

# Penetration Testing

## Lecture 09

Instructor: Dr. Cong Pu, Ph.D.

[cong.pu@okstate.edu](mailto:cong.pu@okstate.edu)

*Acknowledgment: Adapted partially from course materials from Dr. Wenliang Du at Syracuse University, Dr. Fengwei Zhang at Southern University of Science and Technology, and Dr. Steven M. Bellovin at Columbia University.*

# Review Webpage Comments and Metadata for Information Leakage: Summary

- Programmers including detailed comments and metadata on their source code
  - very common, and even recommended
- Comments and metadata included into the HTML code
  - reveal internal information
    - should not be available to potential attackers
- Comments and metadata review of HTML code
  - determine if any sensitive information is being **leaked**

# Review Webpage Comments and Metadata for Information Leakage: Test Objectives



---

- Review webpage comments and metadata to better
  - understand the application
  - find any information leakage

# Review Webpage Comments and Metadata for Information Leakage: How to Test

- Developers using HTML comments
  - include debugging information about the application
  - sometimes, they forget about the comments
    - *leave them on in production*
- Testers should look for HTML comments and metadatas which start with
  - `<!-- -->`
  - `<meta >`

# Review Webpage Comments and Metadata for Information Leakage: Example

- Check HTML source code for comments containing sensitive information that can help the attacker gain more insight about the application
  - e.g., *SQL code, usernames and passwords, internal IP addresses, or debugging information*

```
...
<div class="table2">
  <div class="col1">1</div><div class="col2">Mary</div>
  <div class="col1">2</div><div class="col2">Peter</div>
  <div class="col1">3</div><div class="col2">Joe</div>

<!-- Query: SELECT id, name FROM app.users WHERE active='1'
-->

</div>
...
```

```
<!-- Use the DB administrator password for testing: f@keP@
a$$w0rD -->
```

```
<META name="Author" content="Andrew Muller">
```



# Testing for Credentials Transported over an Encrypted Channel: Summary

---

- Testing for credentials transport
  - verifying that the user's authentication data are transferred via an encrypted channel
    - avoid being intercepted by malicious users
  - objective: to *understand*
    - if the data travels unencrypted from the web browser to the server  
or
    - if the web application takes the appropriate security measures using a protocol like HTTPS
      - the HTTPS protocol is built on TLS/SSL to *encrypt the data*



# Testing for Credentials Transported over an Encrypted Channel: Summary

---

- Clearly, the fact that traffic is encrypted does not necessarily mean that it's completely safe
  - the encryption algorithm used
  - the robustness of the keys that the application is using



# Testing for Credentials Transported over an Encrypted Channel: Summary

---

- The most common example of this issue
  - the login page of a web application
- The tester should verify
  - *user's credentials are transmitted via an encrypted channel*
- To login to a web site
  - the user usually has to fill a simple form
    - transmits the inserted data to the web application with the POST method
  - what is less obvious:
    - this data can be passed using the HTTP protocol, which transmits the data in a non-secure, clear text form, or
    - using the HTTPS protocol, which encrypts the data during the transmission





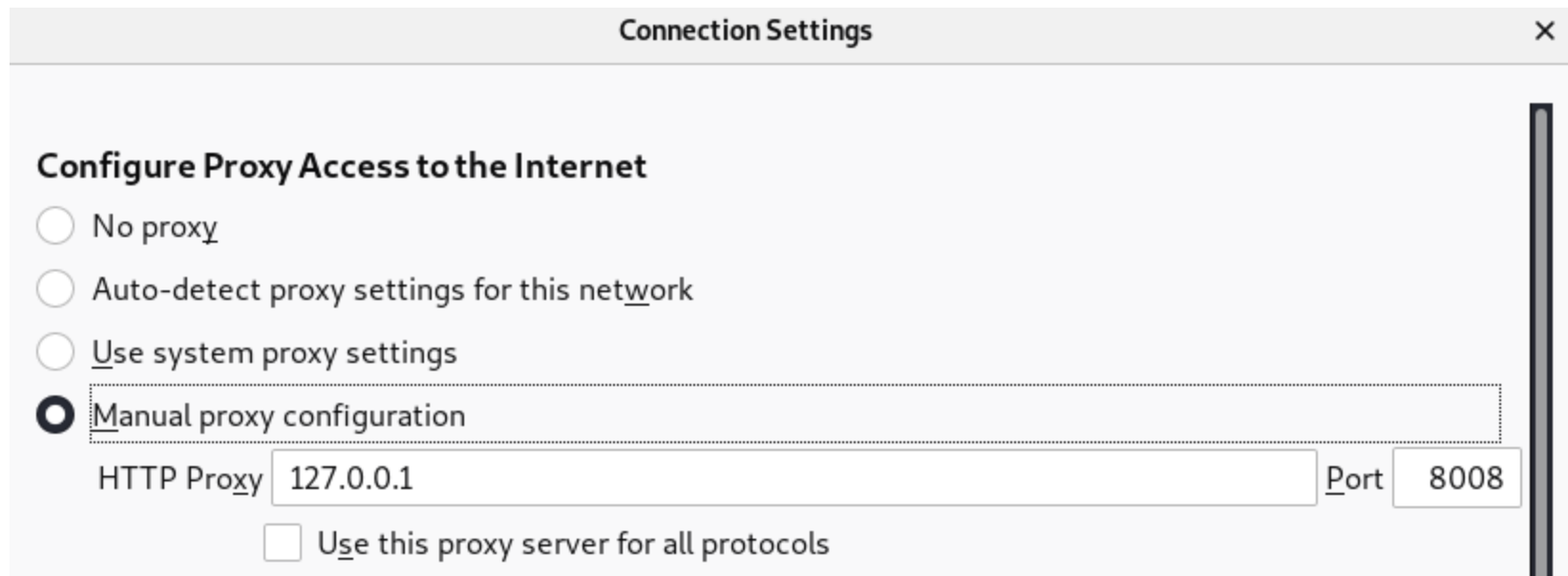
# Testing for Credentials Transported over an Encrypted Channel: Summary

---

- This test is to ensure that an attacker cannot retrieve sensitive information by simply sniffing the network with a sniffer tool

# Testing for Credentials Transported over an Encrypted Channel: How to Test

- WebScarab: capture packet headers and to inspect them. You can use any web proxy that you prefer.
  - Configure proxy



The screenshot shows the 'Connection Settings' dialog box with the following configuration:

- Configure Proxy Access to the Internet**
  - ☐ No proxy
  - ☐ Auto-detect proxy settings for this network
  - ☐ Use system proxy settings
  - ☒ Manual proxy configuration
- HTTP Proxy: 127.0.0.1
- Port: 8008
- ☐ Use this proxy server for all protocols

# Testing for Credentials Transported over an Encrypted Channel: How to Test

- WebScarab: capture packet headers and to inspect them. You can use any web proxy that you prefer.
- sending data with POST method through **HTTP**
  - suppose that the login page presents a form with fields User, Password, and the Submit button to authenticate and give access to the application.
  - if we look at the headers of our request with WebScarab, we can get something like this:

```
Parsed Raw
POST http://testfire.net:80/doLogin HTTP/1.1
Host: testfire.net
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://testfire.net/login.jsp
Content-Type: application/x-www-form-urlencoded
Content-length: 37
Connection: keep-alive
Cookie: JSESSIONID=E04185FD9A77FE435AD9CF4D8E9CD0EF; AltoroAccounts=ODAwMDAwfkNvcnBvcnF0ZX41LjJzNzMyMjI2MUU3fDgwMDAwMX5DaGVja2luZ34xMDQ3ODEuNDR8
Upgrade-Insecure-Requests: 1
uid=admin&passw=admin&btnSubmit=Login
```



- POST http://ocsp.pki.goog:80/gts1o1 HTTP/1.1  
Host: ocsp.pki.goog  
User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:68.0) Gecko/20100101 Firefox/68.0  
Accept: \*/\*  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Content-Type: application/ocsp-request  
Content-length: 83  
Connection: keep-alive
- 0Q0O0M0K0I0 [obscured]



# Testing for Insecure Direct Object References: Summary

---

- Insecure Direct Object References:
  - application provides direct access to objects based on user input
  - consequences:
    - attackers can bypass authorization
      - access resources in the system directly, e.g., database records or files
      - access resources directly by modifying the value of a parameter used to directly point to an object
        - such resources can be database entries belonging to other users, files in the system, and more
  - why: application takes user input and uses it to retrieve an object without performing sufficient authorization checks



# Testing for Insecure Direct Object References: How to Test

---

- To test for this vulnerability
  1. map out all locations in the application where user input is used to reference objects directly
    - for example, locations where user input is used to access a database row, a file, application pages and more
  2. modify the value of the parameter used to reference objects and assess
    - whether it is possible to retrieve objects belonging to other users or otherwise bypass authorization



# Testing for Insecure Direct Object References: How to Test

---

- The value of a parameter is used directly to retrieve a database record
  - sample request:

`http://foo.bar/somepage?invoice=12345`

- in this case, the value of the invoice parameter is used as an index in an invoices table in the database.
- the application takes the value of this parameter and uses it in a query to the database.
- the application then returns the invoice information to the user.
- since the value of invoice goes directly into the query, by modifying the value of the parameter it is possible to retrieve any invoice object, regardless of the user to whom the invoice belongs.



# Testing for Insecure Direct Object References: How to Test

---

- The value of a parameter is used directly to perform an operation in the system

- sample request:

```
http://foo.bar/changepassword?user=someuser
```

- in this case, the value of the user parameter is used to tell the application for which user it should change the password.
    - in many cases this step will be a part of a wizard, or a multi-step operation.
    - in the first step the application will get a request stating for which user's password is to be changed, and in the next step the user will provide a new password (without asking for the current one).





# Testing for Insecure Direct Object References: How to Test

---

- The value of a parameter is used directly to perform an operation in the system
  - sample request:

```
http://foo.bar/changepassword?user=someuser
```

- the user parameter is used to directly reference the object of the user for whom the password change operation will be performed.
- to test for this case the tester should attempt to provide a different test username than the one currently logged in, and check whether it is possible to modify the password of another user.

# Testing for Insecure Direct Object References: How to Test

- The value of a parameter is used directly to retrieve a file system resource
  - sample request:

```
http://foo.bar/showImage?img=img00011
```

- in this case, the value of the file parameter is used to tell the application what file the user intends to retrieve.
- by providing the name or identifier of a different file (for example file=image00012.jpg) the attacker will be able to retrieve objects belonging to other users.



# Testing for Insecure Direct Object References: How to Test

- The value of a parameter is used directly to access application functionality
  - sample request:  
`http://foo.bar/accessPage?menuitem=12`
  - in this case, the value of the menuitem parameter is used to tell the application which menu item (and therefore which application functionality) the user is attempting to access.
  - assume the user is supposed to be restricted and therefore has links available only to access to menu items 1, 2 and 3. By modifying the value of menuitem parameter it is possible to bypass authorization and access additional application functionality.
  - to test for this case the tester identifies a location where application functionality is determined by reference to a menu item, maps the values of menu items the given test user can access, and then attempts other menu items.