Quick announcements

▪ No discussion section this Friday
▪ Midterm grades should be out shortly
Goals for today

- Understand (basically) how Web browsing works
- Understand the basic Web security model (same origin policy)
- How cookies work and some ways they get attacked
Web Architecture

- Web browser issues requests
- Web server responds
- Web browser renders response
Web Architecture

- Web browser issues requests. How? Why?
  - User typed in URL
  - User re-loaded a page
  - User clicked on a link
  - Web server responded with a redirect (telling browser to request new page)
  - Web page embedded another page (leading to request for that page)
  - Script within web page issued a request
Web Architecture

- Web server responds. How?
  - Returns a static file
  - Invokes a script and returns output
  - Invokes a plugin
Web Architecture

- Web browser renders response. How?
  - Renders HTML + CSS
  - Executes embedded JavaScript
  - Invokes a plugin (e.g., PDF)
HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):

Part 2: echo in x86 (10 pts)

Files for this sub-assignment are located in the x86 subdirectory of the student user's home directory in the VM image; that is, /home/student/x86. SSH into the VM and cd into that directory to begin working on it.

For this part, you will be implementing a simplified version of the familiar echo command, using raw x86 assembly code. The goal of this assignment is to familiarize you with writing programs directly in x86.

Your echo command must behave as follows:

- When run with a single command line argument (e.g., ./echo Hello):
HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):
HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).
HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).
HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).
HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).
Anatomy of a request

GET /index.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats

headers

body (empty)
Aside: HTTP versions

- Today’s servers generally support a mix of HTTP 1.1, 2 and 3
  - Differences mainly about efficiency
  - Modern browsers generally support everything

- HTTP/2 used by 47% of the web* as of Oct 2021
  - Allows pipelining requests for multiple objects
  - Multiplexing multiple requests over one TCP connection
  - Header compression
  - Server push

- HTTP/3 used by 22% of the web
  - Uses QUIC instead of TCP
Anatomy of a response

<table>
<thead>
<tr>
<th>status code</th>
<th>HTTP/1.0 200 OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>headers</td>
<td>Date: Sun, 21 Apr 1996 02:20:42 GMT</td>
</tr>
<tr>
<td></td>
<td>Server: Microsoft-Internet-Information-Server/5.0</td>
</tr>
<tr>
<td></td>
<td>Connection: keep-alive</td>
</tr>
<tr>
<td></td>
<td>Content-Type: text/html</td>
</tr>
<tr>
<td></td>
<td>Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT</td>
</tr>
<tr>
<td></td>
<td>Set-Cookie: ...</td>
</tr>
<tr>
<td></td>
<td>Content-Length: 2543</td>
</tr>
<tr>
<td>body</td>
<td>&lt;html&gt;Some data... whatever ... &lt;/html&gt;</td>
</tr>
</tbody>
</table>
HTTP Basics

- Client sends requests
  - Typically:
    ▪ **GET**: retrieve a resource
    ▪ **POST**: update a resource (submit a form, publish a post, etc.)
    ▪ There are a few others (PUT (i.e. replace), PATCH (update part), DELETE) but older browsers don’t use

- Server responds
  - Status + optional body
  - Status examples:
    ▪ **200**: OK
    ▪ **303**: See other (redirect)
    ▪ **404**: Not found

- Repeat...
HTTP Basics

- Remember: there are many resources in a Web page
  - Html, CSS, images, scripts, parts of other Web pages...

---

CSE 127: Computer Security

**CSE 127**

**Syllabus**

- **Schedule:** Lecture: TTh 2:30-3:20 (CENTR 119), Discussion F 5:50-6:50 (CENTR 119)
- **Instructor:** Stellino Savage
- **Office hours:** Wednesday 10:00am - 11:00am at EEB/UB 3105

**TA Office Hours:**

- Wednesday 3:30pm - 4:30pm: Sumanta Roy, at EEB/UB Room 826EA
- Thursday 10:30am - 11:30am: Kamal Khudachi, at EEB/UB Room 8210
- Thursday 3:30pm - 7:30pm: Anant Agrawal, at EEB/UB Room 8200
- Friday 11:00am - 11:00am: Lei Cai, at EEB/UB Room 8260

**Teaching Assistants and Tutors:**

- Anant Agrawal (Tutor)
- Lei Cai (TA)
- Kamal Khudachi (TA)
- Sumanta Roy (TA)

**Description**

This course focuses on computer and network security, covering a wide range of topics in both the "defensive" and "offensive" side of the field. Among these will be the core security and exploitation (buffer overflows, non-conditions, SQL injection, etc.), access control and authentication, covert channels, protocol attacks. We will also look at the hardware and software in detail the delivery/transport mechanisms, focus on the system and computer forensics. The goal of the course is to understand the security of these systems and provide you with the tools and knowledge to implement them efficiently.

To complete the projects in this course, you will need the ability to develop software programs using the C programming language, and some understanding of Assembly, PHP and SQL. We will not teach these in class and you will be expected to learn them on your own. If you do not know C, I recommend the class C Programming Language, by Raemann and Rourke, because it is short and simple.

**Logistics**

We will be using Piazza for class discussion. Rather than emailing questions to the teaching staff, please post your questions on Piazza, and we will keep the discussions organized and let everyone benefit from the answers. We may also use it for announcements. The class Piazza can be
Web Sessions

- HTTP is a **stateless protocol**. No notion of *session*.

- But most web applications are session-based
  - Session active until users logs out (or times out)

- How?
  - Cookies.

- Cookies used for variety of things including
  - **Sessions** (e.g., login, shopping carts)
  - **Personalization** (e.g., user preferences, themes, etc.)
  - **Tracking** (e.g., tracking behavior for targeted advertising)
Web Cookies

- The web server provides tokens in its response to the web browser.
  - `Set-Cookie: <cookie-name>=<cookie-value>; Property=property-value`
  - Also define “properties” for each cookie
    - E.g., when they expire, only use with https, what domains they are for, etc...

- Browser attaches those cookies to every subsequent request to that web server

- **Session Cookies:**
  - Expiration property not set
  - Exist only during current browser session
  - Deleted when browser is shut down*
  - *Unless you configured your browser to resume active sessions on re-start

- **Persistent Cookies**
  - Saved until server-defined expiration time
Setting cookies in response

HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: trackingID=3272923427328234
Set-Cookie: userID=F3D947C2
Content-Length: 2543

<html>Some data... whatever ... </html>
Sending cookie with each request

GET /index.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Cookie: trackingID=3272923427328234
Cookie: userID=F3D947C2
Host: www.example.com
Referer: http://www.google.com?q=dingbats
Basic browser execution model

- Each browser window/tab:
  - Loads content
  - Parses HTML and runs Javascript
  - Fetches sub resources (e.g., images, CSS, Javascript)
  - Respond to events like onClick, onMouseover, onLoad, setTimeout
Nested execution model

- Windows may contain frames from different sources
  - Frame: rigid visible division
  - iFrame: floating inline frame

- Why use frames?
  - Delegate screen area to content from another source
  - Browser provides isolation based on frames
  - Parent may work even if frame is broken
How do you communicate with frames?

- Message passing via postMessage API
  - Sender: `targetWindow.postMessage(message, targetOrigin);`
  
  - Receiver: `window.addEventListener("message", receiveMessage, false);`
    ```javascript
    function receiveMessage(event){
      if (event.origin !== "http://example.com")
        return;
      ...
    }
    ```
Document object model (DOM)

- Javascript can read and modify page by interacting with the DOM
  - Object Oriented interface for reading and writing website content

- Includes browser object model
  - Access window, document, and other state like history, browser navigation, and cookies

- Bottom line:
  - Web page javascript can, and does, change Web page contents dynamically
  - Can even change the contents of itself
Modifying the DOM using JS

```html
<html>
  <body>
    <ul id="t1">
      <li>Item 1</li>
    </ul>
    ... 
  </body>
</html>

<script>
  const list = document.getElementById('t1');
  const newItem = document.createElement('li');
  const newText = document.createTextNode('Item 2');
  list.appendChild(newItem);
  newItem.appendChild(newText);
</script>
```
Always remember:
Modern Web sites are **programs**

- Partially executed on the client side
  - HTML rendering, JavaScript, plug-ins (e.g. Java, Flash)

- Partially executed on the server side
  - CGI, PHP, Ruby, ASP, server-side JavaScript, SQL, etc.
Modern websites are complicated

The LA Times homepage includes 540 resources from nearly 270 IP addresses, 58 networks, and 8 countries. Many of these aren’t controlled by the main sites.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Domain</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUID</td>
<td>1656321DA67D6C8404703800A27D6AB3</td>
<td>.bing.com</td>
<td>2020-01-20...</td>
</tr>
<tr>
<td>_EDGE_S</td>
<td>SID=162F6D4DA0E16A823491600AA1516BD0</td>
<td>.bing.com</td>
<td>N/A</td>
</tr>
<tr>
<td>SRCHUID</td>
<td>V=2&amp;GUID=DCDDEA0BD104408B8367486B9E84EA69&amp;...</td>
<td>.bing.com</td>
<td>2020-06-05...</td>
</tr>
<tr>
<td>SRCHID</td>
<td>AF=NOFORM</td>
<td>.bing.com</td>
<td>2020-06-05...</td>
</tr>
<tr>
<td>_SS</td>
<td>SID=162F6D4DA0E16A823491600AA1516BD0</td>
<td>.bing.com</td>
<td>N/A</td>
</tr>
<tr>
<td>bounceClientVisit1762c</td>
<td>%7B%22vid%22%3A1556033812014037%2C%22did%...</td>
<td>.bounceexchan...</td>
<td>2019-04-23...</td>
</tr>
<tr>
<td>ajs_group_id</td>
<td>null</td>
<td>.brightcove.net</td>
<td>2019-12-11...</td>
</tr>
<tr>
<td>ajs_anonymous_id</td>
<td>%2250aa1405-b704-40f4-8d3b-6a29ffa32f73%22</td>
<td>.brightcove.net</td>
<td>2019-12-11...</td>
</tr>
<tr>
<td>ajs_user_id</td>
<td>null</td>
<td>.brightcove.net</td>
<td>2019-12-11...</td>
</tr>
<tr>
<td>__adcontext</td>
<td>{&quot;cookieID&quot;:&quot;JZZ3V2HKBW2KT6EOM02R2AWV7VLWGX...</td>
<td>.cdnwidget.com</td>
<td>2020-05-23...</td>
</tr>
<tr>
<td>__3idcontext</td>
<td>{&quot;cookieID&quot;:&quot;JZZ3V2HKBW2KT6EOM02R2AWV7VLWGX...</td>
<td>.cdnwidget.com</td>
<td>2020-05-23...</td>
</tr>
<tr>
<td>_kuid</td>
<td>DNT</td>
<td>.krxd.net</td>
<td>2019-10-20...</td>
</tr>
<tr>
<td>__idcontext</td>
<td>eyJjb29raWVJRCl6IkpaWjNWmkhLQicyS1Q2RU9NTzJS...</td>
<td>.latimes.com</td>
<td>2020-05-22...</td>
</tr>
<tr>
<td>kw_pv_session</td>
<td>3</td>
<td>.latimes.com</td>
<td>2019-04-24...</td>
</tr>
<tr>
<td>RT</td>
<td>*s=3&amp;ss=1556033808254&amp;tt=9172&amp;obo=0&amp;bcn=%2F%...</td>
<td>.latimes.com</td>
<td>2019-04-30...</td>
</tr>
<tr>
<td>_lb</td>
<td>1</td>
<td>.latimes.com</td>
<td>2019-04-23...</td>
</tr>
<tr>
<td>pcic</td>
<td>5</td>
<td>.latimes.com</td>
<td>2024-04-21...</td>
</tr>
<tr>
<td>_fbp</td>
<td>fb.1.1556033822471.1780534325</td>
<td>.latimes.com</td>
<td>2019-07-22...</td>
</tr>
<tr>
<td>__gads</td>
<td>ID=10641b22d31f2147:T=1556033820:S=ALNI_MYGSPR...</td>
<td>.latimes.com</td>
<td>2021-04-22...</td>
</tr>
<tr>
<td>sc_cc</td>
<td>true</td>
<td>.latimes.com</td>
<td>N/A</td>
</tr>
<tr>
<td>kw_session_ts</td>
<td>1556033812187</td>
<td>.latimes.com</td>
<td>2019-04-23...</td>
</tr>
<tr>
<td>bounceClientVisit1762v</td>
<td>N4lgNgDiBciBYbcEgM4FIDMBBNAmAYngO6kB0YAhg...</td>
<td>.latimes.com</td>
<td>2019-04-23...</td>
</tr>
<tr>
<td>uuid</td>
<td>69953082-e348-4cc7-b37b-b0c14adc7449</td>
<td>.latimes.com</td>
<td>2024-04-21...</td>
</tr>
<tr>
<td>_gid</td>
<td>GA1.2.771043247.1556033809</td>
<td>.latimes.com</td>
<td>2019-04-24...</td>
</tr>
<tr>
<td>_sp_ses.8129</td>
<td>*</td>
<td>.latimes.com</td>
<td>2019-04-23...</td>
</tr>
<tr>
<td>paic</td>
<td>5</td>
<td>.latimes.com</td>
<td>2024-04-21...</td>
</tr>
</tbody>
</table>
Modern websites are complicated

- Third party ads
- Google analytics
- Framed ad
- jQuery library
- Local scripts
- Extensions
Goals for today

- Understand (basically) how Web browsing works
- **Understand the basic Web security model (same origin policy)**
- How cookies work and some ways they get attacked
Relevant attacker models

Network attacker

Web attacker
Relevant attacker models

Gadget attacker
Web attacker with capabilities to inject limited content into honest page
Most of our focus: web attacker
And variants of it

evil.com

evil.com

example.com

example.com
Web security model

- Safely browse the web in the presence of web attackers
  - Browsers are like operating systems
  - Need to isolate different activities
Same origin policy (SOP)

- **Origin**: isolation unit/trust boundary on the web
  - (scheme, domain, port) triple derived from URL
  - Fate sharing: if you come from same places you must be authorized

- **SOP goal**: isolate content of different origins
  - **Confidentiality**: script contained in evil.com should not be able to read data in bank.ch page
  - **Integrity**: script from evil.com should not be able to modify the content of bank.ch page
SOP for the DOM

- Each frame in a window has its own origin
- Frame can only access data with the same origin
  - DOM tree, local storage, cookies, etc.
SOP for HTTP responses

- Pages can perform requests across origins
  - SOP does not prevent a page from leaking data to another origin by encoding it in the URL, request body, etc.

- SOP prevents code from directly inspecting HTTP responses
Documents

- Can load cross-origin HTML in frames, but not inspect or modify the frame content.
Scripts

- **Can** load scripts from across origins
  - Libraries!
- Scripts execute with privileges of the page
- Page can see source via `func.toString()`
Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- But page can see other properties, like `img.width`

```
if (img.width > 40) {
  ...
} else {
  ...
}
```
Goals for today

▪ Understand (basically) how Web browsing works
▪ Understand the basic Web security model (same origin policy)
▪ How cookies work and some ways they get attacked
SOP for cookies

- DOM SOP: origin is a (scheme, domain, port)

- Cookies use a separate definition of origins.
  - Cookie SOP: ([scheme], domain, path)
  - (https,cseweb.ucsd.edu, /classes/fa22/cse127-a)

- Server can declare domain property for any cookie
  - Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>
What cookies can a Web page set?

**domain**: any domain-suffix of URL-hostname, except "public suffixes"

example:

```text
host = "login.site.com"
```

<table>
<thead>
<tr>
<th>allowed domains</th>
<th>disallowed domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>login.site.com</td>
<td>other.site.com</td>
</tr>
<tr>
<td>.site.com</td>
<td>othersite.com</td>
</tr>
<tr>
<td>.com</td>
<td></td>
</tr>
</tbody>
</table>

⇒ login.site.com can set cookies for all of .site.com but not for another site

Note that this creates a some trickiness for places like ucsd.edu (cs.ucsd.edu, can set cookies for ucsd.edu)

**path**: can be set to anything
SOP: Cookie scope setting

PUBLIC SUFFIX LIST
LEARN MORE | THE LIST | SUBMIT AMENDMENTS

A "public suffix" is one under which Internet users can (or historically could) directly register names. Some examples of public suffixes are .com, .co.uk and pvt.k12.ma.us. The Public Suffix List is a list of all known public suffixes.

The Public Suffix List is an initiative of Mozilla, but is maintained as a community resource. It is available for use in any software, but was originally created to meet the needs of browser manufacturers. It allows browsers to, for example:

- Avoid privacy-damaging "supercookies" being set for high-level domain name suffixes
- Highlight the most important part of a domain name in the user interface
- Accurately sort history entries by site

We maintain a fuller (although not exhaustive) list of what people are using it for. If you are using it for something else, you are encouraged to tell us, because it helps us to assess the potential impact of changes. For that, you can use the psl-discuss mailing list, where we consider issues related to the maintenance, format and semantics of the list. Note: please do not use this mailing list to request amendments to the PSL's data.

It is in the interest of Internet registries to see that their section of the list is up to date. If it is not, their customers may have trouble setting cookies, or data about their sites may display sub-optimally. So we encourage them to maintain their section of the list by submitting amendments.
How do we decide to send cookies?

- Browser sends all cookies in a URL’s scope:
  - Cookie’s domain is domain suffix of URL’s domain
  - Cookie’s path is a prefix of the URL path
How do we decide to send cookies?

| Cookie 1: name = mycookie value = mycookielvalue domain = login.site.com path = / |
| Cookie 2: name = cookie2 value = mycookielvalue domain = site.com path = / |
| Cookie 3: name = cookie3 value = mycookielvalue domain = site.com path = /my/home |

<table>
<thead>
<tr>
<th>Request to URL</th>
<th>Cookie 1</th>
<th>Cookie 2</th>
<th>Cookie 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkout.site.com</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>login.site.com</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>login.site.com/my/home</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>site.com/my</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Note: the cookie path does not give us finer-grained isolation than the SOP

- **Cookie SOP:**
  - cseweb.ucsd.edu/~savage does not see cookies for cseweb.ucsd.edu/~nadiah

- **DOM SOP:**
  - cseweb.ucsd.edu/~savage can access the DOM of cseweb.ucsd.edu/~nadiah

**How can you access cookie?**

- ```
  const iframe = document.createElement("iframe");
  iframe.src = "https://cseweb.ucsd.edu/~nadiah";
  document.body.appendChild(iframe);
  alert(iframe.contentWindow.document.cookie);
```
Another example

- What happens when your bank includes Google Analytics Javascript? Can it access your Bank’s authentication cookie?
  - Yes! Javascript is running with origin’s privileges. Can access `document.cookie`.
Another example

- What happens when your bank includes Google Analytics Javascript? Can it access your Bank’s authentication cookie?
  - Yes! Javascript is running with origin’s privileges. Can access `document.cookie` in DOM

- Also, SOP doesn’t prevent leaking data:

```javascript
const img = document.createElement("image");
document.body.appendChild(img);
```
Partial solution: HttpOnly cookies

Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; HttpOnly;

Don’t expose cookie to JavaScript via document.cookie
Which cookies are sent? (Again.)
Which cookies are sent? (Again.)

```html
<html>
  <img src="https://bank.ch"/>
</html>
```

What if evil.com did this?

<html>
<img src="https://bank.ch/transfer?amt=$1B&to=evil"/>
</html>

Cross-site request forgery (CSRF) attack!
Partial solutions: SameSite cookies

Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; SameSite=Strict;

**Strict**: A same-site cookie is only sent when the request originates from the same site (top-level domain)

**Lax**: Send cookie on top-level “safe” navigations (even if navigating cross-site)

**None**: send cookie without taking context into account.
Which cookies are sent? (SameSite=none)
Which cookies are sent? (SameSite=Strict)
Which cookies are sent? (SameSite=Lax)
Which cookies are sent? (SameSite=Lax)

```html
<html>
  <a href="https://bank.ch">Click me!</a>
</html>
```

http://bank.ch

http://evil.com

http://bank.ch

http://bank.ch

http://4chan.org
Partial solutions: Secure cookies

Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure;

A secure cookie is only sent to the server with an encrypted request over the HTTPS protocol.
Again, why do we care about this stuff?

- Network attacker can steal cookies if server allows unencrypted HTTP traffic

- Don’t need to wait for user to go to the site; web attacker can make cross-origin request
Next time

- Web attacks
  - Injection
  - Cross-site scripting (XSS)
  - Cross-site request forgery (CSRF)
  - Clickjacking
  - Insecure Direct Object References
  - Misc