CSE361 Web Security

How the web works

Nick Nikiforakis
nick@cs.stonybrook.edu
Before we break the web

• We must first understand how it works

• Questions that need to be answered:
  – How the browser works?
  – What happens when we type a URL and hit “Enter”? 
  – How does Facebook remember who I am?
  – ...

Browser and Network

Browser

OS

Hardware

Network

website

request

reply
DNS

• istheinternetonfire.com does not mean anything to a computer

• So first your browser needs to find the IP address belonging to that domain name
The answer from nslookup

nslookup istheinternetonfire.com
Server: 97.107.133.4
Address: 97.107.133.4#53

Non-authoritative answer:
Name: istheinternetonfire.com
Address: 166.84.7.99
How does that work?

• DNS (Domain Name System) works through distributed hierarchical database of DNS servers

• Your computer has what is called a “stub resolver”.
  – This stub resolver does two things:
    • 1. Ask your recursive resolver (typically provided to you by your ISP) to resolve domains for it
    • 2. Remember (cache) the answer of recent queries
How does that work?

• Given that this is the first time you tried to go to this website, your stub resolver asks your network’s recursive resolver the same question
  – If another user asked that question recently, your recursive resolver (like your stub resolver) remembers the answer and provides it immediately
  – If not then the recursive resolver ask the root servers
    • Root server == “Gate keepers of worldwide DNS”
    • 13 Root servers distributed across the world managed by various entities
      – E.g. Verisign operates 2 out of the 13 servers
There are thousands of servers supporting the root, located strategically according to where the most Internet activity occurs. The DNS ensures your query will be sent to a server that isn’t too far away. (*there is a lot more to explain around this, but this is the short version.*) Verisign has committed to develop a truly globally distributed infrastructure. It’s just one of the ways Verisign keeps the Internet fast and reliable for the people who depend on it.

Note: 2 root servers DOES NOT mean two physical machines.
Root servers

- The only thing that root servers know, is where the TLD name servers are
  - Servers for .com, .net, .org, etc.
- When your ISP’s recursive resolver asks a root server for the address of istheinternetonfire.com the answer is:
  - I don’t know, but here is a list of .com nameservers that will probably know
TLD Nameserver

• Q: Hey .com Nameserver, what is the IP address of istheinternetonfire.com?

• A: I don’t know, but go ask the nameservers that are responsible for resolving it, a.dns.gandi.net, b.dns.gandi.net, c.dns.gandi.net
  – Notice that the NS server is located on the .net TLD
  – To save us the trip up to the root and down the .net server, the .com nameserver provides the IP address of the nameserver in its response
    • This is possible because .com and .net are both operated by Verisign
Authoritative Nameserver

• Q: Hey b.dns.gandi.net what is the IP address of istheinternetonfire.com?

• A: The IP address of istheinternetonfire.com is 166.84.7.99

Now the recursive resolver caches the result and returns the address to your stub resolver running in your operating system.
Step 1: Question: what is the IP Address of some-webserver.com? Please reply to My IP Address.

Step 2: Question: where can I find the IP Address of some-webserver.com?

Step 3: Answer: I don't know but .com Namespace should have the answer.

Step 4: Question: What is the IP Address of some-webserver.com?

Step 5: Answer: Primary DNS Server of some-webserver.com knows it.

Step 6: Question: What is the IP Address of some-webserver.com?

Step 7: Answer: Here is the IP Address of some-webserver.com.

Step 8: Answer: Here is the IP Address of some-webserver.com.

User's PC
My IP Address

Primary DNS Server of some-webserver.com

User's Primary DNS Server (Recursion Allowed)

Not authoritative for some-webserver.com

Root Servers

.com Namespace
DNS Hierarchy (it’s a tree!)
DNS demo

• Let’s use `dig` and `wireshark` to better understand this process:

`dig +trace istheinternetonfire.com`
Next step

• Now that your browser knows the address, it can open a TCP/IP socket to that IP address and start sending information
  – The scheme of the URL tells your browser what port to connect to and what protocol to use

• What port is the webservice listening on?
  – By default port 80
  – Unless it’s HTTPS which will then be port 443

• What kind of information do we send the server?
  – We send a request for the main page using the HTTP protocol and the GET method
HTTP: Brief History

• HTTP is a text-based protocol built on top of TCP/IP
• The client (e.g. a web browser) establishes the protocol, sends a message and receives a response to that message from the web server
• The first HTTP protocol was described by Tim Berners-Lee, who is considered the father of the world wide web
  – HTTP 0.9
• HTTP 0.9 was very simple
HTTP/0.9

• Suppose a user clicked on a link that pointed to http://example.com/foo.txt
• The client (web browser) would establish a socket with the server located on the IP address to which example.com resolved
• The browser would then send a single line request

GET /foo.txt
HTTP/0.9

• The server would then immediately respond with the requested data

• Limitations of HTTP/0.9
  – No ability for clients to tell the server something about the user, e.g., his preferred language
  – No standardized way for the server to communicate errors to the client.
    • File not found
    • File has moved to another location
  – No ability to host multiple sites on the same IP address
    • Public IP addresses are not cheap!
HTTP/1.0 and 1.1

• HTTP/1.0 (and later 1.1) fixed these deficiencies
  – Host header that allowed the client to tell the server the website that it wanted
  – HTTP headers (metadata) for both the requests and responses
  – A lot of status messages that servers could use to inform clients about errors, changes, etc.
  – Ability to send multiple HTTP requests and responses over the same connection (this came in HTTP/1.1)
  – Compression support
  – Etc.
• The format of URLs is the following:
  – Items in brackets are optional

scheme://[username:password@]hostname[:port][/path/to/resource][?query_string][#fragment]
URLs

- **https://www.facebook.com**
  - Scheme: https
  - No credentials
  - Hostname: **www.facebook.com**
  - Port: Not specified, therefore default used (443 for HTTPS)
  - Path: /
  - No query string, no fragment
URLs

- **http://example.com/.foo/index.php?a=1&b=2#foo**
  - Scheme: http
  - No credentials
  - Hostname: example.com
  - Port: Not specified, therefore default used (80 for HTTP)
  - Path: /foo/index.php
  - Query string: a=1&b=2
  - Fragment: foo

  - Note that fragments are not sent to the server, they are kept and used only by the client, typically to scroll to a particular location of the incoming document
    - `<a name="#foo"></a>`
  - A website can still access them via JavaScript
HTTP request

Back to our istheinternetonfire.com example:

GET / HTTP/1.1
Host: istheinternetonfire.com
Proxy-Connection: keep-alive
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp, */*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.104 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
HTTP Requests

• A request has the form:

```
<METHOD> /path/to/resource?query_string HTTP/1.1
<header>*
<BODY>
```

• HTTP supports a variety of methods, with two being the most popular ones:
  – GET: intended for information retrieval
    • Typically the BODY is empty
  – POST: intended for submitting information
    • Typically the BODY contains the submitted information
Other HTTP methods

- **HEAD**
  - Works like GET but the server does not send the body of a response (it only sends the appropriate headers)

- **TRACE**
  - Designed for diagnostic purposes. Returns in its response body the exact request it received.

- **OPTIONS**
  - Returns the available methods for a specific resource.

- **PUT**
  - Allows the upload of a file in certain location. This should be disabled by default.
Popular request headers

• All request headers are meant to communicate some information to the server

• User-Agent
  – Family and version of browser, as well as the underlying environment

• Accept
  – Kind of content the client is willing to accept

• Accept-encoding
  – What type of encoding the client supports (e.g. gzip)

• Host
  – The target website of this request

• Cookie
  – Send the server all cookies the browser has for this specific website

• Referer
  – Specifies the URL from which the current request originated
  – Note the misspelling. This is intentional.
HTTP Responses

• A response has the form

```
HTTP/1.1 <STATUS CODE> <STATUS MESSAGE>
<header>*
<BODY>
```

• Important response codes:
  – 2XX: Success, e.g. 200 OK
  – 3XX: Redirection, e.g. 301 Moved Permanently
  – 4XX: Client side error, e.g. 404 Not Found
  – 5XX: Server side error, e.g. 500 Internal Server Error
Popular response headers

• All response headers are meant to communicate some information to the client (browser)

• Cache-control:
  – Passing caching directives to the client (e.g. no-cache)

• Expires:
  – How long the content is valid (and may be cached for)

• Server
  – Provides information about the identity of the server

• Set-Cookie
  – Sets cookies for this website
HTTP 404 – Apache Web Server

Not Found

The requested URL "/foo.txt" was not found on this server.
HTTP 404 – Apache Tomcat

HTTP Status 404 -

type Status report

message

description The requested resource () is not available.

Apache Tomcat/7.0.5
Requesting www.cs.stonybrook.edu

GET / HTTP/1.1
Host: www.cs.stonybrook.edu
Proxy-Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp, */*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.104 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US, en; q=0.8
HTTP 301/302: Redirections

Answer from the web server:

HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Thu, 24 Aug 2017 21:43:26 GMT
Content-Type: text/html; charset=utf-8
X-Drupal-Cache: HIT
Location: https://www.cs.stonybrook.edu/
X-Request-ID: v-42afebea-8915-11e7-aba3-22000a208946
Cache-Control: max-age=900, public
X-Varnish: 136911680 137008170
Age: 113
Content-Length: 21
Connection: keep-alive
HTTP 500

Internal Server Error

The server encountered an internal error or misconfiguration and was unable to complete your request.

Please contact the server administrator, webmaster@gs-example.com and inform them of the time the error occurred, and anything you might have done that may have caused the error.

More information about this error may be available in the server error log.

Apache/2.0.54 Server at gs-example.com Port 80
Browser consumption of response

• The browser gets the response and starts consuming it
  – Drawing on the screen according to the HTML code that was present in the response from the web server

• <u>Lalala</u>

• <hr>

• <a href="http://a.com">Cool site!</a>
Petya makes you WannaCry. Again.

Just your average ETERNALBLUE copy cat.

Petya/Petrwrap ransomware spreading across Europe

Symantec confirms use of ETERNALBLUE, again.

Microsoft SMB Vulnerability MS17-010

@isthenetonfire Find out who cybered you this time.

Made by @jschauma. See other Signs of Triviality.
Automatic loading of remote resources

• As the browser is parsing the HTML, whenever it finds a reference to a remote resource, it will automatically make a request to get it:

• Images
  – <img src="http://foo.com/a.jpg"/>

• Cascading Style sheets
  – <link rel="stylesheet" type="text/css" href="default.css"/>

• Scripts (more on that later)
  – <script src="http..."></script>

• Frames/iframes
  – <iframe src="http..."></iframe>
Where are we at?

• We can ask for pages, and we get back responses
• We can click on links, which will generate GET requests, and navigate around

• Question
  – How about personalization?
  – How does a site know that we are logged in?
No state

• HTTP is, by design, stateless
  – There’s nothing baked in the protocol to identify one request as part of sequence of other requests

• You can try to do it by IP address, but that’s not going to work well
  – Network Address Translation
  – DHCP
  – Mobile devices
HTTP Authentication

• The “oldest” and still present authentication on the web
  – The client asks for a resource and receives an HTTP 401 response header from the server with a message to show to the user
  – The browser renders a popup
  – The user then types a username and a password which are then transferred to the server after performing some local operations, as instructed by the server
    • Basic
    • Digest
How it looks
HTTP Basic Authentication

Diagram source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication
HTTP Basic Authentication

Server response to a request

HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Members only area"
Content-Length: 0

• Depending on whether the server requests “basic” or “digest” the browser will send the credentials in different forms
  – Basic: Send the username and password as unencrypted base64 text
  – Digest: Send the password as a hash to the server
HTTP Basic Authentication

• Assuming “Basic” mode and given the following typed credentials
  – Username: foofoo
  – Password: barbar

• The browser would send a request like the following

```
GET /protected-resource
Host: example.com
Authorization: Basic Zm9vZm9vOmJhcmJhcg==
Content-Length: 0
```

\[\text{base64 encode(“foofoo:barbar”)}\]

• The browser keeps on sending it as long as we are browsing the appropriate website
  – Therefore “reminding” the server of who we are
HTTP Basic Authentication

• At the server-side, our webserver needs to know which directories should be password protected and what are the accepted usernames and passwords

• In Apache, we do this either with .htaccess files or with directives in the configuration of each virtual host
  – .htaccess: Placed in the directory we want to protect
  – .htpassword: Referenced by .htaccess and contains the accepted credentials
HTTP Basic Authentication

.htaccess file, placed in the directory we want to protect

AuthType Basic
AuthName “Restricted Content”
AuthUserFile /etc/apache2/.htpasswd
Require valid-user

.htpasswd file

foofoo:$apr1$DYs1l6.j$Uaab5DBRq5cfx2I2DuQv30
nick:$apr1$BlfUws7P$B6XU96g1HqH1.Sylslf2O0

Type of hashing  Salt  Hash
HTTP Basic Authentication

- **Downsides**
  - No easy way for a user to log out
  - If passwords are stored in files, maintenance is not easy
    - How do we allow new users to be created?
    - How can existing users change their password
  - Passwords are submitted either in the clear (base64 is encoding not encryption) or as a digest (which likely allows replay attacks)
    - HTTPS alleviates this problem, as it does in the more standard authentication methods

- **Modern uses:**
  - Mostly intranet websites
  - Router administrative consoles
  - Any place with just a handful of legitimate users whose credentials do not change often and where usability does not matter
Web-Application-level State

• Using the HTTP Basic Auth, we were reminding the server who we were via our Authorization header, thereby achieving some sort of state.

• The defacto way of having state in modern web applications is through the use of sessions transferred over HTTP cookies

• **Cookie**: A small piece of data sent by the webserver to browsers, which the browsers are to include to all their subsequent requests to that server.
Cookie Mechanics

• Web servers set cookies to browsers via HTTP headers included in their responses

• For example, a response from the web server of www.example.com may contain the following header:
  – Set-Cookie: foo=bar; lang=en;
    • A server can send multiple set-cookie headers in the same response

• The browser stores this cookie in its “cookie jar” associated with the www.example.com website
  – A website cannot store/read cookies to/from a different website
Cookie Mechanics

• With every subsequent request towards www.example.com the browser will be sending the set cookies as HTTP headers
  – Cookie: boo=bar; lang=eng

• A server that sets cookies can also send certain attributes for each cookie
  – Expires: expiration date for the cookie
    • If a date is not provided, then the cookie is deleted when you close your browser
  – Domain: www.example.com can ask that its cookies are valid in all subdomains of example.com
  – Path: Only send the cookie to the server when accessing certain paths
  – httpOnly, Secure: Security-related attributes
What Are Cookies Used For?

• Authentication
  – The cookie proves to the website that the client previously authenticated correctly

• Personalization
  – Helps the website recognize the user from a previous visit

• Tracking
  – Follow the user from site to site; learn his/her browsing behavior, preferences, and so on
Let’s look at a login form

```html
<form method="POST" action="login.php">
Username:
<input type="text" name="username" />
Password:
<input type="password" name="password" />
<input type="submit" />
</form>
```
Let’s look at a login form

- Let’s assume that the user is typing “admin” for username and “letmein” for a password

- The browser will emit a POST request, as instructed by the programmer
HTTP POST request

POST /login.php HTTP/1.1
Host: example.com
Proxy-Connection: keep-alive
Content-Length: 31
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp, */*;q=0.8
Origin: null
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.104 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Accept-Language: en-US, en; q=0.8

username=admin&password=letmein
Server-side

• The webserver receives this request, passes it to a web application and then the web application checks to see whether such a user really exists
  – Typically in a database, present on the same machine or on other dedicated servers

• Assume that the username and password are correct. Now what?
  – We will give the proper page to the user (e.g. wall/list of emails/banking page, etc.)
  – How will we remember the user in the next request?
Cookie-based Session Management

Browser

Request `example.com/index.html`
Response `Set-Cookie: SID=12345`

Request `example.com/login.html`
Cookie: `SID=12345`
Response

Request `example.com/login.php`
Cookie: `SID=12345`

Server

Cookie Jar
`example.com`  
SID=12345

Session Store
`12345`  
auth: false
Cookie-based Session Management

Request `example.com/index.html`
Response `Set-Cookie: SID=12345`

Request `example.com/login.html`
Cookie: SID=12345
Response

Request `example.com/login.php`
Cookie: SID=12345
Response

Browser

Cookie Jar
`example.com`
SID=12345

Server

Session Store
12345
auth: true
user: Bob
Sessions

• As long as different users have different session identifiers (present in their cookies), the web server will be able to tell them apart
  – Regardless of their IP address

<table>
<thead>
<tr>
<th>Session ID</th>
<th>Stored data</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>{user: “Bob”, auth: True}</td>
</tr>
<tr>
<td>56789</td>
<td>{user: “Alice”, auth: True}</td>
</tr>
<tr>
<td>12889</td>
<td>{auth: False}</td>
</tr>
</tbody>
</table>

• When users delete their cookies, the browsers no longer send out the appropriate session identifier, and thus the web server “forgets” about them
Session Identifiers

- Long pseudo-random strings
- Unique per visiting client
- Each identifier is associated with a specific visitor
  - ID A -> User A
- As sensitive as credentials (per session)
Token-based authentication: Alternative design for keeping track of users

• Conceptually, a session id is a key that we give to users. The data associated with that key is at the server-side (session store)

• Alternatively, we could encrypt session data and put it in a cookie
  – IV: initialization vector (a random number, conceptually very similar to a salt)
  – state:
    AES_cbc(“userid=123&name=Nick”,secret_key,IV)
  – state_hash: hmac-sha256(state, secret_key)
  – Set-Cookie: base64(state +”:”+ state_hash + “:” +IV)

Symmetric encryption method
Token-based authentication: Alternative design for keeping track of users

• If we did not use the state_hash, we could not be sure that the user did not tamper with the cookie

• Benefits:
  – No need for server-side data store
    • Scalable as the number of users grow

• Drawbacks
  – Hard to revoke sessions
  – Easy to make crypto mistakes if you are implementing this yourself
    • E.g. forget to use a keyed hash (like HMAC)
One missing piece

• We can create websites
• And we can have state, enabling us to have a personalized web
  – Banking, Email, Social networks, etc.

• But our pages are still static
  – The server sent some HTML, the browser drew it on the screen, and that’s it
JavaScript

• “The world’s most misunderstood programming language”

• Language executed by the Web browser
  – Scripts are embedded in webpages
  – Can run before HTML is loaded, before page is viewed, while it is being viewed, or when leaving the page

• Used to implement “active” webpages and Web applications

• A potentially malicious webpage gets to execute some code on user’s machine
JavaScript History

- Developed by Brendan Eich at Netscape
  - Scripting language for Navigator 2
- Later standardized for browser compatibility
  - ECMAScript Edition 3 (aka JavaScript 1.5)
- Related to Java in name only
  - Name was part of a marketing deal
  - “Java is to JavaScript as car is to carpet”
- Various implementations available
  - SpiderMonkey, RhinoJava, others
Common Uses of JavaScript

• Page embellishments and special effects
• Dynamic content manipulation
• Form validation
• Navigation systems
• Hundreds of applications
  – Google Docs, Google Maps, dashboard widgets in Mac OS X, Philips universal remotes ...
JavaScript in Webpages

• Embedded in HTML as a `<script>` element
  – Written directly inside a `<script>` element
    • `<script>` alert("Hello World!") `</script>`
  – In a file linked as src attribute of a `<script>` element
    `<script type="text/JavaScript" src="functions.js"></script>`

• Event handler attribute
  `<a href="http://www.yahoo.com" onmouseover="alert('hi');">`

• Pseudo-URL referenced by a link
  `<a href=javascript: alert(‘You clicked’);”>Click me</a>`
Document Object Model (DOM)

• HTML page is structured data
• DOM is object-oriented representation of the hierarchical HTML structure
  – Properties: document.alinkColor, document.URL, document.forms[], document.links[], ...
  – Methods: document.write(document.referrer)
    • These change the content of the page!
• Also Browser Object Model (BOM)
  – Window, Document, Frames[], History, Location, Navigator (type and version of browser)
Browser and Document Structure
Reading Properties with JavaScript

Sample script

1. `document.getElementById('t1').nodeName`
2. `document.getElementById('t1').nodeValue`
3. `document.getElementById('t1').firstChild.nodeName`
4. `document.getElementById('t1').firstChild.firstChild.nodeName`
5. `document.getElementById('t1').firstChild.firstChild.nodeValue`

– Example 1 returns "ul"
– Example 2 returns "null"
– Example 3 returns "li"
– Example 4 returns "text"
   • A text node below the "li" which holds the actual text data as its value
– Example 5 returns " Item 1 "

Sample HTML

```
<ul id="t1">
    <li>Item 1</li>
</ul>
```
Page Manipulation with JavaScript

• Some possibilities
  – createElement(elementName)
  – createTextNode(text)
  – appendChild(newChild)
  – removeChild(node)

• Example: add a new list item

```javascript
var list = document.getElementById('t1');
var newitem = document.createElement('li');
var newtext = document.createTextNode(text);
list.appendChild(newitem);
newitem.appendChild(newtext);
```

Sample HTML

```html
<ul id="t1">
  <li>Item 1</li>
</ul>
```
All the functional pieces are in place

• Now we can create personalized and dynamic websites. Yay!

• But what about security?
  – How do we stop websites from snooping around in each other’s business?

• An example with frames
Contact

Email
nick[at]cs.stonybrook.edu

Address
Nick Nikiforakis
Computer Science Department
Stony Brook University
Stony Brook, NY 11794-4400
USA

© 2013 Nick Nikiforakis. Original Design by Arlosin

securitee.org

iframe with src equal to
https://www.cs.stonybrook.edu/members-only
Problems?

• If there are no restrictions, securitee.org could use the DOM to dive into https://www.cs.stonybrook.edu/members-only and

1. Extract details
2. Make requests in the name of the user
3. Inspect the responses
Content in the Browser

• Origin-based separation of documents
  – Naturally enforced by the Same-Origin Policy
  – Allows you to separate sensitive parts and non-sensitive parts
  – Prevents unintended sharing of information
  – Prevents escalation of successful attack

**Origin**

The triple `<scheme, host, port>` derived from the document’s URL. For `http://example.org/forum/`, the origin is `<http, example.org, 80>`

**Same-Origin Policy**

Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted
Examples of the Same-Origin Policy

**SAME-ORIGIN POLICY**

Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted.
Domains vs Subdomains

• Subdomains
  – E.g. `private.example.com` vs `forum.example.com`
  – Considered different origin
  – Possibility to relax the origin to `example.com` using `document.domain`
  – Possibility to use cookies on `example.com`

• Completely separate domains
  – E.g. `private.example.com` vs `exampleforum.com`
  – Considered different origin, without possibility of relaxation
  – No possibility of shared cookies
Subdomains and Domain Relaxation

www.example.com

private.example.com

forum.example.com

account.example.com
Subdomains and Domain Relaxation

```
document.domain = "example.com";
```
Subdomains and Domain Relaxation

```
document.domain = "example.com";
```
So, what’s left?

• Same-Origin Policy has our backs, right?
  – It will stop attacker.com from looking into the DOM, requests, and responses.
  – No malicious website can steal a user’s data, right?

• Wrong!
  – We will go over a wide range of attacks when we move into that module of the class
Questions?
Credits

• Slides on JavaScript, DOM, attacker models and the use of cookies from Vitaly Shmatikov