Week of February 15, 2016

Instructions. We will break into groups to discuss the following questions. Please think of as many solutions as you can. Be original! Maybe you will come up with something no one has thought of yet. Be prepared to talk about your solutions with the rest of the section.

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:

```html
<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?
Question 2  SQL Injection  (10 min)

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

```
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?

---

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 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)?

(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?
Question 4  \textit{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, \texttt{<NOJAVASCRIPT>}. In between \texttt{<NOJAVASCRIPT>} and \texttt{</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 \texttt{<NOJAVASCRIPT>} tag. For instance, consider the following vulnerable code:

\begin{verbatim}
w.write("Hello, " + name + "! Welcome back.\n");
\end{verbatim}

Because \texttt{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

\begin{verbatim}
  w.write("Hello, \texttt{<NOJAVASCRIPT>}" + name
              + "</NOJAVASCRIPT>! Welcome back.\n");
\end{verbatim}

Similarly, instead of writing

\begin{verbatim}
  w.write("Today's most popular link is: 
          + "<A HREF=" + url + ">"><A HREF=" + url + "</A>\n");
\end{verbatim}

(which may be vulnerable, since \texttt{url} comes from user input), Bob the Builder proposes the web developer should write

\begin{verbatim}
  w.write("Today's most popular link is: 
          + "<NOJAVASCRIPT><A HREF=" + url
          + ">"><A>HREF=" + url + "</A></NOJAVASCRIPT>\n");
\end{verbatim}

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