CS 380S

0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/
Browser and Network

Browser

OS

Hardware

Network

website

request

reply
Web Threat Models

◆ Web attacker

◆ Network attacker
  - Passive: wireless eavesdropper
  - Active: evil router, DNS poisoning

◆ Malware attacker
  - Malicious code executes directly on victim’s computer
  - To infect victim’s computer, can exploit software bugs (e.g., buffer overflow) or convince user to install malicious content
    - Masquerade as an antivirus program, video codec, etc.
Web Attacker

- Controls malicious website (attacker.com)
  - Can even obtain a SSL/TLS certificate for his site ($0)
- User visits attacker.com – why?
  - Phishing email, enticing content, search results, placed by ad network, blind luck ...
  - Attacker’s Facebook app
- Attacker has no other access to user machine!
- Variation: gadget attacker
  - Bad gadget included in an otherwise honest mashup
OS vs. Browser Analogies

**Operating system**

- **Primitives**
  - System calls
  - Processes
  - Disk

- **Principals: Users**
  - Discretionary access control

- **Vulnerabilities**
  - Buffer overflow
  - Root exploit

**Web browser**

- **Primitives**
  - Document object model
  - Frames
  - Cookies / localStorage

- **Principals: “Origins”**
  - Mandatory access control

- **Vulnerabilities**
  - Cross-site scripting
  - Universal scripting
Browser: Basic Execution Model

Each browser window or frame:

- Loads content
- Renders
  - Processes HTML and scripts to display the page
  - May involve images, subframes, etc.
- Responds to events

Events

- User actions: OnClick, OnMouseover
- Rendering: OnLoad, OnUnload
- Timing: setTimeout(), clearTimeout()
“The world’s most misunderstood programming language”

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

Used to implement “active” web pages
- AJAX, huge number of Web-based applications

Potentially malicious website 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
JavaScript in Web Pages

◆ Embedded in HTML page as `<script>` element
  - JavaScript written directly inside `<script>` element
    – `<script>` alert("Hello World!") </script>
  - Linked file as src attribute of the `<script>` element
    `<script type="text/JavaScript" src="functions.js"></script>

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

◆ Pseudo-URL referenced by a link
  `<a href="JavaScript: alert('You clicked');">Click me</a>"
Event-Driven Script Execution

```html
<script type="text/javascript">
function whichButton(event) {
  if (event.button == 1) {
    alert("You clicked the left mouse button!")
  } else {
    alert("You clicked the right mouse button!")
  }
}
</script>

<script type="text/javascript">
function whichButton(event) {
  if (event.button == 1) {
    alert("You clicked the left mouse button!")
  } else {
    alert("You clicked the right mouse button!")
  }
}
</script>

<body onmousedown="whichButton(event)">
...  
</body>
```

- Script defines a page-specific function.
- Function gets executed when some event happens.
Click in the document. An alert box will alert which mouse button you clicked.

Windows Internet Explorer

You clicked the left mouse button!

OK
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

W3C standard differs from models supported in existing browsers
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>
```
Content Comes from Many Sources

◆ Frames

```html
<iframe src="//site.com/frame.html"> </iframe>
```

◆ Scripts

```html
<script src="//site.com/script.js"> </script>
```

◆ Stylesheets (CSS)

```html
<link rel="stylesheet" type="text/css" href="//site.com/theme.css" />
```

◆ Objects (Flash) - using swfobject.js script

```html
<script>
var so = new SWFObject('//site.com/flash.swf', ...,);
so.addParam('allowscriptaccess', 'always');
so.write('flashdiv');
</script>
```

Allows Flash object to communicate with external scripts, navigate frames, open windows
**Browser Sandbox**

- **Goal:** safely execute JavaScript code provided by a remote website
  - No direct file access, limited access to OS, network, browser data, content that came from other websites

- **Same origin policy (SOP)**
  - Can only read properties of documents and windows from the same **scheme**, **domain**, and **port**

- **User can grant privileges to signed scripts**
  - UniversalBrowserRead/Write, UniversalFileRead, UniversalSendMessage
C. Jackson and A. Barth

Beware of Finer-Grained Origins

(W2SP 2008)
SOP Often Misunderstood

- Often simply stated as “same origin policy”
  - This usually just refers to “can script from origin A access content from origin B”?

- Full policy of current browsers is complex
  - Evolved via “penetrate-and-patch”
  - Different features evolved slightly different policies

- Common scripting and cookie policies
  - Script access to DOM considers scheme, domain, port
  - Cookie reading considers scheme, domain, path
  - Cookie writing considers domain
Same Origin Policy (High Level)

Same Origin Policy (SOP) for DOM:
Origin A can access origin B’s DOM if A and B have same \((\text{scheme}, \text{domain}, \text{port})\)

Same Origin Policy (SOP) for cookies:
Generally, based on \(([[\text{scheme}], \text{domain}, \text{path}])\)

optional
Setting Cookies by Server

- Delete cookie by setting “expires” to date in past
- Default scope is domain and path of setting URL

HTTP Header:
Set-cookie: NAME=VALUE;

domain = (when to send);
path = (when to send);
secure = (only send over HTTPS);
expires = (when expires);
HttpOnly

if expires=NULL:
this session only

Browser → GET ...
HTTP Header:
Set-cookie: NAME=VALUE;

Server → Browser
Both cookies stored in browser’s cookie jar, both are in scope of *login.site.com*

Cookies are identified by \((\text{name}, \text{domain}, \text{path})\)

**cookie 1**
- name = *userid*
- value = *test*
- domain = *login.site.com*
- path = / secure

**cookie 2**
- name = *userid*
- value = *test123*
- domain = *.site.com*
- path = / secure

distinct cookies
SOP for Writing Cookies

**domain:** any domain suffix of URL-hostname, except top-level domain (TLD)

Which cookies can be set by **login.site.com**?

- **allowed domains**
  - ✔ login.site.com
  - ✔ .site.com

- **disallowed domains**
  - ❌ user.site.com
  - ❌ othersite.com
  - ❌ .com

**login.site.com** can set cookies for all of **.site.com**
but not for another site or TLD

Problematic for sites like .utexas.edu

**path:** anything
SOP for Reading Cookies

Browser sends all cookies in **URL scope**:  
- cookie-domain is domain-suffix of URL-domain  
- cookie-path is prefix of URL-path  
- protocol=HTTPS if cookie is “secure”
Examples of Cookie Reading SOP

cookie 1
  name = **userid**
  value = u1
  domain = **login.site.com**
  path = /
  secure

cookie 2
  name = **userid**
  value = u2
  domain = **.site.com**
  path = /
  non-secure

both set by **login.site.com**

http://checkout.site.com/
cookie: userid=u2

http://login.site.com/
cookie: userid=u2

https://login.site.com/
cookie: userid=u1; userid=u2
(arbitrary order; in FF3 most specific first)
SOP for JavaScript in the Browser

◆ Same scope rules as sending cookies to server
◆ `document.cookie` returns a string with all cookies available for document
  • Based on [scheme], domain, path
  • Often used in JavaScript to customize page
◆ Setting a cookie in Javascript
  – `document.cookie = “name=value; expires=...;”`
  To delete:
  – `document.cookie = “name=; expires= Thu, 01-Jan-70”`
Cookie Protocol Issues

- What does the server know about the cookie sent to it by the browser?
- Server only sees **Cookie: Name=Value**
  - does **not** see cookie attributes (e.g., “secure”)
  - does **not** see which domain set the cookie
  - RFC 2109 (cookie RFC) has an option for including domain, path in Cookie header, but not supported by browsers
Who Set The Cookie?

◆ Alice logs in at login.site.com
  • login.site.com sets session-id cookie for .site.com
◆ Alice visits evil.site.com
  • Overwrites .site.com session-id cookie with session-id of user “badguy” - not a violation of SOP! (why?)
◆ Alice visits cs380s.site.com to submit homework
  • cs380s.site.com thinks it is talking to “badguy”
◆ Problem: cs380s.site.com expects session-id from login.site.com, cannot tell that session-id cookie has been overwritten by a “sibling” domain
Path Separation Is Not Secure

Cookie SOP: path separation

\texttt{x.com/A} does not receive cookies of \texttt{x.com/B}

This is done for efficiency, not security!

DOM SOP: no path separation

\texttt{x.com/A} can read DOM of \texttt{x.com/B}

\texttt{<iframe src="x.com/B"> </iframe>}

\texttt{alert(frames[0].document.cookie);}
“Secure” Cookies Are Not Secure

◆ Alice logs in at https://www.google.com

Set-Cookie: LSID=EXPIRED;Domain=.google.com;Path=/;Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=EXPIRED;Domain=www.google.com;Path=/accounts;Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=EXPIRED;Domain=www.google.com;Path=/;Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=cl:DQAAAhsAAACn3h7GCpKUNxckr79Ce3BUCJtual9a7e5oPvByTr0HUQiFjECYqr5r0q2cH1Cql
Set-Cookie: GAUSR=dabo123@gmail.com;Path=/accounts;Secure

◆ Alice visits http://www.google.com
  • Automatically, due to the phishing filter

◆ Network attacker can inject into response

Set-Cookie: LSID=badguy; secure

and overwrite secure cookie over HTTP

LSID, GAUSR are “secure” cookies
Surf Jacking  ("HTTPS will not save you")

• Victim logs into https://bank.com using HTTPS
  - Non-secure cookie sent back, but protected by HTTPS
• Victim visits http://foo.com in another window
• Network attacker sends “301 Moved Permanently” in response to cleartext request to foo.com
  - Response contains header “Location http://bank.com”
  - Browser thinks foo.com is redirected to bank.com
• Browser starts a new HTTP connection to bank.com, sends cookie in the clear
• Network attacker gets the cookie!
Flash

- **HTTP cookies**: max 4K, can delete from browser
- **Flash cookies / LSO (Local Shared Object)**
  - Up to 100K
  - No expiration date
  - Cannot be deleted by browser user
- **Flash language supports XMLSockets**
  - Can only access high ports in Flash app’s domain
  - Scenario: malicious Flash game, attacker runs a proxy on a high port on the game-hosting site...
  - Consequences?
Frame and iFrame

Window may contain frames from different sources

- Frame: rigid division as part of frameset
- iFrame: floating inline frame

```html
<IFRAME SRC="hello.html" WIDTH=450 HEIGHT=100>
If you can see this, your browser doesn't understand IFRAME.
</IFRAME>
```

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
Mashups
Welcome to your Google homepage. Make it your own.

iGoogle
Cross-Frame Navigation

Frame A can execute a script that manipulates arbitrary DOM elements of Frame B only if Origin(A) = Origin(B)

- Basic same origin policy, where origin is the scheme, domain, and port from which the frame was loaded

How about one frame navigating another?

- Navigate = change where the content in the frame is loaded from
Frame SOP Examples

Suppose the following HTML is hosted at site.com

Disallowed access

```html
<iframe src="http://othersite.com"></iframe>
alert( frames[0].contentDocument.body.innerHTML )
alert( frames[0].src )
```

Allowed access

```html
<img src="http://othersite.com/logo.gif">
alert( images[0].height )
or
frames[0].location.href = "http://mysite.com/
```

Navigating child frame is allowed, but reading frame[0].src is not
Guninski Attack

If bad frame can navigate good frame, attacker gets password!
Gadget Hijacking in Mashups

top.frames[1].location = "http:/www.attacker.com/...";
top.frames[2].location = "http:/www.attacker.com/...";
Gadget Hijacking

Modern browsers only allow a frame to navigate its enclosed frames
Recent Developments

◆ Cross-origin network requests
  • Access-Control-Allow-Origin: <list of domains>
  • Access-Control-Allow-Origin: *

◆ Cross-origin client-side communication
  • Client-side messaging via navigation (older browsers)
  • postMessage (newer browsers)
Library Import

- Same origin policy does not apply to scripts loaded in enclosing frame from arbitrary site

```html
<script type="text/javascript"
src=https://seal.verisign.com/getseal?host_name=A.com>
</script>
```

- This script has privileges of A.com, not source server
  - Can script other pages from A.com origin, load more scripts

- Other forms of importing
SOP Does Not Control Sending

- Same origin policy (SOP) controls access to DOM
- Active content (scripts) can **send** anywhere!
  - No user involvement required
  - Can only read response from same origin
Sending a Cross-Domain GET

Data must be URL encoded

\[
\text{<img src="http://othersite.com/file.cgi?foo=1&bar=x\%20y">}
\]

Browser sends

GET file.cgi?foo=1&bar=x%20y HTTP/1.1 to othersite.com

Can’t send to some restricted ports

- For example, port 25 (SMTP)

Can use GET for denial of service (DoS) attacks

- A popular site can DoS another site [Puppetnets]
Using Images to Send Data

◆ Communicate with other sites

◆ Hide resulting image
  `<img src="..." height="1" width="1">`

Very important point:
a web page can send information to any site!
S. Stamm, Z. Ramzan, M. Jakobsson

Drive-by Pharming

(Symantec report, 2006)
Drive-By Pharming

- User is tricked into visiting a malicious site
- Malicious script detects victim’s address
  - Socket back to malicious host, read socket’s address
- Next step: reprogram the router

[Stamm et al.]
Port Scanning Behind Firewall

- Request images from internal IP addresses
  - Example: `<img src="192.168.0.4:8080"/>

- Use timeout/onError to determine success/failure

- Fingerprint webpages using known image names
Finding the Router

- Script from malicious site can scan home network without violating same origin policy!
  - Pretend to “fetch an image” from an IP address
  - Detect success using `onError`
    
    ```html
    <IMG SRC=192.168.0.1 onError = do()>
    ```

- Determine router type by the image it serves
When response header indicates that page is not an image, the browser stops and notifies JavaScript via the onError handle.
Reprogramming the Router

Fact: 50% of home users use a broadband router with a default or no password

- Log into router
  `<script src="http://admin:password@192.168.0.1"></script>`
- Replace DNS server address with address of attacker-controlled DNS server
Risks of Drive-By Pharming

- Complete Ownership of victim’s Internet cnxn
- Undetectable phishing: user goes to a financial site, attacker’s DNS gives IP of attacker’s site
- Subvert anti-virus updates, etc.

[Stamm et al.]