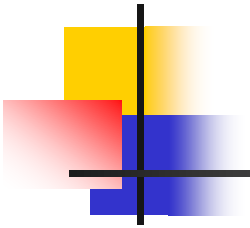
# Session Management Testing

Instructor: C. Pu (Ph.D., Assistant Professor)

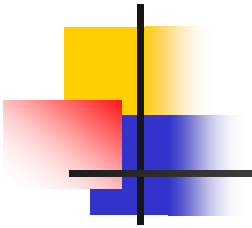
Lecture 07

*puc@marshall.edu*



# Introduction

- One of the core components of any web-based application is the mechanism by which it controls and maintains the state for a user interacting with it.
- This is referred to as *Session Management* and is defined as the set of all controls governing state-full interaction between a user and the web-based application.
- This broadly covers anything from how user authentication is performed, to what happens upon them logging out.



# Introduction

- HTTP is a *stateless* protocol, meaning that web servers respond to client requests without linking them to each other.
- Even simple application logic requires a user's multiple requests to be associated with each other across a "session".
- Most popular web application environments, such as ASP and PHP, provide developers with built-in session handling routines.
- Some kind of identification token will typically be issued, which will be referred to as a "Session ID" or Cookie.

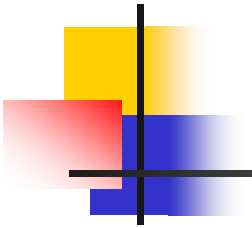
# Testing for Session Management Schema

- In order to avoid continuous authentication for each page of a website or service, web applications implement various mechanisms to store and validate credentials for a pre-determined timespan.
- These mechanisms are known as Session Management and while they are important in order to increase the ease of use and user-friendliness of the application, they can be exploited by a penetration tester to gain access to a user account, without the need to provide correct credentials.
- In this test, the tester wants to check that cookies and other session tokens are created in a secure and unpredictable way.
- An attacker who is able to predict and forge a weak cookie can easily hijack the sessions of legitimate users.

# Testing for Session Management Schema

- In a nutshell, when a user accesses an application which needs to keep track of the actions and identity of that user across multiple requests, a cookie (or cookies) is generated by the server and sent to the client.
- The client will then send the cookie back to the server in all following connections until the *cookie expires* or is destroyed.
- The data stored in the cookie can provide to the server a large spectrum of information about who the user is, what actions he has performed so far, what his preferences are, etc. therefore providing a state to a stateless protocol like HTTP.

# Testing for Session Management Schema



- A typical example is provided by an online shopping cart.
- Throughout the session of a user, the application must keep track of his identity, his profile, the products that he has chosen to buy, the quantity, the individual prices, the discounts, etc.
- Cookies are an efficient way to store and pass this information back and forth (other methods are URL parameters and hidden fields).

# Testing for Session Management Schema

- Due to the importance of the data that they store, cookies are therefore vital in the overall security of the application.
- Being able to tamper with cookies may result in hijacking the sessions of legitimate users, gaining higher privileges in an active session, and in general influencing the operations of the application in an unauthorized way.

# Testing for Session Management Schema

- In this test, the tester has to check whether the cookies issued to clients can resist a wide range of attacks aimed to interfere with the sessions of legitimate users and with the application itself.

- The overall goal is to be able to forge a cookie that will be considered valid by the application and that will provide some kind of unauthorized access (session hijacking, privilege escalation, ...).

# Testing for Session Management Schema

- Usually the main steps of the attack pattern are the following:
  - cookie collection:
    - collection of a sufficient number of cookie samples;
  - cookie reverse engineering:
    - analysis of the cookie generation algorithm;
  - cookie manipulation:
    - forging of a valid cookie in order to perform the attack.
    - this last step might require a large number of attempts, depending on how the cookie is created (cookie brute-force attack).

# Testing for Session Management Schema: How to Test

- All interaction between the client and application should be tested at least against the following criteria:
  - Are all Set-Cookie directives tagged as Secure?
  - Do any Cookie operations take place over unencrypted transport?
  - Are any Cookies persistent?
  - What Expires= times are used on persistent cookies, and are they reasonable?
  - What HTTP/1.1 Cache-Control settings are used to protect Cookies?

# Testing for Session Fixation

- When an application does not renew its session cookie(s) after a successful user authentication, it could be possible to find a session fixation vulnerability and force a user to utilize a cookie known by the attacker.
- In that case, an attacker could steal the user session (session hijacking).
- Session fixation vulnerabilities occur when:
  - A web application authenticates a user without first *invalidating* the existing session ID, thereby continuing to use the session ID already associated with the user.
  - An attacker is able to forge a known session ID on a user so that, once the user authenticates, the attacker has access to the authenticated session.

# Testing for Session Fixation

- In the generic exploit of session fixation vulnerabilities, an attacker creates a new session on a web application and records the associated session identifier.
- The attacker then causes the victim to authenticate against the server using the same session identifier, giving the attacker access to the user's account through the active session.
- Furthermore, the issue described above is problematic for sites that issue a session identifier over HTTP and then redirect the user to a HTTPS log in form.
- If the session identifier is not reissued upon authentication, the attacker can eavesdrop and steal the identifier and then use it to hijack the session.

# Testing for Session Fixation: How to Test

- Testing for Session Fixation vulnerabilities:
  - The first step is to make a request to the site to be tested (example www.example.com). If the tester requests the following:

```
GET www.example.com
```

  - They will obtain the following answer:

```
HTTP/1.1 200 OK
Date: Wed, 14 Aug 2008 08:45:11 GMT
Server: IBM_HTTP_Server
Set-Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-
rt4:-1; Path=/; secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=Cp1254
Content-Language: en-US
```

# Testing for Session Fixation: How to Test

- Testing for Session Fixation vulnerabilities:
  - The application sets a new session identifier JSESSIONID=0000d-8eyYq3L0z2fgq10m4v-rt4:-1 for the client.

```
HTTP/1.1 200 OK
Date: Wed, 14 Aug 2008 08:45:11 GMT
Server: IBM_HTTP_Server
Set-Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-
rt4:-1; Path=/; secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=Cp1254
Content-Language: en-US
```

# Testing for Session Fixation: How to Test

- Testing for Session Fixation vulnerabilities:
  - Next, if the tester successfully authenticates to the application with the following POST HTTPS:

```
POST https://www.example.com/authentication.php HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it;
rv:1.8.1.16) Gecko/20080702 Firefox/2.0.0.16
Accept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com
Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1
Content-Type: application/x-www-form-urlencoded
Content-length: 57

Name=Meucci&wpPassword=secret!&wpLoginattempt=Log+in
```

# Testing for Session Fixation: How to Test

- Testing for Session Fixation vulnerabilities:
  - The tester observes the following response from the server:
    - Any new cookie?

```
HTTP/1.1 200 OK
Date: Thu, 14 Aug 2008 14:52:58 GMT
Server: Apache/2.2.2 (Fedora)
X-Powered-By: PHP/5.1.6
Content-language: en
Cache-Control: private, must-revalidate, max-age=0
X-Content-Encoding: gzip
Content-length: 4090
Connection: close
Content-Type: text/html; charset=UTF-8
...
HTML data
...
```

# Testing for Session Fixation: How to Test

- Testing for Session Fixation vulnerabilities:
  - As no new cookie has been issued upon a successful authentication the tester knows that it is possible to perform session hijacking.

# Testing for Cross-Site Request Forgery

- CSRF is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated.
- With a little help of social engineering (like sending a link via email or chat), an attacker may force the users of a web application to execute actions of the attacker's choosing.
- A successful CSRF exploit can compromise end user data and operation, when it targets a normal user.
- If the targeted end user is the administrator account, a CSRF attack can compromise the entire web application.

# Testing for CSRF

- CSRF relies on the following:
  - [1] Web browser behavior regarding the handling of session-related information such as cookies and http authentication information;
  - [2] Knowledge by the attacker of valid web application URLs;
  - [3] Application session management relying only on information which is known by the browser;
  - [4] Existence of HTML tags whose presence cause immediate access to an http[s] resource; for example the image tag img.

# Testing for CSRF

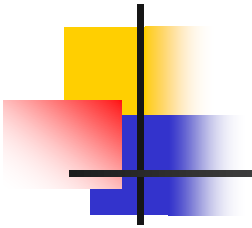
- Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.
  - Point 1)
    - Browsers automatically send information which is used to identify a user session.
    - Suppose a site hosting a web application, and the user victim has just authenticated himself to site.
    - In response, site sends victim a cookie which identifies requests sent by victim as belonging to victim's authenticated session.
    - Basically, once the browser receives the cookie set by site, it will automatically send it along with any further requests directed to site.

# Testing for CSRF

- Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.
  - Point 2)
    - If the application does not make use of session-related information in URLs, then it means that the application URLs, their parameters, and legitimate values may be identified (either by code analysis or by accessing the application and taking note of forms and URLs embedded in the HTML/JavaScript).

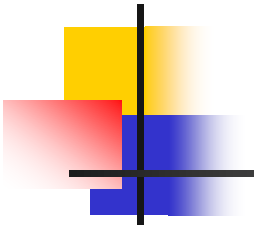
# Testing for CSRF

- Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.
  - Point 3)
    - ”Known by the browser” refers to information such as cookies, or http-based authentication information (such as Basic Authentication; and not form-based authentication), which are stored by the browser and subsequently resent at each request directed towards an application area requesting that authentication.
    - The vulnerabilities discussed next apply to applications which rely entirely on this kind of information to identify a user session.



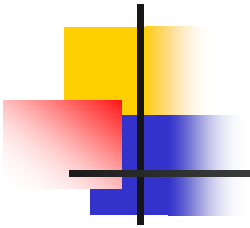
# Testing for CSRF

- Suppose, for simplicity's sake, to refer to GET-accessible URLs (though the discussion applies as well to POST requests).
- If victim has already authenticated himself, submitting another request causes the cookie to be automatically sent with it.



# Testing for CSRF

- The GET request could be originated in several different ways:
  - by the user, who is using the actual web application;
  - by the user, who types the URL directly in the browser;
  - by the user, who follows a link (external to the application) pointing to the URL.



# Testing for CSRF

- These invocations are indistinguishable by the application.
- In particular, the third may be quite dangerous.
- There are a number of techniques (and of vulnerabilities) which can disguise the real properties of a link.
- The link can be embedded in an email message, or appear in a malicious web site where the user is lured, i.e., the link appears in content hosted elsewhere (another web site, an HTML email message, etc.) and points to a resource of the application.

# Testing for CSRF

- If the user clicks on the link, since it was already authenticated by the web application on site, the browser will issue a GET request to the web application, accompanied by authentication information (the session id cookie).
- This results in a valid operation performed on the web application and probably not what the user expects to happen.
- Think of a malicious link causing a fund transfer on a web banking application to appreciate the implications.

# Testing for CSRF

- By using a tag such as img, as specified in point 4 above, it is not even necessary that the user follows a particular link.
- Suppose the attacker sends the user an email inducing him to visit an URL referring to a page containing the following (oversimplified) HTML:

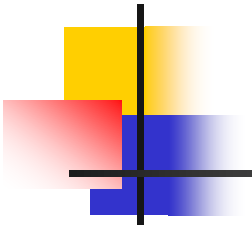
```
<html><body>

...

<img src="https://www.company.example/action" width="0"
height="0">

...

</body></html>
```



# Testing for CSRF

- The problem here is a consequence of the following facts:
  - there are HTML tags whose appearance in a page result in automatic http request execution (img being one of those);
  - the browser has no way to tell that the resource referenced by img is not actually an image and is in fact not legitimate;

# Testing for CSRF

- It is the fact that HTML content unrelated to the web application may refer components in the application, and the fact that the browser automatically composes a valid request towards the application, that allows such kind of attacks.
- As no standards are defined right now, there is no way to prohibit this behavior unless it is made impossible for the attacker to specify valid application URLs.
- This means that valid URLs must contain information related to the user session, which is supposedly not known to the attacker and therefore make the identification of such URLs impossible.

# Testing for CSRF

- The problem might be even worse, since in integrated mail/browser environments simply displaying an email message containing the image would result in the execution of the request to the web application with the associated browser cookie.
- Things may be obfuscated further, by referencing seemingly valid image URLs such as

```
<img src="https://[attacker]/picture.gif" width="0"
height="0">
```

- where [attacker] is a site controlled by the attacker, and by utilizing a redirect mechanism on

```
http://[attacker]/picture.gif to http://[thirdparty]/action.
```

# Testing for CSRF: How to Test

- For a black box test, the tester must know URLs in the restricted (authenticated) area.
- If they possess valid credentials, they can assume both roles – the attacker and the victim.
- In this case, testers know the URLs to be tested just by browsing around the application.
- Otherwise, if testers don't have valid credentials available, they have to organize a real attack, and so induce a legitimate, logged in user into following an appropriate link.
- This may involve a substantial level of social engineering.

# Testing for CSRF: How to Test

- Either way, a test case can be constructed as follows:
  - let u be the URL being tested; for example, u = http://www.example.com/action
  - build an html page containing the http request referencing URL u (specifying all relevant parameters; in the case of http GET this is straightforward, while to a POST request you need to resort to some Javascript);
  - make sure that the valid user is logged on the application;
  - induce him into following the link pointing to the URL to be tested (social engineering involved if you cannot impersonate the user yourself);
  - observe the result, i.e. check if the web server executed the request.