CS 253: Web Security
DNS, HTTP, Cookies
Admin

- Assignment 0 is out!
- Dropping one assignment, for total of 5 assignments
- Office hours this week
  - Feross: Today, 3-5pm, Gates 323
  - Esther: Friday, 1-3pm, Huang Basement
What happens when you type a URL and press enter?
Domain Name System (DNS)
Client

stanford.edu?

171.67.215.200

DNS Server
How does the "DNS server" work?
DNS

Client → stanford.edu? → DNS Recursive Resolver

Root Nameserver

Feross Aboukhadjej
DNS

Client

stanford.edu?

DNS Recursive Resolver

stanford.edu?

See ".edu" NS

Root Nameserver

".edu" Nameserver
DNS

Client → DNS Recursive Resolver

Root Nameserver

".edu" Nameserver

stanford.edu?

See ".edu" NS

stanford.edu?

See "stanford.edu" NS

stanford.edu?
The diagram illustrates the Domain Name System (DNS) process. A client initiates a DNS query for "stanford.edu". The DNS resolver then queries the root nameserver for "stanford.edu", which in turn queries the .edu nameserver. The .edu nameserver refers to the "stanford.edu" nameserver, which returns the IP address 171.67.215.200 to the resolver. The resolver then returns this information to the client.
What happens when you type a URL and press enter?

1. **Client** asks **DNS Recursive Resolver** to lookup a hostname (**stanford.edu**).
2. **DNS Recursive Resolver** sends DNS query to **Root Nameserver**
   - **Root Nameserver** responds with IP address of **TLD Nameserver** (".edu" Nameserver)
3. **DNS Recursive Resolver** sends DNS query to **TLD Nameserver**
   - **TLD Nameserver** responds with IP address of **Domain Nameserver** ("stanford.edu" Nameserver)
4. **DNS Recursive Resolver** sends DNS query to **Domain Nameserver**
   - **Domain Nameserver** is authoritative, so replies with server IP address.
5. **DNS Recursive Resolver** finally responds to **Client**, sending server IP address (171.67.215.200)
DNS + HTTP

Client

DNS Recursive Resolver

Server

1.2.3.4
DNS + HTTP

Client

stanford.edu?

DNS Recursive Resolver

Server

1.2.3.4
DNS + HTTP

Client

DNS Recursive Resolver

stanford.edu?

1.2.3.4

Server

1.2.3.4
DNS + HTTP

Client

stanford.edu?
1.2.3.4

DNS Recursive Resolver

HTTP Request

Server

1.2.3.4
DNS + HTTP

Client → DNS Recursive Resolver

stanford.edu?
1.2.3.4

DNS Recursive Resolver → Server

HTTP Request

HTTP Response

1.2.3.4

Server
Attacks on DNS
DNS hijacking

- Attacker changes DNS records of target to point to own IP address
- All site visitors are directed to attacker’s web server
- Motivation
  - Phishing
  - Revenue through ads, cryptocurrency mining, etc.
- How do they do it?
DNS hijacking vectors

- Malware changes user's local DNS settings
- Hacked recursive DNS resolver
- Hacked router
- Hacked DNS nameserver
- Compromised user account at DNS provider
DNS hijacking

Client

Hijacked DNS Resolver

Malicious Server
9.9.9.9

Server
1.2.3.4
DNS hijacking

Client → Hijacked DNS Resolver

stanford.edu?

Malicious Server

Server

9.9.9.9

1.2.3.4
DNS hijacking

Client → Hijacked DNS Resolver

stanford.edu?
9.9.9.9

Hijacked DNS Resolver → Malicious Server

9.9.9.9

Malicious Server → Server

1.2.3.4

Feross Aboukhadijeh
DNS hijacking

Client -> 9.9.9.9

stanford.edu?

HTTP Request

Hijacked DNS Resolver

9.9.9.9

Malicious Server

9.9.9.9

Server

1.2.3.4

34 Feross Aboukhadieh
DNS hijacking

Client

Hijacked DNS Resolver

stanford.edu?

HTTP Request

HTTP Response

Malicious Server

9.9.9.9

Server

1.2.3.4

9.9.9.9

Feross Aboukhadieh
86% of Education Industry Experienced DNS Attack in Past Year

The education industry also has the lowest adoption of network security policy management automation at only 8%, according to a new report.
DNS privacy

- Queries are in plaintext
- ISPs have been known to sell this data
- **Pro tip:** Consider switching your DNS settings to Cloudflare (1.1.1.1) or another provider with a good privacy policy
What’s next in making Encrypted DNS-over-HTTPS the Default

Selena Deckelmann | September 6, 2019

In 2017, Mozilla began working on the DNS-over-HTTPS (DoH) protocol, and since June 2018 we’ve been running experiments in...
What happens when you type a URL and press enter?
HTTP

Client \hspace{2cm} Request \hspace{2cm} Server
Demo: Make an HTTP request
Demo: Make an HTTP request

curl https://twitter.com

curl https://twitter.com > twitter.html

open twitter.html
HTTP request

GET / HTTP/1.1
Host: twitter.com
User-Agent: Mozilla/5.0 ...
GET / HTTP/1.1

Method Path Protocol Version
HTTP

- **Client-server model** - Client asks server for resource, server replies
- **Simple** - Human-readable text protocol
- **Extensible** - Just add HTTP headers
- **Stateless** - Two requests have no relation to each other
- **Transport protocol agnostic** - Only requirement is reliability
HTTP is stateless?

- Obviously, we interact with "stateful" servers all the time
- "Stateless" means the HTTP protocol itself does not store state
- If state is desired, is implemented as a layer on top of HTTP
HTTP Status Codes

- **1xx** - Informational ("Hold on")
- **2xx** - Success ("Here you go")
- **3xx** - Redirection ("Go away")
- **4xx** - Client error ("You messed up")
- **5xx** - Server error ("I messed up")
HTTP Success Codes

- 200 OK - Request succeeded
- 206 Partial Content - Request for specific byte range succeeded
HTTP Redirection Codes

- **301 Moved Permanently** - Resource has a new permanent URL
- **302 Found** - Resource temporarily resides at a different URL
- **304 Not Modified** - Resource has not been modified since last cached
HTTP Client Error Codes

- **400 Bad Request** - Malformed request
- **401 Unauthorized** - Resource is protected, need to authorize
- **403 Forbidden** - Resource is protected, denying access
- **404 Not Found** - Ya'll know this one
HTTP Server Error Codes

- 500 Internal Server Error - Generic server error
- 502 Bad Gateway - Server is a proxy; backend server is unreachable
- 503 Service Unavailable - Server is overloaded or down for maintenance
- 504 Gateway Timeout - Server is a proxy; backend server responded too slowly
HTTP proxy servers

- Can cache content
- Can block content (e.g. malware, adult content)
- Can modify content
- Can sit in front of many servers ("reverse proxy")
HTTP with a proxy server

- Client
- Proxy
- Server
HTTP with a proxy server

Client \rightarrow Proxy \rightarrow Server

Request
HTTP with a proxy server

Request

Client ➔ Proxy ➔ Server
HTTP with a proxy server

Client → Request → Proxy → Request → Response → Server
HTTP with a proxy server
HTTP request

GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 ...
Host: example.com
HTTP headers

- Let the client and the server pass additional information with an HTTP request or response
- Essentially a map of key-value pairs
- Allow experimental extensions to HTTP without requiring protocol changes
Useful HTTP request headers

- **Host** - The domain name of the server (e.g. `example.com`)
- **User-Agent** - The name of your browser and operating system
- **Referer** - The webpage which led you to this page (misspelled)
- **Cookie** - The cookie server gave you earlier; keeps you logged in
- **Range** - Specifies a subset of bytes to fetch
Useful HTTP request headers

- **Cache-Control** - Specifies if you want a cached response or not
- **If-Modified-Since** - Only send resource if it changed recently
- **Connection** - Control TCP socket (e.g. **keep-alive** or **close**)
- **Accept** - Which type of content we want (e.g. **text/html**)
- **Accept-Encoding** - Encoding algorithms we understand (e.g. **gzip**)
- **Accept-Language** - What language we want (e.g. **es**)

67 Feross Aboukhadijeh
Demo: Make an HTTP request with headers
Demo: Make an HTTP request with headers

curl https://twitter.com --header "Accept-Language: es" --silent | grep JavaScript

curl https://twitter.com --header "Accept-Language: ar" --silent | grep JavaScript
HTTP response

HTTP/1.1 200 OK
Content-Length: 9001
Content-Type: text/html; charset=UTF-8
Date: Tue, 24 Sep 2019 20:30:00 GMT

<!DOCTYPE html ...
Useful HTTP response headers

- **Date** - When response was sent
- **Last-Modified** - When content was last modified
- **Cache-Control** - Specifies whether to cache response or not
- **Expires** - Discard response from cache after this date
- **Vary** - List of headers which affect response; used by cache
- **Set-Cookie** - Set a cookie on the client
Useful HTTP response headers

- **Location** - URL to redirect the client to (used with 3xx responses)
- **Connection** - Control TCP socket (e.g. `keep-alive` or `close`)
- **Content-Type** - Type of content in response (e.g. `text/html`)
- **Content-Encoding** - Encoding of the response (e.g. `gzip`)
- **Content-Language** - Language of the response (e.g. `ar`)
- **Content-Length** - Length of the response in bytes
Demo: Implement an HTTP client

- Not magic!
- Steps:
  - Open a TCP socket
  - Send HTTP request text over the socket
  - Read the HTTP response text from the socket
const net = require('net')

const socket = net.createConnection({
  host: 'example.com',
  port: 80
})

const request = `GET / HTTP/1.1
Host: example.com

`.slice(1)

socket.write(request)
socket.pipe(process.stdout)
const dns = require('dns')
const net = require('net')

dns.lookup('example.com', (err, address) => {
  if (err) throw err

  const socket = net.createConnection({
    host: address,
    port: 80
  })

  const request = `GET / HTTP/1.1
Host: example.com

`.slice(1)

  socket.write(request)
  socket.pipe(process.stdout)
})
Demo: Chrome DevTools
What happens when you type a URL and press enter?

1. Perform a **DNS lookup** on the hostname (**example.com**) to get an IP address (**1.2.3.4**)

2. Open a **TCP socket** to **1.2.3.4** on port **80** (the HTTP port)

3. Send an **HTTP request** that includes the desired path (**/**)  

4. Read the **HTTP response** from the socket  

5. Display the response to the user
Missing some steps!

- HTML subresources!
What happens when you type a URL and press enter?

1. Perform a DNS lookup on the hostname (example.com) to get an IP address (1.2.3.4)
2. Open a TCP socket to 1.2.3.4 on port 80 (the HTTP port)
3. Send an HTTP request that includes the desired path (/)
4. Read the HTTP response from the socket
5. Parse the HTML into the DOM
6. Render the page based on the DOM
7. Repeat until all external resources are loaded:
   - If there are pending external resources, make HTTP requests for these (run steps 1-4)
   - Render the resources into the page
Client

DNS Recursive Resolver

171.67.215.200

stanford.edu?

NS
NS
NS
DNS Recursive Resolver

GET /
200 OK, <!doctype html...>

GET /style.css

GET /logo.png
200 OK, body { color: hot-pink; }

200 OK, <binary image data>
Cookies
Server sets a cookie on the client

Set-Cookie: theme=dark;
Client sends cookie to the server

Cookie: theme=dark;
Demos: Cookies
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8' />
    <title>My Cool Site</title>
  </head>
  <body>
    <form method='POST' action='/login'>
      Username:
      <input name='username' />
      Password:
      <input name='password' type='password' />
      <input type='submit' value='Login' />
    </form>
  </body>
</html>
Demo: Insecure Login 1

```javascript
const express = require('express')
const cookieParser = require('cookie-parser')
const { createReadStream } = require('fs')
const bodyParser = require('body-parser')

const app = express()
app.use(cookieParser())
app.use(bodyParser.urlencoded({ extended: false }))

// Routes go here!

app.listen(4000)
```
Demo: Insecure Login 1

```javascript
const USERS = { alice: 'password', bob: '50505' }
const BALANCES = { alice: 500, bob: 100 }

app.get('/', (req, res) => {
  const username = req.cookies.username
  if (username) {
    res.send(`Hi ${username}. Your balance is ${BALANCES[username]}.`)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
  if (password === req.body.password) {
    res.cookie('username', username)
    res.redirect('/', 302)
  } else {
    res.send('fail!')
  }
})
```