CS155 Homework 2

Leo Martel SUNet ID: lmartel

May 29, 2015

Worked with Daria Lamberson

Problem 1

(a) P should verify the cert in the usual way: by getting the public key from the CA, then using it to verify the signature inside the cert, as well as checking that the cert is for the company's website and has not expired.

However, using SSL correctly is not sufficient to guard against ssl-strip (as the name implies). We could protect against this attack by coding P to use exclusively $HTTPS$, and refuse to fall back to $HTTP$. ssl-strip is a downgrade attack, so it would no longer work.

(b) The company could issue its own certificate (a self-signed certificate). The only disadvantage to self-signed certs is that browsers warn users about them, but P is not running in a browser so it can choose not to display a self-signed cert warning. We can trust the self-signed cert since we control both the program and the update server.

(c)
- Assuming that the distribution of P is centralized/secure, then both designs are secure. The cert method is secure from (a), and the signed update method is secure because the public key is embedded in P, so it can't be replaced by an attacker, and therefore an attacker can't trick any user into accepting a faulty patch.

  If we're distributing the patch through BitTorrent, then we're probably trying to distribute the workload as much as possible. In the self-signed-cert approach, every user needs to connect to the webserver to get the public key for the cert. In the signed patch approach, each user can download the patch and then verify its integrity locally using the public key embedded in P. Then our webserver only needs to seed the update into BitTorrent; no other interaction is necessary. The signed patch approach, then, scales better bandwidth-wise, so we should use it here.
- Self-signed-cert approach: the server needs to generate and sign a certificate once. Signed patch approach: the server needs to generate a signature for each new patch (i.e. one signature for v1.1, one for v1.2, etc). The cert approach requires fewer crypto operations but neither one is crypto-intensive, and the cert approach requires a lot more bandwidth for distributing the public key.

Problem 2

(a) John can exploit this situation by impersonating Amazon.com. The user's browser will give them a green lock icon, and they'll happily enter their Amazon credentials, sending them to John over SSL with his cert (so he can decrypt the traffic and steal their password).

(b) Symantec must set the "CA bit" on John's cert to false, indicating that they have not authorized him as a sub-CA. The browser must traverse up the certification path from the "amazon" cert to get the cert of the issuer (John's cert). He must be authorized as a CA for the cert he issued for Amazon to validate.

Problem 3

(a) "attacker.com" could embed an image from "sensitiveinfo.com" (which would be loaded with the browser's credentials) in a canvas with origin "attacker.com", then use getImageData to read information from the image and send it to the attacker's server.

(b) GetImageData should only be usable on **images** with the same origin, not anything inside a canvas with the same origin.

(c) If an attacker can serve a canvas on "sensitiveinfo.com", then they can make their canvas take up the whole content area and make it transparent, then use *GetImageData* to see everything the user does on "sensitiveinfo.com".

(d) "GetImageData" should return the color (with an alpha channel) at that location **in the canvas**, not the color at that location on the screen. So in the attack from part (*c*) with a transparent canvas image, *GetImageData* would return $(255, 255, 255, 0)$ – white with opacity 0 – rather than whatever color was underneath.

Problem 4

(a) Requests to "foo.com", even sent from "attacker.com", will automatically include all the browser's cookies for "foo.com". Thus the attacker can send AJAX requests to submit forms or load sensitive data using the user's session.

(b) A form submitted from "attacker.com" will not (and cannot) include the CSRF token, because it's embedded in the "foo.com" HTML (and not stored in the cookie session). The attacker can load "foo.com" in a frame, but then they can't read the DOM; or they can redirect to "foo.com" and obviously lose control of user.

(c) This is secure, because the only way for the attacker to get the right token is to load a page **with the user's session** and read the token out of the HTML. This is impossible, as discussed in class, so the scheme is secuer.

(d) This approach is not secure, because it uses the same token for all users. The attacker's webserver could load "foo.com" with its own session, get the CSRF token, then send it to the "attacker.com" frontend which sends the AJAX request. Assuming this happens reasonably quickly, the token will still be valid (for all users of the website), and the cross-site form submission will succeed.

(e) The same-origin policy dictates that javascript cannot access data on a website with a different origin. The SOP is the reason that the attacker's javascript cannot read the HTML of "foo.com". Without SOP, the attacker could simply load "foo.com" with AJAX and read the token off the page, defeating any kind of DOM-embedded-token CSRF defense.

Problem 5

(a) `steve' OR 1 = 1; --`

This will run the command

```
SELECT * FROM usertable WHERE username = 'steve' OR 1 = 1;
```

and get every user in the database, so the *numrows* of the result will be greater than 0, which triggers the login-success code.

(b) The backslash "escapes" the quote, meaning that the query interprets the entire string

```
steve' OR 1 = 1 --''
```

as a username, instead of interpreting the first part as the username and the second part as a separate piece of the query.

(c) The attacker can take their attack string and replace the quote $0x27$ with $0xbf$ 27. Then when the *addslashes* function runs, it will insert a slash between the two bytes, yielding $0xbf$ $5c$ 27, which will then be interpreted by the database as a chinese character $0xbf$ $5c$ followed by an unescaped quote $0x27$.

Problem 6

(a) N could prevent user R from accessing "foo.com" by spamming out $DOES-NOT-EXIST$ responses, impersonating S, for that domain. This is a denial of service attack.

(b) N could contact S to look up "obviously-doesnt-exist-fmutchiiughssfgdfglxmdkf.com", receiving back a $DOES-NOT-EXIST$ response signed by S. N could then spam this response at R, causing a denial of service, since the signed non-specific DNE is a valid response to any lookup request.

(c) R should verify the signature on the response, defeating the attack from (a).

R should then verify that the queried domain d does in fact fall lexicographically between the previous and next existent names d_p and d_n. This defeats the attack in (b) because the attacker cannot just store one signed DNE response, but needs one with the right range. The attacker needs to either get very very lucky, prefetch and cache every possible ranged DNE response (not feasible if S has many domain records), or read the query from R and then fetch a DNE response to send them before S responds, which is highly unlikely if S processes requests in $FIFO$ order as most webservers do.

(d) The attacker can query for $aaaaaaaaa.suffix.com$, and the *next* field in the $DOES-NOT-EXIST$ response tells them the first existant host with that suffix, $[host1].suffix.com$. The attacker can then query for $[host1]aaaa.suffix.com$, and get back a $DOES-NOT-EXIST$ record with $prev = [host1]$ and $next = [host2]$, repeating the process. The attacker needs only one query per existent host to get all of them.

This information can be useful by leaking secrets. Without $NSEC$ an attacker cannot enumerate every possible host, so not knowing about $NSEC$ a website might put secret information in an internal-only subdomain, e.g. *spyinglogin.nsa.gov*. The attacker would find that a host exists at this domain, revealing the secret.

(e) The resolver simply needs to hash their query, and verify that the hashed value properly falls between the $hashed-prev$ and $hashed-next$ values in the response.