Computer Science 75 — Lecture 11: November 30, 2009
Fall 2009 — Andrew Sellergren
Scribe Notes

Contents

1 Other APIs (0:00–9:00)

- Last week we spoke in depth about the HarvardFood API. We intended this as an exercise in screen-scraping, but we realize that the data is probably not of interest to most of you being that you're not undergrads on Harvard's campus. Still, many of you are affiliated with Harvard, so we thought we'd take a look at the other RESTful APIs that we've created for some Harvard-related data sets.

- Years ago, the standard for interacting with external data sets was to create XML files that described how to interact with the data. Then, using various free tools, source code in language like PHP or Java could be generated to manipulate and extract the data. This was always a somewhat arduous process however.

  To simplify this approach, the REST (representational state transfer) interface was developed. With this interface, the underlying structure of the data doesn't need to be known because it is outputted in a universal format. And there's no need to download software because the data is collected simply by hitting a given URL.

  Take a look at Twitter's API Documentation for example. As you can see, using various GET parameters, you can obtain a wealth of data from Twitter's servers in either JSON or Atom formats, both of which are easily parsed. The "methods" for interacting with Twitter are really just URLs that can be requested.

  In the spirit of Twitter, the Shuttleboy API has "methods" for stops and trips which take various parameters and return data in JSON, PHP, or CSV formats.

2 Security (9:00–103:00)

2.1 Obvious Threats

- Let's begin by taking a look at some obvious threats:
  - Telnet
  - FTP
  - HTTP
  - MySQL

- You can think of Telnet as a less-secure form of SSH. It is still used fairly widely these days, including by Cisco and TiVo, and is generally safe when used to connect two computers on the same subnet. Being that its transmissions are unencrypted, however, what risk does it pose? Well, usernames and passwords are transmitted in the clear. Pretty easy for a malicious user to sniff your network traffic and steal your information, especially with the advent of wifi these days.

- Next on the list is FTP. As you may have guessed, this is a less-secure version of SFTP. Like Telnet, passwords are transmitted in the clear. FTP is still useful in certain circumstances—for example, software distribution—since it allows anonymous logins.

- As for HTTP, you might ask why we weren't using the more-secure alternative HTTPS even for C$75 Finance. Well, the simple answer is that in order to support HTTPS, or more properly SSL, your server must have a unique IP address for your domain. Since we only get 4 unique IP addresses, not 160, with our hosting deal, this wasn't a realistic option.

- If you tried connecting from your own machine to our server using the hostname `cs75.net` as your MySQL connection parameter, you found that it failed. This is because we have a firewall running on the server that blocks all connections on port 3306, the default MySQL port. This prevents the transmission of your credentials in the clear, which the `mysql_connect()` function does by default. There are ways to encrypt this data before sending it over the wire, but we chose to take the stricter precaution of only allowing database connections from the local server.

2.2 suPHP

- If you recall from a previous lecture, the course server is running a tool called suPHP which forces any PHP script to run on Apache as the owner of the PHP file. In other words, if the PHP file is owned by `malan`, then it will execute on the server as user `malan`, rather than the default `apache` or, even worse, `root`. If scripts by default execute as `root` on a server, then all it takes is one mistake by a programmer—like using a system call which would be vulnerable to an injection attack—to allow a malicious user to wreak havoc on the entire server. If instead the script is running as `malan`, then only `malan`'s files will be at risk if an injection attack succeeds.

- What's the downside of every user's scripts running as user `apache`? Well, that means that everyone on the server has the same access. Anyone who has access to the server can possibly take a peek at your files. For example, even if your directories aren't world-readable, the files within them might be. If a malicious user guesses that you have a file called `index.php`, then he might be able to take a peek at it while running a PHP script as user `apache`.

- Question: how do you allow collaboration in a Linux environment? You can either create a new user account which everyone in the group will share or you could create a user group and place all the collaborators in it. Then you need to make sure to `chgrp` all the files to the appropriate group name. Of course, the better option is probably to use some kind of subversion control system.

- To demonstrate suPHP, we might write a simple program like the following:

  ```
  <?php
       echo `/usr/bin/whoami`;
  ?>
  ```

  The backtick quotes cause a local command to be executed. In this case, we're executing `whoami`, which tells us what user the script is executing as. If we create this file as user `cs75`, then we're going to get `cs75` as output for this script. If we change the ownership of the file to a different user, the script will actually not execute at all because we've configured the server to throw an error in the case that a file in a user's directory is owned by someone other than that user.

2.3 Session Hijacking

- Take a look at these two lines from an HTTP response header:

  ```
  Set-Cookie: PHPSESSID=5899f546557421d38d74b659e5bf384f; path=/
  Set-Cookie: secret=12345
  ```

- Recall that the server sends a Set-Cookie header whereas the client will send a Cookie header. Tucked into the Set-Cookie header is the `path` parameter which specifies which part of the domain the cookie is valid for—in this case, it's the whole domain. The long alphanumeric string after `PHPSESSID` is the unique identifier for this user.

- The second line seems to be a rookie mistake. The server is storing the user's password in a cookie, which is insecure on multiple levels. First, this information will be transmitted in the clear, so anyone sniffing network traffic can steal the password. Second, the cookie will be stored on the user's computer, so the password will be readily accessible to anyone who gets control of the user's machine. Obviously, this would be the case for a machine in one of the computer labs here on campus, for example.

  If we still wanted to implement the ability to "remember" users between sessions—i.e. to fill in their usernames even if they're not logged in—then we can instead store the username in the cookie along with a long alphanumeric string that we generate server-side. When the user goes to log in, that long alphanumeric string will be sent to the server and checked against a database. Since it's not the user's actual password, we can set the cookie to expire so that that long alphanumeric string won't always be able to validate a user.

- So what is session hijacking? Given that a long alphanumeric string sent to the server is capable of authenticating a user, it's easy to see that if a malicious user gets a hold of this string, he can impersonate another user. For example, with a packet sniffer, could hijack a session simply by grabbing the session ID number as it's transmitted over the wire in the clear. That includes anyone sitting next to you in a Starbucks.

- How to defend against this? We might use a VPN to establish an encrypted connection with a middle-man server like Harvard, for example. Another option is SSL. With SSL, we can encrypt all of the information in the HTTP headers. This way, the person sitting next to you in Starbucks doesn't have direct access to your session ID number. The problem is that this isn't supported by all servers. In fact, even if sites support SSL for login, they often bounce the user back to non-SSL connections simply because they require less overhead. Bank sites tend to support SSL, of course, because they have a lot more to lose if a user's account is compromised.

  Of course, to fully protect against session hijacking, you could simply disable cookies in your browser. But then you can't use your Facebook account at all. Recall that HTTP is a stateless protocol, which means that it's low-cost in terms of performance, but it poses problems when trying to maintain connections and remember users. Cookies are a way of solving these problems.

  Ultimately, your best defense against session hijacking is to avoid checking your Facebook profile in public places. Though, for the most part, you're at least protected by the fact that the person sitting next to you probably doesn't have the technical knowledge combined with the desire to hijack your session.

- Question: browsers ensure that a given site only has access to its own cookies on a user's machine, not the cookies that were placed there by other sites.

- Cookies also pose a major threat to privacy. Consider Google Analytics, which is a service provided by Google that allows a site to track information about its visitors. It's wonderfully useful to the site itself (we ourselves use it on the course website), but if a large number of sites use it, then Google becomes the one site that has access to almost all of a user's browsing history via the numerous cookies it has placed on a user's machine. People are becoming more and more concerned with this these days.

  One defense against this is to frequently clear your cache, which would prevent Google from knowing your browsing history across large spans of time. Another defense is to disable third-party cookies. This is possible in almost all modern browsers and prevents any site other than the one you're currently visiting from planting cookies on your machine. For the most part, this won't disrupt your browsing experience at all and will help protect your privacy.

Realize that even image files can send cookies along with their headers, so you're not off the hook simply because a site has no JavaScript on it.

- In summary, here are a few technical terms for the attacks we've already mentioned:
  - Physical Access
  - Packet Sniffing
  - Session Fixation
  - XSS

  Physical access to the server, of course, translates to the ability to poke around and yank sensitive information on the server. Packet sniffing, as we've already mentioned, is the threat posed by the person sitting next to you in Starbucks who may be grabbing your session ID number from unencrypted HTTP traffic.[1] Session fixation is a fancy term for the guessing of a valid session ID number. XSS stands for cross-site scripting, which we'll discuss a little later.

- What defenses are there against session hijacking?
  - Hard-to-guess session keys?
  - Rekey session?
  - Check IP address?
  - Encryption?

  Very long session keys ensure that they're both hard to guess and that there won't be many collisions between users on the site. Rekeying sessions should be done with some frequency, although it presents a problem if a user opens multiple browser windows, for instance. Checking a user's IP address is a possibility, but since IP addresses are often shared and subject to change, this isn't perfectly reliable.

- Question: could you use the MAC address of the computer to identify it? Not really, no. HTTP operates several layers above this in the TCP/IP, so websites don't have access to your computer's MAC address.

- Although encryption seems to be the silver bullet in terms of addressing session hijacking, there's (at least) one gotcha we should mention: in order to support SSL, your server must have a unique IP address for your domain. This is because along with all the other data that's being transmitted across the wire, the the Host parameter of the HTTP header would also be encrypted if we were to use it to try to support multiple domains on a single IP address. Without a unique IP address, how will the request be properly directed if the Host parameter is encrypted? In

[1] You can try it for yourself in the comfort of your own home using Wireshark.

order to decrypt it, we need to know where to send it, but in order to know where to send it, we need to decrypt it. So we have a chicken-and-the-egg problem.

Moreover, the world is running out of unique IP addresses since there are only 4 billion total (for IPv4, at least). As IP addresses become scarcer, obtaining a unique IP address will become more costly. In order to use SSL, you need an SSL certificate for each domain name. For subdomains, you can use a cover-all wildcard certificate, but these tend to be quite expensive ($199 per year for the course website, for example).

An SSL certificate, more properly speaking, is simply a *signed public key*. That is, a public key generated for the website has been verified by a third party such as GoDaddy. When a user downloads this public key, his browser will be able to verify that it was signed by a trustworthy entity.

However, SSL certificates provide a layer of security only in theory. Third parties like GoDaddy don't exactly have rigorous standards for verifying the identities of those who wish to purchase SSL certificates. In practice, it's very easy to purchase SSL certificates with fake credentials. So the third-party verification that GoDaddy might provide isn't exactly reliable. And yet, if an SSL certificate hasn't been provided by a well-known third party like GoDaddy or VeriSign, browsers will throw an error when visiting SSL-encrypted sites. What you're paying for, then, is a "trusted" name. You might pay more to a company like VeriSign, but you're not really paying for increased security[2], but rather a more widely known name that is less likely to be rejected by a browser. Although we provide no guarantees, we highly doubt that you'll run into any trouble paying $30 for an SSL certificate as opposed to $1000 or more.

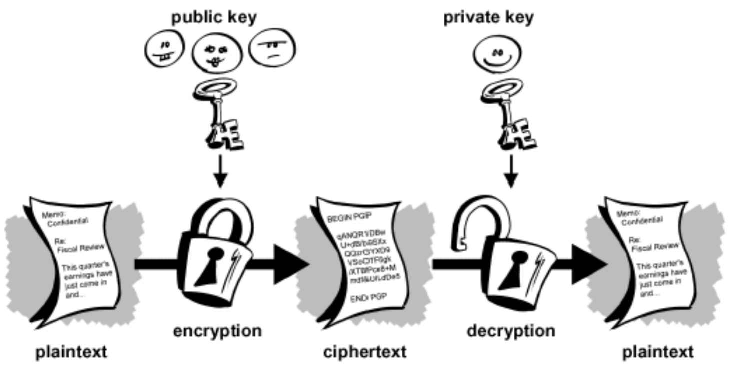
You're probably better off buying a certificate for multiple years if you know you're going to keep your domain around for a while. Otherwise, you'll have to go through the process of generating the certificate and linking to it in your `httpd.conf` file every year. It's quite annoying. Also, if you're going to use as many as 7 subdomains that you'd like to protect with SSL, it probably makes sense to buy a wildcard certificate, which allows for unlimited subdomains and prevents you from having to buy and install a new certificate each time you add a subdomain.

- Question: if you sniff your HTTP traffic using Live HTTP Headers, you're probably going to see the headers sent before they are encrypted with SSL. However, if you use Charles Debugging Proxy, you can see the encrypted traffic, which will just appear as 0's and 1's.

[2]Assuming the number of bits used for encryption is the same across several certificate suppliers, the security they provide will be essentially equivalent.

Computer Science 75 Lecture 11: November 30, 2009
Fall 2009 Andrew Sellergren
Scribe Notes

2.4 Cryptography



- How do we communicate sensitive information between two parties who have never met and therefore can't have agreed upon an encryption scheme ahead of time? In other words, if I want to send my credit card information to Amazon and encrypt it using a password of my choice, how do I securely communicate this password to Amazon? Again, we have a chicken and the egg problem.

- Enter public key and private key pairs. The public key and private key are two very large numbers that have a special mathematical relationship we won't discuss here. Suffice it to say that having only the public key is useless. An adversary could steal this number, but can only use it to *encrypt* data. In order to decrypt the data, he would need the private key.

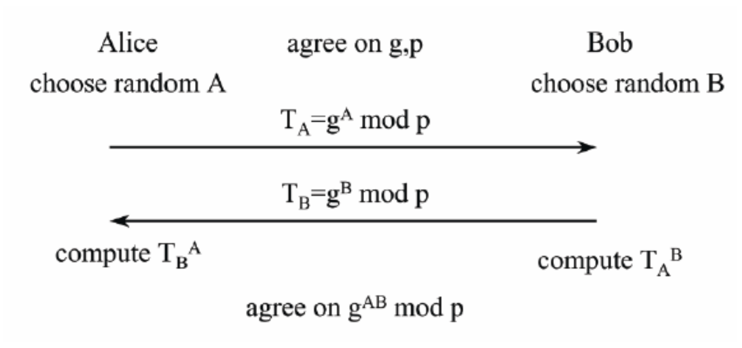
- Now, when I transmit information to Amazon, I encrypt it using Amazon's public key. Amazon, with its private key, is the only other entity (at least in theory) that can decrypt this information. Likewise, when Amazon sends me data, it encrypts it using *my* public key, which enables me to decrypt it using my private key. Whew.

- Generally speaking, RSA encryption is computationally intensive which means it's expensive. What can we do instead? Well, there are many symmetric-key encryption algorithms, including DES and AES, which are much faster. They rely on a single secret key to both encrypt and decrypt data. Now we face the problem of how to transmit that secret key securely between two entities. Well, we can use RSA encryption to transmit this secret key and for all subsequent transactions, use a much faster algorithm like DES or AES to encrypt data. This is precisely what SSL does.

- Unfortunately, these methods are still susceptible to man-in-the-middle attacks. If an adversary is between you and Amazon, he can receive your transmission of the secret key and then respond with his own public key.

Then your browser would establish an encrypted connection, but with the wrong person. In this case, there's nothing wrong with the encryption scheme itself, it's simply that trust has been misplaced. No encryption scheme is perfect!

- Let's take a look at the finer details of public key cryptography using an algorithm called Diffie-Hellman (DLP) which is much less complicated than RSA:



In this example, T_A is effectively Alice's public key and T_B is effectively Bob's public key. g and p are two prime numbers known to both Alice and Bob, g standing for *generator* and usually being 2. Although the number A is known only to Alice and the number B is known only to Bob (think of A and B as Alice and Bob's private keys, respectively), the number $g^{AB} \mod p$, without ever having been sent across the wire, is known to both and thus can be used to encrypt information.

2.5 SQL Injection Attacks

- As a fun exercise, try using a name with an apostrophe in it when logging into your favorite website and see if it breaks. If it does, it's probably because they didn't properly handle single quotes before performing SQL queries.
- Remember the function `mysql_real_escape_string`? Besides being an example of annoying style, it's very useful toward protecting against SQL injection attacks.
- Suppose a malicious user wants to hack into your server and your database. If we pass his input directly to our `mysql_query` function using the `$_POST` array, we've left ourselves vulnerable.
- Let's say our SQL query for looking up a user in our database looks like the following:

```
$result = mysql_query(sprintf("SELECT uid FROM users
                               WHERE username='%s' AND password='%s' ",
                               $_POST["username"], $_POST["password"]));
```

As you can see, we're passing the user's input directly into the query and executing it immediately.

- What if instead of a password, our user types in a SQL-like string such as `12345' OR '1' = '1` at the login page? Since 1 always equals 1, the user will always be assigned a `uid` even if he doesn't have a valid login. Oops!
- The simple fix is to *escape* any user input that you pass to a database. This will add a backslash before any single quotes which will prevent a malicious user from closing the single quotes in our SQL query. You can do this by passing the user input to `mysql_real_escape_string` before inserting it into the query to be executed.

2.6 Same Origin Policy

- Let's return to Project 3 for a moment. Why were we unable to use our JavaScript code to query Google News directly? The same origin policy prevents JavaScript on one server from executing on another.
- Without the same origin policy, a malicious user would be able to launch a denial of service attack on one website using another. For example, a script on Facebook would be able to query Google News. In this way, the hundreds of thousands of requests to Google News would be coming not from a single server but from hundreds of thousands of different users. Thus, the requests would be nearly impossible to block by IP address.
- The SOP applies to the following:
  - Windows
  - Frames
  - Embedded Objects
  - Cookies
  - XMLHttpRequest

  Because SOP applies to windows, scripts embedded in a site can't access the navigation bar. If they could, then a third-party advertiser would be able to know exactly what websites you're visiting.

2.7 Cross-site Attacks

- There are two cross-site attacks which we'll discuss: cross-site request forgery (CSRF/XSRF) and cross-site scripting (XSS). Cross-site request forgery is simply the spoofing of a URL which uses the `GET` method, for example, to cause a user to unwittingly take an action—for example, buying a stock he didn't intend to—simply by clicking a disguised link. Consider the following scenario:

1. You log into project2.domain.tld.
2. You then visit a bad guy's site.
3. Bad guy's site contains a link to `http://project2.domain.tld/buy.php?symbol=INFX.PK`
4. You unwittingly buy the penny stock!

Because you were already logged into the stock-buying website in the same browser window, this GET request succeeds.

- How to defend against CSRF/XSRF? Well, you could simply disable the GET method. Browsers nowadays also warn you when form data is going to be resubmitted. Amazon also defends against this by requiring you to login a second time before completing checkout.

- What about the HTTP Referer header? Well, not only can it be spoofed, but it sometimes is removed entirely by antivirus software and the like.

- Basically, to defend against CSRF/XSRF, you just want to add a small speedbump so that the process isn't entirely automated. Challenging the user with a CAPTCHA is a good example.

- XSS attacks are more real than you might imagine. In your previous projects, you might've implemented some kind of "remember me" feature whereby the value of e-mail stored in `$_POST` would pre-populate the e-mail field like so:

```
<input name="email" type="text" value="<?= $_POST["email"] ?>" />
```

  Now what if a user enters as his e-mail address the following:

```
" /><script> //JS code </script><span style="
```

  Now, if you bounce the user back to the login page and insert this value in the e-mail field, you'll have embedded his JavaScript code in your DOM. He might thereby access your cookies, for example.

- One defense against this would be to disable access to the cookies via JavaScript. Of course, the better defense is to simply escape all user input before spitting it out to the browser window by calling `htmlspecialchars()` for example.