PV222
Security Architectures

Lecture 2

Web Security

Lecture Overview

- What is the web?
- The web components: HTTP and HTML
- HTTP, state and cookies
- OWASP Top 10
- SSL/TLS

HTML, HTTP and the Web

The World Wide Web: In the beginning…

- The Internet began as a US Government experiment in the late 1960s.
- However, it was the early 1990s before the Internet was widely available outside of the government and academic sectors.
- At the same time some scientists in CERN (the European Particle Physics Laboratory) released an authoring language and distribution system.
- This was the birth of the **Hypertext Markup Language** (**HTML**).
- At its core it's a multimedia-enabled, integrated electronic document language.
- Key to its success was the **hypertext linking** of documents, whereby documents automatically reference other documents.

Clients, Servers and Browsers

- To access HTML documents, we run **browsers** on **client machines**.
- The browser links to **web severs** over the Internet to access and retrieve electronic documents.
- All web activity begins on the client side, when a users starts his or her browser.
- The browser begins by loading a **home page** HTML document from either local storage, or from a server over some network.
- This request (and the server's reply) is formatted according to the **HyperText Transfer Protocol** (**HTTP**) standard.

HTML

- HTML is a **document-layout** and **hyperlink-specification** language.
- It defines the syntax and placement of special, embedded directions that aren't displayed by the browser, but tell it how to display the contents of the document.
  - This includes the text, images, and other supported media.
- It also tells the browser how to make the document interactive through special hypertext links.
  - These connect one document with other documents – on any other computer – as well as with other Internet resources, such as FTP.

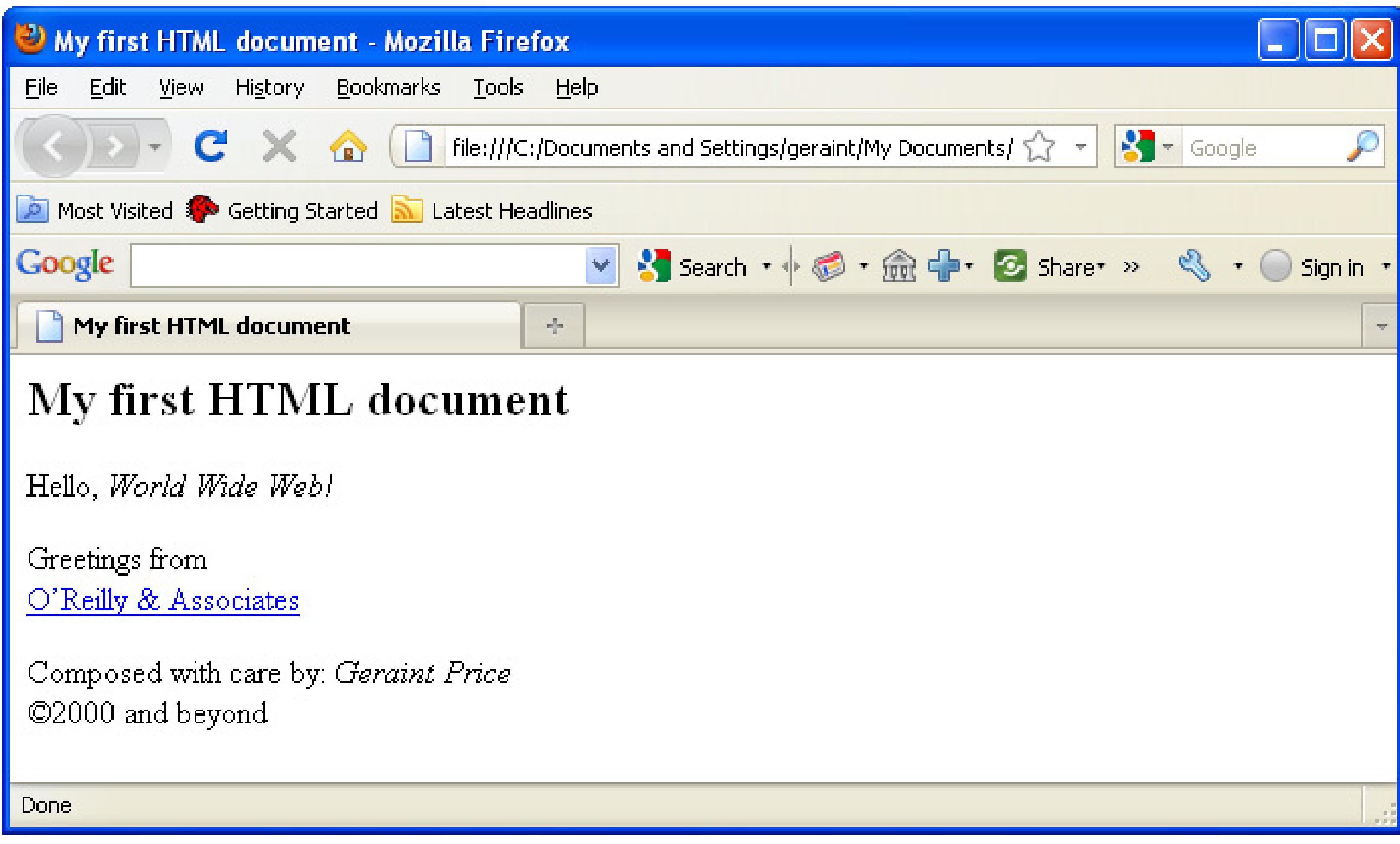
The Web Standards

- The **World Wide Web Consortium (W3C)** was formed with the charter to define the standard versions of HTML.
- Beyond HTML, the W3C has the broader responsibility of standardising any technology related to the World Wide Web.
  - `http://www.w3c.org`
- Even broader in each than W3C, the **Internet Engineering Task Force (IETF)** is responsible for defining and managing every aspect of Internet technology.
- The IETF defines all of the technology of the Internet via official documents known as Requests For Comments or RFCs.
  - `http://www.ietf.org`

A First HTML Document

```
<html>
<head>
<title>My first HTML document</title>
</head>
<body>
<h2>My first HTML document</h2>
Hello, <i>World Wide Web!</i>
 <!-- No “Hello, World” for us -->
<p>
               Greetings from<br>
<a href=“http://www.ora.com”>O’Reilly & Associates</a>
<p>
Composed with care by:
<cite>Geraint Price</cite>
<br>&copy;2000 and beyond
</body>
</html>
```

A First HTML Document



HTML Embedded Tags

- HTML is an **embedded language**: you insert the language's directions or **tags** into the same document that you and your readers load into a browser to view.
- The browser uses the information inside the HTML tags to decide how to display or otherwise tread the subsequent contents.
  - For instance, the `<i>` tag that follows the word "Hello" in the simple example tells the browser to display the following text in italic.
- Most tags define and affect a discrete region of your HTML document.
  - The region begins with the **start tag**, and finishes with the **end tag**. An end tag is the start tag's name preceded by a forward slash (`/`).
  - For example, the end tag that matches the "start italicizing" `<i>` tag is `</i>`.

Hyperlinks

- What makes HTML so useful on the Internet is **hypertext**.
- Hypertext gives you the ability to retrieve and display a different document simply by clicking on an associated word or phrase (**hyperlink**) in the HTML document.
- To include a hyperlink to some other document, you need to know the document's unique address and how to put an **anchor** in the HTML document.

URLs – I

- Every document and resource on the Internet has a unique address known as its **uniform resource locator (URL)**.
- A URL consists of:
  - the document's name preceded by the hierarchy of directory names in which the file is stored (**pathname**);
  - the Internet **domain name** of the server that hosts the file;
  - the software and manner by which the browser and the document's host server communicate to exchange the document (**protocol**).
- This information is arranged:
  - `protocol://server_domain_name/pathname`

URLs – II

- Here are some example URLs:
  - `http://www.kumquat.com/docs/catalog/price_list.html`
  - `price_list.html`
  - `http://www.kumquat.com/`
  - `ftp://ftp.netcom.com/pub/`
- The first example is what's known as an **absolute** or complete URL, as it includes every part of the URL format: protocol, server, and pathname.
- Browsers also let you use **relative** URLs and automatically fill in any missing portions.
  - The second example is the simplest relative URL.
  - Relative URLs are also useful if you don't know a directory or document name.
  - The third example points to a `kumquat.com`'s server, and leaves is up to the server to decide which file to send back to the client.

The HTTP Protocol

- The web protocol, i.e. the set of rules by which data is transferred between web browsers and web servers is called **HTTP**, for **HyperText Transfer Protocol**.
- This is a very simple “request/reply” protocol running over **TCP** (the **Transmission Control Protocol**).
- Requests are directed from a web browser to a resource at a specific **address**.
- HTTP is an application-level protocol.

HTTP: Overview

- There are two main versions of HTTP:
  - Version 1.0 (HTTP/1.0 defined in RFC 1945) and Version 1.1 (HTTP/1.1 defined in RFC 2616).
- The fundamental unit of HTTP communication is a **message** (a structured sequence of bytes).
- HTTP is a request-response protocol.
  - The client (e.g. web browser) submits a **request** message to the server.
  - The server, which stores content, or provides resources, returns **response** message to the client.

HTTP: Requests

- A client's HTTP request message:
  - **Request** line, such as `GET /images/logo.png HTTP/1.1`
  - **Headers**, such as `Host` and `Date`.
  - An empty line.
  - An optional message body.
- In the HTTP/1.1 protocol, all headers except `Host` are optional.

HTTP: Responses

- A server's response consists of:
  - a **status line**, including the protocol version number, and a success/error code, and
  - a MIME-like message, containing server information, content **meta-information** (**headers**), and content.
- The content will typically be written in HTML.

Web Security

What is Web Security?

- Garfinkel and Spafford (in *Web Security, Privacy & Commerce*) define web security as:
  1. *"Securing the web server and the data that is on it."*
  2. *"Securing information that travels between the web server and the user."*
  3. *"Securing the end user's computer and other devices that people use to access the Internet."*

Securing the Web Server

- Securing the web server is a three part process:
  - First, the computer itself must be secured using traditional computer security techniques.
  - Second, special programs that provide web services must be secured.
  - Finally, you need to examine the operating system and the web service to see if there are any unexpected interactions between the two.

Simplification of Services

- One of the best strategies for improving a web server’s security is to minimise the number of services provided by the host on which the web server is running.
  - If you have to provide both a web server and a mail server, the safest strategy is to put them on different computers.
  - Chose an operating system that and web server that don’t come with lots of extra defaults and unnecessary options.
  - The more complex the system, the more interactions and the more that can go wrong.
  - You should limit the number of users who have the ability to log into the computer.

Securing Information in Transit

- Much of the initial emphasis in the field of web security surrounded securing the information as it travels over the Internet.
- There are many ways in which you can secure information that travels through a network:
  - Physically securing the network, so that eavesdropping is impossible.
  - Hide the information that you wish to secure within information that appears innocuous.
  - Encrypt the information so that it cannot be decoded by any party who is not in possession of the proper key.
- Of these option, encryption is the only technique that is feasible on a large-scale public network.

Securing the User's Computer

- In the early days of the Web, browsers were regularly being exposed as insecure.
- However, during that period, the main cause of problems for end users were viruses and worms.
- Many computer security professionals had long maintained that education was the way to secure end user's computers.
- In recent years, however, some people have revised their opinion, and are now putting their hopes on strong end user computer security technology.
- The reason is that computer systems are getting too complicated for most end users to make rational security decisions.

HTTP is stateless

- The HTTP protocol does not require the server to maintain any protocol state.
- That is, the server does not keep any information to enable consecutive requests from a single user agent to be linked.
- Hence HTTP does not support "sessions", e.g. as might be required to support e-commerce.

Cookies

- HTTP Cookies are simple means of enabling browser sessions with a server.
- The idea is that the server sends back state information in its response header, in the form of a *Cookie*.
- The Cookie is then resubmitted with the next request to the same server.
- A Cookie might, for example, specify the current contents of your shopping basket.

Cookie contents

- A cookie header (in a response header) contains:
  - *attribute*, the data payload;
  - *domain scope*, enables sharing of cookies by web hosts with specified domain name;
  - *path scope*, limits the URI path to which the cookie should be sent back;
  - *expiration*, the expiry date of the Cookie;
  - *SSL flag*, if set the Cookie should only be sent back via an HTTPS (HTTP over SSL) connection.

Cookies and privacy

- Whilst Cookies are an invaluable tool for e-commerce and other uses of the web, they also constitute a privacy threat.
- Clearly, a server can use Cookies to track individual user PCs (even if the server cannot automatically discover the owner of a particular PC).
- We look at one way this tracking can pose a threat.

Tracking cookies

- Web-based advertising agencies, e.g. DoubleClick, Focalink, Globaltrack, and ADSmart put advertisements on web sites.
- These web pages contain an `<IMG>` tag, pointing to a URL on the advertising agency's server.
- When a web browser sees this `<IMG>` tag, it contacts the agency server to retrieve the graphic.
- The first time the graphic is downloaded, the user browser will receive an agency cookie containing a random ID.

Tracking cookies

- Every time the browser visits a site containing the agency’s advertisements, it sends the cookie (the random ID) along with the URL of the page that is being read (using the referer field) to the agency.
- This enables the agency to track a single user’s behaviour across multiple web sites.

Countermeasures

- Software can be used to detect tracking cookies and eliminate them (and, in some cases, even prevent them being loaded).
- Sources of software include:
  - www.spybot.info (for Spybot Search and Destroy), and
  - www.lavasoftusa.com (for Ad-Aware 6.0)

Referer field

- One of the fields in the header of an HTTP request message is the **Referer** field.
- This allows the client to specify, for the server's benefit, the address (URI) of the resource from which the URI of this request was obtained.
- In most browsers, when you look at a new page, the browser will send the URL of the current page in the referer field.
- Under the HTTP definitions, this is means to be an option for the user, but according to Garfinkel and Spafford, they have never seen a browser where it is optional.

OWASP Top Ten

OWASP Top Ten – I

- The *Open Web Application Security Project* (OWASP) is an open community dedicated to improving the security of web applications.
- The OWASP Top Ten is a project to collate information on what the most critical web application security flaws are.
- Designed to educate designers, developers and architects about the consequences of the most common web application security vulnerabilities.
- Goal is education… it is not a standard or policy.

OWASP Top Ten – II

1. Cross Site Scripting (XSS)
2. Injection Flaws
3. Malicious File Execution
4. Insecure Direct Object Reference
5. Cross Site Request Forgery (CSRF)
6. Information Leakage and Improper Error Handling
7. Broken Authentication and Session Management
8. Insecure Cryptographic Storage
9. Insecure Communications
10. Failure to Restrict URL Access

Cross-Site Scripting (XSS) – I

- XSS flaws occur whenever an application takes data that originated from a user and sends it to a browser without first validating that content.
- XSS allows attackers to execute scripts in the victim's browser.
- Using this technique, the attacker can:
  - Hijack user sessions;
  - Deface websites;
  - Inset hostile content;
  - Conduct phishing attacks;
  - etc…
- Usually JavaScript, but any scripting language which is supported by the victim's browser can be misused.

Cross-Site Scripting (XSS) – II

- Three types of XSS.
- **Reflected**:
  - A page will reflect user supplied data directly back to the user.
- **Stored**:
  - Takes hostile data, stores it in a file, database or other backend system, then at a later stage displays the data to the user.
  - Very dangerous in Blogs, forums, etc. where a large number of users will see input from other users.
- **DOM injected**:
  - The site's JavaScript code and variables are manipulated rather than the HTML.

Cross-Site Scripting (XSS) – III

- The best protection for XSS is a combination of:
  - "whitelist" validation of all incoming data;
  - appropriate encoding of all outgoing data.
- Here are some of the validation principles:
  - **Input validation**: validate against length, type, syntax, etc. Use "known good" acceptance strategy.
  - **Strong output encoding**: ensure that all user-supplied data is correctly encoded (e.g. HTML or XML).
  - **Specify the output encoding**: for example, ISO 8859-1 character encoding.
  - **Do not use "blacklist" validation**: it's stronger to use a known "good list" than a known "bad list".

Injection Flaws – I

- Injection occurs when user-supplied data is sent to a command interpreter as part of a comment or query.
- Attackers trick the interpreter into executing unexpected commands via supplying specially crafted data.
- Injection flaws allow attackers to create, read, update, delete any data available to the application.
- Most famous type of injection query is *SQL Injection* attack.

Injection Flaws – II

- SQL Injection occurs when:
  - data is entered to a program from an untrusted source;
  - the data is used to dynamically generate an SQL query.
- This can lead to the loss of the following services:
  - **Confidentiality**: any data held in the database can be read.
  - **Authentication**: if SQL commands are used to verify username and passwords, then an attacker can possibly log-in without prior knowledge of the password.
  - **Authorisation**: if the authorisation data is held by a database, then this can be modified.
  - **Integrity**: it may be possible to modify or delete sensitive information.

Injection Flaws – III

- Avoid the use of interpreters where possible!
- If you must use an interpreter, then use a safe API (e.g. one with strong typing).
- Even if you use a strong API, validation is still recommended.
- Examples of recommended precautions:
  - **Input validation**: use a “known good” acceptance strategy.
  - **Use strongly types APIs**: this helps reduce the types of input that will automatically be accepted.
  - **Enforce least privilege**: when connecting to databases and other back-ends.
  - **Avoid detailed error messages**: these might be useful to an attacker.

Malicious File Execution – I

- Any web server is vulnerable where the application allows filenames or files from the user.
- This allows attackers to perform:
  - Remote code execution;
  - Remote root kit execution.
- As well as allowing remote execution, it can be used to access local file systems.
- Other methods of attack:
  - Hostile data being uploaded to session files, log data, and via image uploads (e.g. in a forum environment).

Malicious File Execution – II

- Some ways in which this attack can be prevented:
  - **Strongly validate user input**: again, accept only “known good” input.
  - **Add firewall rules**: these should prevent web servers from making connections to external websites.
  - **Use a taint checking mechanism**: some languages allow any variables that have user input to be flagged as potentially dangerous.
  - **Implement a chroot jail**: this is a type of sandboxing technique to isolate applications from each other.

Insecure Direct Object Reference – I

- A *direct object reference* when some internal object (e.g. file, directory, database record or key) is used as a URL or form parameter.
- This can allow an attacker to manipulate other objects without authorisation.
- For example:
  - In internet banking applications, the account number is often used as the database primary key.
  - It is then tempting to use this account number directly on a web interface.
  - If no extra check is done to verify the user, an attacker can manipulate this parameter to see or change **all** accounts.

Insecure Direct Object Reference – II

- The best way to protect against this is to not to use a direct object reference.
  - Instead use an indirect mapping which is easier to validate.
- Avoid exposing your private object references to users wherever possible.
- Validate any private object reference with the "known good" approach.
- Verify authorisation to all referenced objects.

Cross Site Request Forgery – I

- A CSRF attack forces a logged-on victim's browser to send a request to a vulnerable web application.
- The vulnerable application then performs the chosen action on behalf of the victim.
- The malicious code is usually not on the attacked side – hence "*Cross Site*".
- A typical CSRF attack against a forum might take the form of directing a user to invoke some function, such as the application's logout page:
- The following tag in any webpage viewed by the user will automatically log them out:
  - `<img src="http://www.example.com/logout.php">`

Cross Site Request Forgery – II

- Applications must ensure that they are not relying on credentials or tokens that are automatically submitted by browsers.
- The only solution is to use a custom token that the browser will not “remember” and then automatically include with a CSRF attack.
  - Insert custom random tokens into every form and URL.
- Also check that there are no XSS vulnerabilities in your application.
- For sensitive data or value transactions, re-authenticate or use transaction signing.

Information Leakage and Improper Error Handling – I

- Applications can unintentionally leak information about their configuration, internal workings or violate privacy through a variety of application problems.
- Web applications will often leak information about their internal state through detailed or debug error messages.
- Often, this information can be leveraged to launch or even automate more powerful attacks.

Information Leakage and Improper Error Handling – II

- Applications frequently generate error messages and display them to users.
- Sometimes this can reveal useful information to an attacker.
- Some common ways in which this might happen:
  - Where an error displays too much detailed information: e.g. a stack trace; failed SQL statements.
  - Functions that respond with different results based on different inputs: for example, responding to an incorrect username/password combination with different error codes depending on which parts were wrong.

Information Leakage and Improper Error Handling – III

- Applications should use a standard exception handling architecture to prevent additional information leaking to an attacker.
- Effective practice might include:
  - Disable or limit detailed error handling.
  - Ensure that secure paths that have multiple outcomes return similar or identical error messages.
  - Different layers (e.g. database layer, web server layer) will return exceptional results. Ensuring that these are sanitised to prevent information leakage is important.
  - Always return a “standard” error screen can prevent an automated tool from finding out if a serious error occurred.

Broken Authentication and Session Management – I

- Many web applications allow authentication or session management through tokens or session cookies.
- Failure to protect these tokens or cookies can allow an attacker to hijack a user's account or login session.
- Authentication relies on secure credential communication and storage.
  - Ensure that SSL is the only option for authenticated parts of the application.
  - All credentials are stored in a hashed or encrypted form.

Broken Authentication and Session Management – II

- Other things that might be done:
  - Only use the inbuilt session management mechanism.
  - Do not allow the login process to start from an unencrypted page.
  - Ensure that every page has a logout link, and that upon logout all server side state and client side cookies are destroyed.
  - Use a timeout period that automatically logs out an inactive session.
  - Do not rely on spoofable credentials as the sole form of authentication: e.g. IP addresses; referrer headers.

Insecure Cryptographic Storage – I

- Protecting key sensitive data using cryptography has become an important part of application security.
- The most common application flaws are:
  - Not encrypting sensitive data.
  - Using home grown algorithms.
  - Insecure use of strong algorithms.
  - Continued use of proven weak algorithms (MD5, SHA-1, RC3, RC4, etc…)
  - Hard coding keys, and not protecting key storage.

Insecure Cryptographic Storage – II

- Ensure that everything that should be encrypted is actually encrypted.
- Ensure that the cryptography is properly implemented:
  - Do not create cryptographic algorithms.
  - Do not use weak algorithms.
  - Generate keys offline and store private keys with care.
  - Ensure that encrypted data stored on disk is not easy to decrypt.

Insecure Communications

- Applications frequently fail to encrypt sensitive network traffic.
- Encryption (usually SSL) should be used for all authenticated connections, especially Internet-accessible web pages.
- In addition, encryption should be used whenever sensitive data (e.g. credit card, health information) is transmitted.
- Applications that can fallback or forced out of encrypting can be abused by attackers.
- Encrypting communications with back-end servers is also important, as the information they carry is more sensitive and more extensive.
- Under PCI (Payment Card Industry) Data Security Standard requirement 4, you must protect cardholder data in transit.

Failure to Restrict URL Access

- Frequently, the only protection for a URL is that links to that page are not presented to an unauthorised user.
- However, a motivated, skilled or lucky attacker may be able to find and access these pages.
- Web applications must enforce access control on all URLs and business functions.
- It is also important that the authorisation is checked regularly during the process.
  - Otherwise an attacker might be able to skip the authentication phase and forge the parameters for the subsequent steps.

Other Attacks on Servers

- Web servers themselves may be the victims of attacks via HTTP requests.
- For example, to cause *buffer overflow* in a web server, an attacker might induce errors at Web traffic ports by entering large character strings to find a susceptible overflow field.
- Once a field spills over into a code-executing field, an attacker will enter another string that will spill a command into the executable field.
- Buffer overflows can give an attacker access to a range of sensitive server functions.

URL Obfuscation Attacks

- *URL Obfuscation* attacks are mechanisms used to trick users to visit an attacker's website.
  - Examples of such attacks are: *using strings*; *using @ sign*; *URL redirection*.
  - Using strings:
    - `http://254.231.52.42/ebay/account_update/now.php`
  - Using @ sign:
    - `http://www.citybank.com/update.pl@254.231.52.42/usb/upd.pl`
  - URL redirection:
    - `http://usa.visa.com/track/dyredir.jsp?rDirl=http://200.251.251.10/.verified/`

SSL/TLS

SSL/TLS overview

- SSL = Secure Sockets Layer. Current version is v3.
- TLS = Transport Layer Security. TLS 1.0 is similar to SSL 3.0 with minor tweaks.
- TLS is defined in RFC 2246.
- SSL/TLS provides security “at TCP layer”. In fact, it usually provides a thin layer between TCP and HTTP.

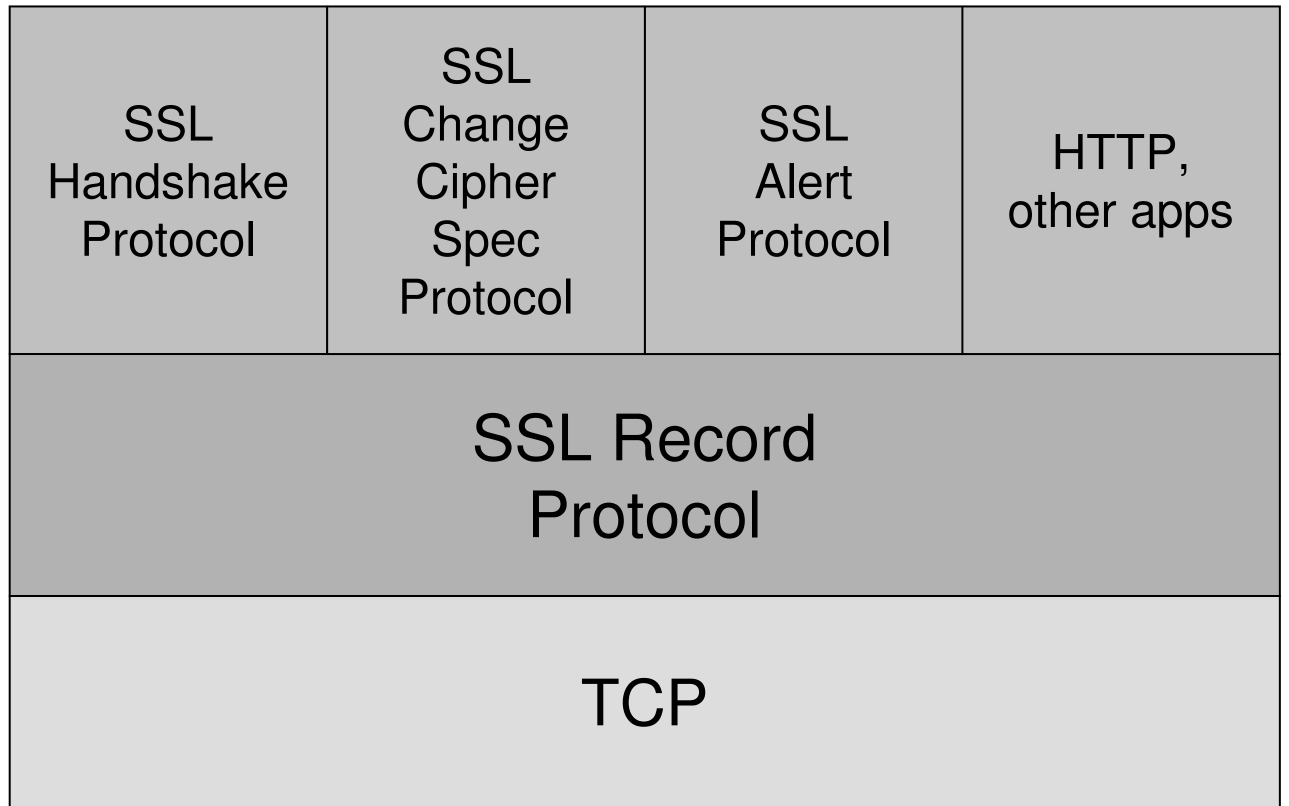
SSL/TLS basic features

- SSL/TLS widely used in Web browsers and servers to support "secure e-commerce" over HTTP.
  - Built into Microsoft IE, Netscape, Mozilla, Apache, IIS, …
  - Presence of SSL protected link indicated by the browser padlock symbol.

SSL architecture

- SSL architecture involves two layers:
  - *SSL Record Protocol*
    - Lower layer providing secure, reliable channel to upper layer.
  - *Upper layer* carrying:
    - SSL Handshake Protocol,
    - Change Cipher Spec. Protocol,
    - Alert Protocol,
    - HTTP,
    - Any other application protocols.

SSL architecture



SSL Record Protocol

- Carries application data and “management” data.
- Sessions:
  - Sessions created by handshake protocol.
  - Defines set of cryptographic parameters (encryption and hash algorithm, master secret, certificates).
  - Carries multiple connections to avoid repeated use of expensive handshake protocol.
- Connections:
  - State defined by nonces, secret keys for MAC and encryption, IVs, sequence numbers.
  - Keys for many connections derived from single master secret created during handshake protocol.

SSL Record Protocol

- SSL Record Protocol provides:
  - Data origin authentication and integrity.
    - MAC using algorithm similar to HMAC, based on MD5 or SHA-1 hash algorithms.
    - MAC protects 64 bit sequence numb for anti-replay.
  - Confidentiality.
    - Bulk encryption using symmetric algorithm (IDEA, RC2-40, DES-40 (exportable), DES, 3DES, RC4-40 and RC4-128.

SSL Record Protocol

- Data from application/upper layer SSL protocol partitioned into fragments (max size 2^{14} bytes).
- MAC first, then pad (if needed), and finally encrypt.
- Prepend header containing: Content type, version, length of fragment.
- Submit to TCP.

SSL Handshake Protocol

- SSL needs secret keys:
  - Used for MAC & encryption at Record Layer.
  - Different keys in each direction.
- These keys are established as part of the SSL Handshake Protocol.
- The SSL Handshake Protocol is a complex protocol with many options.

SSL Handshake Protocol security goals

- Entity authentication of participating parties ("client" and "server").
  - Server nearly always authenticated, client more rarely.
  - Appropriate for most e-commerce applications.
- Establishment of a fresh, shared secret.
  - Shared secret used to derive further keys for SSL Record Protocol.
- Secure ciphersuite negotiation (including encryption and hash algorithms).

SSL Handshake Protocol – key exchange

- SSL supports several key establishment mechanisms.
- Most common is RSA encryption.
  - Client chooses `pre_master_secret`, encrypts it using public RSA key of server, and sends to server.
- Can also create `pre_master_secret` using one of several variants of Diffie-Hellman key establishment protocol.

SSL Handshake Protocol – entity authentication

- SSL supports several different entity authentication mechanisms.
- Most common based on RSA:
  - The ability to decrypt `pre_master_secret` and generate correct MAC using keys derived from `pre_master_secret` authenticates the server to the client.
- DSS or RSA signatures on nonces (and other fields, e.g. Diffie-Hellman values).

SSL key deriviation

- Keys used for MAC and encryption derived from `pre_master_secret`:
  - Derive `master_secret` from `pre_master_secret` and client/server nonces using MD5 and SHA-1.
  - Derive key material from `master_secret` and client/server nonces, by repeated use of hash functions.
  - Split key material into MAC and encryption keys as needed.

SSL Handshake Protocol run

- We choose the most common use of SSL for illustration:
  - No client authentication.
  - Client sends `pre_master_secret` using Server's public encryption key from Server certificate.
  - Server authenticated by ability to decrypt to obtain `pre_master_secret`, and construct correct finished message.

SSL Handshake Protocol run

M1: C → S: `ClientHello`

- Client initiates connection.
- Sends client version number.
  - 3.1 for TLS.
- Sends `ClientNonce`.
  - 28 random bytes plus 4 bytes of time.
- Offers list of ciphersuites.
  - key exchange and authentication options, encryption algorithms, hash functions, e.g. `TLS_RSA_WITH_3DES_EDE_CBC_SHA`.

SSL Handshake Protocol run

M2: S → C: `ServerHello`, `ServerCertChain`, `ServerHelloDone`

- Sends server version number.
- Sends `ServerNonce` and `SessionID`.
- Selects single ciphersuite from list offered by client, e.g. `TLS_RSA_WITH_3DES_EDE_CBC_SHA`.
- Sends `ServerCertChain` message.
  - Allows client to validate server's public key.
- (optional) `CertRequest` message.
  - Omitted in this protocol run – no client authentication.
- Finally, `ServerHelloDone`.

SSL Handshake Protocol run

M3: C → S: `ClientKeyExchange`, `ChangeCipherSpec`, `ClientFinished`

- `ClientKeyExchange` contains encryption of `pre_master_secret` under server's public key.
- `ChangeCipherSpec` indicates that client is updating cipher suite to be used in this session.
  - Sent using SSL Change Cipher Spec. Protocol.
- (optional) `ClientCertificate`, `ClientCertificateVerify` messages.
  - Only when client is authenticated.
- Finally, `ClientFinished` message.
  - A MAC on all messages sent so far (both sides).
  - MAC computed using `master_secret`.

SSL Handshake Protocol run

M4: S → C: `ChangeCipherSpec`, `ServerFinished`

- `ChangeCipherSpec` indicates that server is updating cipher suite to be used on this session.
  - Sent using SSL Change Cipher Spec. Protocol.
- Finally, `ServerFinished` message.
  - A MAC on all messages sent so far (both sides).
  - MAC computed using `master_secret`.
  - Server can only compute MAC if it can decrypt `pre_master_secret` in M3.

SSL Handshake Protocol run

Summary:

M1: C $\rightarrow$ S: `ClientHello`

M2: S $\rightarrow$ C: `ServerHello`, `ServerCertChain`, `ServerHelloDone`

M3: C $\rightarrow$ S: `ClientKeyExchange`, `ChangeCipherSpec`, `ClientFinished`

M4: S $\rightarrow$ C: `ChangeCipherSpec`, `ServerFinished`

SSL Handshake Protocol run

1. Is the client authenticated to the server in this protocol run?
2. Can an adversary learn the value of `pre_master_secret`?
3. Is the server authenticated to the client?
   1. No.
   2. No. Client has validated server's public key; only the holder of the private key can decrypt `ClientKeyExchange` to learn `pre_master_secret`.
   3. Yes. `ServerFinished` includes MAC on nonces computed using key derived from `pre_master_secret`.

Other SSL Handshake options

- Many optional/situation-dependent protocol messages:
  - M2 (S → C) can include:
    - `ServerKeyExchange` (e.g. for DH key exchange).
    - `CertRequest` (for client authentication).
  - M3 (C → S) can include:
    - `ClientCert` (for client authentication).
    - `ClientCertVerify` (for client authentication).

Other SSL protocols

- Alert protocol.
  - Management of SSL session, error messages.
  - Fatal errors and warnings.
- Change cipher spec protocol.
  - Not part of SSL Handshake Protocol.
  - Used to indicate that entity is changing to recently agreed ciphersuite.
- Both protocols run over Record Protocol (so peers of Handshake Protocol).

SSL and TLS

- TLS 1.0 = SSL 3.0 with minor differences:
  - TLS signalled by version number 3.1
  - Use of HMAC for MAC algorithm.
  - Different method for deriving key material (`master-secret` and `key-block`).
    - Pseudo-random function based on HMAC with MD5 and SHA-1.
  - Additional alert codes.
  - More client certificate types.
  - Variable length padding (can be used to hide lengths of short messages and so frustrate traffic analysis).
  - And more…

SSL/TLS applications

- Secure e-commerce using SSL/TLS.
  - Client authentication not needed until client decides to buy something.
  - SSL provides secure channel for sending credit card information.
  - Client authenticated using credit card information, merchant bears (most of) risk.
  - Very widely used.

SSL/TLS application issues

- Secure e-commerce: some issues.
  - No guarantees about what happens to client data (including credit card details) after session: may be stored on insecure server.
  - Does client understand meaning of certificate expiry and other security warnings?
  - Does client software actually check complete certificate chain?
  - Does the name in certificate match the URL of e-commerce site? Does the user check this?
  - Is the site the one the client thinks it is?
  - Is the client software proposing appropriate ciphersuites?

SSL/TLS application issues

- Secure electronic banking:
  - Client authentication may be enabled using client certificates.
    - Issues of registration, secure storage of private keys, revocation and re-issue.
  - Otherwise, SSL provides secure channel for sending username, password, mother's maiden name, …
    - What else does client use same password for?
  - Does client understand meaning of certificate expiry and other security warnings?
  - Is client software proposing appropriate ciphersuites?
    - Enforce from server.

Standards

- All IETF RFCs can be obtained from: `www.ietf.org`
- The W3C recommendations are available at: `www.w3c.org`
- For general information about security standards see: A. W. Dent and C. J. Mitchell: *User's guide to cryptography and standards* (Artech House, 2004).
  - `http://www.isg.rhul.ac.uk/ugcs`

Acknowledgements

- SSL/TLS discussion based on (an abbreviated version of) Kenny Paterson's lecture for IY5511 course.
- Information on the OWASP Top 10 taken from:
  - `http://www.owasp.org/`
- Information on the basics of HTML taken from: C Musciano & B Kennedy, "*HTML: The Definitive Guide*", O'Reilly.
  - Plenty of information on HTML available from a number of web sites, including the W3C web site (lots of useful tutorial information there).
- Basic information on web security taken from: S. Garfinkel and G. Spafford – "*Web Security, Privacy & Commerce*", O'Reilly.
- Some additional information taken from original lecture notes by Chris Mitchell.

Conclusions

- After today's lecture you should:
  - Have a basic understanding of how the components that make up the web work.
  - Understand what are the security problems faced by clients and servers using the web as an interface.
  - Be able to describe a high level overview of how SSL allows us to build secure connections between clients and servers.
  - Be able to appreciate that security of web applications does not just start and end with SSL.