Web Security: Vulnerabilities & Attacks
### Primitives:
- Processes
- System calls
- File system
- Frames
- Content (including JavaScript, ...)
- Document object model, cookies, localStorage

### Principals:
- Users:
  - Discretionary access control

- "Origins":
  - Mandatory access control

### Vulnerabilities:
- Buffer overflow
- Root exploit
- Cross-site scripting
- Cross-site request forgery
- Cache history attacks
Browser security mechanism

• Each frame of a page has an origin
  – Origin = protocol://host:port
• Frame can access its own origin
  – Network access, Read/write DOM, Storage (cookies)
• Frame cannot access data associated with a different origin

(Origin is “http://www.catville.com:80”)

(Origin is “http://www.rooster-flakes.com:80”)

Dawn Song
Components of browser security policy

Frame-Frame relationships
- $\text{canScript}(A,B)$
  - Can Frame A execute a script that manipulates arbitrary/nontrivial DOM elements of Frame B?
- $\text{canNavigate}(A,B)$
  - Can Frame A change the origin of content for Frame B?

Frame-principal relationships
- $\text{readCookie}(A,S)$, $\text{writeCookie}(A,S)$
  - Can Frame A read/write cookies from Site Y?
Origin of Browser Primitives

**Cookies**
- Setting Cookies:
  - Default origin is domain and path of setting URL

**Javascript**
- Imported in a page or frame:
  - Has the same origin as that page or frame
- Embedded in a page or frame:
  - Has the same origin as that page or frame

**DOM**
- Each frame of a page:
  - Origin is protocol://host:port
Library import

<script src=https://seal.verisign.com/getseal?host_name=safebank.com></script>

- Script has privileges of imported page, NOT source server.
- Can script other pages in this origin, load more scripts
- Other forms of importing
# Same-origin policy

## Goal: To isolate content retrieved by different parties

### Same-origin policy for Javascript/DOM

Two documents have the same origin if:

- **Same protocol** (https, http, ftp, etc)
- **Same domain** (safebank.com, etc)
- **Same port** (80, 23, 8080, etc)

Results of same-origin checks against “http://cards.safebank.com/c1/info.html”

<table>
<thead>
<tr>
<th>Same origin:</th>
<th>Different origin:</th>
</tr>
</thead>
</table>

Results of same-origin checks against “http://cards.safebank.com/c1/info.html”

<table>
<thead>
<tr>
<th>Same origin:</th>
<th>Different origin:</th>
</tr>
</thead>
</table>

There are some exceptions to this rule. (for example, a document can change its domain to be any suffix of its domain, evil.catville.com -> catville.com)

### Same-origin policy for Cookies

Two documents have the same origin if:

- **Same protocol** (https, http, ftp, etc)
- **Same domain** (safebank.com, etc)
- **Same Path** (/, /c1/, etc)

*any domain-suffix or URL-hostname, except Top Level Domain*

Results of same-origin checks against “http://cards.safebank.com/c1/info.html”

<table>
<thead>
<tr>
<th>Same origin:</th>
<th>Different origin:</th>
</tr>
</thead>
</table>

*however, cookies can be accessed across different paths via the DOM

<table>
<thead>
<tr>
<th>allowed domains:</th>
<th>disallowed domains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>cards.safebank.com</td>
<td>tos.safebank.com</td>
</tr>
<tr>
<td>.safebank.com</td>
<td>catville.com</td>
</tr>
<tr>
<td></td>
<td>.com</td>
</tr>
</tbody>
</table>

There is no single same-origin policy
**Same-origin policy**

**Goal:** To isolate content retrieved by different parties

<table>
<thead>
<tr>
<th>Same-origin policy for Javascript/DOM</th>
<th>Same-origin policy for Cookies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two documents have the same origin if:</td>
<td>Two documents have the same origin if:</td>
</tr>
<tr>
<td><strong>Same</strong> protocol (https, http, ftp, etc)</td>
<td>(optional) <strong>Same</strong> protocol (https, http, ftp, etc)</td>
</tr>
<tr>
<td><strong>Same</strong> domain (safebank.com, etc)</td>
<td><strong>Same</strong> domain* (safebank.com, etc)</td>
</tr>
<tr>
<td><strong>Same</strong> port (80, 23, 8080, etc)</td>
<td><strong>Same</strong> Path** (/, /c1/, etc)</td>
</tr>
</tbody>
</table>

Results of same-origin checks against “http://cards.safebank.com/c1/info.html”

**Same origin:**

**Different origin:**
“http://www.cards.safebank.com”
“http://catville.com”
“https://cards.safebank.com”
“http://cards.safebank:8080”

**allowed domains:**
cards.safebank.com
.safebank.com

disallowed domains:
tos.safebank.com
catville.com
.com

**host=“cards.safebank.com”**

****however, cookies can be accessed across different paths via the DOM

**There is no single same-origin policy**
Security User Interface
Safe to type your password?
Safe to type your password?
Safe to type your password?
Safe to type your password?
Safe to type your password?
Mixed Content: HTTP and HTTPS

• Problem
  – Page loads over HTTPS, but has HTTP content
  – Network attacker can control page
• IE: displays mixed-content dialog to user
  – Flash files over HTTP loaded with no warning (!)
  – Note: Flash can script the embedding page
• Firefox: red slash over lock icon (no dialog)
  – Flash files over HTTP do not trigger the slash
• Safari: does not detect mixed content
Mixed Content: HTTP and HTTPS
Mixed content and network attacks

• banks: after login all content over HTTPS
  – Developer error: Somewhere on bank site write
    `<script src=http://www.site.com/script.js> </script>`
  – Active network attacker can now hijack any session

• Better way to include content:
  `<script src=//www.site.com/script.js> </script>`
  – served over the same protocol as embedding page
Lock Icon 2.0

- Extended validation (EV) certs

- Prominent security indicator for EV certificates
- note: EV site loading content from non-EV site does not trigger mixed content warning
Finally: the status Bar

- Trivially spoofable

```html
<a href="http://www.paypal.com/">
    onclick="this.href = ‘http://www.evil.com/’;”>
    PayPal
</a>
```
Cookies

Slides credit: John Mitchell
Cookies

• Used to store state on user’s machine

HTTP Header:
Set-cookie: NAME=VALUE ;
domain = (who can read) ;
expires = (when expires) ;
secure = (only over SSL)

Important Point: HTTP is a stateless protocol; cookies add state
Cookie authentication

Browser

POST login.cgi
Username & pwd
Set-cookie: **auth=val**

Web Server

Validate user
**auth=val**

Auth server

Store val

restricted.html

Check val

YES/NO

If YES, restricted.html

Cookie: **auth=val**
Cookie Security Policy

• Uses:
  – User authentication
  – Personalization
  – User tracking: e.g. Doubleclick (3rd party cookies)

• Browser will store:
  – At most 20 cookies/site, 3 KB / cookie

• Origin is the tuple \(<\text{domain}, \text{path}>\)
  – Can set cookies valid across a domain suffix
Secure Cookies

- Provides confidentiality against network attacker
  - Browser will only send cookie back over HTTPS
- ... but no integrity
  - Can rewrite secure cookies over HTTP
    ⇒ network attacker can rewrite secure cookies
    ⇒ can log user into attacker’s account
Command Injection
Background

Client Browser

URI

foo.php

Web Page

Web Server

UID: www

PHP -> WEB PAGE
Quick Background on PHP

**display.php**: `<? echo system("cat ".$_GET['file']); ?>`

**IN THIS EXAMPLE**

- `<? php-code ?>` executes php-code at this point in the document
- `echo expr:` evaluates expr and embeds in doc
- `system(call, args)` performs a system call in the working directory
- “.....”, ‘.....’ String literal. Double-quotes has more possible escaped characters.
- `_GET['key']` returns value corresponding to the key/value pair sent as extra data in the HTTP GET request

**LATER IN THIS LECTURE**

- `preg_match(Regex, Stiring)` Performs a regular expression match.
- `proc_open` Executes a command and opens file pointers for input/output.
- `escapeshellarg()` Adds single quotes around a string and quotes/escapes any existing single quotes.
- `file_get_contents(file)` Retrieves the contents of file.
Background

display.php: `<? echo system("cat ".$_GET['file']); ?>`

Client Browser → display.php?file=notes.txt

URI

Web Server

UID: www

display.php

system("cat ".$_GET['file'])

Web Page

Shell Command

cat notes.txt

Dawn Song
Today we are learning about Web Security.

Content of notes.txt

display.php: `<? echo system("cat ".$_GET['file']); ?>`
Q: Assuming the script we’ve been dealing with (reproduced above) for http://www.example.net/display.php. Which one of the following URIs is an attack URI?

Hint: Search for a URI Decoder to figure out values seen by the PHP code.

c. http://www.example.net/display.php?file=notes.txt%3B%20rm%20-rf%20%2F%3B%0A%0A
Q: Assuming the script we’ve been dealing with (reproduced above) for http://www.example.net/display.php. Which one of the following URIs is an attack URI?

Hint: Search for a URI Decoder to figure out values seen by the PHP code.

(URIs decoded)

c. http://www.example.net/display.php?file=notes.txt; rm -rf /;
Command Injection

display.php: `<? echo system("cat ".$_GET['file']); ?>`

Q: Assuming the script we’ve been dealing with (reproduced above) for
http://www.example.net/display.php. Which one of the following
URIs is an attack URI?

   Hint: Search for a URI Decoder to figure out values seen by the PHP code.

(Resulting php)

a. `<? echo system("cat rm"); ?>`
b. `<? echo system("cat rm -rf /; "); ?>`
c. `<? echo system("cat notes.txt; rm -rf /; "); ?>`
d. `<? echo system("cat "); ?>`
Injection

• Injection is a general problem:
  – Typically, caused when data and code share the same *channel*.
  – For example, the code is *“cat”* and the filename the data.
    • But ‘;’ allows attacker to start a new command.