CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 17th, 2011

Cross-site request forgeries

Request forgeries

Cross-site request forgeries (CSRFs) are one example of the **confused deputy problem.**

You log in to website, the website sets a cookie. You fill out a form to transfer money, then when you click submit it sends a request to bank.com like the following: `bank.com/xfer?frm=…&to=…&amount=…`

Attack: Someone can send this link to another person, who is already logged in and get him to transfer the money by tricking him into clicking the link. The browser will automatically send the correct authentication cookie with request.

Ambient authority: When you the users says "do it" and the software looks around for some indicator of its authority to do the action. The user is not telling the software what authority to use.

Fix:

- User confirmation
  - Problem: The attacker can just copy the request link after the user confirmed it
- HTTP referrer field: the bank.com website can check that the website from which the request is coming is indeed bank.com and something like gmail.com, where the user might have received a malicious link in his email
- Add a token in the URL: `bank.com/xfer?frm=…&to=…&amount=…&token=big_random_number`

User ID-associated token solution

Associate each user ID with a random token, so that one user cannot pass a link that was generated for him to another user. The attacker is now stuck, he would have to guess other user's tokens.

On top of that, tokens would only be generated for users when they are attempting to make a request and deleted after the request has gone through the server. The tokens can be stored on the server-side in a database. This way an attacker would not only have to guess the random token of an user but he would also have to time the attack properly, which makes his job even "more impossible."

Instead of having to store tokens in the database, we can send as a token the output of a MAC:

$$token = MAC_k(form_{id}, user_{id}, cookie)$$

This is set as the token of the transfer link. The server will check that the token corresponds to the user who sent the request. The server will just compute its own MAC on $form_{id}, user_{id}$ and $cookie$ and check that the result equal's the user's token.

The attacker cannot use his token in the forged request since the MAC verification will fail on the server side. The attacker will never be able to compute the MAC for other user IDs since he does not have the secret key k.

How do you make sure you never forgot to apply such a MAC on every generated request link?

- Use Apache's mod_rewrite to append the &token to every URL generated on the page
- Handle all GET requests on your website in one place where you check the token

HTTP referrer solution

The bank can check the referrer field to see that your request is originating from the website. The attacker can't modify the victim's referrer field after he clicked on the link unless he does a MITM attack.

Path traversal bug

Back in the early days, you could do something like this to a webserver:

http://server.com/../../etc/passwd

This would work is because most webservers would work on UNIX, and the `htdocs` **document root** would be in `/var/htdocs`, and the webserver unknowingly allowed the user to refer to a file outside the document root.

`dumbserver.c:`

```
filename = docroot + url;
open(filename);
```

Wrong solutions: If I see a double dot (..), I'll just delete them. Many bugs arise with this approach

Good solutions (which also handle symbolic links):

Get the canonical path of the request file

```
realfilename = realpath(docroot + url);
if(docroot is not a prefix of realfilename )
    error
```

Use chroot jails

Imagine the directory structure is this:

- The `httpd` server daemon's path is: `/var/httpd/usr/bin/httpd`
- The document root is located at: `/var/httpd/docroot`

You can have a launcher program `launcher.c` that does the following to launch the server:

```
chroot("/var/httpd");
chdir("/");
exec("/usr/bin/httpd");
```

One problem that arises with this is the need to copy all frequently used programs into: `/var/httpd/usr/bin/` since after the `httpd` program has been chrooted it cannot access any file outside its new root, such as the system utilities in `/usr/bin` or `/usr/sbin`

Forced browsing, forgetting to check authentication

Forced browsing bugs arise when you depend on **security by obscurity** in a website.

I have a file and I name it **superduperuniquely** and I give you the link to it. I feel safe because I think no one will guess my superduperunique name, but eventually, people figure it out.

Examples: CSE373 and HW solutions.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 17th, 2011
Consider a website that has a collection of administrator pages. Someone logs in and they see "my account" and "history."

The administrator sees a link with "administration" that the users don't see. But the administration page does not check whether the user logged in is an administrator, assuming that if the user got to it, he had to be an administrator, since such a page is not show up for normal users. The website fails to consider that someone could guess the link.

Each administration page needs to check that you're an admin.

In a PHP webpage, people would put: `require "auth.php".` But programmers are sloppy and some pages might lack that require statement

There was a research paper that did a good job at finding forced browsing bugs:

- crawled pages with user not logged in
- crawled pages with user logged in
    - this would isolate the admin page in the previous example
- tried to access pages that were visible only after login, without logging in to see if it works

Mashups

A **mashup** is a webpage that uses a bunch of different web-services to provide new functionality.

Example: housingmaps.com used craigslist.com and Google Maps to plot all the housing offers on a map.

- A simple example because there's no need for authentication

Another **more complicated mashup** would be one that mapped all of your Facebook friends on Google Maps because it would need your Facebook authentication credentials to access all your friends.

Organization one:

- **Server A and server B** talk to **mashup server** which talks to the **browser**

You have to trust the mashup server, so that's something you do not want to do.

Organization two:

- You load the **mashup server** page in your browser and then your browser goes and talks to server A and B.
- The **same origin policy**: mashup server delivers some JavaScript to the browser with the origin set to mashup.com. That JS won't be able to read server A's or B's cookies or send an AJAX request to A or B.