Raluca Ada Popa
Spring 2018

CS 161
Computer Security

Discussion 10

Week of April 9, 2018

Question 1 ***Session Fixation*** **(15 min)**

Some web application frameworks allow cookies to be set by the URL. For example, visiting the URL

`http://foobar.edu/page.html?sessionid=42`.

will result in the server setting the `sessionid` cookie to the value "42".

(a) Can you spot an attack on this scheme?

(b) Suppose the problem you spotted has been fixed as follows. `foobar.edu` now establishes new sessions with session IDs based on a hash of the tuple `(username, time of connection)`. Is this secure? If not, what would be a better approach?

Solution:

(a) The main attack is known as *session fixation*. Say the attacker establishes a session with `foobar.edu`, receives a session ID of 42, and then tricks the victim into visiting `http://foobar.edu/browse.html?sessionid=42` (maybe through an `img` tag). The victim is now browsing `foobar.edu` with the attacker's account. Depending on the application, this could have serious implications. For example, the attacker could trick the victim to pay his bills instead of the victim's (as intended).

Another possibility is for the attacker to fix the session ID and then send the user a link to the log-in page. Depending on how the application is coded, it might so happen that the application allows the user to log-in but reuses the previous (attacker-set) session ID. For example, if the victim types in his username and password at `http://foobar.edu/login.html?sessionid=42`, then the session ID 42 would be bound to his identity. In such a scenario, the attacker could impersonate the victim on the site. This is uncommon nowadays, as most login pages reset the session ID to a new random value instead of reusing an old one.

(b) The proposed fix is not secure since it solves the wrong problem, per the discussion in part (a). Even if it were the right approach, timestamps and user names do not provide enough *entropy*, and could be guessable with a few thousand tries.

The correct fix is for the server to generate cookie values afresh, rather than setting them based on the session ID provided via URL parameters.

Question 2 ***Cross Site Request Forgery (CSRF)*** **(15 min)**

In a CSRF attack, a malicious user is able to take action on behalf of the victim. Consider the following example. Mallory posts the following in a comment on a chat forum:

```
<img src="http://patsy-bank.com/transfer?amt=1000&to=mallory"/>
```

Of course, Patsy-Bank won't let just anyone request a transaction on behalf of any given account name. Users first need to authenticate with a password. However, once a user has authenticated, Patsy-Bank associates their session ID with an authenticated session state.

(a) Sketch out the process that occurs if Alice wants to transfer money to Bob. Explain what happens in Alice's browser and patsy-bank.com's server, as well as what information is communicated and how.

(b) Explain what could happen when Alice visits the chat forum and views Mallory's comment.

(c) What are possible defenses against this attack?

Solution:

(a) Alice fills out the form on patsy-bank.com. When she clicks submit, the information she entered into the form are converted into parameters in an HTTP GET. Alice's browser will then bundle the cookies for patsy-bank.comand send them along with the GET to patsy-bank.com's server. The server will then check the validity of Alice's cookie before processing the request.

(b) The `img` tag embedded in the form causes the browser to make a request to `http://patsy-bank.com/transfer?amt=1000&to=mallory` with Patsy-Bank's cookie. If Alice was previously logged in (and didn't log out), Patsy-Bank might assume Alice is authorizing a transfer of 1000 USD to Mallory.

(c) CSRF is caused by the inability of Patsy-Bank to differentiate between requests from arbitrary untrusted pages and requests from Patsy-Bank form submissions. The best way to fix this today is to use a **token** to bind the requests to the form. For example, if a request to `http://patsy-bank.com/transfer` is normally made from a form at `http://patsy-bank.com/askpermission`, then the form in the latter should include a random token that the server remembers. The form submission to `http://patsy-bank.com/transfer` includes the random token and Patsy-Bank can then compare the token received with the one remembered and allow the transaction to go through only if the comparison succeeds.

It is also possible to check the `Referer` header sent along with any requests. This header contains the URL of the previous, or referring, web page. Patsy-Bank can check whether the URL is `http://patsy-bank.com` and not proceed otherwise. A problem with this method is that not all browsers send the `Referer` header, and even when they do, not all requests include it.

Another problem is that when Patsy-Bank has a so-called "open redirect" `http://patsy-bank.com/redirect?to=`*url*, the referrer for the redirected request will be `http://patsy-bank.com/redirect?to=...`. An attacker can abuse this functionality by causing a victim's browser to fetch a URL like `http://patsy-bank.com/redirect?to=http://patsy-bank.com/transfer`..., and from patsy-bank.com's perspective, it will see a subequent request `http://patsy-bank.com/transfer`... that indeed has a `Referer` from patsy-bank.com.

The modern and more flexible way to protect against CSRF is via the `Origin` header. This works by browsers including an `Origin` header in the requests they send to web servers. The header lists the sites that were involved in the creation of the request. So in the example above, the `Origin` header would include the chat forum in the `Origin` header. Patsy-Bank will then drop the request, since it did not originate from a site trusted by the bank (an instance of *default deny*). This approach is more flexible because unlike the token solution above, you can allow multiple sites to cause the transaction. For example, Patsy-Bank might trust `http://www.trustedcreditcardcompany.com` to directly transfer money from a user's account. This is a use-case that the token-based solution doesn't support cleanly. Currently, many modern browsers support the `Origin` header, but there is still a sizeable chunk of users with browsers that don't support it.

Question 3 ***CSRF++*** **(15 min)**

Patsy-Bank learned about the CSRF flaw on their site described above. They hired a security consultant who helped them fix it by adding a random CSRF token to the sensitive `/transfer` request. A valid request now looks like:

```
https://patsy-bank.com/transfer?to=bob&amount=10&token=<random>
```

The CSRF token is chosen randomly, separately for each user.

Not one to give up easily, Mallory starts looking at the welcome page. She loads the following URL in her browser:

```
https://patsy-bank.com/welcome?name=<script>alert("Jackpot!");</script>
```

When this page loaded, Mallory saw an alert pop up that says "Jackpot!". She smiles, knowing she can now force other bank customers to send her money.

(a) What kind of attack is the welcome page vulnerable to? Provide the name of the category of attack.

(b) Mallory plans to use this vulnerability to bypass the CSRF token defense. She'll replace the `alert("Jackpot!");` with some carefully chosen JavaScript. What should her JavaScript do?

(c) Mallory wants to attack Bob, a customer of Patsy-Bank. Name one way that Mallory could try to get Bob to click on a link she constructed.

Solution:

(a) Reflected XSS

(b) Load a payment form, extract the CSRF token, and then submit a transfer request with that CSRF token.

Or: Load a payment form, extract the CSRF token, and send it to Mallory.

(c) Send him an email with this link (making it look like a link to somewhere interesting). Post the link on a forum he visits. Set up a website that Bob will visit, and have the website open that link in an iframe. Send Bob a text message or a message in Facebook with the link.

(There are many possible answers.)

Question 4 ***Bonus Cookie (optional)*** **(5 min)**

The same origin policy requires that browsers isolate the cookies of URLs with different domains. However, this also means that `https://www.google.com` and `https://mail.google.com` can't share the same cookies. How would you design a system to get around this (without jeopardizing security)?

Solution: The `document.domain` property allows a page to set its own domain, so the above addresses can share the same domain (and thus the cookies).

Cross-origin resource sharing: extend HTTP request/response headers to allow a server to specify origins that are allowed to access a cookie. Read more on

Wikipedia.

A final note: do not hesitate to ask for help! Our office hours exist to help you. Please visit us if you have any questions or doubts about the material.