Oct 13: Web security II

Scribe: Jingcheng Liu

Recall from last lecture, there are simple yet popular attacks. Here is a list from OWASP 2013:

1. SQL Injection
2. Hacking session management
3. XSS
4. Insecure direct object reference: this happens with the web application, say an application with a line of code as follows:

```
SELCT * FROM table WHERE name = user_supplied;
```

5. Security misconfiguration
6. Sensitive data exposure
7. Mising proper access control
8. CSRF

1 How to secure web applications?

Unfortunately there is no systematic theory to secure web applications. Existing approaches are based on experience and case-studies. Here are some good practices:

1. use a web framework that has security mechanisms built-in. e.g. Instagram uses Django.
2. protect against top 10 attacks.
3. design web servers using privilege separation (split in different origins), isolate trusted vs. untrusted.
4. set various flags/policies that restrict attackers.
   - e.g. httponly, secure cookies
   - CSP (content security policy)
   - research projects e.g. Mylar, Hails;
5. think hard(er)!

2 (Django) security measures

Next we have a look at some common practices (especially what's adopted by Django) against some of the attacks.

2.1 SQL injection

This happens when a web application constructs a SQL query directly from user input. Here is an example:

```
server: process_request(req)
user=req.GET('username');
sql="SELECT * FROM table WHERE username='"+user+"'";
send results to client;
```

Given such a web application, an example attack would go like:

```
user="'alice' or '1'='1"
```

This would traverse the whole table. If one replace the predicate with some other test predicate, one could use the standard self-reduction of search problems to decision problem, and simulate searching the whole database just by testing the predicate.

Solution: escape user input, Django abstracts construction of a SQL query, not allowing direct manipulation of SQL.

3 Email header injection

Here is a sample web feedback form, where the user types in the subject and body, hit send and the web application constructs an email that sends email to admin@web.com.

```
subject:
body:
[send] sends email to admin@...
```

The attacker can then inject END-OF-LINE character to insert or overwrite fields in the email header.

```
subject: hi \n cc: ...
body: ....
[send] sends email to admin@...
```

Solution: escape user input!

4 XSS (cross-site scripting)

Here is a typical scenario. An attacker submits a script in a comment to a blog post, which renders into an executable script to other user viewing the blog post.

Solution:

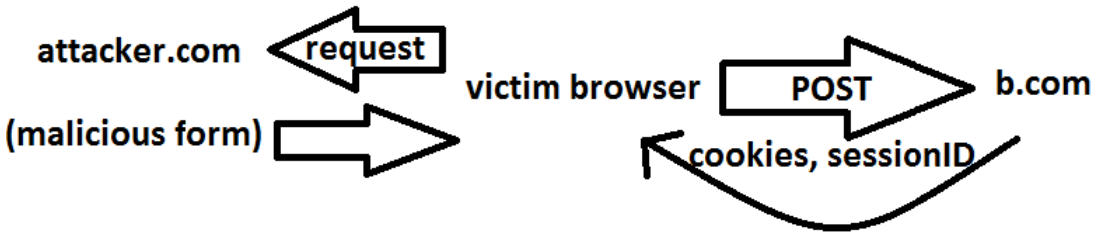
- escape user input, Django would render: `< script &gt`
- disallow `http://foo.com? q=<script src=.../>`
- Privilege separation(e.g. multiple origins for web server)
  - googleusercontent.com vs. google.com
  - script, apply same-origin policy to protect script from accessing sensitive data
- content security policy: default-src 'self': *.mydomain.com, also specify source of scripts, frames, images.

5 CSRF: cross-site request forgery

Here is an artificial BOA transfer form.

```
<form action="bankofamerica.com/transfer" method=POST>
<input name=recipient value="alice">
<input name=amount value=$100>
</form>
<script> form.submit()</script>
```

The attacker would attempt to forge a user's request as follows:




```
<form action...>
<input ... value='attacker'>
<input ... value=100k>
</form>
<script> form.submit()</script>
```

CSRF fix: each form embeds a random id, server checks ID upon form post.

As an aside, for backward compatibility, HTTP referer checking could be bypassed by opening a new tab and execute the html there.

6 Session forging/hijacking

Here is a typical session management scheme through cookies:

user browser: cookie (contains session ID); Server: session ID, user, time-duration.

This is not a good practice as the user could forge the cookie, and attackers could steal the cookie.

Here is another one that puts the session id in URL: `http://example.com/?PHPSSEID=fa2921....`

This is not a good idea either. As a user may share the URL directly to his/her friends or in the public. Don't put session id in URL.

Stateless cookies: sign `{cookie: userID, info about user, session ID}` with a MAC, or encrypt it. The caveat is that revocation is hard; it cannot be remotely revoked.

7 Directory traversal

Say a web application takes user input filename and outputs its content.

```
def dump_file(request):
filename = request.GET("filename")
filename=os.path.join(base_path, filename)
open(filename).read()
```

A malicious user may try: `filename="../../../etc/passwd"` or to some setting files.

Solution: sanitize the input, check directory, access control etc.