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

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

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

<b>bold text</b>
<html><body>...
</body></html>

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>
  <li>list item 2</li>
</ul>
<a href="site.com/boat.jpg">Link!</a>
<script>
  document.location = ...
</script>
<img src="boat.jpg" />

Introduction

Introduction
HTTP over Secure Socket Layer (HTTPS)

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 Cert_{chase.com}, containing its public key \( P_{\text{chase.com}} \).
6. The browser verifies the authenticity of Cert_{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 \( E_K(M) \), \( H(K||M) \).
Cross-Site Request Forgery — Login Attack.

Digital Certificates

- 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.

Extended Validation Digital Certificates

- **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.

Extended Validation Digital Certificates...

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.

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

Source: [http://javascript.info/tutorial/dom-nodes](http://javascript.info/tutorial/dom-nodes)

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.:
  ```
  <img src="..." onMouseOver="javascript:hello()">
  ```
DOM Tree Traversal

DOM tree node properties:

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstChild, lastChild</td>
<td>start/end of this node's list of children</td>
</tr>
<tr>
<td>childNodes</td>
<td>array of all this node's children</td>
</tr>
<tr>
<td>nextSibling, previousSibling</td>
<td>neighboring nodes with the same parent</td>
</tr>
<tr>
<td>parentNode</td>
<td>the element that contains this node</td>
</tr>
</tbody>
</table>

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

```javascript
```

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
<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="53 kg"/>
    <input type="submit"/>
</form>
</body></html>
```

HTTP is sent in cleartext — susceptible to MITM attack.

Sessions Using Hidden Form Fields...

```html
<input type="hidden" name="name" value="Alice"/>
<input type="hidden" name="height" value="170cm"/>
<input type="hidden" name="weight" value="53 kg"/>
```

Use HTTPS instead.
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.

Let’s assume Alice is browsing to [http://www.victoriassecret.com](http://www.victoriassecret.com).

She fills out a form with her personal data:

```html
<html>
<body>
<form action="http://www.victoriassecret.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>
```

**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

- mail.example.com
- example.com
- one.mail.example.com
- example.com
- .com

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.

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
<html>
<body>
<form action="http://www.victoriassecret.com/buy.jsp" method="get">
  <input type="hidden" name="sessionID" value="0x324A...">
</form>
</body>
</html>
```

- The session ID should be hard to guess.
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 — 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:
   - add random numbers to client-side/server-side tokens,
   - change session tokens frequently.
Click-Jacking

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

  ```html
  ```

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

Privacy Attacks — Third-party cookies

1. You browse to http://www.example1.com:

   ```html
   <HTML><BODY>
   <img src="http://ads.evil.com">
   </BODY></HTML>
   ```

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

3. You browse to http://www.example2.com:

   ```html
   <HTML><BODY>
   <img src="http://ads.evil.com">
   </BODY></HTML>
   ```

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

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

Cross-Site Scripting (XSS)

- Idea:
  1. attacker injects code C into a web site,
  2. C makes its way into generated web page P,
  3. a user is served the P page,
  4. 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
  <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
<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>
```

- What if Eve had instead added the text

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

  as her comment?
- Then Alice would be executing this page:

```html
<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!<br>
  </BODY>
</HTML>
```

Cross-Site Scripting...

- Obviously, Eve could insert more harmful code:

  ```html
  <script>
    document.cookie;
  </script>
  ```

  This redirects the browser to the evil site, and passes along Alice’s cookies.
- Eve could be more cunning:

  ```html
  <script>
    img = new Image();
    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:

```html
<iframe frameborder=0 src="" height=0 width=0 id="XSS" name="XSS">
</iframe>
<script>
frames["XSS"].location.href = 
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 — 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.

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

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

The attacker constructs this URL:

```html
<script>
document.location=
document.cookie
</script>
```

When the victim navigates to the URL, the payload will be executed in their browser.
Cross-Site Scripting — Counter-Countermeasures

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

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

  turns into

  ```html
  <script>alert(%27hello%27);</script>
  ```

Cross-Site Request Forgery (CSRF)

- **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).
  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.

Cross-Site Request Forgery (CSRF)

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

  ```html
  <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.

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, ....
PHP

- `$_GET['variable']` — array of GET input variables.
- No typing.

```php
<?php
// insert code here

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

Remote File Inclusion (RFI)

- Let this be index.php:

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

- Most sites now forbid RFI.
Local File Inclusion (LFI)

- As RFI, but a local file gets executed

- Getting the password file:
  \[http://www.cnn.com/index.php?page=/etc/passwd%00\]
  %00 is a null byte, effectively removing the .php extension.

- Attack: The attacker
  1. uploads a file (a php script hiding as a .jpg file, for example).
  2. tricks the site to execute the uploaded file using 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.
### SQL tables

- SQL databases store records as tables:

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>author</th>
<th>body</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Databases</td>
<td>John</td>
<td>Story 1</td>
</tr>
<tr>
<td>2</td>
<td>Computers</td>
<td>Joe</td>
<td>Story 2</td>
</tr>
<tr>
<td>3</td>
<td>Security</td>
<td>Jane</td>
<td>Story 3</td>
</tr>
<tr>
<td>4</td>
<td>Technology</td>
<td>Julia</td>
<td>Story 4</td>
</tr>
</tbody>
</table>

### SQL commands

- SQL commands for accessing a relational database:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>extract records from tables</td>
</tr>
<tr>
<td>INSERT</td>
<td>insert new records in a table</td>
</tr>
<tr>
<td>UPDATE</td>
<td>alter a record in a table</td>
</tr>
<tr>
<td>DELETE</td>
<td>remove a record in a table</td>
</tr>
<tr>
<td>UNION</td>
<td>combine the results of multiple queries</td>
</tr>
</tbody>
</table>

### SQL queries

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

### SQL Injection Attack

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

Consider this URL:

```
```

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

Consider instead

```
```

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";
}
?>
```

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 users table to the attacker.
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.