Week of February 15, 2016

Question 1  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.com and 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 subsequent 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 2  SQL Injection**  (10 min)

(a) Explain the bug in this pseudocode. How would you exploit it?

```python
UID = request_parameters["UID"]
query = "SELECT name FROM users WHERE uid = " + UID;
// Then execute the query.
```

(b) What is the best way to fix this bug?

**Solution:**

(a) The bug is that the uid parameter can be interpreted as a command when properly formatted. For example, to delete the users table, pass in the following as the uid:

```plaintext
0; DROP TABLE users;
```

(b) In this case, a simple fix would be to use a whitelist since uid only needs digits. In essence, you are constraining the type of UID to an integer. Such a whitelisting approach can also work for strings, but is prone to errors. See below for a better solution.

The underlying issue is that data can be interpreted as a command. The solution to this general issue is to separate the parsing of the query from the execution (when the data is supplied). **Prepared statements** (or parameterized queries) offer exactly this. The SQL expression is only parsed once, with placeholders for data. In a second step, the placeholders are replaced with the user input, without changing the intent of the SQL expression. Consider the following example:

```php
query = db->prepare('SELECT name FROM users WHERE uid = :user');
query->execute(array(':user' => UID));
```

The first line defines the SQL expression with a placeholder “:user” that is substituted with user input in the second line. (This placeholder was a “?” instead in the Java example shown in lecture. Same idea.) Note that the substituted input is not parsed as SQL anymore as this already happened in the first line. Therefore an attacker cannot provide bogus SQL commands because they will only be interpreted as data that is bound to the variable :user.

**Question 3  Cookies and Other Food for Thought**  (10 min)

In this question we’ll consider some loopholes that attackers can manipulate.

(a) An iframe can be loaded transparently on top of other elements in a page. Assuming an attacker can get users to visit a malicious site, how can they get them to like their Facebook page? How would you prevent this?

(b) Same origin policy requires that browsers isolate the cookies of URLs with dif-
ferent 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)?

(c) A “Browser in browser” attack involves creating what seems to be a new browser window that actually exists on a malicious page. How could an attacker use this? How would you defend against it?

Solution:

1. You can position the transparent iframe on top of some flashy button saying something like “Click here to claim your $10000” When users click the button, they will actually be clicking the hidden “like” button in the iframe. There are various ways to fix this, but generally you want to make sure your webpage isn’t loaded in the background like that with “framebusting”; for example, site makers used to use “top.location = location”.

2. The `document.domain` property allows a page to set its own domain, so the above addresses can share the same domain (and this 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.

3. An attacker can use this to phish the victim, such as causing the victim to enter his password to his bank account. Defenses against this are open-ended :)

Question 4  
**XSS Defense?**  
(15 min)

Bob the Builder comes up with what he thinks is a great solution to the problem of cross-site scripting. He suggests introducing a new HTML tag, `<NOJAVASCRIPT>`. In between `<NOJAVASCRIPT>` and `</NOJAVASCRIPT>`, JavaScript is disabled: browsers are should not execute any JavaScript between these two tags. Bob the Builder suggests that web developers can use this to avoid cross-site scripting attacks: they should surround every place in their HTML page where they are including untrusted content with a `<NOJAVASCRIPT>` tag. For instance, consider the following vulnerable code:

```javascript
w.write("Hello, " + name + "! Welcome back.\n");
```

Because `name` comes from user input, the above code has an XSS vulnerability. Bob the Builder proposes that instead of writing the above, the web developer should use

```javascript
w.write("Hello, <NOJAVASCRIPT>" + name 
+ "</NOJAVASCRIPT>! Welcome back.\n");
```

Similarly, instead of writing

```javascript
w.write("Today’s most popular link is: " 
+ "<A HREF=" + url + ">
" + url + "</A>\n");
```
which may be vulnerable, since url comes from user input), Bob the Builder proposes the web developer should write

```javascript
w.write("Today’s most popular link is: 
   + "<NOJAVASCRIPT><A HREF="" + url
   + "">" + url + "</A></NOJAVASCRIPT>\n");
```

List at least two problems with Bob the Builder’s proposal.

Solution:

1. If the attacker injects </NOJAVASCRIPT> ... malicious stuff ... <NOJAVASCRIPT>, he can escape the special tags.

2. JavaScript is not the only kind of malicious content. The attacker can inject other kinds of malicious active content, such as Flash, Java, etc. This is a case of incomplete mediation.

3. It might be possible to inject a link tag that executes a CSRF attack against the site itself, if the site does not protect itself from CSRF. Or it might be possible to inject malicious HTML content onto the page without introducing any Javascript: for instance, the attacker might be able to inject a login form that is designed to capture credentials from the user and send them elsewhere, essentially mounting a phishing attack against the user viewing the page.

4. Bob the Builder’s proposal only protects against traditional XSS attacks. It does not protect against DOM-based XSS attacks (also known as client-side XSS), a type of XSS attack that attacks JavaScript that is already running on the page. In particular, if the existing JavaScript uses user input safely (e.g., extracting text from the page and then using it to dynamically add HTML to the page, using `document.write` or `setInnerHtml`), then it might be vulnerable to XSS attacks. These attacks are known as DOM-based XSS, and are not prevented by Bob the Builder’s proposal. Thus, if your page introduces legitimate JavaScript at some other point, your page may still be at risk, depending upon how carefully that JavaScript was written.

5. Depending on how <NOJAVASCRIPT><IFRAME SRC="http://othersite.com/..."> is handled, it might be the case that you could disable JavaScript on the page included in the iframe. This would defeat frame busting and maybe other security mechanisms that the iframe’d website put in place. Thus, this defense against XSS could have the unintended consequence of introducing other security problems unrelated to XSS.

6. There are major deployment issues with this approach. You still have all of the old web pages out there that do not use the special tags, and it does not protect users with legacy browsers.