Cross-site Requests

One Mechanism
Many Attacks

Martin Johns, SAP Research, RUB, 16.06.2010
Me, myself and I

• Senior Researcher at SAP Research Karlsruhe
• PhD on XSS
  – Universities of Hamburg and Passau
  – Worked in various research projects with Siemens, Fraunhofer, SAP, Commerzbank and others
• In a former life webapp developer
  – Wrote a lot of very insecure code
Agenda

- History / technical background
- Cross-site Request Forgeries
- Backchannels
- Further attacks
- Bonus track: DNS rebinding
Agenda

- History / technical background
- Cross-site Request Forgeries
- Backchannels
- Further attacks
- Bonus track: DNS rebinding
At the beginning there was hypertext

• When Tim Berners Lee invented the WWW in 1990 he was primarily interested in hypertext
• His vision: A hypertext-based replacement for document delivery systems, such as Gopher
  – Consequence: There was no need for a session concept
• Essential part of his vision: Blurring the boundaries between different physical servers
Multi origin documents

- Hypertext documents were able to link to images
- Marc Andressen (1993): “Hmmm, wouldn’t it be nifty if these images were displayed inline???”
- Hence, the <img>-tag was born
Then there was JavaScript

• Netscape: “This web thing has to become more interactive. Hey, Brendon Eich, how about putting your LISP dialect into the browser?”
• Brendon: “Hmmm, is LISP still hip?”
• Netscape: “We call it ‘JavaScript’ and hide the functional part behind C-like syntax and nobody will notice”
• The concerned public: “And what about privacy???”
The Same Origin Policy (SOP)

Basically the only security mechanism applicable to webapps
  – Restricts active client-side code

Designed to prevent cross-domain read/write access
  – Applies to JavaScript
  – Affects cookie-access, cross-document interaction and networking communication
  – The SOP is satisfied iff
    • the protocol,
    • the domain and
    • the port
  of two elements match
  – Java and Flash have similar policies
The Same-Origin Policy (II)

So, the SOP provides a nice sandbox:

1. No direct access to the local file system
   (Protocol-rule)
2. No direct access to other hosts
   (Domain-rule)
3. No direct access to other applications on the same host
   (Port- and protocol-rule)

⇒ The SOP restricts JavaScript to “its” web application...
So...

- We have a same-origin policy for authentication and authorization
- ...which is applied at multi-origin documents
- Where could that go wrong?
  - ... remember: Tags, such as `<img>` or `<iframe>` are not restricted by the SOP
- Introducing: Cross-site Request Forgery (CSRF)
  - a.k.a: XSRF, Sea Surf, Session Riding, ...
    - (Microsoft also invented their own name, but I forgot how they called it)
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
• Further attacks
• Bonus track: DNS rebinding
Interlude: Authentication tracking with cookies

– After the authentication form the server sets a cookie at the client’s browser
– The browser sends this cookie along with all requests to the domain of the web application
Implicit authentication

“Implicit” authentication:
– After the initial authentication step the browser takes care of sending the authentication credentials

Methods
– Cookies
– HTTP Authentication
– Client-Side SSL

Alternative: “Explicit” authentication
– The web application is responsible
– Methods
  • URL rewriting
  • Form based tracking (hidden form fields)
www.bank.com

CSRF (I)

Cookie: auth_ok
CSRF (I)

www.bank.com

GET transfer.cgi?am=10000&an=3422421

Cookie: auth_ok

www.attacker.org
CSRF (II)

Cause: The web application does not verify that state changing request were created “within” the web application

• Attack methods:
  – Forging GET requests:
    • Image tag with SRC attribute that points to a state changing URL
    • The URL might be obfuscated by a http redirect
  – Forging POST request:
    • Attacker creates an IFRAME (or a pop-up window)
    • The frame is filled with a HTML form
    • This form is submitted via JavaScript

• Compared to XSS, CSRF’s impact is underestimated
  – In 2006 the following numbers were collected by CWE:
    • XSS: 1282 issues
    • CSRF: 5 issues
  – This is slowly getting better, though
Example 1: Breaking Applications

• Vulnerable: digg.com
  – digg.com’s frontpage is determined by the number of “diggs” a certain story gets
  – Using CSRF a webpage was able to cause the victim’s browser to “digg” an arbitrary URL
  – The demo page “digged” itself
Example 2: Causing Financial Loss

• Vulnerable: Netflix.com
  – Add movies to your rental queue
  – Add a movie to the top of your rental queue
  – Change the name and address on your account
  – Change the email address and password on your account (i.e., takeover your account)
  – Cancel your account (Unconfirmed/Conjectured)
Example 3: Owning the Server

• Vulnerable: Wordpress 2.02
  – Wordpress’ theme editor was susceptible to CSRF
  – Wordpress theme-files can be php-files
  – Via CSRF an attacker could modify those files to contain arbitrary php-code
Login CSRF

• Exploiting CSRF does not necessarily need an pre-existing authentication context
  – Instead, the attacker can force the user to be logged on to an attacker owned account
    • For this purpose, the user’s logout might have to be CSRFed as well

• Attack surface
  – Web applications that track the user’s actions for future reference and
  – in which the specific user context is not too significant
    • Example: Google search history
Login CSRF

[Barth et al. 2008]
Avoiding CSRF

General problem:
- CSRF vulnerabilities are NOT caused by programming mistakes
- Completely correct code can be vulnerable
- The reason for CSRF lies within http:
  - No dedicated authentication credential
  - State-changing GET requests
  - JavaScript

“Preventing CSRF” is actually “fixing the protocol”
Avoidance: Misconceptions

• Only accepting POST requests
  – Defends against local attacks
  – On foreign web pages hidden POST requests can be created with frames

• Referrer checking
  – Some users prohibit referrers
    → referrerless requests have to be accepted
  – Techniques to selectively create http request without referrers exist

• Origin Header
  – Currently in the proposal phase
  – Might be able to solve the mess it adopted widely
Avoidance 1: Switch to explicit authentication

URL rewriting: Session token is included in every URL
  – Attention: Token leakage via proxy-logs/referrers
  – Does not protect against local attacks
    • All URLs produced by the application contain the token

Form based session tokens
  – Session token is communicated via “hidden” form fields
  – Attention: May break the “Back” button

Combination of explicit and implicit mechanisms
  – e.g. Cookies AND URL-rewriting
  – Has to be supported by the used framework
  – SID-leakage may still be a problem
  – Browser bugs are also a problem

Attention: Does not prevent login CSRF
Avoidance 2

Use nonces

– This assures that the POST request’s origin was an HTML form that was provided by the web application in the first place

Example:

```html
<form action="submit.cgi" method="POST">
  <input type="text" name="foo">
  <input type="hidden" name="nonce"
       value="xulkjsf22enbsc">
</form>
```
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
• Further attacks
• Bonus track: DNS rebinding
Backchannels

• CSRF only allows to cause state changing actions on behalf of the victim
• Due to the same-origin policy gaining information is hard for the attacker
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
  – JSON Hijacking
• Further attacks
• Bonus track: DNS rebinding
JSON (and associates)

• Disclaimer: Not everything discussed in the following slides is “JSON” in a strict sense

• However all techniques follow the same patterns:
  – Designed for asynchronous communication
  – Data encapsulated in executable code

• Advantage of this approach:
  – No client-side parser necessary

• Disadvantage:
  – Communicating data through executable code?
    • Come on...
JSON-driven data communication

• Situation: The web application dynamically requests further data from the server without a page refresh...

```javascript
var p = eval("var a =" + JSON_text );
```

• Variant A: XMLHttpRequest:
  – The received data is “evaluated”

• Variant B: <script>-Tag:
  ```html
  <script src="json.php?data=foo"></script>
  ```
  – The received data is automatically executed
    • Via new global variables
    • Via callback function
Frequently used patterns

There are 4 commonly used techniques:

1. Array format (*)
2. Object format (*)
3. Variable setter format
4. Callback function

(*) To be precise, only these two are “officially” JSON (according to RFC 4627)
Data leakage attack

• Loading the data’s URL in a `<script>`-tag
  – The script-tag’s URL is not subject to the SOP
  – If the data is indeed executable, it is executed in the security-context of the attacker’s domain

```html
<script src="http://webapp.com/URL_to_JSON.data"></script>
```
CSRF (I)

www.fancy.com

Cookie: auth_ok
CSRF (I)

www.fancy.com

GET private_stuff_in_json.php

Data

Cookie: auth_ok

www.attacker.org

Data
Data leakage attack

<script src="http://webapp.com/URL_to_JSON.data"></script>

- Exploitation:
  - Array format ⇒ Overwriting of the global Array()-Object
  - Variable setter ⇒ Access the new global variable
  - Callback function ⇒ Provide an own function with the same name
  - Object format ⇒ Redefine the __defineSetter__() function of the global Object()-object
    - modern browsers only
Example: GMail Addressbook disclosure

- Address-book data was communicated in JSON
  
  ```json
  ["ct","YourName","foo@gmail.com"],
  ["ct","AnotherName","bar@gmail.com"]
  ```

- But this URL content could also be accessed via script injection
  
  <script src="http://mail.google.com/mail/?_url_scrubbed_"/>

- By overwriting the global array-constructor this data could be read cross-domain
Defense

• Make sure that your DATA is not EXECUTABLE!!!!!!
• The correct JSON-object notation is actually safe:

```json
{
    "person":{
        "name":"Joe Smith",
        "location":"London",
        "fruit":"Apples"
    }
}
```

• Other techniques:
  – Start data with `while(1) {}`
  – Enclose data in `/* ... */`
  – Use a different format
    • Additional bonus: Something that is not `eval()`-ed is also not so prone to XSS
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
  – Timing
• Further attacks
• Bonus track: DNS rebinding
Backchannels

Back channels:
- Timing
- DOM changes
- Events
- JS (errors)
CSRF privacy attacks

Mechanism:

- Create a cross-site request and observe side effects or measure its completion time within the browser
  - [Felten & Schneider ‘00]
  - Browser cache disclosure through timing
  - [Grossman ’06 / Hansen ‘06]
  - Leaking application state through restricted pictures and JavaScript error console
  - [Bortz et al. ‘07]
  - Application state disclosure
    - Example: Number of items in a shopping cart
  - [Meer & Slaviero ‘07]
    - Refined time measuring through Java’s nanotime()
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
  – Intranet attacks
• Further attacks
• Bonus track: DNS rebinding
IP based authentication

Firewall

Intranet webserver
Subverting the Firewall with CSRF

- Firewall == implicit mean of authentication
  \[\Rightarrow\] Susceptible to CSRF

Diagram:
- Malicious host
- Firewall
- Intranet Server (10.10.10.10)
Circumventing the SOP (II)

The basic reconnaissance attack (BRA)

– Question: Does the element with URL U exist?

Method:

– Construct URL U pointing to the target of the examination
– Start a timeout-event t
– Include a suitting network aware element using U
– Use JavaScript’s eventhandler-framework to determine the result:
  • The timeout t occurs ⇒ The target does not exist
  • onload() event ⇒ The target exists
  • onerror() event ⇒ (specific result depends on the element and target - stay tuned)
Intranet exploration with the BRA

Intranet Server (10.10.10.10)

Malicious host

<img src="http://10.10.10.10">

onload()/onerror()/timeout()-events,

Host / URL / element exists

Firewall
But where to start?

• “My hosts are NATed and use obscure private IPs”
• Java to the rescue:
  – Java applets provide low level sockets
  – The target of these sockets is restricted by the SOP
  – This does not matter as we are interested in the origin of the connection (the local IP)
  – On modern browsers even more convenient with LiveConnect:

```javascript
function natIP() {
  var w = window.location;
  var host = w.host;
  var port = w.port || 80;
  var Socket = (new java.net.Socket(host, port)).
              .getLocalAddress().getHostAddress();
  return Socket;
}
```
Intranet exploration with the BRA (II)

• Ping sweep / http-server discovery:
  – Iterate through the subnet using the BRA
  – E.g. using images:
    <IMG src="http://10.10.10.1">,
    <IMG src="http://10.10.10.2">,
    <IMG src="http://10.10.10.3">,
    ...
  – Events (depend on element type):
    • Timeout-event: Host does not exist
    • OnLoad-event: Host probably runs a webserver
      – IFrame can confirm
    • OnError-event: Host exists but the port is closed (RST package)
      – OnError fires on most browsers only for the IMG element
  – Varying the port might locate https or development servers
Intranet exploration with the BRA (III)

- Server/application discovery/fingerprinting
  - Known “special” DNS names
    - `<IFRAME src="http://fritz.box"> (home router)
  - Known image-URLs
    - `<IMG src="http://10.10.10.10/icons/c.gif"> (Apache)
  - Web page fingerprinting based on JavaScript errors
    - `<SCRIPT src="http://10.10.10.10/index.php">

```html
<script>
function err(msg, url, code) {
    if ((msg == "missing } in XML expression" ) && (code == 1)) {
        // Wordpress
    } else if ((msg == "syntax error" ) && (code == 3)) {
        // Squirrelmail
    } else
        // unknown
}
window.onerror = err;
</script>
```
Intranet exploration with the BRA (IV)

- HTTP-authentication
  - If the scanned server is protected by HTTP-auth the browsers displays a login-dialogue
  - This should at least startle the browser’s user

- Avoiding HTTP-authentication pop-ups (Stefan Esser)
  - The trick is to cause the server to drop the request before it is processed
  - This can be achieved by malformed URLs
  - Incomplete entities:
    - http://host/%
  - Excessively long URLs
    - http://host/AAA ... AAA
  - Breaks fingerprinting
Exploiting the intranet

• The attacker is able to:
  – locate intranet hosts and
  – fingerprint applications/routers/devices

• Several promising points for CSRF attacks:
  – Unchanged default passwords on appliances
    • “Drive by Pharming”
  – Unpatched servers
    • The old and almost forgotten IIS in the basement
  – Outdated intranet applications
    • Wordpress 2.0 for internal communication

• With DNS rebinding things get ever more interesting
  – ...but this is a different talk
Some limitations...

• Timing
  – Working with timeout-events takes... Time
  – Using parallelization can speed the process up
  – But various restrictions on connections limits exit
    • Windows XPSP2 and later

• Port restrictions
  – Most browsers only allow HTTP and high-number ports
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
• **Further attacks**
• Bonus track: DNS rebinding
Cross-application interaction

URL schemes act as application launchers
- e.g., referencing a mailto-URL in an <iframe> launches the default mail application
- Using CSRF techniques to exploit vulnerabilities within third party applications [McFeters and Rios ’07]
- Allows exploit creation which involves more than one application (safari carpet bomb)
  - Side question: Who is responsible to fix such issues?
Targets in the Internet

• Convenient way for the attacker to cover his tracks
  – Scanning potential future victims for botnet recruiting
  – Vulnerability scanning via Google Translate
  – CSRF DDoS
Even more attacks...

– Fingerprinting / attacking non HTTP-protocols via multi-part HTML forms
– Attacks that don’t require JavaScript

Convenient attack tools exist

– E.g., Browser Exploitation Framework (BeEF)
  • One line XSS-payload
– Backframe
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
• Further attacks
• Bonus track: DNS rebinding
DNS spoofing / anti-DNS-pinning / DNS-rebinding

• Originally invented 1996 to subvert Java applets

• General technique
  – The attacker dynamically created DNS entries assigned to local IP addresses
  – This way the SOP can be circumvented
DNS Rebinding

10.10.10.10 = attacker.org

10.10.10.10 = attacker.org

http://attacker.org/index.html

http://attacker.org/foo.html

Contents of the server

200.200.200.200 = attacker.org

attacker.org == attacker.org

⇒ The SOP is satisfied
DNS pinning

Counter Measure
- Keep the DNS binding for the livetime of the browser session
- Breaks, e.g., dynamic DNS, certain load balancing techniques
- Further problem: nowadays our browser sessions are quite long
- Violates RFC 2616

“Anti-DNS-pinning”
- In Firefox, IE and Opera were not fully implemented
  • Unknown if and how it will be fixed
  • Last year Firefox was still vulnerable
- Potential methods to cause the browser to drop the pinning
  • Close the original port on attacker.org
  • Request a resource on a closed port on attacker.org
DNS Rebinding – complex to solve

- Cached scripts and applets
- Proxies
- Heterogenous network paths
- Multiple engines
- ... and ???
Countering DNS rebinding attacks

- Host header:
  - All requests created through JavaScript are within the domain “attacker.org”
    - Dictated by the SOP
  ⇒ Host-header == “attacker.org”
  ⇒ Web content for other virtual hosts is unreachable for JavaScript
Agenda

• History / technical background
• Cross-site Request Forgeries
• Backchannels
• Further attacks
• Bonus track: DNS rebinding
• Conclusion
Conclusion

• The Web is broken
  – Use NoScript

• We need a new browser security model
The End

• Thanks for listening
• Questions?
• CSFire