Computer Science 75 Lecture 11: April 27, 2009
Spring 2009 Andrew Sellergren
Scribe Notes

Contents

1 Introduction (0:00–2:00) 2

2 Rudimentary Security (2:00–13:00) 2

3 suPHP (13:00–18:00) 3

4 Session Hijacking (18:00–40:00) 4

5 Cryptography (40:00–55:00) 6

6 SQL Injection Attacks (55:00–63:00) 7

7 Same Origin Policy (63:00–75:00) 8

Computer Science 75
Spring 2009
Scribe Notes

Lecture 11: April 27, 2009
Andrew Sellergren

1 Introduction (0:00–2:00)

- 1 handout this week.
- Bug of the week, brought to you by yours truly! Zipcar sent me an e-mail with the first line "Hi, #first name#." Oops.
- Today's lecture will focus on the gotchas of web development, specifically the ones that allow malicious users to break or manipulate your application.

2 Rudimentary Security (2:00–13:00)

- 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.
- In fact, only recently did Harvard's FAS servers stop supporting Telnet. Most of Harvard's ethernet jacks are shared via hubs which, upon receiving information, spit it out to all connected computers in the hopes that one of them is the correct destination. That means any of the other computers can access information that wasn't intended for them!
- Next on the list is FTP. As you may have guessed, this is a less-secure version of SFTP. Moving forward, if you're using any kind of third-party hosting service like GoDaddy, you should look to use SFTP rather than FTP for transferring files.
- 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.
- What about MySQL? If we're passing in the hostname `localhost`, is there any security risk? Well, not as much as if we were accessing a remote host, but there's still a potential gotcha. If your PHP file containing the username and password for your database were world-readable (`chmod 644`),

then anyone with an account on that server can open them up and yank your database login credentials.

- 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.

3 suPHP (13:00–18:00)

- 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`.
- What does this accomplish? If, for example, the PHP script in question tried to execute the shell command `rm -rf /etc/passwd/`, the script would fail because user `malan` doesn't have permissions to run this shell command.
- 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`.
- In the same vein, files that are uploaded to the server via SFTP might have their owner set as `apache` by default, in which case, others might have access to them.
- What's one way to take care of all these headaches? Well, if your server isn't shared, then you don't have to worry about other users having access via this ownership loophole. Simple as that.
- The `formatter` key is where we can tell YUI to use a function of our own definition, namely `googleURL` and `countryURL`, which make the contents of the column's cells into links to Google and Wikipedia, respectively.
- Below the `columns` definition, we have several other important snippets of code we should examine. First, we're instantiating a new YUI DataSource and telling it that the response we'll be getting will be of type JSON.

Then, we're also telling YUI to queue our Ajax requests if several are made close together. Below that, we have to define the response schema of our DataSource. This may sound a little cryptic, but know that it's an extension of our earlier definition of the `columns` array. And trust that it's all in the YUI documentation somwhere! Finally, we tell YUI that the DataTable will live in the `div` that has `id="results"`. That's it!

- Installing suPHP is as easy as adding a few lines of code to the `http.conf` file which essentially tell Apache to pass all PHP files through suPHP before executing.

4 Session Hijacking (18:00–40:00)

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

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

  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.

- What about the first line? Well, for starters, anyone with access to the server might be able to poke around the `tmp` directory to find the cookies. By default, these files will be stored with the owner being the appropriate user. As we saw once before, however, the *name* of the file is actually the session ID number. With these session ID numbers, a malicious user could hijack a session. Oops.

- Even worse, a malicious user with a packet sniffer could hijack a session simply by grabbing the session ID number as it's transmitted over the wire. That includes anyone sitting next to you in a Starbucks.

- If the session ID numbers are being generated in a predictable fashion, then a malicious user could also spoof a session simply by guessing a legitimate session ID number.

- 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. Because the `HOST` parameter of the HTTP header will also be encrypted, the server must have a unique IP address. Otherwise, how would it know where to send the HTTP request if it can't decrypt the `HOST` parameter?

- There are a few workarounds to this problem. SSL depends on public and private key pairs. If we have a unique one for each domain that's hosted on a server, we could simply try decrypting the HTTP header with all possible key pairs. This gets a little hairy when you have a lot of key pairs, however. A second option would be to have a shared key pair for all domains hosted on the server.

- 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. 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?

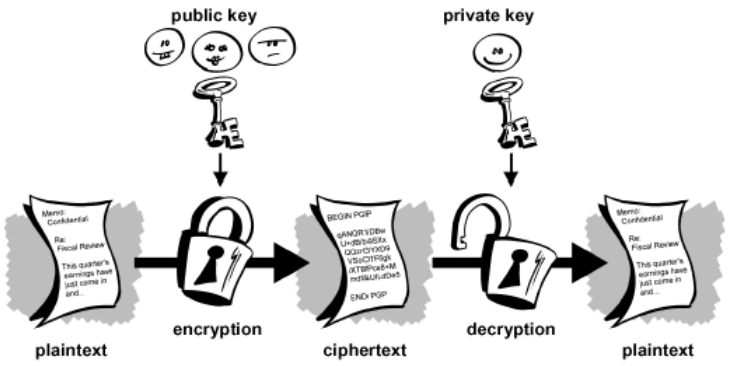
  Very long session keys ensure that they're both hard to guess and that there won't be many collisions between users on the site. Rekeying sessions should be done with some frequency, although it presents a problem if a user opens multiple browser windows, for instance. Checking a user's IP address is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

- In theory, the idea of SSL certificates is good, but generally speaking, it fails in practice. For example, certificates can be revoked, but they rarely are and even if they are, browsers don't usually check for it. Not to mention that SSL has been cracked!

- SSL certificates range in sophistication and price, although they're all basically the same magic act. You can pay extra to have the VeriSign logo appear on your website or even have a logo appear in the address bar, but really, it's not that much more secure.

- Know that it's possible to generate your own SSL certificate for free, but that most browsers will throw an error when a user tries to access your site, saying something to the effect of "We cannot verify the identity of this site." This is because you didn't go through the normal channels of having a recognized third party, such as GoDaddy, vouch for your SSL certificate. Most users will probably click through to your website anyway, but it's slightly annoying.

5 Cryptography (40:00–55:00)



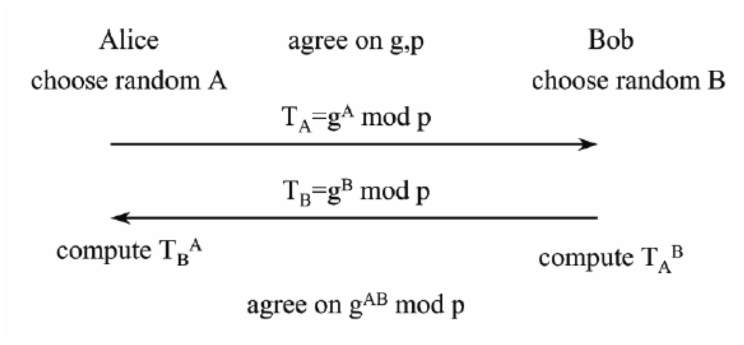
- How do we communicate sensitive information between two parties who have never met and therefore can't have agreed upon an encryption scheme ahead of time? In other words, if I want to send my credit card information to Amazon and encrypt it using a password of my choice, how do I securely communicate 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. Web giants like Facebook and Amazon tend to avoid it on as many pages as possible because more CPU cycles means more money.

- 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.
- 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.
- What's interesting is that the public key and private key pair have a second use in this verification process. Not only can the public key be used to encrypt data which is then decrypted using the private key, the private key can be used to *sign* a website's public key and the signer's public key can be used to verify the signer's identity.
- 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.

6 SQL Injection Attacks (55:00–63:00)

- 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.

7 Same Origin Policy (63:00–75:00)

- 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.
- The SOP is a way of separating DOMs. This is true even of iframes such that the JavaScript for the outer container site can't access or traverse the DOM of the inner frame if that DOM comes from a different domain.
- Browser plugins like Firebug and Greasemonkey are something of an exception because they're really manipulating a local copy of the content rather than the content on the server.
- The SOP applies to the following:
  - Windows

- Frames
- Embedded Objects
- Cookies
- XmlHttpRequest

The related attacks are 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.

- 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.
- One interesting security defense called SafePass has been implemented by Bank of America. When you go to login, it will text you a small passcode that you use to login. This is definitely a step up from the SiteKey, which is highly susceptible to a man-in-the-middle attack.
- XSS attacks are more real than you might imagine. In your previous projects, you've surely implemented some kind of error message system whereby a user's bad input was spit out to the browser window so he could see where he went wrong. What if that bad input was a script tag linking to a malicious script? Now, if you haven't sanitized this input, you'll be embedding into your DOM a malicious script. This script might, for example, send the value of `document.cookie` to the malicious user's server.
- 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 `htmlentities()` for example.
- You can't go wrong escaping user input!