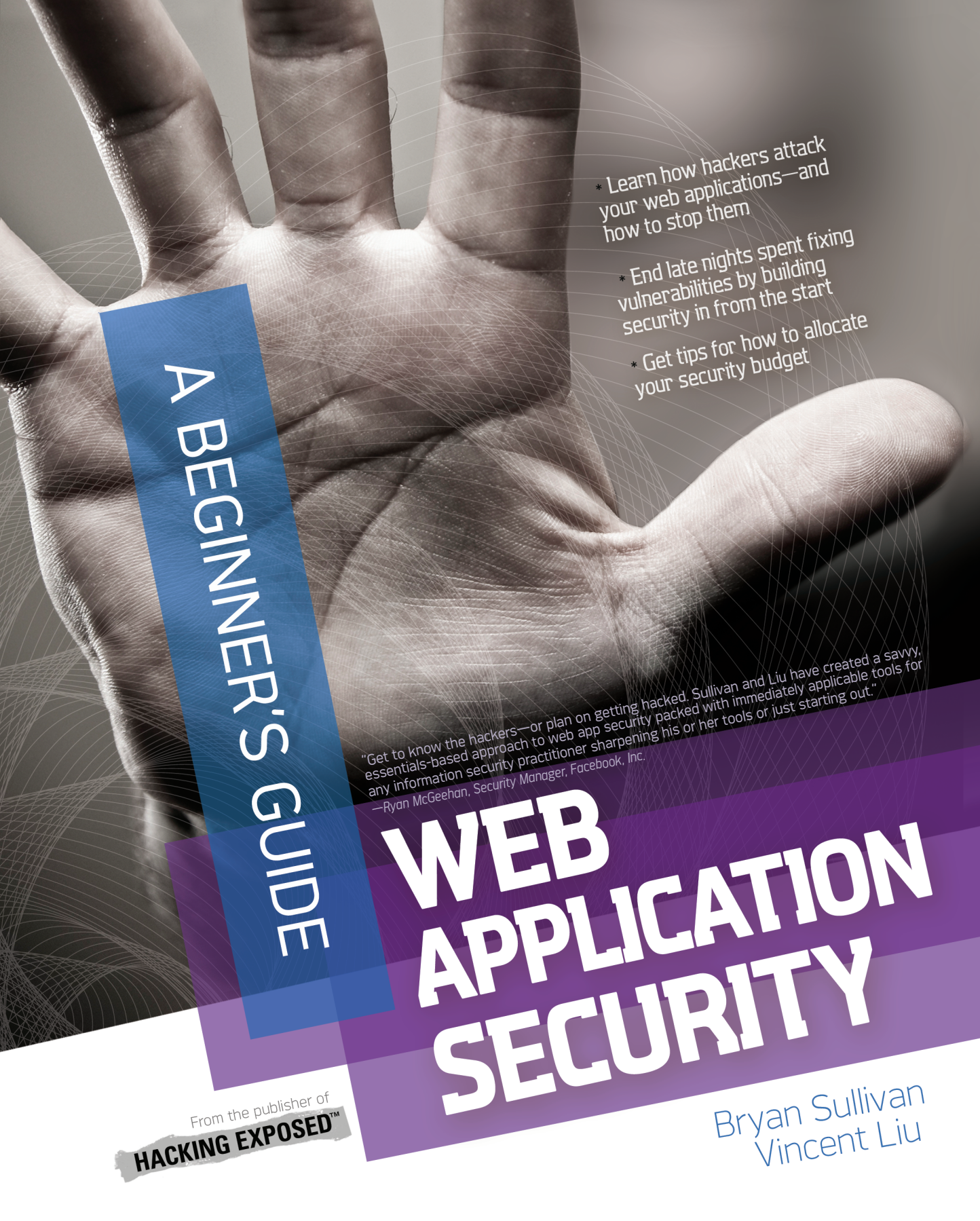
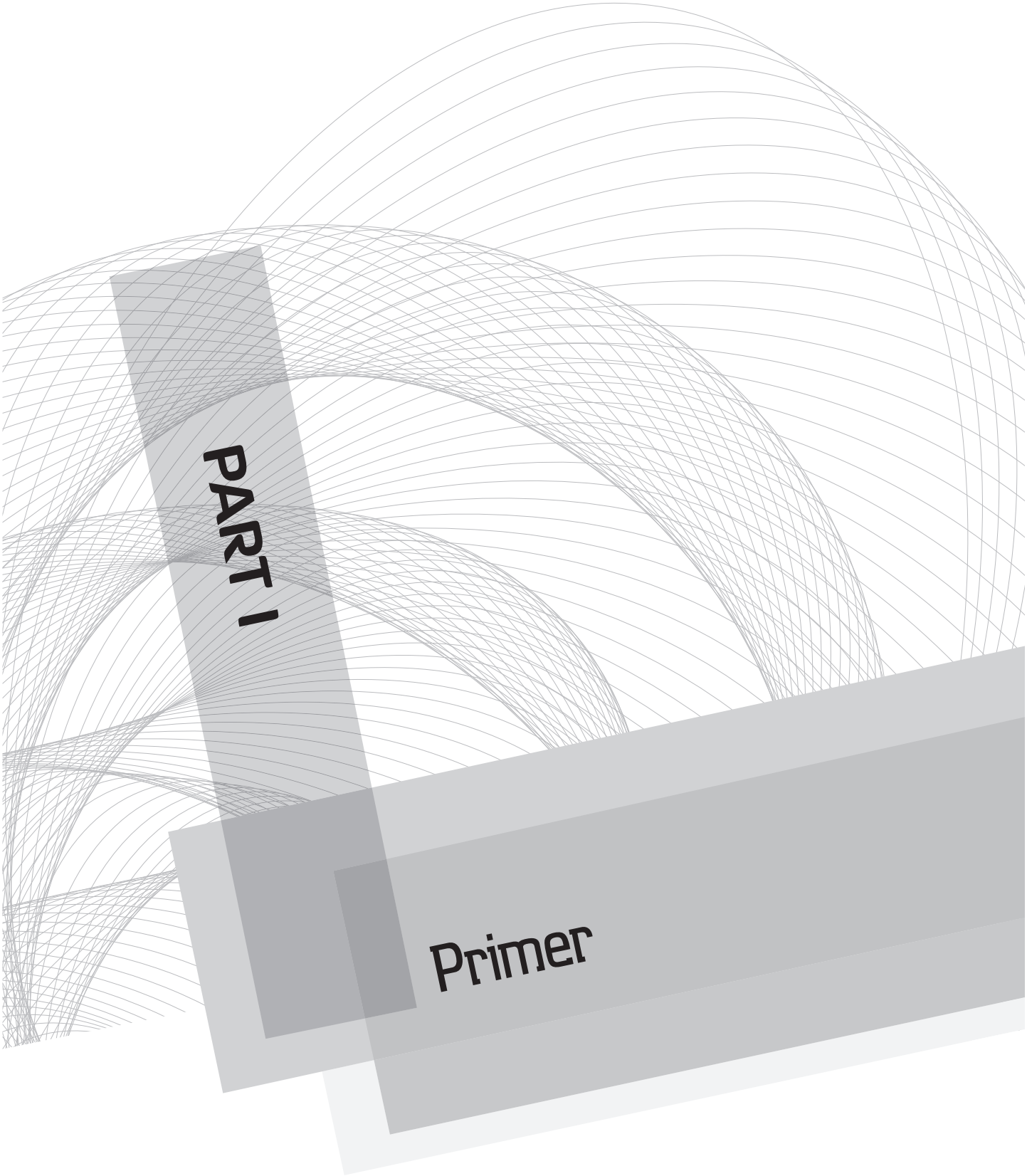

A BEGINNER'S GUIDE
* Learn how hackers attack your web applications—and how to stop them
* End late nights spent fixing vulnerabilities by building security in from the start
* Get tips for how to allocate your security budget
"Get to know the hackers—or plan on getting hacked. Sullivan and Liu have created a savvy, essentials-based approach to web app security packed with immediately applicable tools for any information security practitioner sharpening his or her tools or just starting out."
—Ryan McGeehan, Security Manager, Facebook, Inc.
WEB APPLICATION SECURITY
Bryan Sullivan
Vincent Liu
From the publisher of
HACKING EXPOSED™


PART I
Primer

CHAPTER 1

Welcome to the Wide World of Web Application Security

We'll Cover

- Misplaced priorities and the need for a new focus
- Network security versus application security: The parable of the wizard and the magic fruit trees
- Thinking like a defender
- The OWASP Top Ten List
- Secure features, not just security features

The information technology industry has a big problem—a 60-billion-dollar problem, in fact.

Sixty billion dollars is what the global IT industry spends on security in one year. That's more than the gross domestic product of two-thirds of the countries in the world. And it doesn't seem as if we're getting a lot for our money, either. Every week, there's a new report of some data breach where thousands of credit card numbers were stolen or millions of e-mail addresses were sold to spammers. Every week, there's some new security update for us to install on all of our work and home computers. If we're spending so much money on security, why are we still getting hacked? The answer is simple: we're spending money, but we're spending it on the wrong things.

Misplaced Priorities and the Need for a New Focus

A recent survey of security executives from Fortune 1000 companies (http://www.fishnetsecurity.com/News-Release/Firewalls-Top-Purchase-Priority-In-2010-Survey-Says-) showed that the number one IT security spending priority was network firewalls. Given that, you'd guess that the number one way these companies are getting attacked is through open ports on their networks, wouldn't you? In fact, if you did, you'd be dead wrong. The number one way Fortune 1000 companies and other organizations of all sizes get attacked is through their web applications.

How often do web applications get attacked? Security industry analysts suggest that as much as 70 percent of attacks come through web applications. And that 70 percent figure doesn't just represent a large number of small nuisance attacks like the site defacements

that were so common in the early days of the Web. Vulnerabilities in web applications have been responsible for some of the most damaging, high-profile breaches in recent news. Just a small sample of attacks in the first half of 2011 alone includes:

- The SQL injection attacks on the Sony Music web sites in May 2011 by the LulzSec organization. While unconfirmed by Sony, it's also believed that SQL injection vulnerabilities were responsible for the attacks against the Sony PlayStation Network and Qriocity that leaked the private data of 77 million users and led Sony to shut down the services for over a month. The overall cost of this breach to Sony has been estimated to exceed 171 million dollars (US).
- A cross-site scripting vulnerability in the Android Market discovered in March 2011 that allowed attackers to remotely install apps onto users' Android devices without their knowledge or consent.
- The attack on information security firm HBGary Federal in February 2011 by the hacker group Anonymous. Another simple SQL injection vulnerability in the www .hbgaryfederal.com website, combined with a poorly implemented use of cryptographic hash functions, enabled Anonymous to extract the company officers' usernames and passwords, which then enabled them to read the officers' confidential internal e-mails. The CEO of HBGary Federal resigned from the company shortly thereafter, citing a need to "take care of his family and rebuild his reputation."

None of these attacks were stopped by the sites' firewalls! But IT budgets still focus primarily on firewall defenses. This is puzzling, since network firewalls are completely useless to prevent almost any web application attack. You can't use firewalls to close off ports from which your web applications are being served, because then nobody could come to your web site. Organizations spend billions of dollars a year on advertising to get people to come to their sites; they're certainly not going to close them up with firewalls. Figure 1-1 shows a diagram of an attacker evading a server's firewall defenses by simply entering through the web site port 80.

We as an industry definitely have some misplaced priorities when it comes to security spending, but the magnitude of the imbalance is simply staggering. In another recent survey of IT professionals (http://www.barracudanetworks.com/ns/downloads/White_Papers/Barracuda_Web_App_Firewall_WP_Cenzic_Exec_Summary.pdf), almost 90 percent of companies reported that they spend less money on web application security than they spend on coffee: less than $1 per day per employee. We're willing to spend billions of dollars a year to protect our networks, but when it comes to the targets that are really getting hit the hardest, we skimp and cut corners. To repeat an often-used analogy,

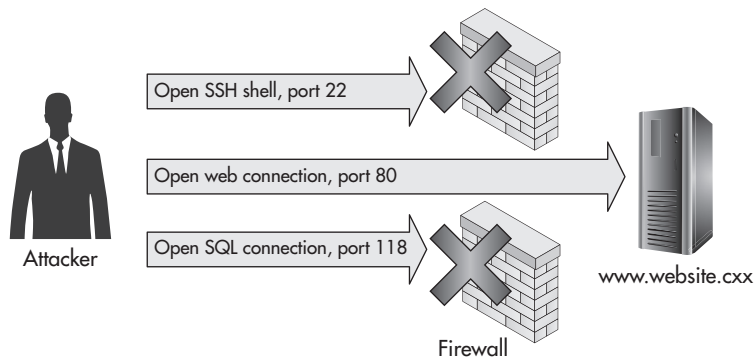


Figure 1-1 A server firewall preventing users (and attackers) from accessing most server ports but leaving port 80 open for web site traffic

this is like installing burglar alarms and steel bars on all of the windows in your home, but leaving the front door wide open.

Since the same survey showed that almost 70 percent of organizations rely on network firewalls for their web application defense—which is essentially the same as having no defense at all—it's hard to see this as anything besides an issue of being appropriately educated on web application security. People know their web applications are important, but they don't know how to secure them.

That's where this book comes in.

Network Security versus Application Security: The Parable of the Wizard and the Magic Fruit Trees

In order to understand the difference between network security issues and application security issues a little better, consider this parable of the wizard and the magic fruit trees.

Once upon a time there lived a kindly old wizard who loved fruit. He used his magic spells to create a magnificent orchard full of all different kinds of fruit trees. He created apple trees, banana trees, and plum trees. He conjured up entirely new kinds of fruit trees that never existed in nature, fields of vineyards that grew cherries the size of cantaloupes, and shrubs that grew oranges with purple skin and tasted like watermelons.

As we said, this wizard was a kindly wizard, and he didn't mind sharing his magical fruit with all the people of the village. He let them all come and go as they pleased through

his groves, picking as much fruit as they wanted—after all, the trees were magic and grew new fruit the second the old fruit was picked. Life was good and everyone was happy, until one day the wizard caught a lovesick young farm boy carving his sweetheart's initials into one of the lemonpear trees. He scolded the boy, sent him away, and turned the boy's ears into big floppy donkey ears as a punishment (just for a few hours, of course).

The wizard thought that was that and started to get back to his scrolls, but then he saw another villager trying to dig up a tree so he could take it back to his house and plant it there. He rushed over to stop this thief but an even more horrific sight caught his eye first. Two apprentices of the evil wizard who lived across the valley had come with torches and were trying to burn down the whole orchard to exact revenge for their master's embarrassing defeat at wizard chess earlier that month.

Now the wizard had had enough! He threw everyone out, and cast a spell that opened up a moat of boiling lava to surround the orchard. Now no one could get in to vandalize his beloved fruit trees, or to steal them, or try to burn them down. The trees were safe—but the wizard felt unhappy that now he wasn't able to share his fruit with everyone. And the villagers did tend to spend a lot of gold pieces buying potions from him while they were there picking his fruit. To solve this problem, he came up with an ingenious new solution.

The wizard invited his friend the giant to come live in the orchard. Now whenever someone wanted a piece of fruit, he would just shout what he wanted to the giant. The giant would go pick the fruit for them, jump over the lava moat, and then hand them the fruit. This was a better deal for both the wizard and the villagers. The wizard knew that all the miscreants would be kept away from the trees, and the villagers didn't even have to climb trees any more: the fruit came right to them.

Again, life was good and everyone was happy, until one day one very clever young man walked up to the edge of the lava where the giant was standing. Instead of asking the giant to bring him back a basket of persimmons or a fresh raisinmelon, he asked the giant to go up into the tower and fetch him the wizard's scrolls. The giant thought this request was a little strange, but the wizard had just told him to get the people whatever they asked for. So he went to the tower and brought back the magic scrolls for the young man, who then ran off with all of the wizard's precious secrets.

Real-World Parallels

If you were hoping for a happy end to this story, there isn't one—not yet, at least. First, let's take a look at the parallels between this story and the real-world security issues that organizations like yours face every day.

You (the wizard) have data (fruit) that you'd like to share with some people. But you know that if you just let everyone have free access to your server farm (orchard), there'll be disastrous results. People will deface your servers (vandalize the trees), they'll install botnets or other malware to take the servers over for themselves (steal the trees), and they'll try to deny service to the servers so no one can use them (burn the trees down).

In response to these threats, you erect a firewall (lava moat) to keep everyone out. This is good because it keeps the attackers out, but unfortunately it keeps all your legitimate users out too, as you can see in Figure 1-2. So, you write a web application (a giant) that can pass through the firewall. The web application needs a lot of privileges on the server (the way a giant is very powerful and strong) so it can access the system's database and the file system. However, while the web application is very powerful, it's not necessarily very smart, and this is where web application vulnerabilities come in.

By exploiting logic flaws in the web application, an attacker can essentially "trick" the web application into performing attacks on his behalf (getting the giant to do his bidding). He may not be able to just connect into the servers directly to vandalize them or steal

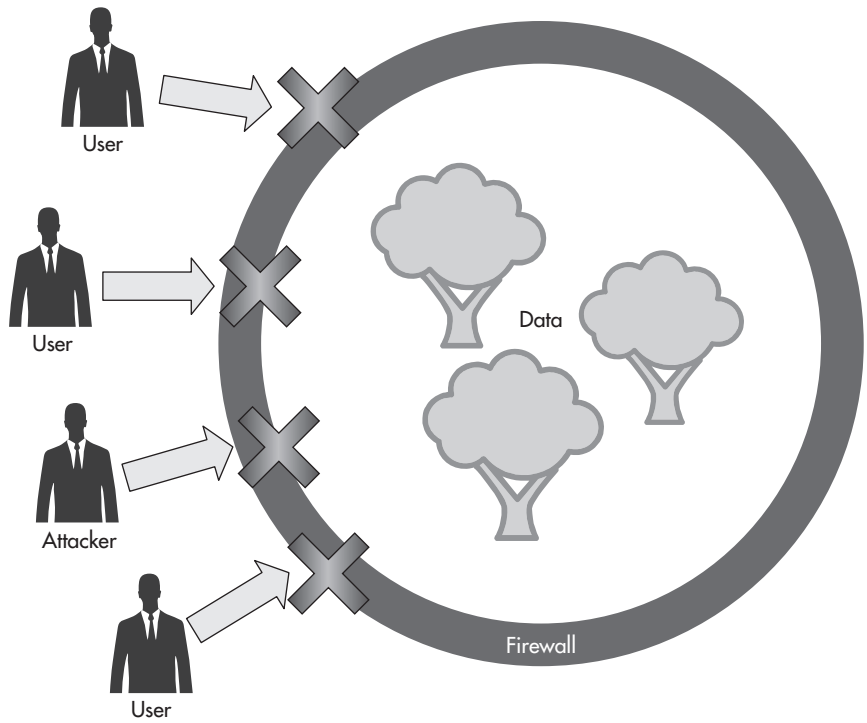


Figure 1-2 A firewall (lava moat) keeps attackers out, but keeps legitimate users out as well.

from them any more, but if he can get a highly privileged application to do it for him, then that's just as good. He may even be able to read the application source code (the wizard's scrolls) out of the file system.

The moral of the story is that it's necessary to use network-level defenses like firewalls to keep attackers out, but network-level defenses alone are not sufficient. You also need to weed out all the logic flaws in your web applications so that they can't be subverted.

Thinking like a Defender

The goal of this book is to help you prevent the logic flaws that lead to web application vulnerabilities, and we'll do this in two ways. First, we'll examine the code and configuration problems underlying specific web application vulnerabilities like cross-site scripting and SQL injection. It's crucial to be properly educated in defense techniques for these vulnerabilities, because you will need to put them to the test.

Note

A lot of people think that they're safe from attack because their company is too small to be noticed by attackers. Hackers only go after the big guys like Google and Microsoft, right? Think again: According to statistics from the IBM X-Force security research team, products from the top ten software vendors accounted for only 20 percent of reported vulnerabilities in 2010 (as seen in Table 1-1), and this number is down from 23 percent in 2009. Attackers are increasingly targeting the "long tail" of smaller organizations' web applications, so never think that you're too small to slip under their radar.

Rank	Vendor	Disclosure Frequency
1.	Apple	4.0%
2.	Microsoft	3.4%
3.	Adobe	2.4%
4.	Cisco	1.9%
5.	Oracle	1.7%
6.	Google	1.6%
7.	IBM	1.5%
8.	Mozilla	1.4%
9.	Linux	1.4%
10.	Sun	1.1%
N/A	All others	79.6%

Table 1-1 2010 First-Half Vulnerability Disclosure Rates per Vendor (IBM X-Force 2010 Mid-Year Trend and Risk Report)

However, as with firewalls, knowing how to defend against specific web application attacks is necessary but not sufficient by itself. Beyond just looking at specific attacks, we also want to educate you on larger, more general security principles.

This is important because attack methods change all the time. Attackers refine their methods, finding new ways to break into systems that were previously thought to be secure. Every year, some security researcher will present a paper at the BlackHat or DefCon security conference that negates a built-in browser or operating system defense that developers had come to rely on.

You need to be prepared not just for the attacks that are going to come today, but for the new attacks that are going to come tomorrow. You can do this not by thinking like an attacker (which you're not), but by learning to think like a defender (which you now are). This is why it's so important to learn the general security principles behind the specific defenses.

In Actual Practice

The way that a lot of security experts want you to solve this problem is for you to "think like an attacker." In their opinion, if you just think the way the attackers do, you'll be able to anticipate their moves and block them. What ridiculous advice! You're not an attacker—at least, I certainly hope you're not. If you want any degree of confidence in your results at all, it's just not possible for you to snap your fingers and start thinking like someone with years of experience in a completely different field of expertise.

To show what an unrealistic expectation this is, when I give presentations to groups of security professionals, I'll sometimes challenge them to think like a dentist. I'll tell them that my tooth hurts and ask what they plan to do for me. They'll take an X-ray, they say. "Fine," I reply, "what are you going to look for in the image?" They don't know. "Have you ever operated an X-ray machine before?" They haven't. "Are you sure you're not going to give me a lethal dose of radiation?" They're not. This could be a problem!

When you try to think like an attacker, it's likely that you'll not only be lulled into a false sense of security—thinking you've protected yourself when you really haven't—but there's also a good chance that you'll make matters even worse than they were before. Maybe we'll all be better off letting developers be developers, and letting security researchers be security researchers.

It's good to know how to appropriately encode HTML output to prevent cross-site scripting attacks, but it's even better to know that mixing code and data can lead the application to interpret user input as command instructions. Knowing this—and developing your applications with an eye to avoiding this—can help you prevent not just cross-site scripting, but SQL injection, buffer overflows, request header forgery, and many others. And there's a good chance it'll help you prevent whatever new attacks get dropped at DefCon next year. The methods may change from year to year, but the underlying principles will always remain the same.

The OWASP Top Ten List

We'll spend most of the rest of this book talking about web security vulnerabilities and principles, but just to whet your appetite for what's to come, let's start by getting familiar with the OWASP Top Ten List.

One of the most-respected authorities in the field of web application security is the organization *OWASP*, short for the Open Web Application Security Project. As its name implies, OWASP is an open-source project with the goal of improving web application security. (You can see a screenshot of the OWASP web site, www.owasp.org, in Figure 1-3.)

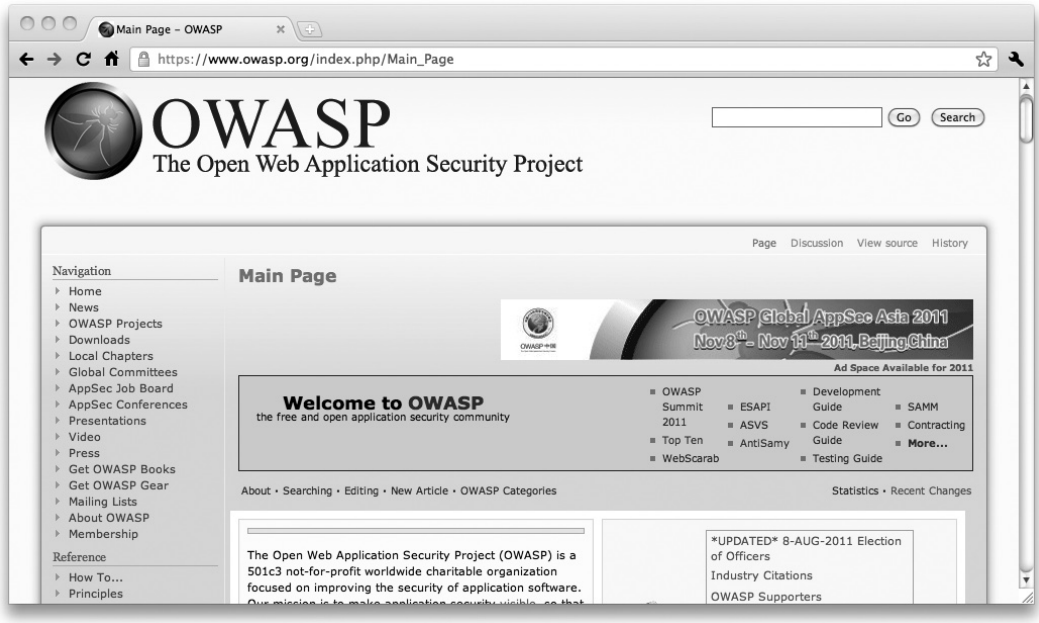


Figure 1-3 The OWASP web site www.owasp.org

OWASP is basically a loose coalition of individual contributors and sponsor companies who come together to contribute resources to the project. These resources include guidance documents to explain how to write more secure code, scanning tools to help you find vulnerabilities in your applications, and secure coding libraries you can use to prevent vulnerabilities from getting into your applications in the first place. But the best-known OWASP resource by far is its Top Ten List.

The OWASP Top Ten List of the Most Critical Web Application Security Risks is compiled from both objective and subjective data. OWASP sponsor organizations contribute objective data on the prevalence of different types of web application vulnerabilities: how many database attacks they've seen, how many browser attacks, and so on. OWASP-selected industry experts also contribute more subjective rankings of the severity or potential damage of these vulnerabilities.

As we mentioned earlier, web security risks change over time as new vulnerabilities are discovered (or invented). And it's not all doom and gloom; new defenses are developed every year too. New versions of application frameworks, web servers, operating systems, and web browsers all often add defensive technology to prevent vulnerabilities or limit the impact of a successful attack.

Tip

Built-in browser defenses can be a great help, but don't rely on them. It's very unusual to be in a situation where you can guarantee that all your users are using the exact same browser. Certainly this won't ever be the case if you have any public-facing web applications. And even if you're only developing web sites for use inside an organizational intranet where you can mandate a specific browser, it's likely that some users might configure their settings differently, inadvertently disabling the browser defenses. The bottom line here is that you should treat browser defenses as an unexpected bonus and not take them for granted. You are the one who needs to take responsibility for protecting your users. Don't count on them to do it for you.

Since web application vulnerability risks change, becoming comparatively more or less critical over time, the OWASP Top Ten List is periodically updated to reflect these changes. The first version of the list was created in 2004, then updated in 2007 and again in 2010 (its most recent version as of this writing). The list is ranked from most risk to least risk, so the #1 issue (injection) is considered to be a bigger problem than the #2 issue (cross-site scripting), which is a bigger problem than broken authentication and session management, and so on.

As of 2010, the current version of the OWASP Top Ten List is as described in the following sections.

Figure 1-4 An injection attack against an application's SQL database

#1. Injection

One of an attacker's primary goals is to find a way to run his own code on your web server. If he can do this, he may be able to read valuable confidential data stored in your databases or conscript it into a remote-controllable *botnet* of "zombie" machines. To accomplish this through a network-level attack, he might have to find a way to sneak an executable binary file through your firewall and run it. But with an application-level attack, he can accomplish his goal through much more subtle means.

A typical web application will pass user input to several other server-side applications for processing. For example, a search engine application will take the search text entered by the user, create a database query term from that text, and send that query to the database. However, unless you take special precautions to prevent it, an attacker may be able to input code that the application will then execute. In the example of the search engine, the attacker may enter database commands as his search text. The application then builds a query term from this text that includes the attacker's commands, and sends it to the database where it's executed. You can see a diagram of this attack in action in Figure 1-4.

This particular attack is called *SQL injection* and is the most widespread form of injection attack, but there are many others. We see injection vulnerabilities in XML parsing code (XPath/XQuery injection), LDAP lookups (LDAP injection), and in an especially dangerous case where user input is passed directly as a command-line parameter to an operating system shell (command injection).

#2. Cross-Site Scripting (XSS)

Cross-site scripting vulnerabilities are actually a specific type of injection vulnerability in which the attacker injects his own script code (such as JavaScript) or HTML into a vulnerable web page. At first glance, this may not seem like an incredibly critical vulnerability, but attackers have used cross-site scripting holes to steal victims' login passwords, set up phishing sites, and even to create self-replicating worms that spread throughout the target web site.

Cross-site scripting is dangerous not just because it can have such high-impact effects, but also because it's the most pervasive web application vulnerability. You're potentially creating cross-site scripting vulnerabilities whenever you accept input from a user and then display that input back to them—and this happens all the time. Think about blogs that let users write their own comments and replies to posts. Or collaborative wikis, which let the users themselves create the site content. Or even something as seemingly innocent as a search feature: if you display the user's search term back to them (for example, "Your search for 'pandas' found 2498 results"), you could be opening the door to cross-site scripting attacks.

#3. Broken Authentication and Session Management

Authentication and authorization are usually considered to be network-level defenses, but web applications add some unique new possibilities for attackers. When you use a web application, your browser communicates with the application web server by sending and receiving messages using the Hypertext Transfer Protocol (HTTP). HTTP is a stateless protocol, which means that the server does not "remember" who you are between requests. It treats every message you send to it as being completely independent and disconnected from every other message you send to it. But web applications almost always need to associate incoming messages with a particular user. Since the underlying HTTP protocol doesn't keep state, web applications are forced to implement their own state keeping methods.

Usually, the way they do this is to generate a unique token (a *session identifier*) for each user, associate that user's state data with the token value, and then send the token back to the user. Then, whenever the user makes a subsequent request to the web application, he includes his session identifier token along with the request. When the application gets this request, it sees that the request includes an identifier token and pulls the corresponding state data for that token into memory.

There's nothing inherently insecure with this design, but problems do come about because of insecure ways of implementing this design. For example, instead of using cryptographically strong random numbers for session identifiers, an application might be programmed to use incrementing integers. If you and I started sessions right after each other, my token value would be 1337 and yours would be 1338. It would be trivial for an attacker to alter his identifier token to different valid values and just walk through the list of everyone's sessions.

Another example of a poor state management implementation is when the application returns the session token as part of the page URL, like www.site.cxx/page?sessionid=12345. It's easy for a user to accidentally reveal this token. If a user copies and pastes the page URL

from her browser and posts it on a blog, not only is she posting a link to the page she was looking at but she's also posting her personal token, and now anyone who follows the link can impersonate her session.

#4. Insecure Direct Object References

There's usually no good reason for a web application to reveal any internal resource names such as data file names. When an attacker sees a web application displaying internal references in its URL, like the "datafile" parameter in the URL http://www.myapp.cxx/page?datafile=12345.txt, he'll certainly take the opportunity to change that parameter and see what other internal data he can get access to. He might set up an automated crawler to find all the datafiles in the system, from "1.txt" through "99999999.txt". Or he might get even sneakier and try to break out of the application's data directory entirely, by entering a datafile parameter like "../../../passwords.txt".

Note

Throughout this book, you'll see us use example URLs with a top-level domain of ".cxx", like "http://www.myapp.cxx". We do this because—as of this writing—there is no such real top-level domain ".cxx", so there's no chance that the example site actually exists. We don't want to accidentally name a real web site when we're talking about security vulnerabilities!

#5. Cross-Site Request Forgery

Cross-site request forgery (CSRF) attacks are another type of attack that takes advantage of the disconnected, stateless nature of HTTP. A web browser will automatically send any cookies it's holding for a web site back to that web site every time it makes a request there. This includes any active session identification or authentication token cookies it has for that site too.

By sending you a specially crafted e-mail message or by luring you to a malicious web site, it's very easy for an attacker to trick your browser into sending requests to any site on the Internet. The site receives the request, sees that the request includes your current session token, and assumes that you really did mean to send it.

The worst part about cross-site request forgery is that every site on the Internet that relies on cookies to identify its users—and there are millions of these sites—is vulnerable to this attack by default. You'll need to use additional measures beyond just session identification cookies to properly validate that incoming requests are legitimate and not forgeries.

#6. Security Misconfiguration

You can code your application with every security best practice there is, crossing every "t" and dotting every "i", but you can still end up with vulnerabilities if that application isn't properly configured. You'll often see these kinds of configuration vulnerabilities when development settings are accidentally carried over into production environments.

Web applications in particular are designed to be easy to deploy. Sometimes deployment is as simple as copying the files from the developer's machine to the production server. However, developers usually set their configuration settings to give them as much debugging information as possible, to make it easier for them to fix bugs. If a developer accidentally deploys his configuration settings files onto the server, then that whole treasure trove of internal data may now be visible to potential attackers. This may not be a vulnerability in and of itself, but it can make it much easier for the attacker to exploit any other vulnerabilities he may find on the system.

#7. Insecure Cryptographic Storage

Sensitive data like passwords should never be stored unencrypted in plaintext on the server. In fact, it's rarely necessary for passwords to be stored at all. Whenever you can, it's better to store a one-way cryptographic hash of a user's password rather than the password itself.

For example, instead of storing my password "CarrotCake143", a web application could just store the Secure Hash Algorithm (SHA-1) digest value of "CarrotCake143", which is a 40-character-long string of hexadecimal characters starting with "2d9b0". When I go to log in to this web application and give it my username and password, it computes a new SHA-1 hash from the password that I give it. If the new hash matches the old hash, it figures that I knew the correct password and it lets me in. If the hashes don't match, then I didn't know the password, and it doesn't let me in.

The benefit of this approach is that hash functions only work in one direction: it's easy to compute the hash of a string, but it's impossible to recompute the original string from the hash. Even if an attacker somehow manages to obtain the list of password hashes, he'll still have to take a brute-force approach to testing for an original value that matches my "2d9b0…" SHA-1 hash. On the other hand, if the application stores my password in plaintext and an attacker manages to get ahold of it in that unprotected form, then he's already won—and this is just one example of one misuse of one particular form of cryptography.

Tip

For an even better way to secure password hashes, you should add a random value (or "salt") to the plaintext password before computing its hash value. This approach has multiple benefits. First, in case the hash value is ever leaked, it makes an attacker's job of reverse-engineering the original password text from a pre-computed lookup table (or "rainbow table") much more difficult. (In Figure 1-5, you can see a screenshot of a web site offering rainbow tables for download, which can be used to crack Windows XP user accounts.)

And second, without salt values, whenever two users have the same password, they'll have the same password hash as well. Cracking just one user's password from a leak of the hash list could end up revealing account information for potentially hundreds or thousands of other users as well.

#8. Failure to Restrict URL Access

One way that web applications sometimes keep unauthorized users out of certain pages on the site is to selectively hide or display the links to those pages. For example, if you're the administrator for www.site.cxx, when you log in to the web site's home page, you might see a link for "Administration" that takes you to admin.site.cxx. But if I log in to www.site.cxx, I won't see that link since I'm not an authorized administrator there.

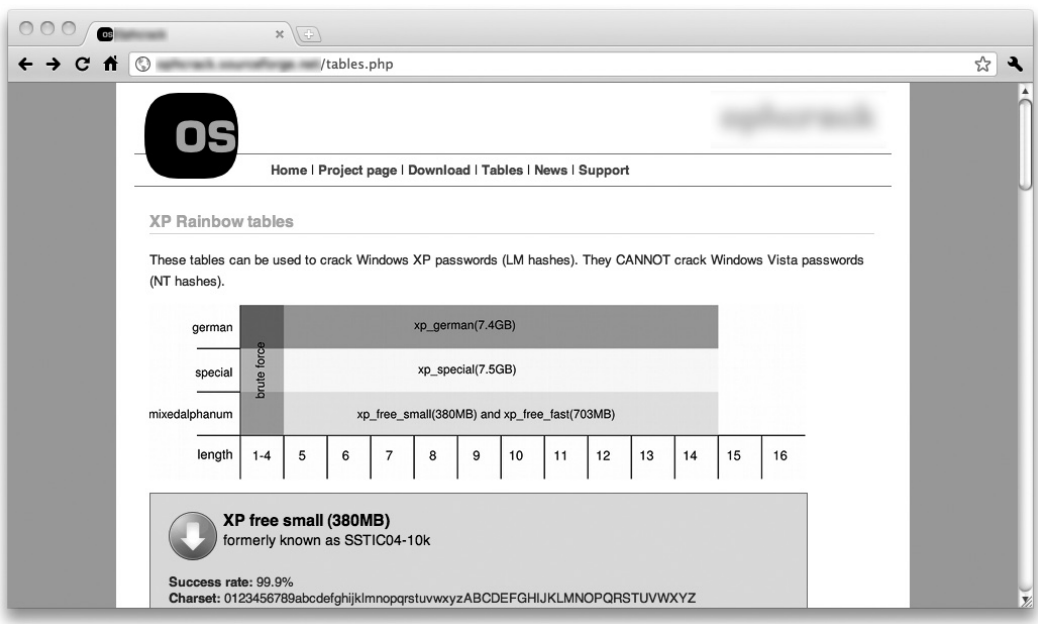


Figure 1-5 A web site offering Windows XP password rainbow tables for download

This design is fine as long as there's some other kind of authorization mechanism in place to prevent me from accessing the administration site. If the only thing keeping me out is the fact that I'm not supposed to know the site is there, that's not sufficient protection. If someone on the inside accidentally reveals the secret site, or if I just happen to guess it, then I'll be able to just get straight in.

#9. Insufficient Transport Layer Protection

Using Hypertext Transfer Protocol Secure (HTTPS) for your web site gives you many security benefits that regular vanilla HTTP does not. HTTPS uses either the Secure Sockets Layer (SSL) protocol or, better yet, the Transport Layer Security (TLS) protocol, which provides cryptographic defenses against eavesdropping attackers or "men-in-the-middle." SSL/TLS encrypts messages sent between the client and the web server, preventing eavesdroppers from reading the contents of those messages. But just preventing someone from reading your private messages isn't enough—you also need to make sure that nobody changes or tampers with the message data as well—so SSL/TLS also uses message authentication codes (MACs) to ensure that the messages haven't been modified in transit.

Finally, you need to know that the server you're sending a message to is actually the server you want. Otherwise, an attacker could still intercept your messages, claim to be that server, and get you to send "secure" messages straight to him. SSL/TLS can prevent this scenario as well, by supporting authentication of the server (and optionally the client) through the use of verified, trusted digital certificates.

Without these protections, secure communications across the Internet would basically be impossible. You'd never send your credit card number to a web site, since you'd never know who else might be listening in on the conversation.

Unfortunately, because HTTPS is slower than standard HTTP (and therefore more expensive since you need more servers to serve the same number of users), many web applications don't use HTTPS as thoroughly as they should. A classic example of this is when a web site only uses HTTPS to protect its login page. Now, protecting the login page is critical: Otherwise, an attacker could intercept the user's unencrypted password. But it's not enough just to protect that one message.

Assuming the user logs in successfully, the web site will return an authentication token to the user, usually in the form of a cookie. (Remember, HTTP is stateless.) If all of the subsequent pages that the user visits after he's authenticated are not also served over HTTPS, an attacker could read the authentication token out of the message and then start using it for himself, impersonating the legitimate user.

Note

While getting transport layer security right is a critical part of your application's security, it should be evident by now that it's not the only part of your application's security. As with firewalls, far too many people tend to put far too much trust in the little HTTPS lock icon in their browser. Take SQL injection, for example: if your site is vulnerable to a SQL injection attack, all that you'll get from using HTTPS is to create a secure channel that an attacker can use to exploit you.

#10. Unvalidated Redirects and Forwards

With web applications, it's often the most simple and seemingly innocent functions of the application that lead to surprisingly damaging vulnerabilities. This is certainly the case with OWASP #10, Unvalidated Redirects and Forwards (usually just referred to as *open redirect* vulnerabilities).

Let's say that you open your browser and browse to the page www.site.cxx/myaccount. This page is only accessible to authenticated users, so the application first redirects you to a login page, www.site.cxx/login. But once you've logged in, the site wants to send you to the myaccount page that you originally tried to go to. So when it redirects you to the login page, it keeps that original page you asked for as a parameter in the URL, like www.site.cxx/login?page=myaccount. After you successfully pass the login challenge, the application reads the parameter from the URL and redirects you there.

Again, it sounds very simple and innocent. But suppose an attacker were to send you a link to www.site.cxx/login?page=www.evilsite.cxx? You might follow the link and log in without noticing where the page was redirecting you to. And if the site www.evilsite.cxx was set up as a phishing site to impersonate the real www.site.cxx, you might keep using evilsite without realizing that you're now getting phished.

Wrapping Up the OWASP Top Ten

You shouldn't worry if you're unfamiliar with some of the vulnerabilities in the Top Ten list or even all of them. We'll cover all of these vulnerabilities and others in detail over the course of this book, starting with the very basic principles of the attack: Which targets is the attacker trying to compromise? What does he want to accomplish? What am I doing that allows him to do this? And most importantly: What can I do to stop him?

And again, remember that each of these vulnerabilities is just a symptom of a larger, more general security issue. Our real goal is to educate you on these larger principles. We don't just want to "give you a fish" and tell you about the OWASP Top Ten, we want to "teach you to fish" so that if OWASP expands their list next year to be a Top 20 or Top 100, you'll already have your applications covered.

Secure Features, Not Just Security Features

Just as the IT professionals we talked about at the beginning of the chapter had some misconceptions about network security defenses versus application security defenses, developers also often have some mistaken beliefs concerning security. Next time you pass a developer in the hallway, stop him and ask him what he knows about security. He'll probably answer with some information about firewalls, antivirus, or SSL. If he's a Neal Stephenson fan, maybe he'll corner you and start ranting on the inherent superiority of the Blowfish cryptography algorithm over the Advanced Encryption Standard algorithm. (If this happens to you, we apologize for getting you in this situation.)

And there's nothing wrong with any of this—firewalls, antivirus, SSL, and cryptography are all important security features. But there's a lot more to creating secure web applications than just knowing about security features. It's actually much more important to know how to apply security to the routine development tasks that programmers tackle every day, like parsing strings or querying databases. In short, it's more important to know how to write *secure* features than it is to know how to write *security* features.

Look back at the OWASP Top Ten one more time. It's telling that for the majority of these vulnerabilities, the way that you solve the problem is usually found in a secure coding technique rather than in the application of a security feature. This is especially true when you look at the earlier, more critical vulnerabilities on the list. Of the top six, only one (#3, Broken Authentication and Session Management) can be attributed to misuse of a security feature. The rest are all caused by improperly coding the "normal," everyday features that make up the majority of the work that applications perform.

IMHO

It's disappointing to me that so many people think of security as just being security features. If you go to your local bookstore and randomly pick a book from the computing section, that book will probably have one short chapter on security, and 99 percent of that chapter will cover authentication and authorization methods. I've even seen entire books titled something like "Web Security" that only covered authentication and authorization.

We're certainly showing our bias here regarding the value of secure features versus security features. But don't take that to mean that security features are unimportant. If you don't implement appropriate authentication and authorization checks, or if you use easily crackable homegrown cryptography, your users' data will be stolen and they won't be happy about it. They won't care whether it was a cross-site scripting vulnerability or improper use of SSL that led to their credit card being hijacked. They probably won't even

understand the difference. All they'll know is that they were hacked, and you're the one responsible. So cover all your bases, both secure features and security features.

Final Thoughts

We'll meet up with our friend the wizard again at the end of the book to see what he's learned to make his magic fruit orchard a safer place. Of course, we know that the wizard is wise enough not to test out his new spells on anyone's trees except his own. This goes for you too. Virtually all of the attack techniques we'll be describing are illegal for you to test against any web site, unless you own that site yourself or have explicit permission from the owner.

We've Covered

Misplaced priorities and the need for a new focus

- Seventy percent of attacks come in through a site's web applications.
- Spending money on network firewalls isn't going to help this problem.

Network security versus application security: The parable of the wizard and the magic fruit trees

- Web applications are like giants: they're very powerful, but not very smart.

Thinking like a defender

- Application-level attacks are caused by logic flaws in your application.
- You need to find and fix these flaws to be secure.
- You're not going to do this by pretending to "think like an attacker."
- But you can do this by learning security principles and starting to think like a defender.

The OWASP Top Ten List

- The Open Web Application Security Project (OWASP) organization periodically publishes a list of the current top ten most critical web application vulnerabilities.
- This list is very widely referenced, and you should become familiar with the vulnerabilities and the underlying causes.

Secure features, not just security features

- It's important to know how to write everyday application functionality in a secure manner, not just how to use special security features like cryptography and SSL.