# March 31 & April 1, 2015

**Question 1** ***Cross Site Request Forgery (CSRF)*** **(10 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) Explain what could happen when Victim Vern visits the chat forum and views Mallory's comment.

(b) What are possible defenses against this attack?

**Solution:**

(a) 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 Victim Vern was previously logged in (and didn't log out), Patsy-Bank might assume Vern is authorizing a transfer of 1000 USD to Mallory.

(b) 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.

**Question 2** ***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.

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.

**Question 3** ***Encryption Modes*** **(15 min)**

Consider the following encryption mode for applying AES-128 with a key $K$ to a message $M$ that consists of $l$ 128-bit blocks $M_1$,...,$M_l$. The sender first picks a random 128-bit string, $C_0$, which is the first block of the ciphertext. Then for $i > 0$, the $i^{th}$ ciphertext block is given by $C_i = C_{i-1} \oplus$ AES-128$_K(M_i)$. The ciphertext is the concatenation of these individual blocks: $C = C_0 \parallel C_1 \parallel C_2 \parallel ... \parallel C_l$.

(a) What is the intent behind the random value $C_0$? (I.e., what is it meant to achieve.)

**Solution:** $C_0$ is an *Initialization Vector*. The intent behind it is to ensure that if the same text is encrypted in two distinct messages, the ciphertexts will differ, so an eavesdropper can't infer the relationship between the messages.

(b) Is this mode of encryption secure? If so, state what the desirable properties it has that make it secure. If not, sketch a weakness.

**Solution:** It is not secure. Since the ciphertext is visible to an eavesdropper, the eavesdropper knows $C_i$ for all values of $i$. This allows them to directly determine AES-128$_K(M_i)$ for all $i$ due to the inverse nature of exclusive-or, which makes the scheme equivalent to ECB in terms of revealing whenever two message blocks the same text.

Another valid criticism is that because the scheme uses the Initialization Vector $C_0$ in a *reversible* manner, an attacker can deduce when the two separate ciphertexts in fact encode the same text.

(c) Suppose we replace the computation of $C_i$ with $C_i =$ AES-128$_k(C_{i-1} \oplus M_i)$. Does this make the mode of encryption more secure, less secure, or unchanged? Briefly explain your answer.

**Solution:** This mode is more secure. This alternate form is exactly the definition of CBC mode, which has been proven secure in the face of chosen plaintext attacks.