32548-48730 Cybersecurity

Lecture-2

Web Security: Web- based attacks, mitigation strategy

***Reading**:*
Wenliang Du: Chapter 10, 11, 12,18
Chwan-Hwa (john) Wu and David Irwin, Chapter 26, Stallings: Chapter 8

Dr. Priyadarsi Nanda
Priyadarsi.Nanda@uts.edu.au

Our Focus

- **Web Based Attacks**
  - –DNS attack (Cache poisoning)
  - –SQL injection attack
  - –CSRF and XSS attack
- Various security mitigation strategy

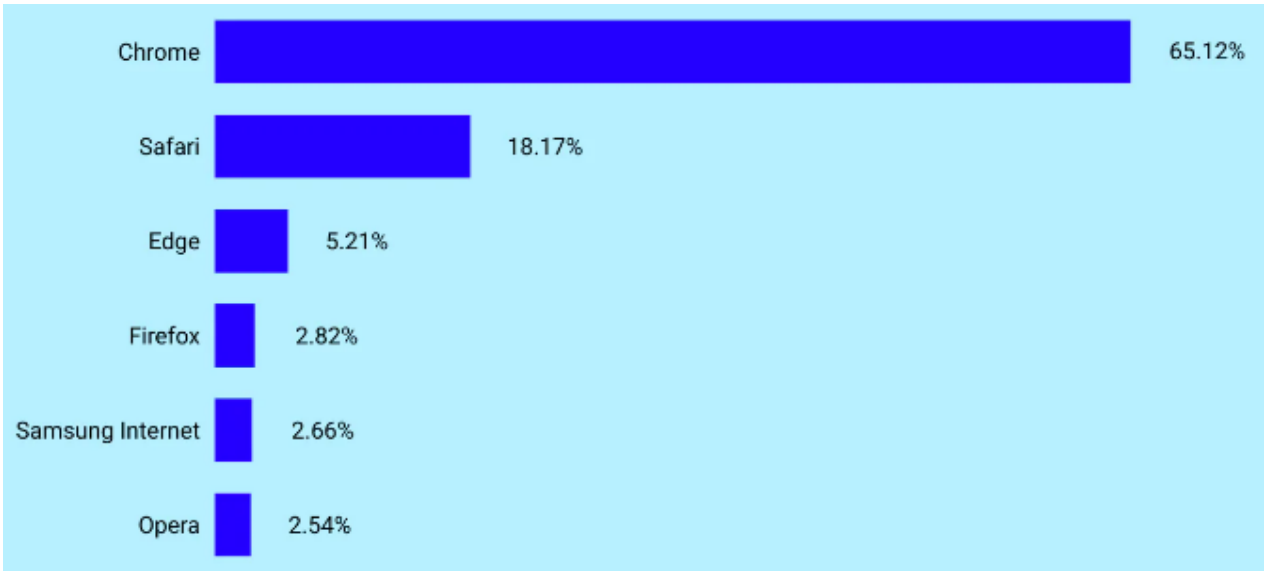
- **Test your knowledge:**
  - –Explain how DNS cache poisoning may lead to different cyberattacks?
  - –What is Cross Site Scripting (XSS) attack? Explain how to defend against such attack?
  - –What is Cross-Site Request Forgery (CSRF)?
  - –Some common password crack techniques?
  - –What is SQL injection attack? Explain defence mechanisms to prevent such attack?

Browser and Web based Cyber threats

- Browser is the most exposed software to interact with Internet based resource
  - Online shopping, Online banking, accessing database etc.

- Cyber criminals establish a foothold on networked device through the browser using a number of schemes:
  - Exploiting vulnerabilities associated with programming languages: Java
  - script, Flash, ActiveX etc.
    - install ransomware, exfiltration of data, and stealing intellectual property
  - Difficult to detect attack as conventional approach (involving file scanning) does not work (Why?)
    - Attacker may Embed obfuscated Adobe Flash file within the JavaScript.

Most popular Browser 2024

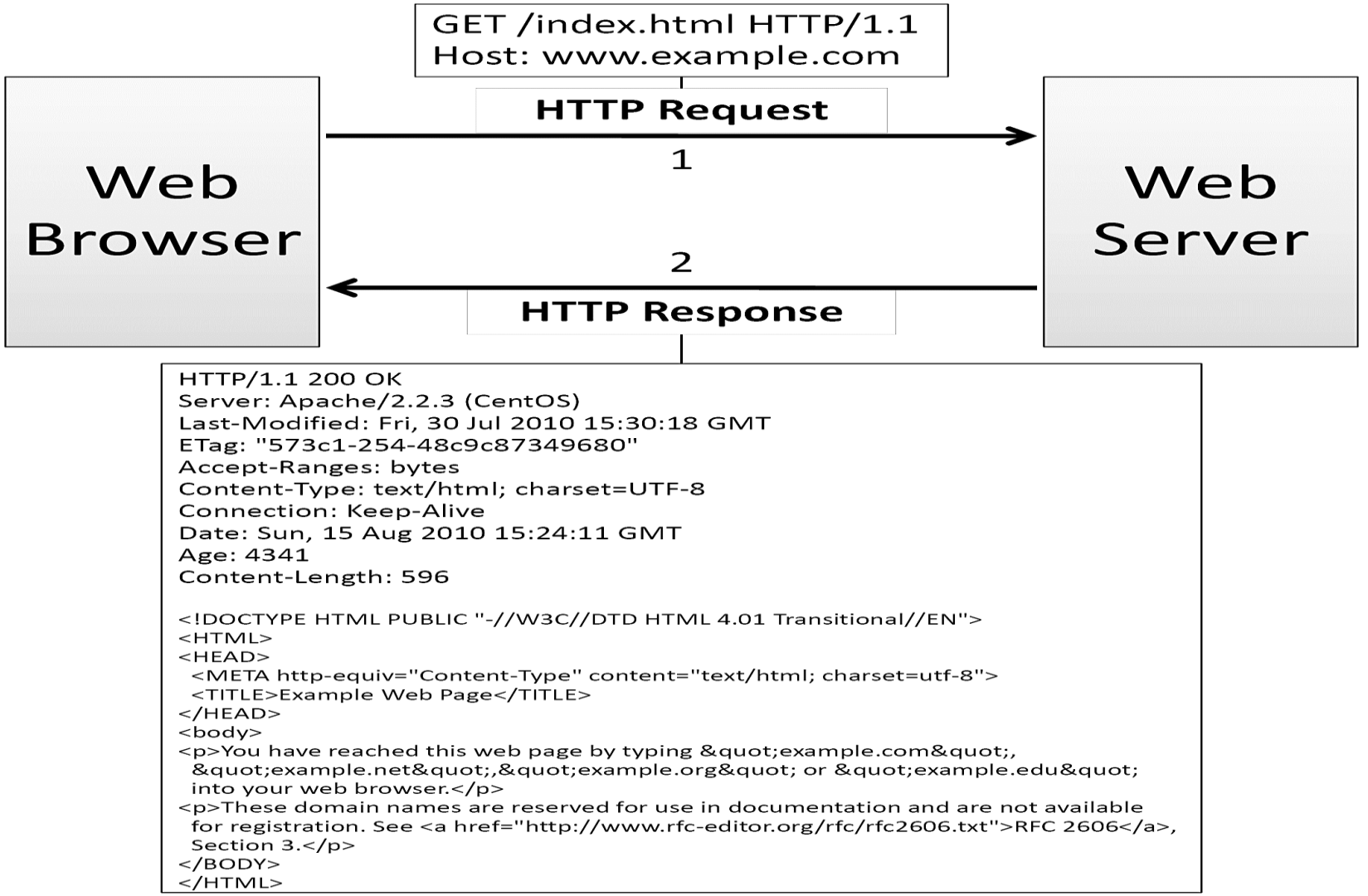


- According to Oberlo, Chrome accounts for 65.12% percent of the total market share for all browsers worldwide:
- What users look for: Better feeling, Privacy and Security
- Look for browser related statistics and user adaptation trend at:

https://www.oberlo.com/statistics/browser-market-share

  - Statistics can be misleading: The pure and simple truth is rarely pure and never simple- *Oscar wilde*

How Web Browser Work

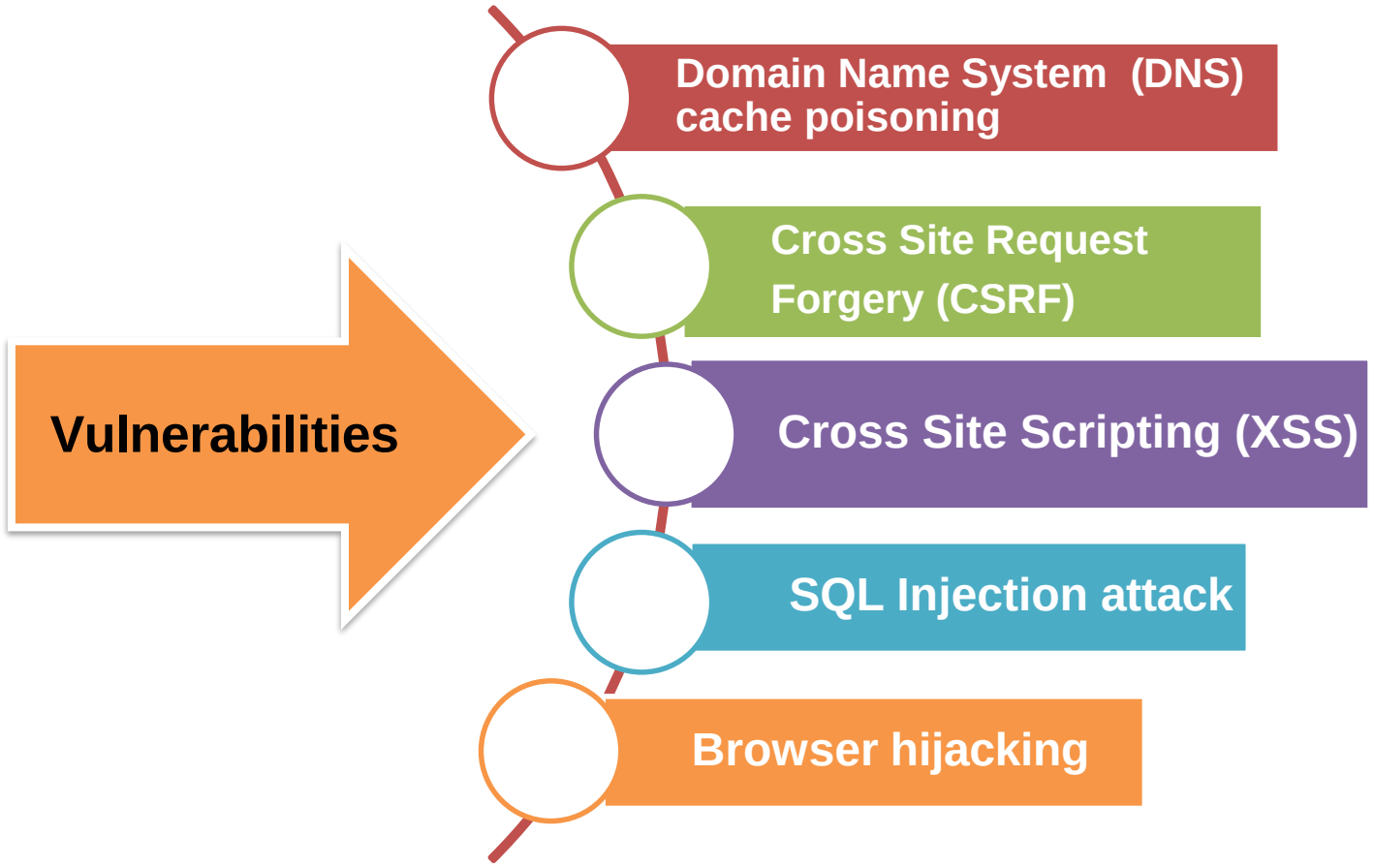


Some Http Response Codes

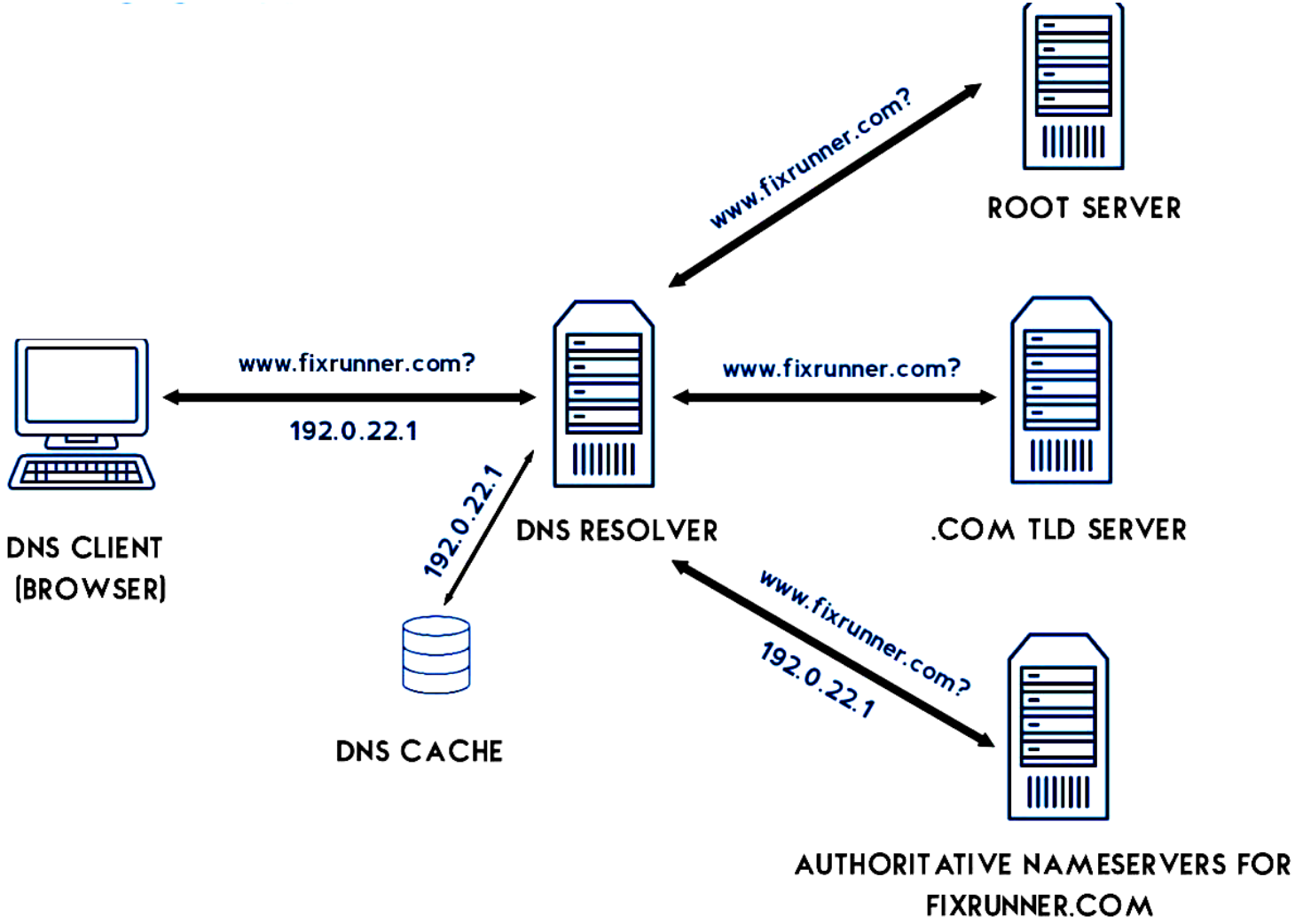
Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

Vulnerabilities in Web-based Systems

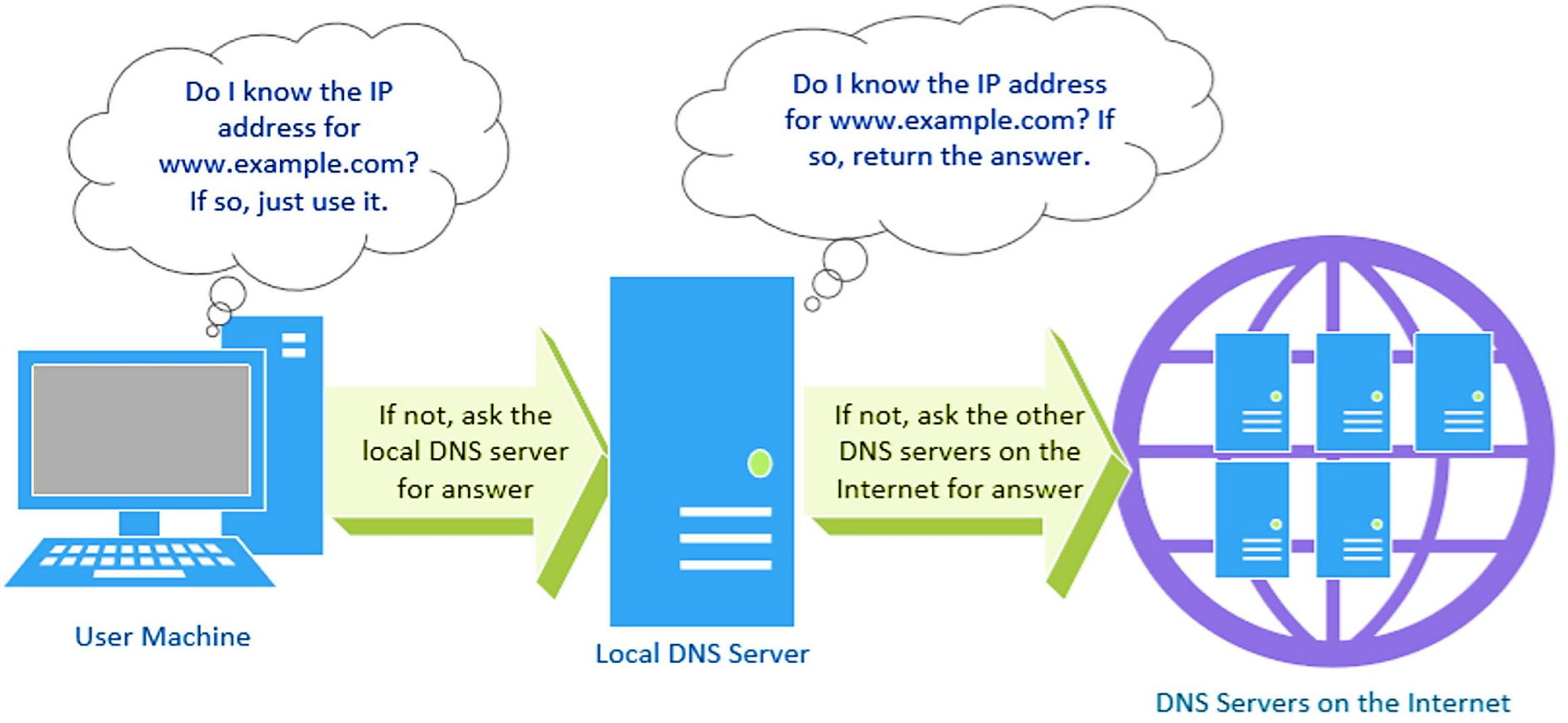
- Mainly due to inadequate validation of user input resulting different web-based attacks;



How DNS Works

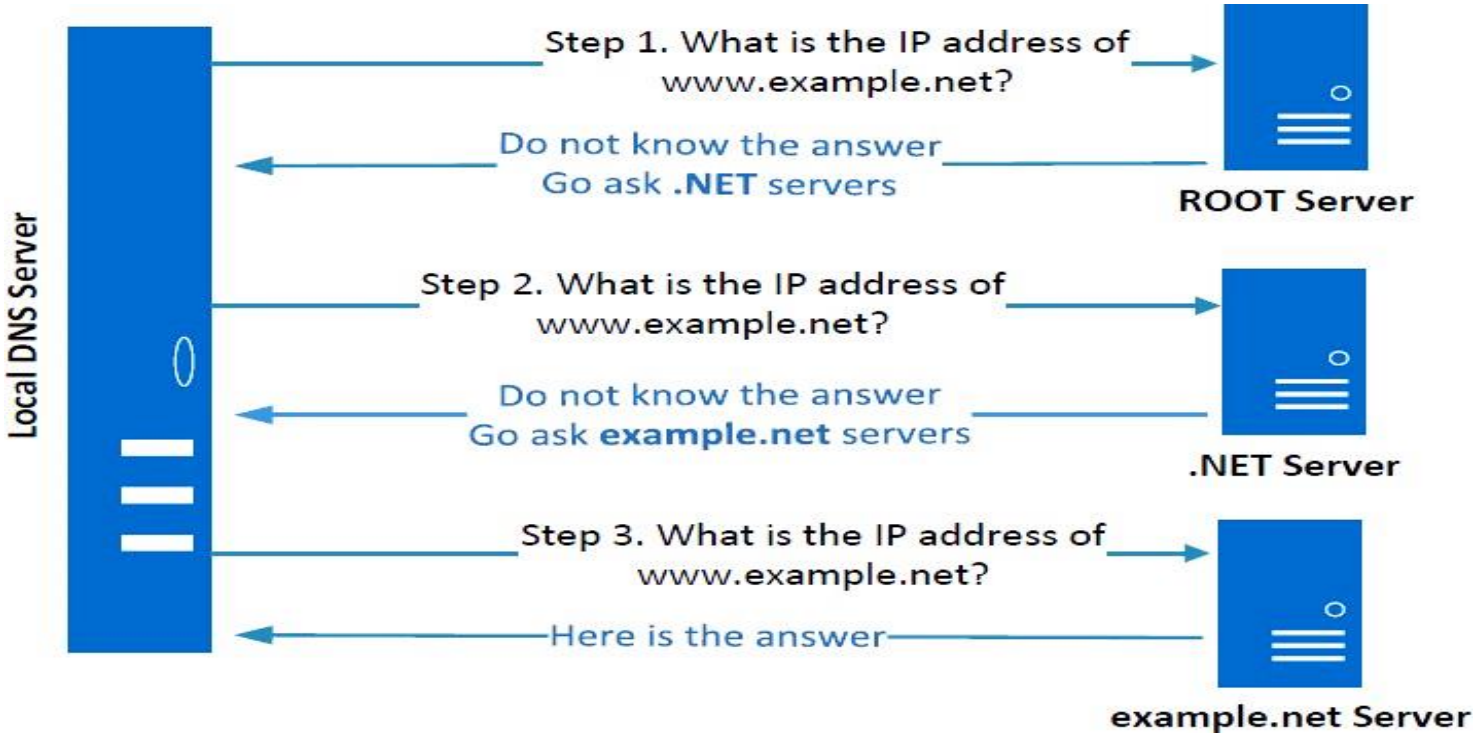


DNS Query Process



Local DNS Server and Iterative Query Process

- The iterative process starts from the ROOT Server. If it doesn't know the IP address, it sends back the IP address of the nameservers of the next level server (.NET server) and then the last level server (example.net) which provides the answer.



DNS Response

- **There are 4 types of sections in a DNS response :**

  - Question section : Describes a question to a nameserver
  - Answer section : Records that answer the question
  - Authority section : Records that point toward authoritative nameservers
  - Additional section : Records that are related to the query.

- As root server doesn't know the answer there is no answer section, but, it tells us about the authoritative nameservers (NS Record) along with their IP addresses in the Additional section (A record).

- When the local DNS server gets information from other DNS servers, it caches the information.

- Each piece of information in the cache has a time-to-live value, so it will be eventually time out and removed from the cache.

Steps 2-3: Ask .net & example.net servers


```
seed@ubuntu:~$ dig @m.gtld-servers.net www.example.net

;; QUESTION SECTION:
;www.example.net.               IN      A

;; AUTHORITY SECTION:
example.net.            172800  IN      NS      a.iana-servers.net.
example.net.            172800  IN      NS      b.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.     172800  IN      A       199.43.132.53
b.iana-servers.net.     172800  IN      A       199.43.133.53
```

Ask a .net nameservers.

Go ask them!

```
seed@ubuntu:$ dig @a.iana-servers.net www.example.net

;; QUESTION SECTION:
;www.example.net.               IN      A

;; ANSWER SECTION:
www.example.net.        86400   IN      A       93.184.216.34
```

Ask an example.net nameservers.

Finally got the answer

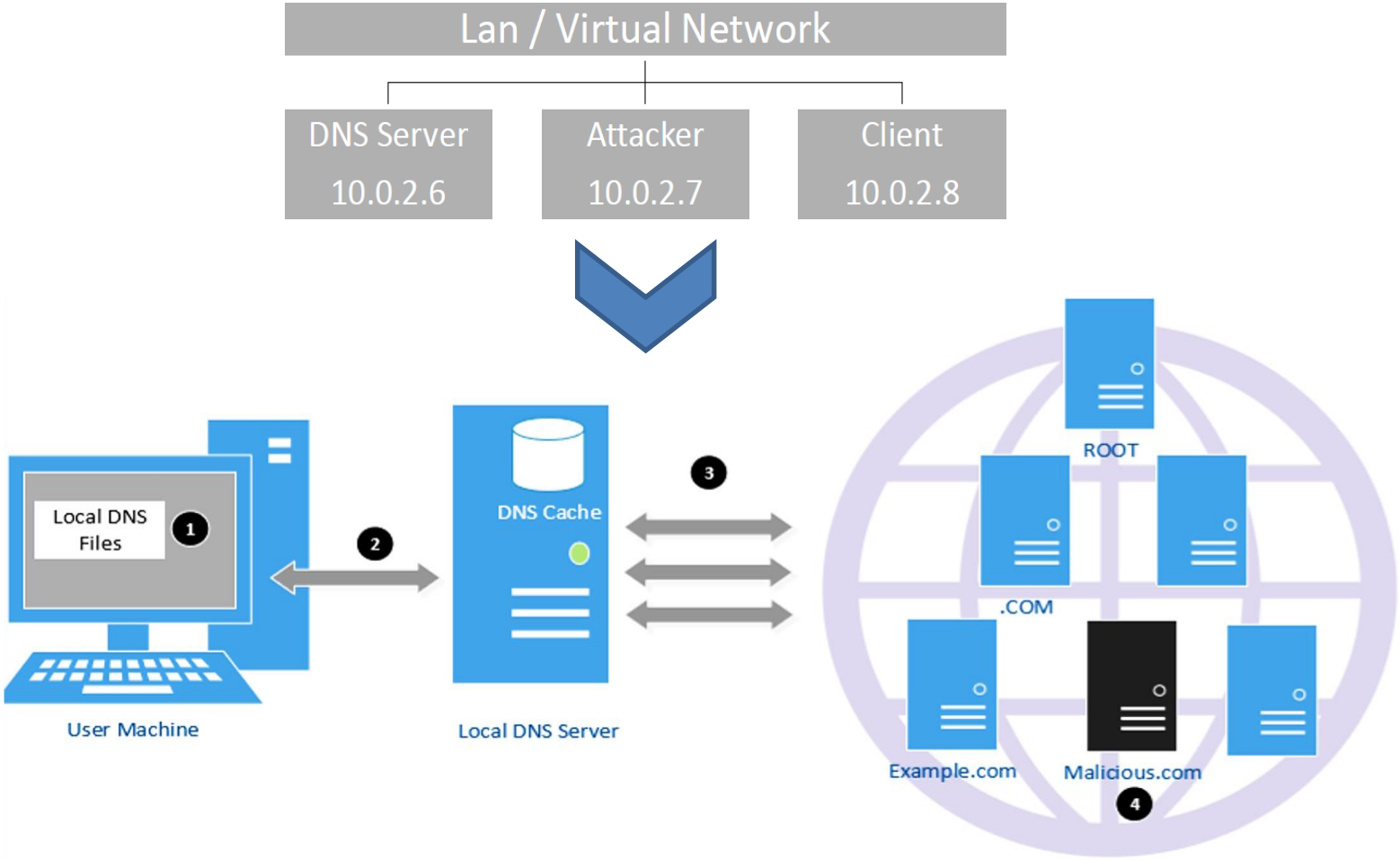
DNS Attacks Types

- DNS Spoofing
- DNS Flood Attack
- Distributed reflection DDoS
- Domain lock-up
- DNS tunnelling

DNS Attacks

Realizing DNS attacks

- Following setup is being used for our lab activities (Lab-2)



DNS Attacks on Compromised Machines

- If attackers have gained the root privileges on a machine, they can

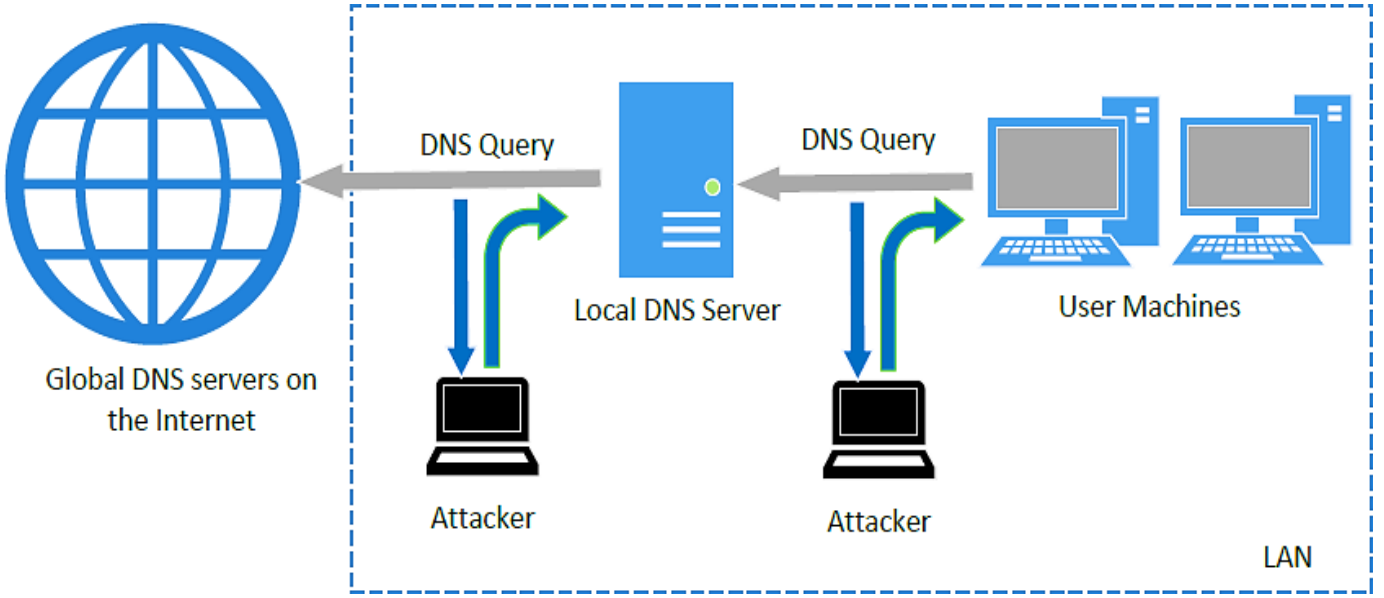
  - Modify /etc/resolv.conf: use malicious DNS server as the machine's local DNS server and can control the entire DNS process.

  - Modify /etc/hosts: add new records to the file, providing the IP addresses for some selected domains. For example, attackers can modify IP address of www.bank32.com which can lead to attacker's machine.

  - In your lab, you will use Netwag / Netwox tool to understand and launch attacks.

Local DNS cache poisoning attack

Spoofing DNS Replies (from LAN)



Goal: Forging DNS replies after seeing a query from Local DNS Server

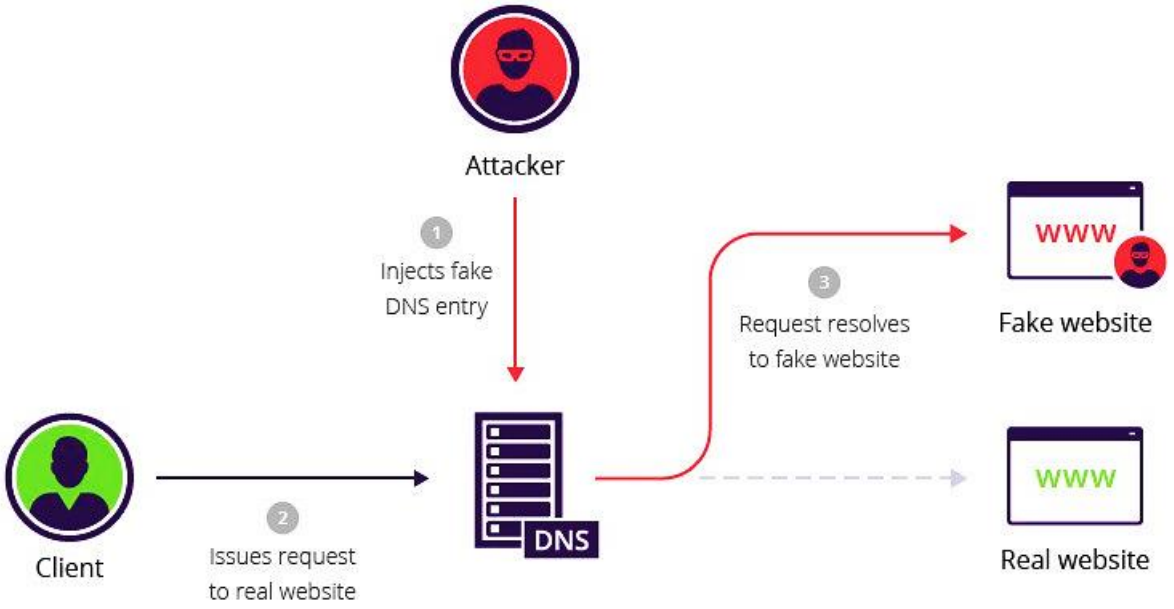
Local/Remote DNS Cache Poisoning Attack

- Consider your query as: www.example.net

Local

```
$ dig www.example.net
; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.example.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61991
;; flags: qr aa ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 0

;; QUESTION SECTION:
;www.example.net.    IN A

;; ANSWER SECTION:
www.example.net.    259200  IN A       1.2.3.4                ①

;; AUTHORITY SECTION:
example.net.        259200  IN NS      ns.attacker32.com.     ②
```

Remote

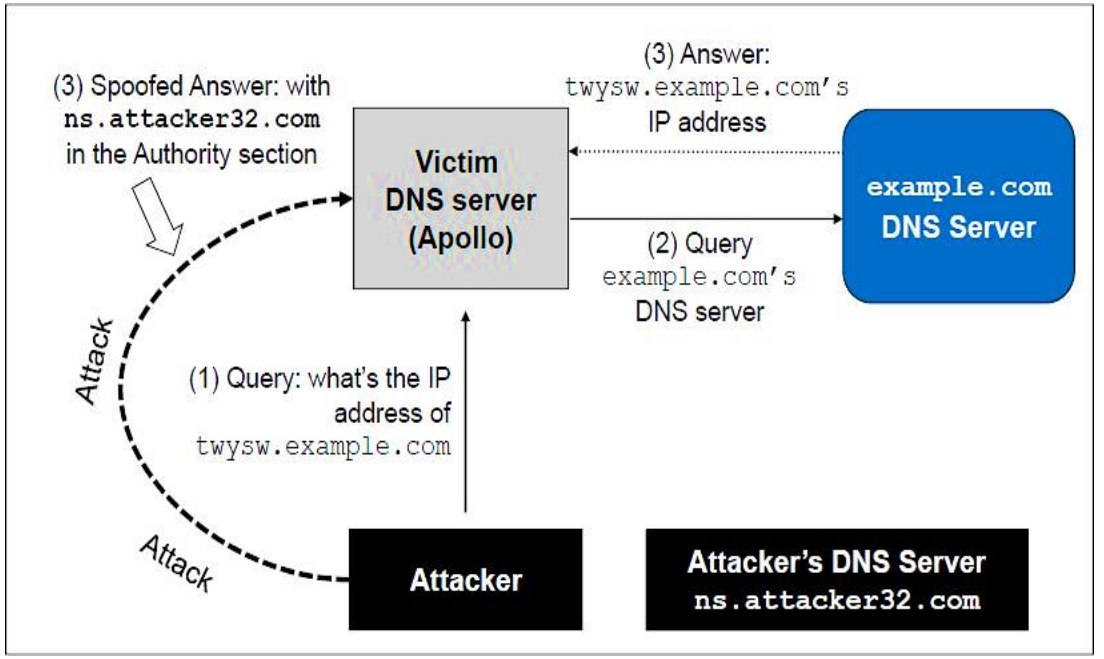


The Most Popular Types of DNS Attacks: https://securitytrails.com/blog/most-popular-types-dns-attacks

The Kaminsky Attack

How can we keep forging replies without worrying about the cache effect?

- **Kaminsky's Idea:**



- Ask a different question every time, so caching the answer does not matter, and the local DNS server will send out a new query each time.
- Provide a forged answer in the Authority section

The Kaminsky Attack: A Sample Response

This random name will change for each attack attempt

This answer does not matter

```
;; QUESTION SECTION:
;twysw.example.com.              IN    A

;; ANSWER SECTION:
twysw.example.com.      259200  IN    A    1.2.3.4

;; AUTHORITY SECTION:
example.com.            259200  IN    NS   ns.attacker32.com
```

This is what we want the local DNS server to cache

Tell the DNS server to use this one as the nameserver for the example.com domain

Attacks from Malicious DNS Server

- When user visits a website, such as attacker32.com:
- A DNS query will eventually come to the authoritative nameserver of the attacker32.com domain.
- In addition to providing an IP address in the answer section of the response, DNS server can also provide information in the authority and additional sections
- Attackers can then use these sections to add fraudulent information

Fake Data in the Additional Section

Additional Section

Additional information is provided

```
;; QUESTION SECTION:
;www.example.net.                IN      A

;; ANSWER SECTION:
www.example.net.        259200  IN      A       192.168.0.101

;; ADDITIONAL SECTION:
www.gmail.com.          259200  IN      A       192.168.0.201
www.facebook.com.       259200  IN      A       192.168.0.202
```

They will be discarded: out of zone. They will cause security problems if not discarded.

Authority Section

This one is allowed

This one is out of zone, and should be discarded

```
;; QUESTION SECTION:
;www.example.net.                IN      A

;; ANSWER SECTION:
www.example.net.        259200  IN      A       192.168.0.101

;; AUTHORITY SECTION:
example.net.            259200  IN      NS      ns.example.net.
facebook.com.           259200  IN      NS      ns.example.net.
```

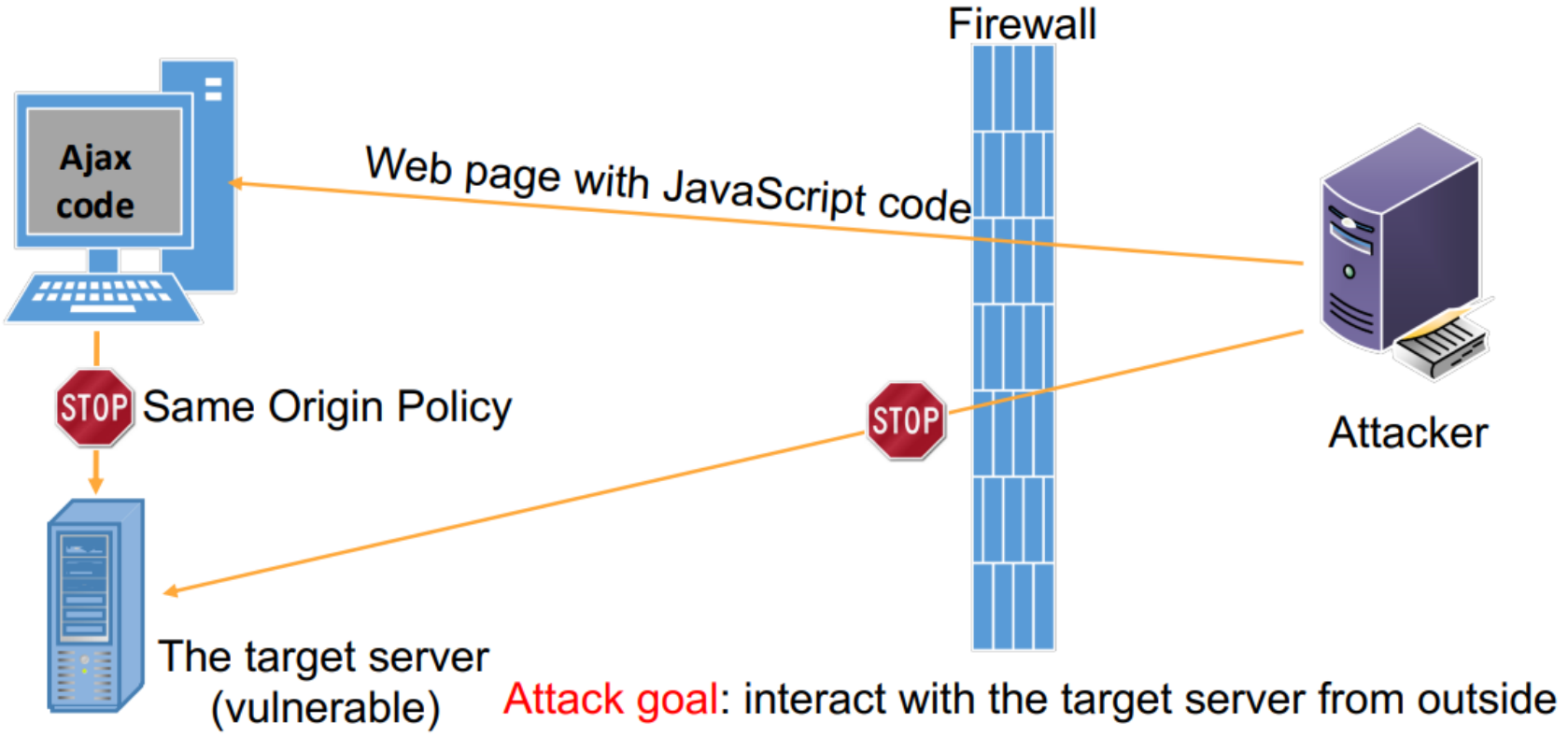
Reply Forgery Attacks from Malicious DNS Servers


```
;; QUESTION SECTION:
;www.example.net.                IN      A

;; ANSWER SECTION:
www.example.net.         259200  IN      A       192.168.0.101

;; AUTHORITY SECTION:
example.net.             259200  IN      NS      www.facebook.com.

;; ADDITIONAL SECTION:
www.facebook.com.        259200  IN      A       192.168.0.201
```

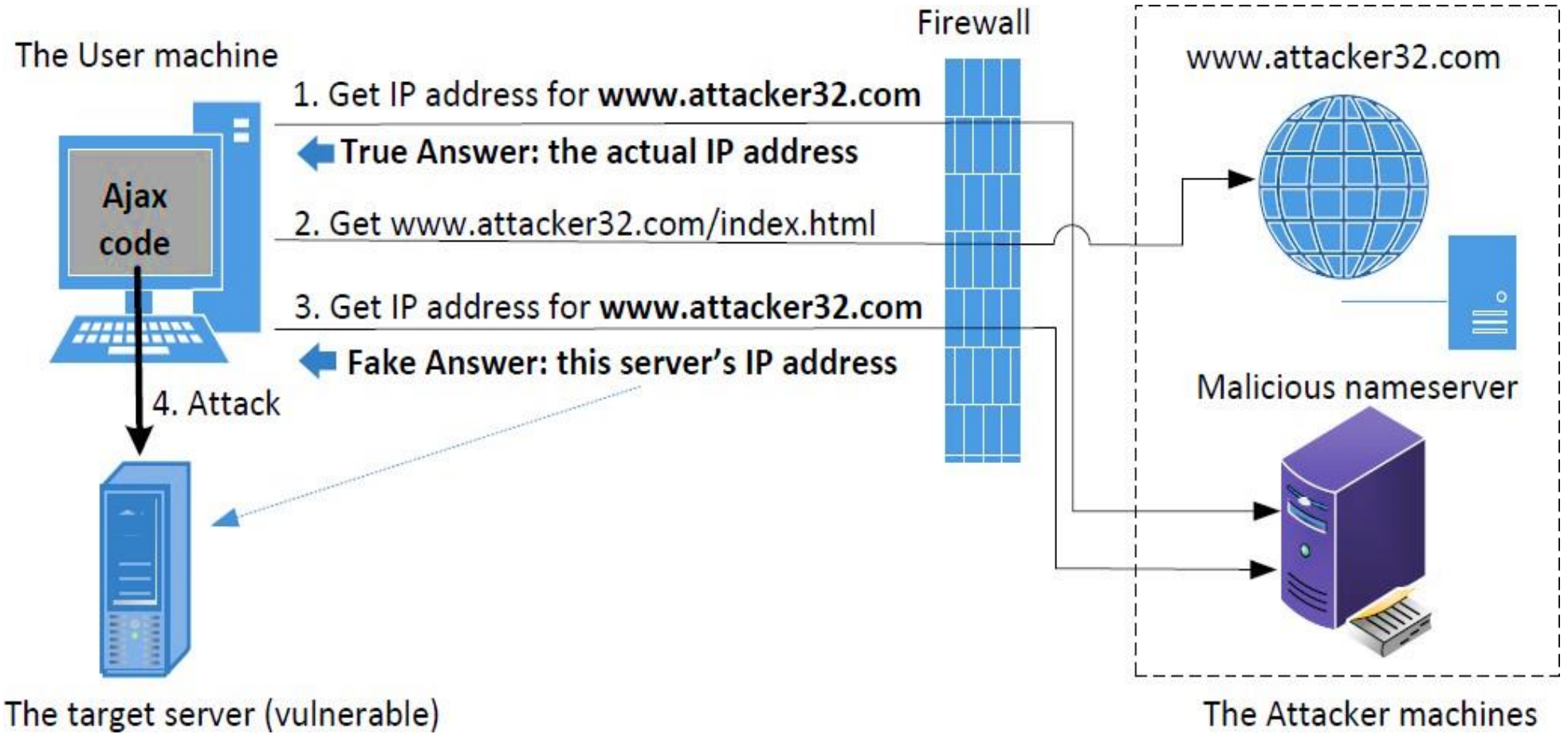
This one is allowed

This one is not allowed (out of zone). The local DNS server will get the IP address of this hostname by itself.

DNS Rebinding Attack



DNS Rebinding Attack



Denial of Service Attacks on Root Servers

Attacks on the Root and TLD Servers :

Root nameservers: If the attackers can bring down the servers of the root zone, they can bring down the entire Internet. However, attack on root servers is difficult:

- The root nameservers are highly distributed. There are 13 (A,B….M) root nameservers (server farm) consisting of a large number of redundant computers to provide reliable services.

- As the nameservers for the TLDs are usually cached in the local DNS servers, the root servers need not be queried till the cache expires (48 hours). Attacks on the root servers must last long to see a significant effect.

Denial of Service Attacks on TLD Servers

- Nameservers for the TLDs are easier to attack. TLDs such as gov, com, net etc have quite resilient infrastructure against DOS attacks. But certain obscure TLDs like country-specific TLDs do not have sufficient infrastructure. Due to this, the attackers can bring down the Internet of a targeted country.

Protection Using TLS/SSL

- Transport Layer Security (TLS/SSL) protocol provides a solution against the cache poisoning attacks. (This part will be covered later)

- After getting the IP address for a domain name (www.example.net) using DNS protocol, a computer will ask the owner (server) of the IP address to prove that it is indeed www.example.net.

- The server has to present a public-key certificate signed by a trusted entity and demonstrates that it knows the corresponding private key associated with www.example.net (i.e., it is the owner of the certificate).

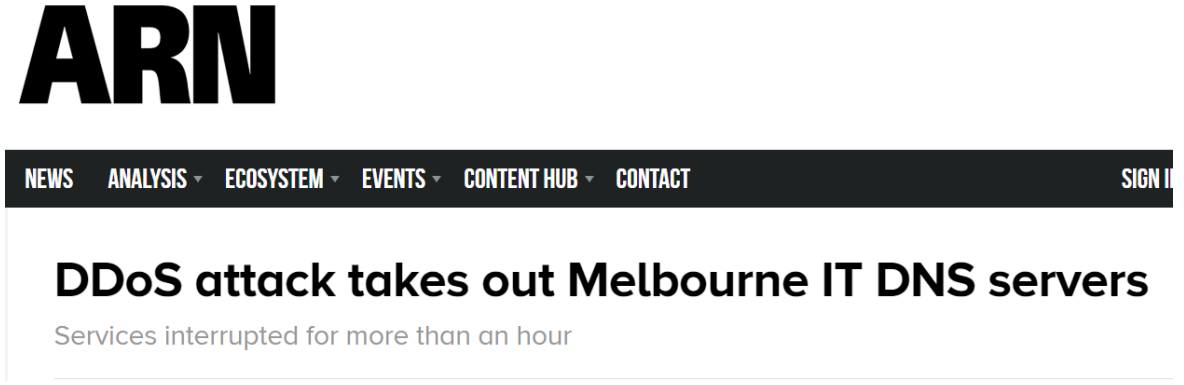
- HTTPS is built on top of TLS/SSL. It defeats DNS cache poisoning attacks. This will be discussed in Transport Layer Security.

Attacks on Nameservers of a Particular Domain

- Australian Telcom Provider Telstra Mistakes DNS Issue for DDoS Attack ☹

https://www.zdnet.com/article/telstra-dns-falls-over-after-denial-of-service-attack/

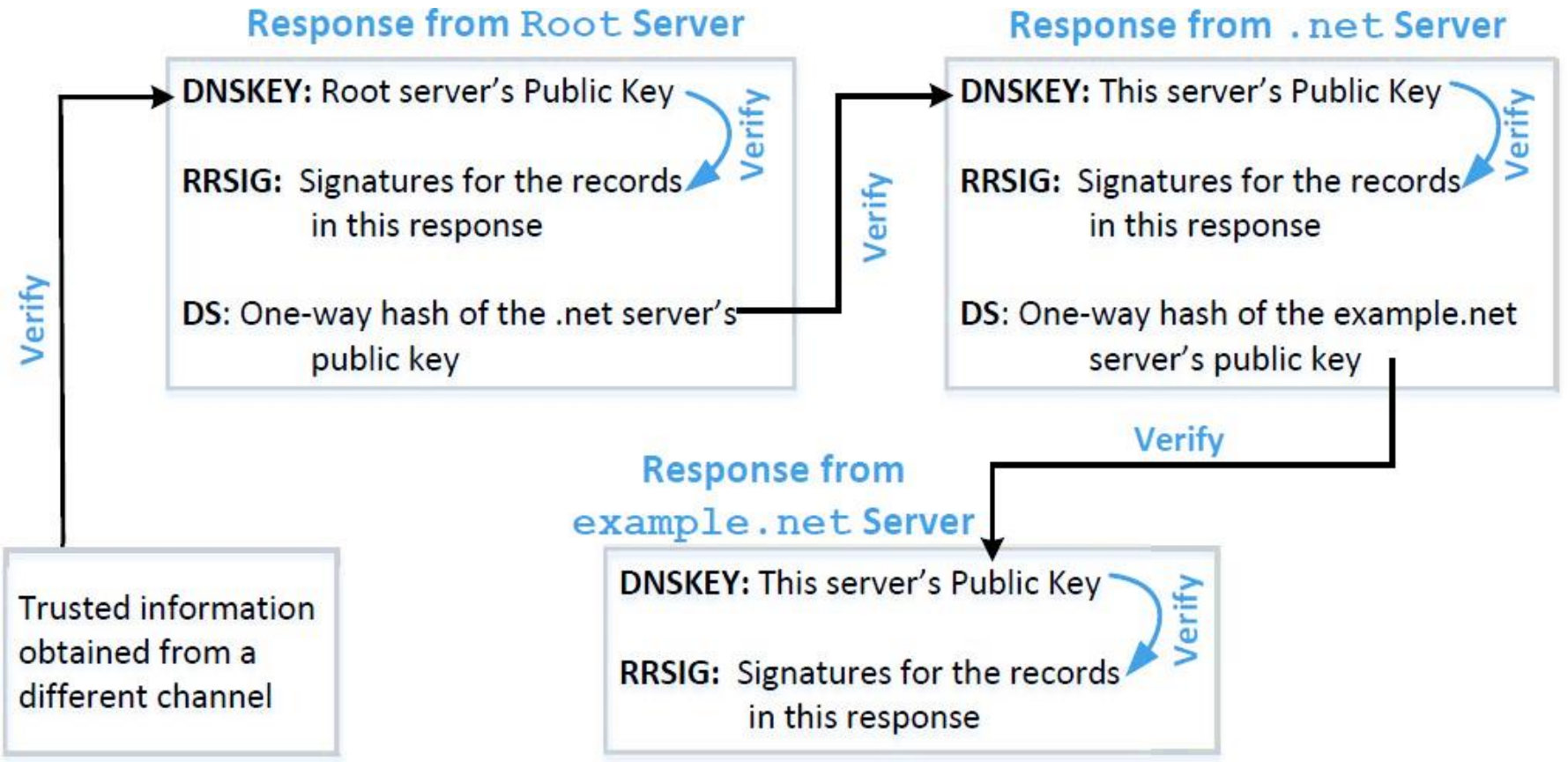
- DDoS attack takes out Melbourne IT DNS servers ☹



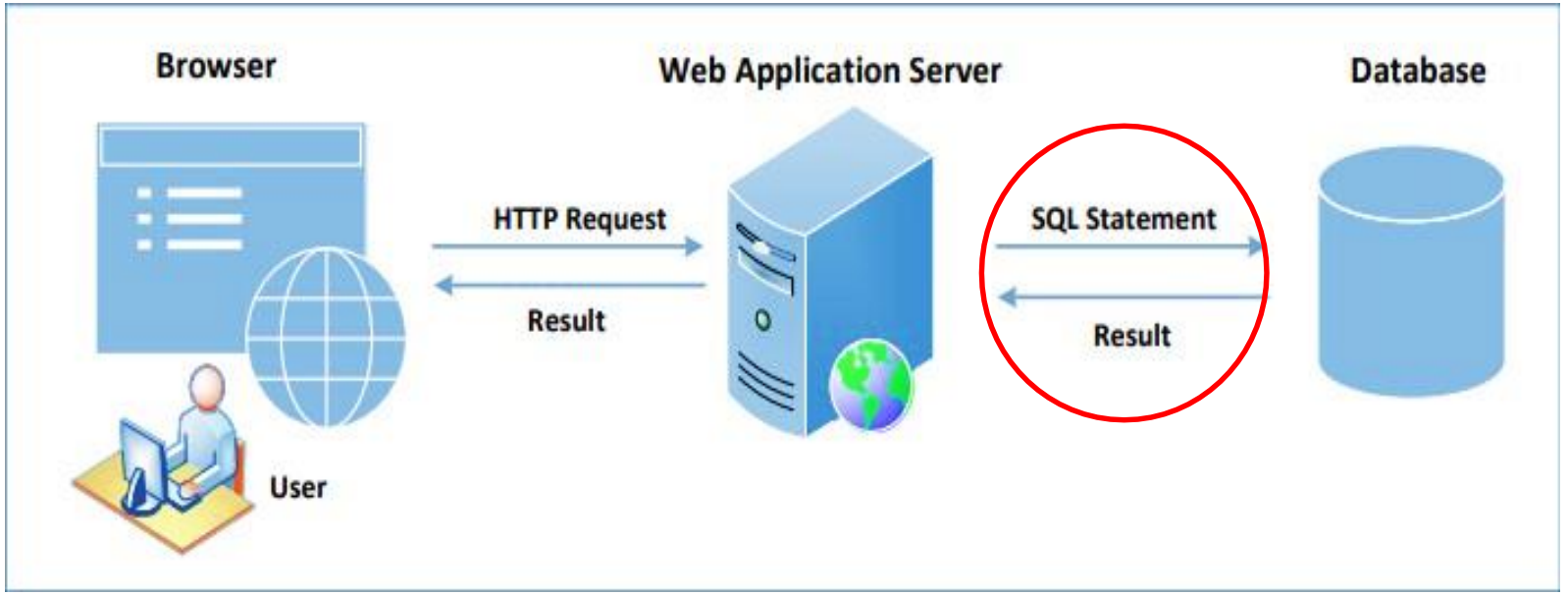
https://www.arnnet.com.au/article/617665/ddos-attack-takes-melbourne-it-dns-servers/

Protection Against DNS Cache Poisoning Attacks

Protection Using DNSSEC



Interacting with Database in Web Application



- A typical web application consists of three major components:
- SQL Injection attacks can cause damage (severity=High) to the database.
- The users do not directly interact with the database but through a web server.
- If this channel is not secured, malicious users can attack the database

Getting Data from User

- A form used by users to type their data
- Once the submit button is clicked, an HTTP request will be sent out with the data attached

EID EID5000
Password paswd123
Submit

- The HTML source of the above form is:

```
<form action="getdata.php" method="get">
  EID:      <input type="text" name="EID"><br>
  Password: <input type="text" name="Password"><br>
            <input type="submit" value="Submit">
</form>
```

- Web browser generates a Request:

```
http://www.example.com/getdata.php?EID=EID5000&Password=paswd123
```

- Once this request reached the target PHP script the parameters inside the HTTP request will be saved to an array $_GET or $_POST. The following example shows a PHP script getting data from a GET request

```
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];
   echo "EID: $eid --- Password: $pwd\n";
?>
```

How Web Applications Interact with Database

Connecting to MySQL Database

```
function getDB() {
   $dbhost="localhost";
   $dbuser="root";
   $dbpass="seedubuntu";
   $dbname="dbtest";

   // Create a DB connection
   $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
   if ($conn->connect_error) {
         die("Connection failed: " . $conn->connect_error . "\n");
   }
   return $conn;
}
```

- PHP program connects to the database server before conducting query on database using:
- The code shown below uses new MySQL(…) along with its 4 arguments to create the database connection.

```
/* getdata.php */
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];

   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $sql = "SELECT Name, Salary, SSN
           FROM employee
           WHERE eid= '$eid' and password='$pwd'";

   $result = $conn->query($sql);
   if ($result) {
      // Print out the result
      while ($row = $result->fetch_assoc()) {
        printf ("Name: %s -- Salary: %s -- SSN: %s\n",
                $row["Name"], $row["Salary"], $row['SSN']);
      }
      $result->free();
   }
   $conn->close();
?>
```

Constructing SQL statement

- Construct the query string and then send it to the database for execution.
- The channel between the user and the database creates a new attack surface for the database.

Launching SQL Injection Attacks

- Everything provided by user will become part of the SQL statement.
- Is it possible for a user to change the meaning of the SQL statement?
- The intention of the web app developer by the following is for the user to provide some data for the blank areas.

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid=' [                    ] ' and password=' [                    ] '
```

- Assume that a user inputs a random string in the password entry and types "EID5002'#" in the eid entry. The SQL statement will become the following

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'EID5002' #' and password='xyz'
```

Launching SQL Injection Attacks

- Everything from the # sign to the end of line is considered as comment
- Hence, the SQL statement will be equivalent to the following:

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'EID5002'
```

- Is this a security breach?
- Let's see if a user can get all the records from the database assuming that we don't know all the EID's in the database.
- We need to create a predicate for WHERE clause so that it is true for all records.

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'a' OR 1=1
```

Launching SQL Injection Attacks using cURL

- More convenient to use a command-line tool to launch attacks.
- Easier to automate attacks without a graphic user interface.
- Using cURL, we can send out a form from a command-line, instead of from a web page.

```
% curl 'www.example.com/getdata.php?EID=a' OR 1=1 #&Password='
```

- The above command will not work. In an HTTP request, special characters are in the attached data needs to be encoded or they maybe mis-interpreted.
- In the above URL we need to encode the apostrophe, whitespace and the # sign and the resulting cURL command is as shown below:

```
% curl 'www.example.com/getdata.php?EID=a%27%20
                                    OR%201=1%20%23&Password='
Name: Alice -- Salary: 80000 -- SSN: 555-55-5555<br>
Name: Bob -- Salary: 82000 -- SSN: 555-66-5555<br>
Name: Charlie -- Salary: 80000 -- SSN: 555-77-5555<br>
Name: David -- Salary: 80000 -- SSN: 555-88-5555<br>
```

Modify Database

- If the statement is UPDATE or INSERT INTO, we will have chance to change the database.
- Consider the form created for changing passwords. It asks users to fill in three pieces of information, EID, old password and new password.

EID	EID5000
Old Password	paswd123
New Password	paswd456

Submit

- When Submit button is clicked, an HTTP POST request will be sent to the server-side script changepassword.php, which uses an UPDATE statement to change the user's password.

```
/* changepasswd.php */
<?php
   $eid = $_POST['EID'];
   $oldpwd = $_POST['OldPassword'];
   $newpwd = $_POST['NewPassword'];

   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $sql = "UPDATE employee
           SET password='$newpwd'
           WHERE eid= '$eid' and password='$oldpwd'";

   $result = $conn->query($sql);
   $conn->close();
?>
```

Modify Database

- Let us assume that Alice (EID5000) is not satisfied with the salary she gets. She would like to increase her own salary using the SQL injection vulnerability. She would type her own EID and old password. The following will be typed into the “New Password” box :

New Password	paswd456’, salary=100000 #

- By typing the above string in “New Password” box, we get the UPDATE statement to set one more attribute for us, the salary attribute. The SQL statement will now look as follows.

```
UPDATE employee
SET password='paswd456', salary=100000 #'
WHERE eid= 'EID5000' and password='paswd123'";
```

- What if Alice doesn’t like Bob and would like to reduce Bob’s salary to 0, but she only knows Bob’s EID (eid5001), not his password. How can she execute the attack?

EID	EID5001’ #
Old Password	anything
New Password	paswd456’, salary=0 #

Multiple SQL Statements

- Damages that can be caused are bounded because we cannot change everything in the existing SQL statement.
- It will be more dangerous if we can cause the database to execute an arbitrary SQL statement.
- To append a new SQL statement “DROP DATABASE dbtest” to the existing SQL statement to delete the entire dbtest database, we can type the following in the EID box

EID a’; DROP DATABASE dbtest; #

- The resulting SQL statement is equivalent to the following, where we have successfully appended a new SQL statement to the existing SQL statement string.
- The above attack doesn’t work against MySQL, because in PHP’s mysqli extension, the mysqli::query() API doesn’t allow multiple queries to run in the database server.

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= ’a’; DROP DATABASE dbtest;
```

Multiple SQL Statements

- The code below tries to execute two SQL statements using the $mysqli->query() API

```
/* testmulti_sql.php */
<?php
$mysqli = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$res    = $mysqli->query("SELECT 1; DROP DATABASE dbtest");
if (!$res) {
  echo "Error executing query: (" .
       $mysqli->errno . ") " . $mysqli->error;
}
?>
```

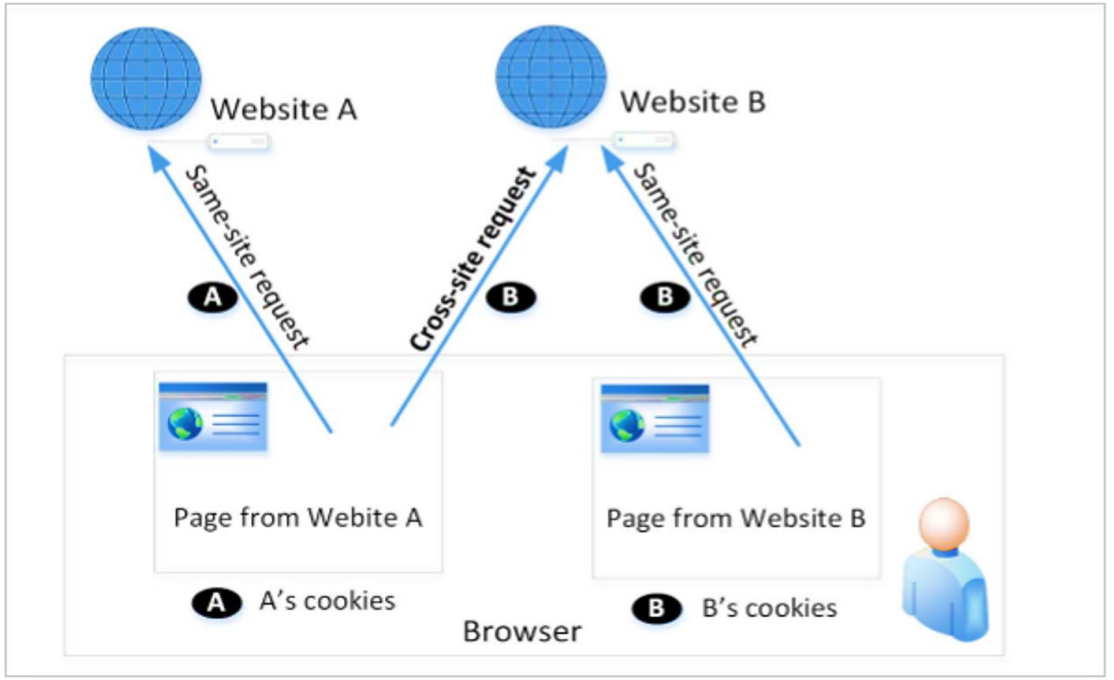
- When we run the code, we get the following error message:

```
$ php testmulti_sql.php
Error executing query: (1064) You have an error in your SQL syntax;
   check the manual that corresponds to your MySQL server version
   for the right syntax to use near 'DROP DATABASE dbtest' at line 1
```

- If we do want to run multiple SQL statements, we can use $mysqli -> multi_query(). [not recommended]

Cross Site Request Forgery (CSRF)

- Also called a one-click attack or session riding
- When a page from a website sends an HTTP request back to the website, it is called a same-site request.



- If a request is sent to a different website, it is called a cross-site request because where the page comes from and where the request goes are different.

Eg: A webpage can include a Facebook link, so when users click on the link, an HTTP request is sent to Facebook.

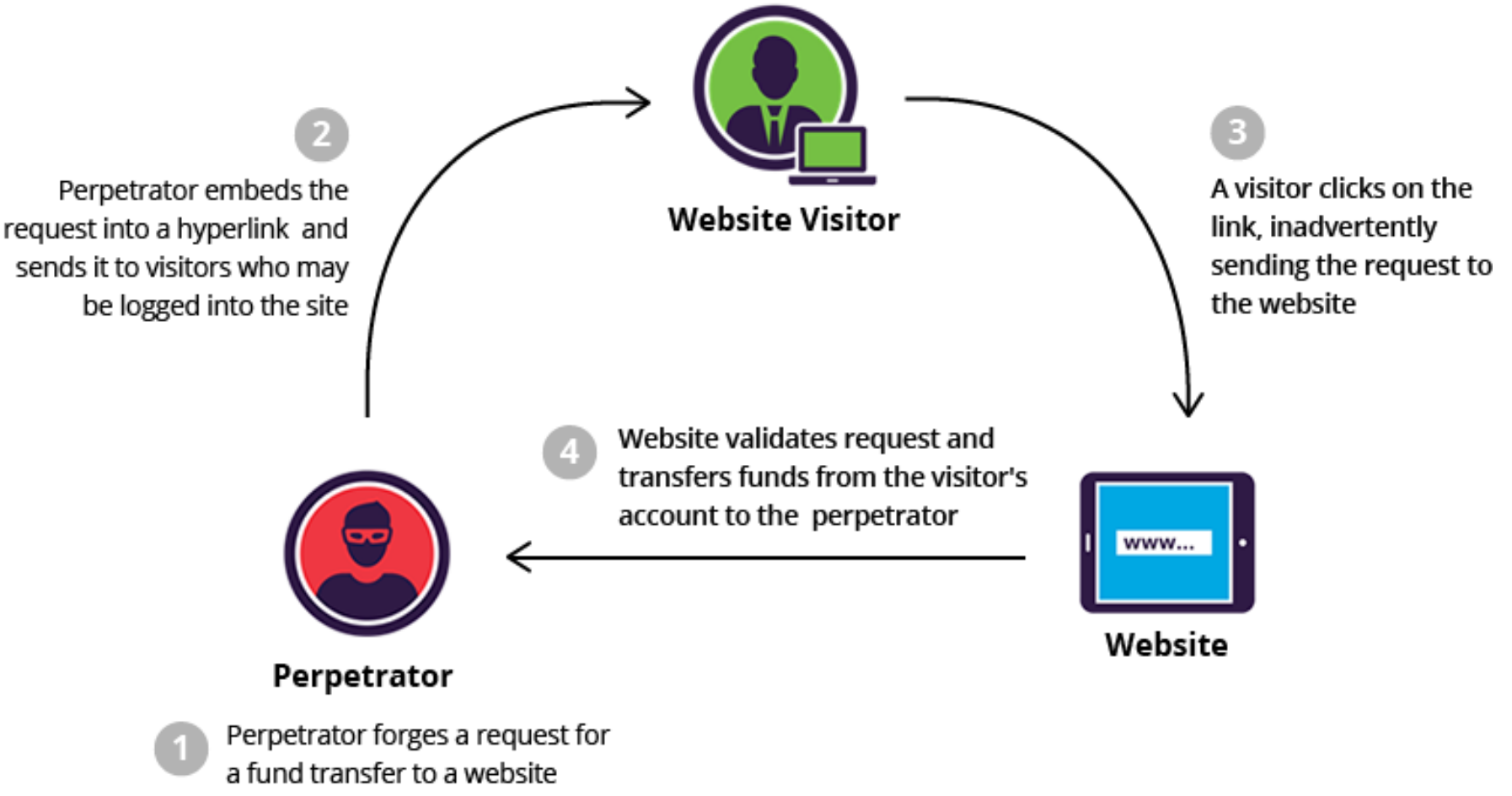
How the attacker launch CSRF Attack ?

- An attacker may Setup:
- A Target website (malicious)
- Get information on the Victim user who has an active session on the target website
    - The user login to a bank and forgets to sign out
- The attacker has full control of the malicious website

Steps:

- The attacker crafts a webpage that can forge a cross-site request to be sent to the targeted website
- The attacker needs to attract the victim user to visit the malicious website
- The victim is logged into the targeted website

How the attacker launch CSRF Attack ?



https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/

CSRF Attack on GET Requests - Basic Idea

- Consider an online banking web application www.bank32.com which allows users to transfer money from their accounts to other people's accounts.

- An user is logged in into the web application and has a session cookie which uniquely identifies the authenticated user.

- The user sends HTTP request to transfer $500 from his/her account to account 3220:

http://www.bank32.com/transfer.php?to=3220&amount=500

- In order to perform the attack:

- The attacker needs to send out the forged request from the victim's machine so that the browsers will attach the victim's session cookies (stolen) with the requests.

CSRF Attack on GET Requests - Basic Idea

- The attacker can place the piece of code (to trigger request) in the form of JavaScript code in the attacker's web page.

- HTML tags like img and iframe can trigger GET requests to the URL specified in src attribute. Response for this request will be an image/webpage.

```
<img src="http://www.bank32.com/transfer.php?to=3220&amount=500">

<iframe
   src="http://www.bank32.com/transfer.php?to=3220&amount=500">
</iframe>
```

Fundamental Causes of CSRF

- The server cannot distinguish whether a request is cross-site or same-site

  - Same-site request: coming from the server's own page. Trusted.
  - Cross-site request: coming from other site's pages. Not Trusted.
  - We cannot treat these two types of requests the same.

- Does the browser know the difference?
  - Of course. The browser knows from which page a request is generated.
  - Can the browser help?

- How to help the server?
  - Referrer header (browser's help)
  - Same-site cookie (browser's help)
  - Secret token (the server helps itself to defend against CSRF)

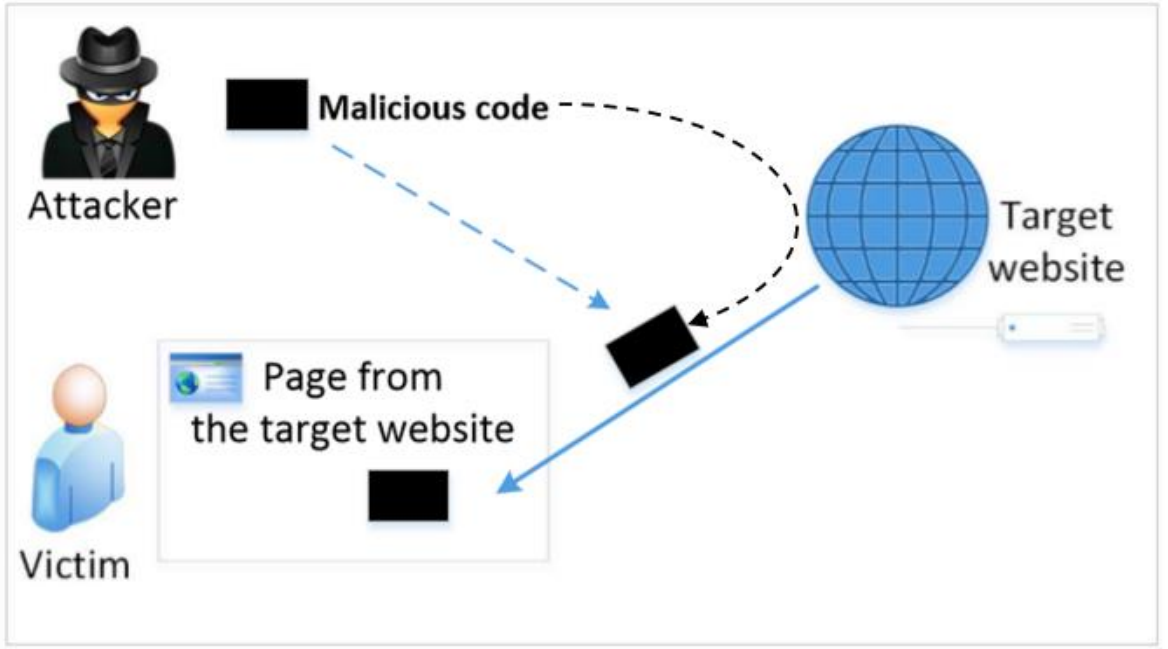
Countermeasures - Referrer Header

- HTTP header field identifying the address of the web page from where the request is generated.
- A server can check whether the request is originated from its own pages or not.
- This field reveals part of browsing history causing privacy concern and hence, this field is mostly removed from the header.
- The server cannot use this unreliable source.
- User side prevention:
  -Logging off from one site before using another
  -Selectively send authentication tokens with request

Countermeasures: Same-Site Cookies

- A special type of cookie in browsers like Chrome and Opera, which provide a special attribute to cookies called Same Site.
- This attribute is set by the servers and it tells the browsers whether a cookie should be attached to a cross-site request or not.
- Cookies with this attribute are always sent along with same-site requests, but whether they are sent along with cross-site depends on the value of this attribute.
- Values
  - Strict (Not sent along with cross-site requests)
  - Lax (Sent with cross-site requests)

The Cross-Site Scripting (XSS)



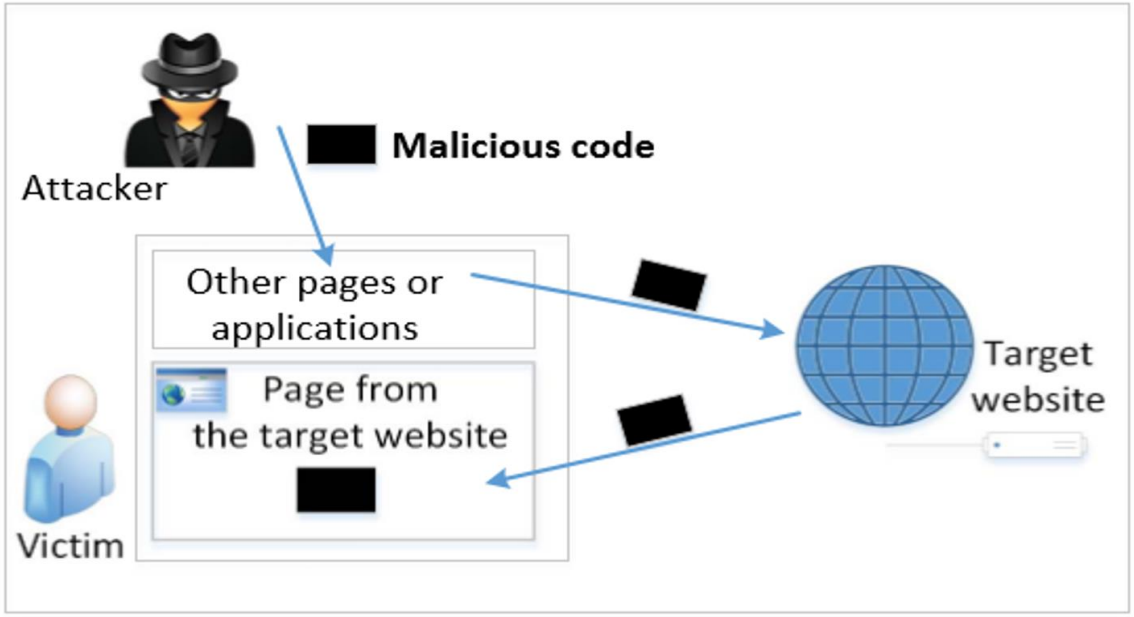
- XSS attack types:
  - Non-persistent (Reflected) XSS Attack
  - Persistent (Stored) XSS attack

- Attacker injects malicious code to the victim's browser via the target website.
  - Web pages (HTML) embed dynamic contents (code)
- When code comes from a website, it is considered as trusted and necessary actions (execution) are performed by the user browser.
- However, code can do whatever the user can do inside the session.

Non-persistent (Reflected) XSS Attack

If a website with a reflective behaviour takes user inputs, then :

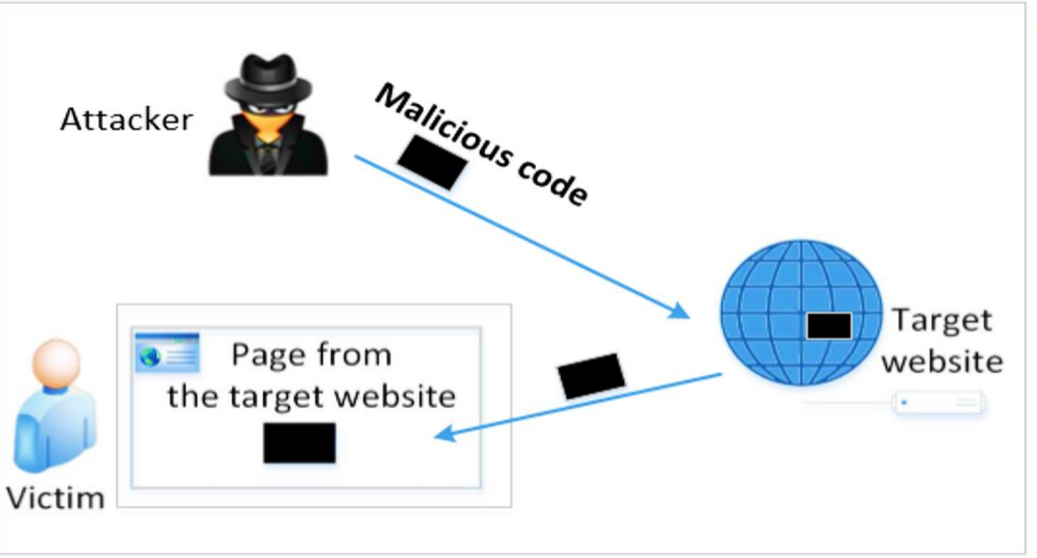
- Attackers can put JavaScript code in the input, so when the input is reflected back, the JavaScript code will be injected into the web page from the website.



Persistent (Stored) XSS Attack

Example : User profile in a social network is a channel as it is set by one user and viewed by another.

- Attackers directly send their data to a target website/server which stores the data in persistent storage.
- If the website later sends the stored data to other users, it creates a channel between the users and the attackers.



- These channels are supposed to be data channels.
- But data provided by users can contain HTML mark-ups and
- JavaScript code.
- If the input is not sanitized properly by the website, it is sent to other users' browsers through the channel and gets executed by the browsers.
- Browsers consider it like any other code coming from the website. Therefore, the code is given the same privileges as that from the website.

Damage Caused by XSS (severity)

- **Web defacing:** JavaScript code can use DOM APIs to access the DOM nodes inside the hosting page. Therefore, the injected JavaScript code can make arbitrary changes to the page. Example: JavaScript code can change a news article page to something fake or change some pictures on the page.

- **Spoofing requests:** The injected JavaScript code can send HTTP requests to the server on behalf of the user.

- **Stealing information:** The injected JavaScript code can also steal the victim's private data including the session cookies, personal data displayed on the web page, data stored locally by the web application

Countermeasures: XSS Attack

Filter Approach

- Remove code from user inputs.
- Use of open-source libraries that can filter out JavaScript code.
- Example : jsoup
- Sandbox
  - Separate or limit running untrusted programs
    - Java scripts, mobile apps, network daemon programs

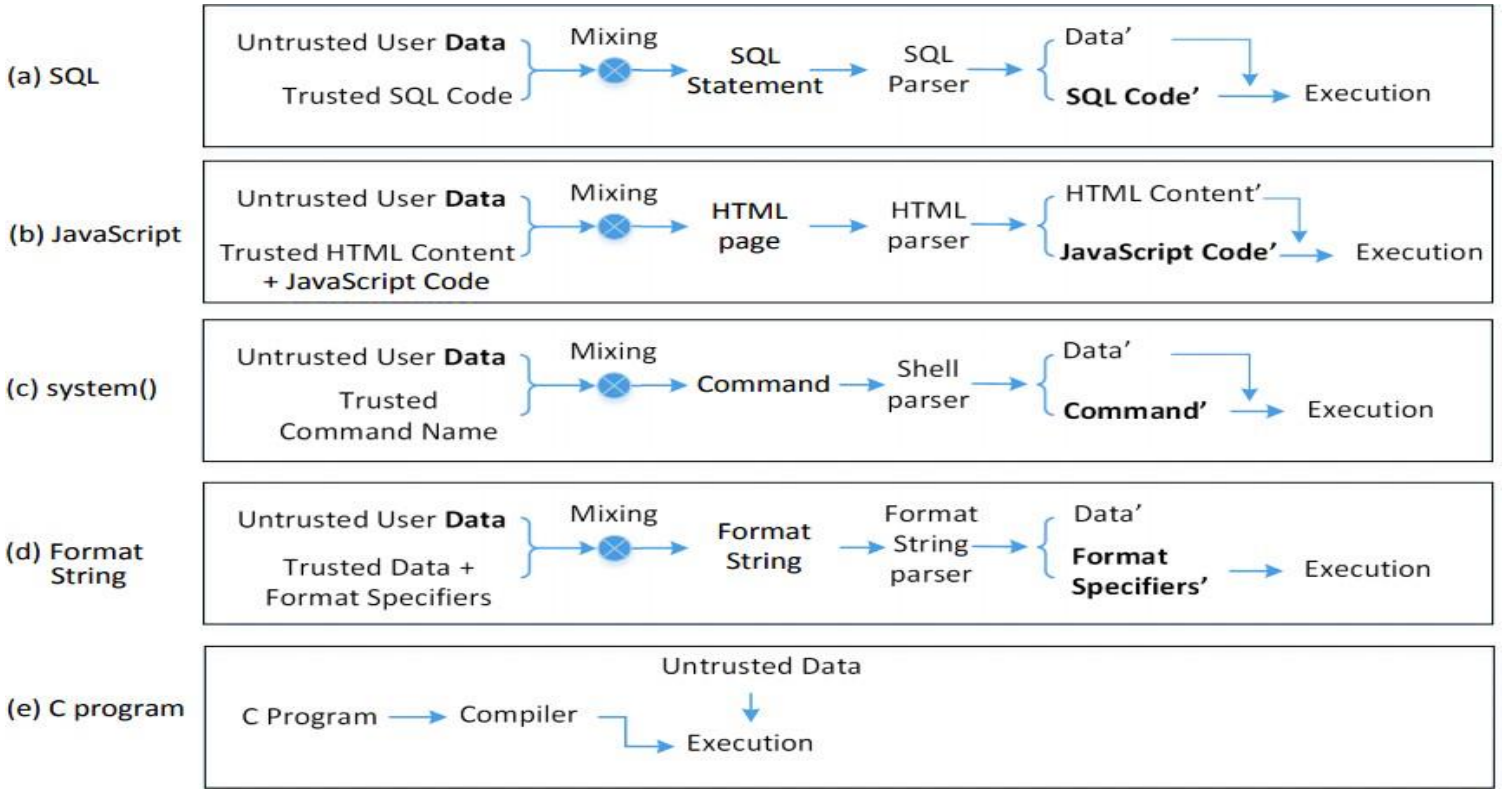
Encoding Approach

- Replaces HTML mark-ups with alternate representations.
- If data containing JavaScript code is encoded before being sent to the browsers, the embedded JavaScript code will be displayed by browsers, not executed by them.
- Convert
  - <script> alert('XSS') </script> to <script>alert('XSS')

Secret Token

- The server embeds a random secret value inside each web page.
- When a request is initiated, the secret value is generated.
- The server checks this value to see whether a request is cross-site or not.
- The secret is randomly generated and is different for different users.

The Fundamental Cause



Mixing data and code together is the cause of several types of vulnerabilities and attacks including SQL Injection attack, XSS attack, attacks on the system() function and format string attacks.

Countermeasures: Filtering and Encoding Data

- Before mixing user-provided data with code, inspect the data. Filter out any character that may be interpreted as code.
- Special characters are commonly used in SQL Injection attacks. To get rid of them, encode them.
- Encoding a special character tells the parser to treat the encoded character as data and not as code. This can be seen in the following example

```
Before encoding:   aaa' OR 1=1 #
After encoding:    aaa\' OR 1=1 #
```

PHP's mysqli extension has a built-in method called mysqli::real_escape_string(). It can be used to encode the characters that have special meanings in SQL. The following code snippet shows how to use this API

```
/* getdata_encoding.php */
<?php
   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $eid = $mysqli->real_escape_string($_GET['EID']);          ①
   $pwd = $mysqli->real_escape_string($_GET['Password'];      ②
   $sql = "SELECT Name, Salary, SSN
           FROM employee
           WHERE eid= '$eid' and password='$pwd'";
?>
```

Countermeasures: Prepared Statement

- **Prepared Statement:** It is an optimized feature that provides improved performance if the same or similar SQL statement needs to be executed repeatedly. Using prepared statements, we send an SQL statement template to the database, with certain values called parameters left unspecified. The database parses, compiles and performs query optimization on the SQL statement template and stores the result without executing it. We later bind data to the prepared statement.

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$sql = "SELECT Name, Salary, SSN
        FROM employee
        WHERE eid= '$eid' and password='$pwd'";
$result = $conn->query($sql);
```

The vulnerable version: code

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$sql = "SELECT Name, Salary, SSN
        FROM employee
        WHERE eid= ? and password=?";                    ①

if ($stmt = $conn->prepare($sql)) {                      ②
   $stmt->bind_param("ss", $eid, $pwd);                  ③
   $stmt->execute();                                     ④

   $stmt->bind_result($name, $salary, $ssn);             ⑤
   while ($stmt->fetch()) {                              ⑥
      printf ("%s %s %s\n", $name, $salary, $ssn);
   }
}
```

Send code

Send data

Start execution

Using prepared statements, we separate code and data.

Why Are Prepared Statements Secure?

- Trusted code is sent via a code channel.
- Untrusted user-provided data is sent via the data channel.
- The database clearly knows the boundary between code and data.
- Data received from the data channel is not parsed.
- The attacker can hide code in data, but the code will never be treated as code, so it will never be attacked.

Summary

- Most cyber attacks are launched through web
- Phishing, Cross site scripting, URL Obfuscation, Mobile codes are some of the largely used web- based attacks
- Even with the use of sophisticated and Secured connection (SSL, HTTPS), new attacks are going to emerge in future