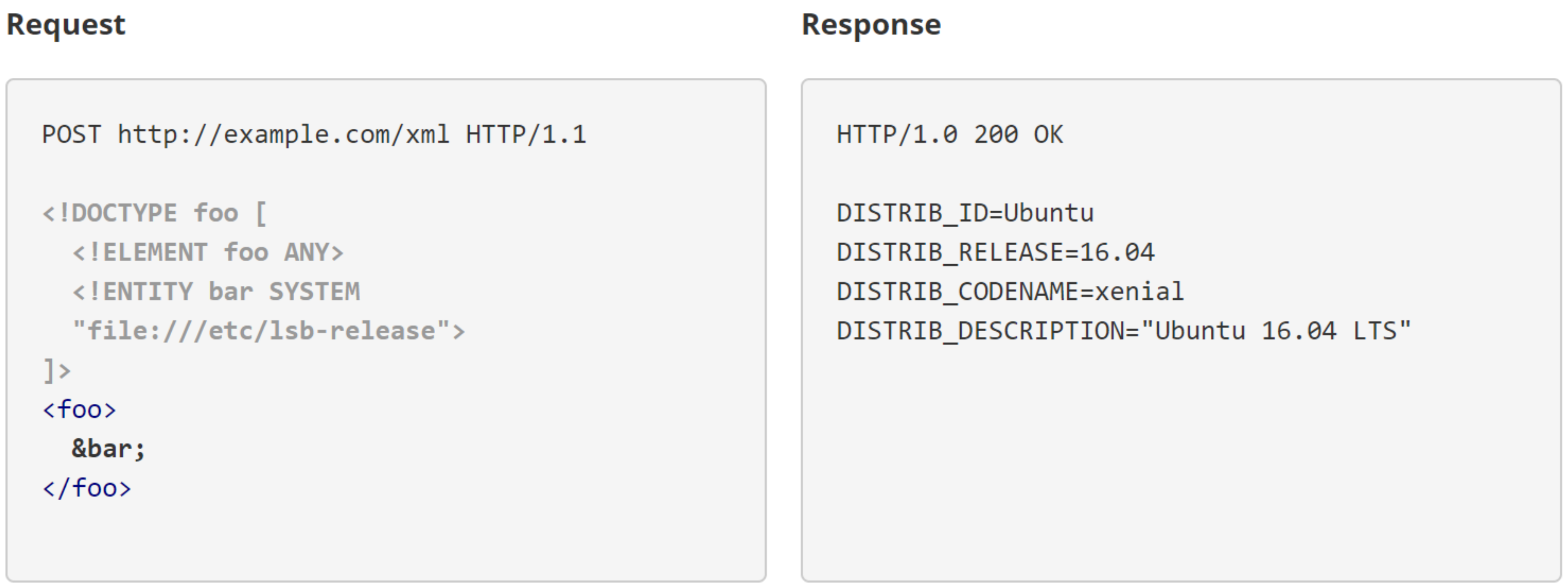


- The application accepts XML documents directly with malicious XML data, which is then parsed by an XML processor that has document type definitions (DTDs) enabled.

Scenario: An attacker attempts a denial-of-service attack by including a potentially endless file:

<!ENTITY xxe SYSTEM "file:///dev/random" >]>

XXE Example



Request

```
POST http://example.com/xml HTTP/1.1

<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM
  "file:///etc/lsb-release">
]>
<foo>
  &bar;
</foo>
```

Response

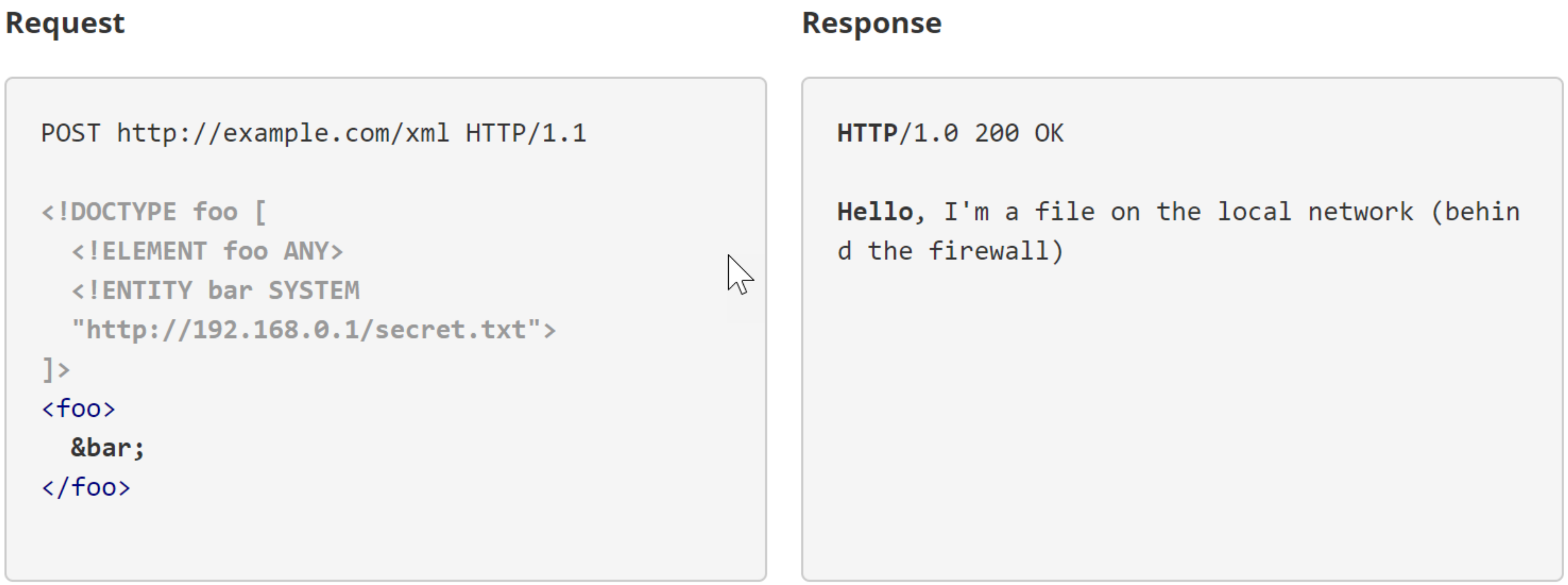
```
HTTP/1.0 200 OK

DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04 LTS"
```

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
  <foo>&xxe;</foo>
```

XXE Example



Request

```
POST http://example.com/xml HTTP/1.1

<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM
  "http://192.168.0.1/secret.txt">
]>
<foo>
  &bar;
</foo>
```

Response

```
HTTP/1.0 200 OK

Hello, I'm a file on the local network (behin
d the firewall)
```

Scenario #2: An attacker probes the server's private network:

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
   <!DOCTYPE foo [
   <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
   <foo>&xxe;</foo>
```

5. Broken Access Control

Insecure Direct Object References

- Attacker manipulates the URL or form values to get unauthorized access
  - to objects (data in a database, objects in memory etc.):

    ```
    http://shop.com/cart?id=413246    (your cart)
    http://shop.com/cart?id=123456    (someone else's cart ?)
    ```
  - to files:

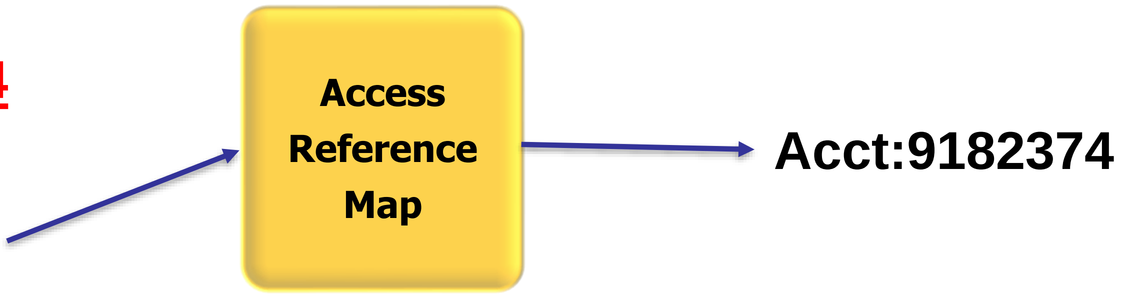
    ```
    http://s.ch/?page=home             -> home
    http://s.ch/?page=/etc/passwd      -> /etc/passwd
    ```
- Solution:
  - avoid exposing IDs, keys, filenames to users if possible
  - validate input, accept only correct values
  - verify authorization to all accessed objects (files, data etc.)

5. Broken Access Control

- Eliminate the direct object reference
  - Replace them with a temporary mapping value such as a random mapping

Instead of :
http://app?id=9182374

Use:
http://app?id=7d3J93



- Validate the direct object reference
  - Verify the user is allowed to access the target object
  - Verify the requested mode of access is allowed to the target object (e.g., read, write, delete)

5. Broken Access Control

Missing Function Level Access Control

- Often Web applications verify function level access rights before making that functionality visible in the UI. However, applications need to **perform the same access control checks on the server when each function is accessed**.
- “Hidden” URLs that don’t require further authorization

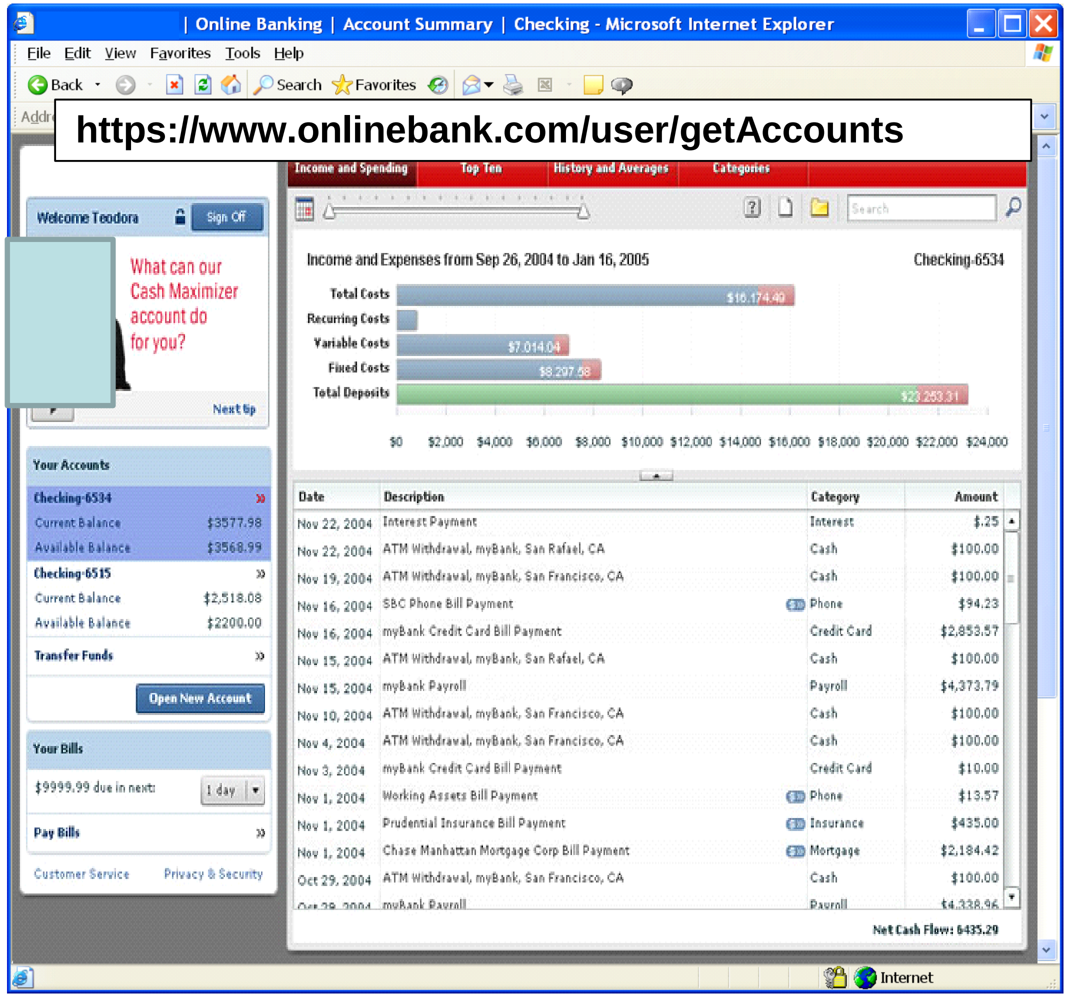
  `http://site.com/`**`admin/adduser?name=x&pwd=x`**

  (even if `http://site.com/`**`admin/`** requires authorization)

- Solution
  - Add missing authorization ☺
  - Don‘t rely on **security by obscurity** – it will not work!

5. Broken Access Control

Missing Function Level Access Control Illustrated



| Online Banking | Account Summary | Checking - Microsoft Internet Explorer

https://www.onlinebank.com/user/getAccounts

Welcome Teodora — Sign Off

What can our Cash Maximizer account do for you?

Next tip

Your Accounts

Checking-6534	»
Current Balance	$3577.98
Available Balance	$3568.99
Checking-6515	»
Current Balance	$2,518.08
Available Balance	$2200.00
Transfer Funds	»

Your Bills

$9999.99 due in next: 1 day

Pay Bills »

Customer Service — Privacy & Security

Income and Spending | Top Ten | History and Averages | Categories

Income and Expenses from Sep 26, 2004 to Jan 16, 2005 — Checking-6534

Total Costs	$16,174.40
Recurring Costs	
Variable Costs	$7,014.04
Fixed Costs	$8,297.58
Total Deposits	$23,253.31

Date	Description	Category	Amount
Nov 22, 2004	Interest Payment	Interest	$.25
Nov 22, 2004	ATM Withdrawal, myBank, San Rafael, CA	Cash	$100.00
Nov 19, 2004	ATM Withdrawal, myBank, San Francisco, CA	Cash	$100.00
Nov 16, 2004	SBC Phone Bill Payment	Phone	$94.23
Nov 16, 2004	myBank Credit Card Bill Payment	Credit Card	$2,853.57
Nov 15, 2004	ATM Withdrawal, myBank, San Rafael, CA	Cash	$100.00
Nov 15, 2004	myBank Payroll	Payroll	$4,373.79
Nov 10, 2004	ATM Withdrawal, myBank, San Francisco, CA	Cash	$100.00
Nov 4, 2004	ATM Withdrawal, myBank, San Francisco, CA	Cash	$100.00
Nov 3, 2004	myBank Credit Card Bill Payment	Credit Card	$10.00
Nov 1, 2004	Working Assets Bill Payment	Phone	$13.57
Nov 1, 2004	Prudential Insurance Bill Payment	Insurance	$435.00
Nov 1, 2004	Chase Manhattan Mortgage Corp Bill Payment	Mortgage	$2,184.42
Oct 29, 2004	ATM Withdrawal, myBank, San Francisco, CA	Cash	$100.00
Oct 29, 2004	myBank Payroll	Payroll	$4,338.96

Net Cash Flow: $435.29

- Attacker notices the URL indicates his role

  /user/getAccounts

- He modifies it to another directory (role)

  /admin/getAccounts, or

  /manager/getAccounts

- Attacker views more accounts than just their own

6 – Security Misconfiguration

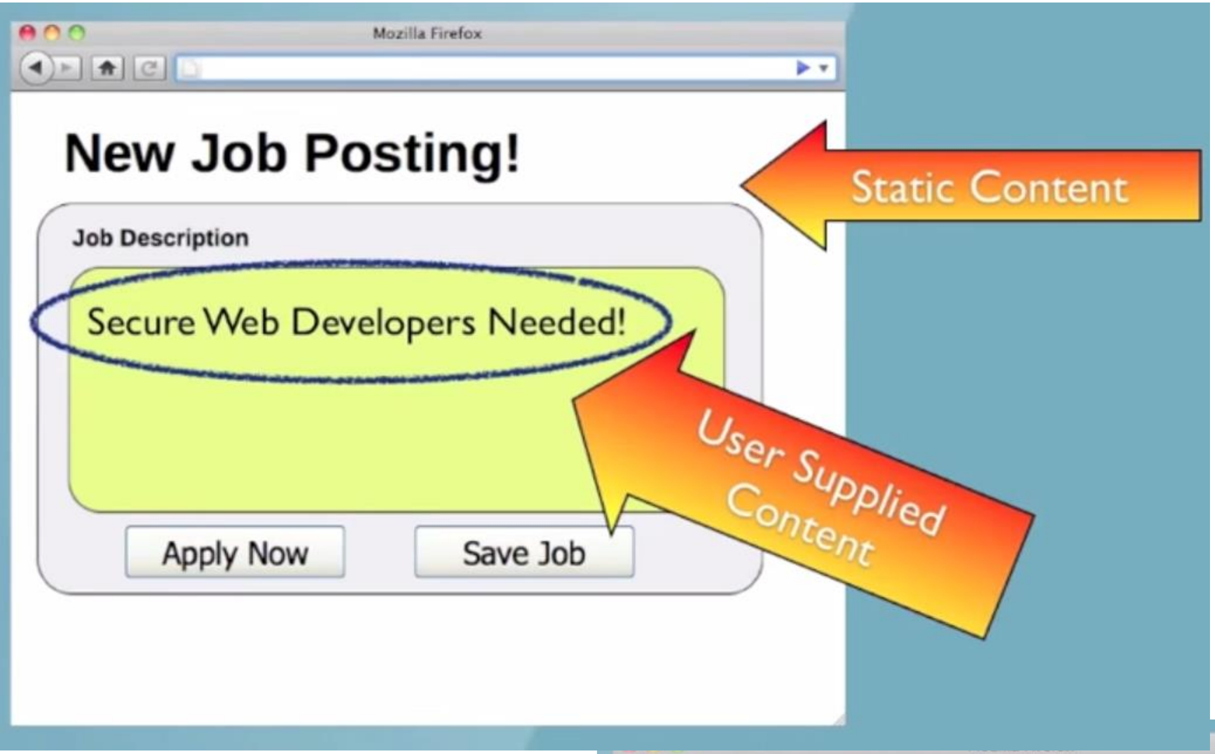
- **Hackers can take advantage of poor server configuration (e.g., default accounts, unpatched flaws, unprotected files and directories) to gain unauthorized access to application functionality or data**
- **Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code.**

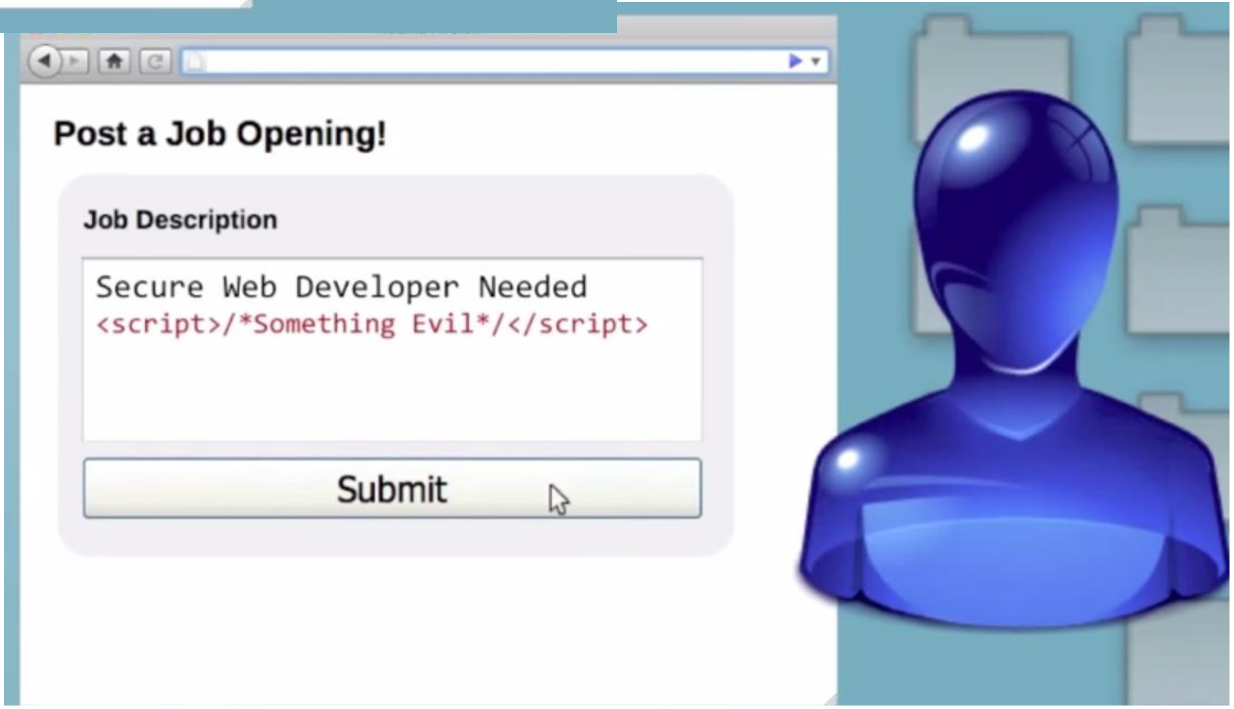
Solution

- Verify your system's configuration by scanning to find misconfiguration or missing patches
- Secure configuration by "**hardening**" the servers:
  - Disable unnecessary packages, accounts, processes & services.
  - Disable default accounts
  - Patch OS, Web server, and Web applications
  - Run Web server user a regular (non-privileged) user

7. Cross-site scripting (XSS)

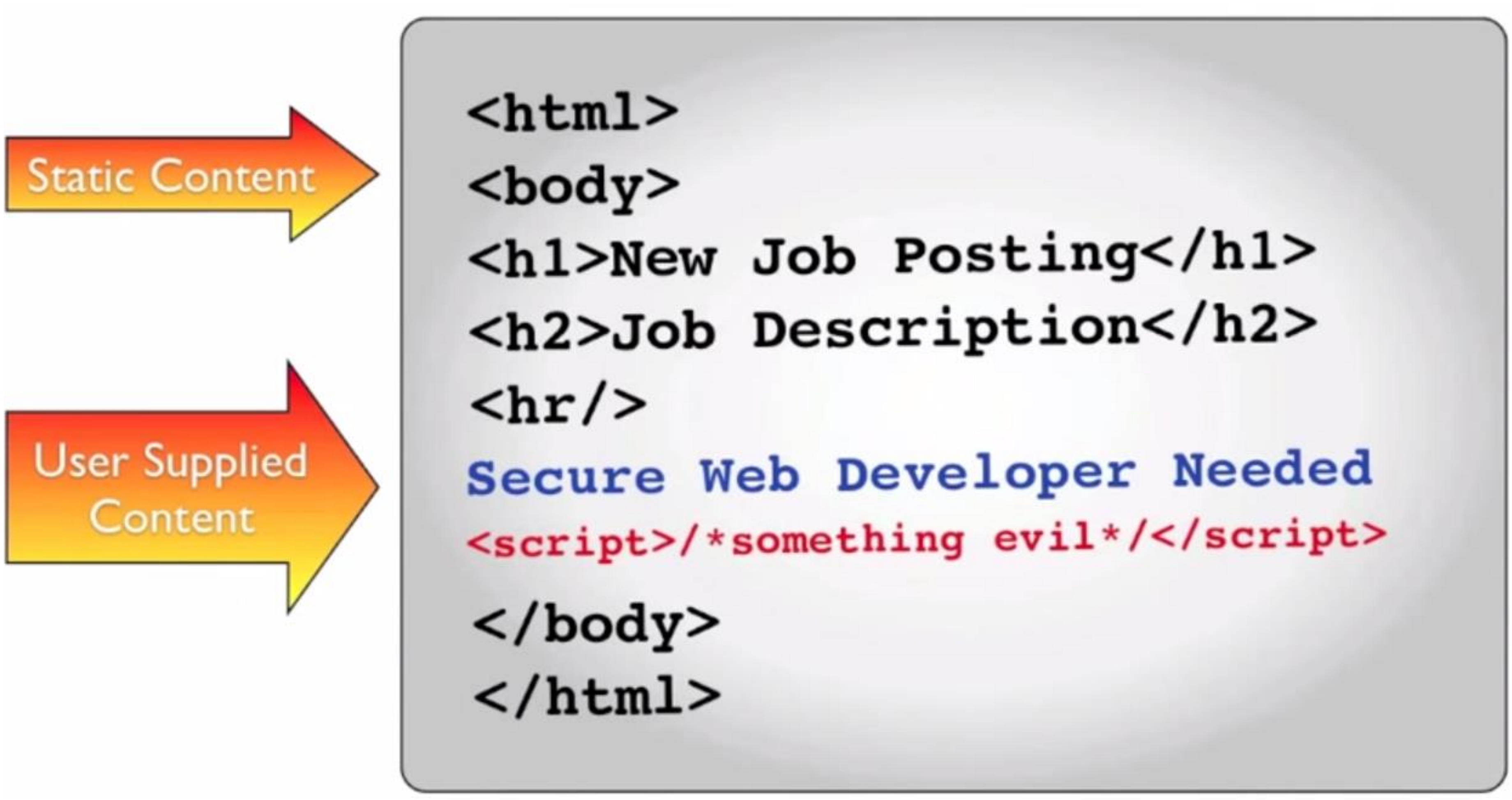
- Cross-site scripting (XSS) vulnerability
  - an application takes user input and sends it to a Web browser without validation or encoding
  - attacker can execute JavaScript code in the victim's browser
  - to hijack user sessions, deface web sites etc.
- Solution: validate user input, encode HTML output



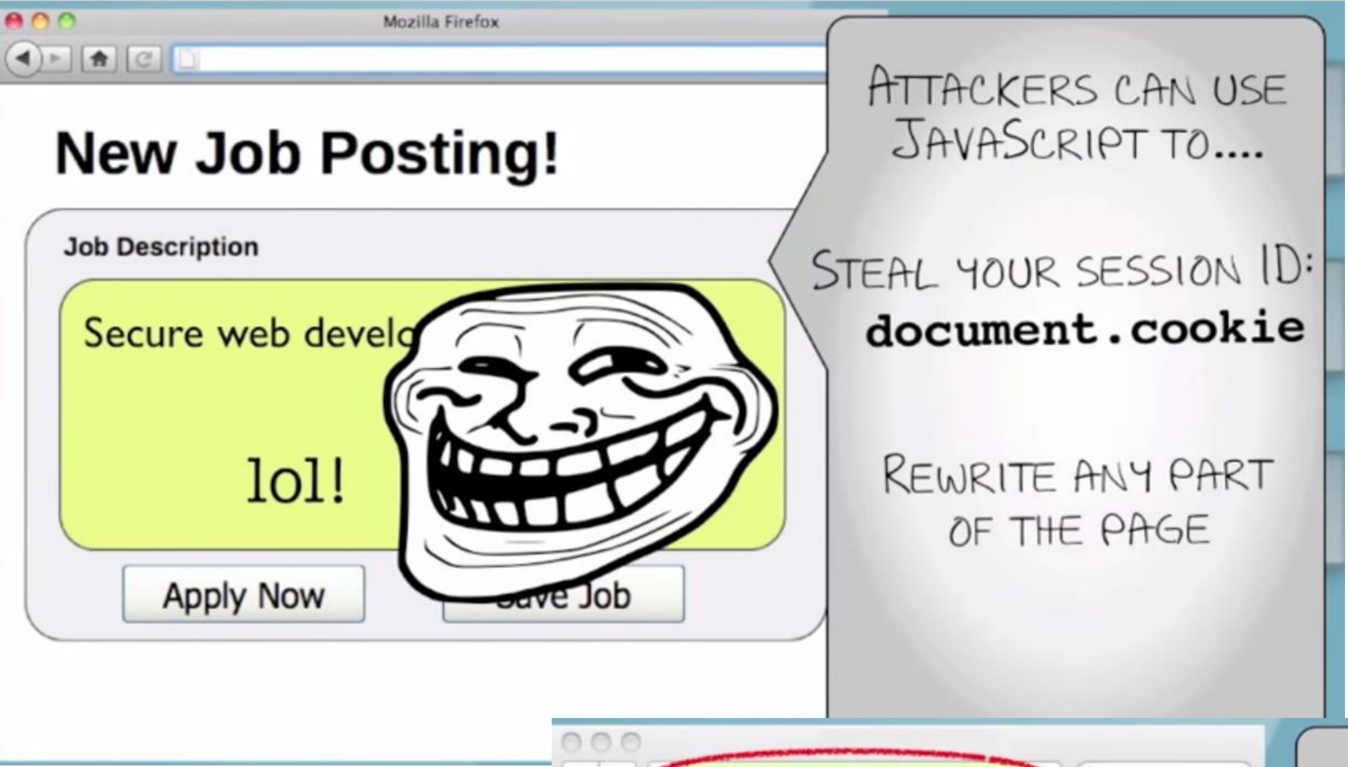


```
Secure Web Developer Needed
<script>/*Something Evil*/</script>
```

Visitors will get the evil script



```
<html>
<body>
<h1>New Job Posting</h1>
<h2>Job Description</h2>
<hr/>
Secure Web Developer Needed
<script>/*something evil*/</script>
</body>
</html>
```



Mozilla Firefox

New Job Posting!

Job Description

Secure web develo

lol!

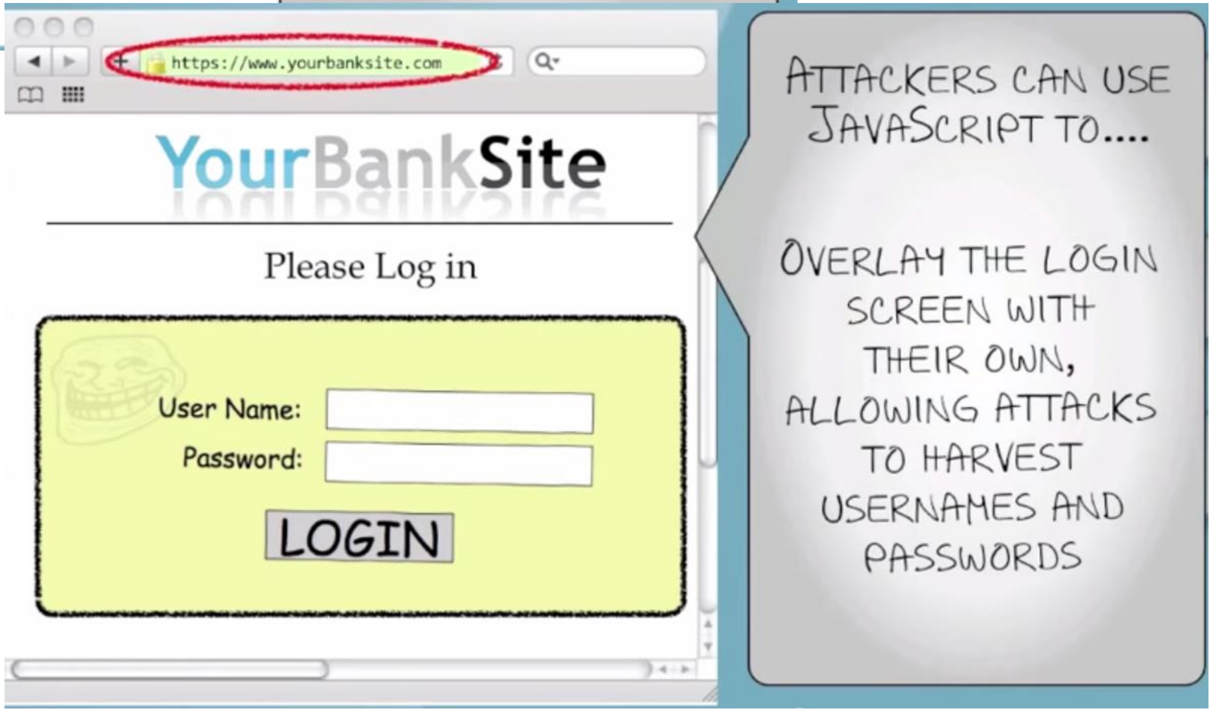
Apply Now

Save Job

ATTACKERS CAN USE JAVASCRIPT TO....

STEAL YOUR SESSION ID:
document.cookie

REWRITE ANY PART OF THE PAGE



https://www.yourbanksite.com

YourBankSite

Please Log in

User Name:

Password:

LOGIN

ATTACKERS CAN USE JAVASCRIPT TO....

OVERLAY THE LOGIN SCREEN WITH THEIR OWN, ALLOWING ATTACKS TO HARVEST USERNAMES AND PASSWORDS

8. Insecure Deserialization

- Applications and APIs will be vulnerable if they deserialize hostile or tampered objects supplied by an attacker.
- This can result in two primary types of attacks:
  - Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution
  - Typical data tampering attacks, such as access-control-related attacks, where existing data structures are used but the content is changed.
- **Solution:**
  - Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
  - Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes.

9. Using Components with Known Vulnerabilities

Vulnerable Components Are Common

- Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools

Widespread

- Virtually every application has these issues because most development teams don't focus on ensuring their components/libraries are up to date

Typical Impact

- Full range of weaknesses is possible, including injection, broken access control, XSS ...
- The impact could range from minimal to complete host takeover and data compromise

10. Insufficient Logging & Monitoring

- Insufficient logging, detection, monitoring and active response to attacks
- Auditable events, such as logins, failed logins, and high-value transactions are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally and not consolidated and analyzed
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by do not trigger alerts.
- The application is unable to detect, escalate, or alert for active attacks in real time or near real time.

Solution

- Establish continuous application security testing and monitoring

Summary

- Understand threats and typical attacks
- Validate, validate, validate (!) user input before using it
- Do not trust the client input
- Avoid the misconception of **security by obscurity**
- Read and follow recommendations for your development platform
- Use web vulnerability scanning tools https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools
- Harden the Web server and programming platform configuration