# Web Application Security Assessment Tools

Security testing a Web application or Web site requires careful thought and planning due to both tool and industry immaturity. Finding the right tools involves several steps, including analyzing the development environment and process, business needs, and the Web application's complexity.

MARK CURPHEY AND RUDOLPH ARAUJO
*Foundstone*

The security industry is undergoing a significant shift in how it perceives its own problem space. A few years ago, if you asked the average corporate information security (IS) department employee to describe his or her primary job functions, the answer would likely include the phrase "network security." In today's distributed architectures, however, applications actually drive most of the security decisions, such as whether to allow a particular customer or partner access to particular data. As a result, the IS industry is having to adapt fast and shift its focus from a network-centric world view to application and software security. Nowhere is this happening faster than with Web-based technologies, giving rise to a demand for technologies that automate the search for security holes in Web applications and Web services. In response, an array of technologies, approaches, and companies have emerged, making it difficult for developers to separate tool facts from marketing hype.

Here, we describe the different technology types for analyzing Web applications and Web services for security vulnerabilities, along with each type's advantages and disadvantages. At Foundstone, we work with some of the world's biggest banks and telecommunications companies to identify and resolve security issues. Together with our clients, we face challenging testing scenarios in the context of demanding applications and complex business environments. We've seen firsthand what works and what doesn't; what's marketing hype and what gets results. Our analysis here is based on our collective experiences and the lessons we've learned along the way.

## A Web security vulnerabilities framework

To choose the appropriate tool, it helps to understand a typical Web site and its security vulnerabilities. As Figure 1 shows, the average real-world Web site bears little resemblance to the average Web site diagram that you'd find in a textbook.

Most Web sites are complex systems that integrate and exchange data with other systems and store and process data in many different places. They typically have administrative interfaces that let users manage their accounts and provide transaction-approval components. Many also link to order processing and inventory systems and management-reporting applications. In short, the customer's user interface is usually only a part of the system; from a security perspective, and perhaps more important, it's not always the most interesting part. Year after year, computer security surveys consistently report that insiders with some legitimate system access launch most system attacks. Nevertheless, most people still spend their security resources knocking on the front door, while the back doors and windows might be wide open.

### Threat modeling

At Foundstone, we recommend that clients use threat modeling to begin assessing their security risks. Threat modeling has become popular recently in part due to Mike Howard and David LeBlanc's excellent book *Writing Secure Code*.[1] Whether you use their Microsoft-based approach or another threat-modeling approach, the

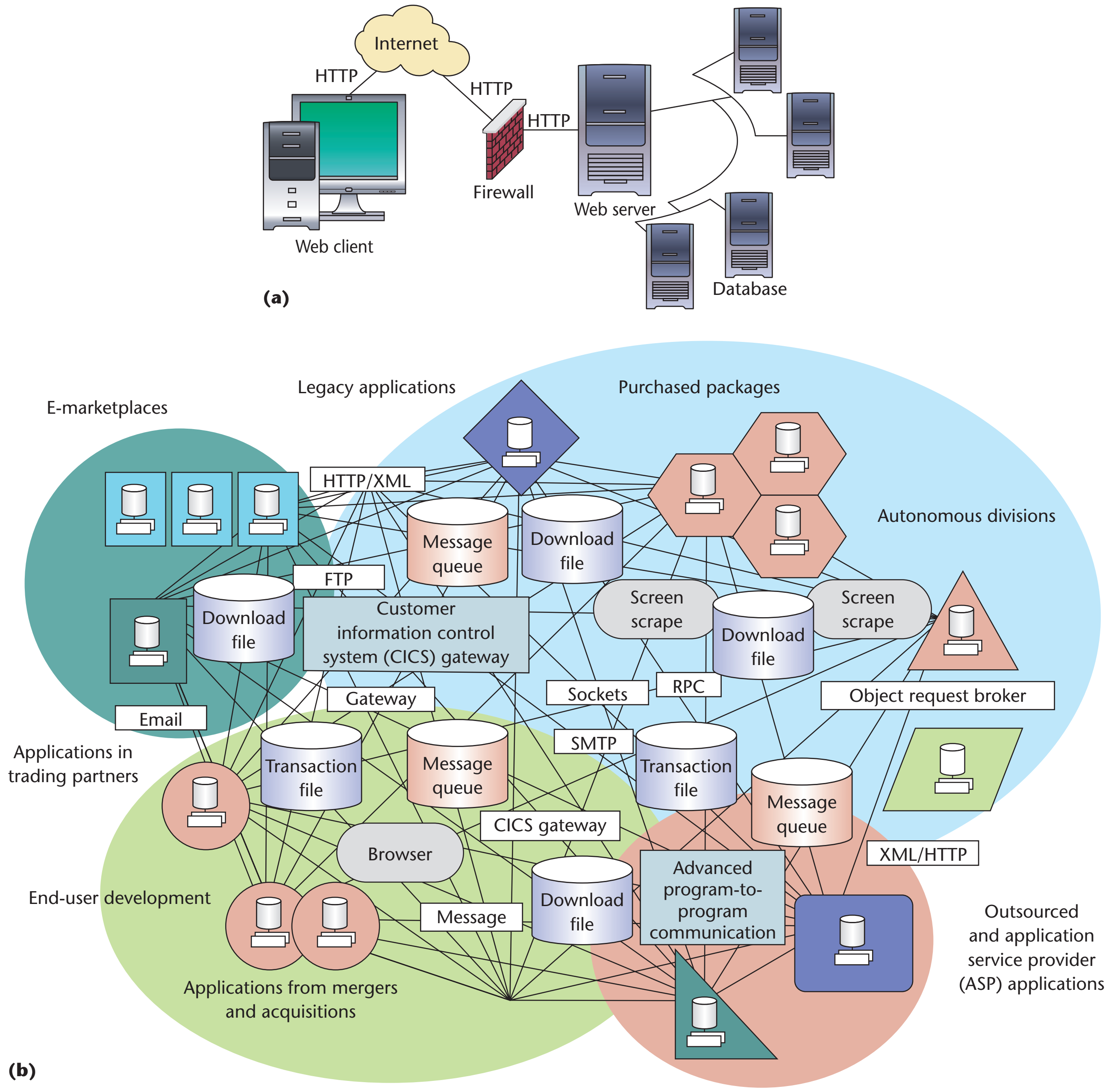

Figure 1. Two views of the Web site environment. (a) The idealized environment is characterized by simple core components and textbook technology and design choices. (b) In the real world, life is more complex. Systems grow organically, compromises are made, and ecosystems form with complex interdependencies.

technique guides you through the process of defining system components, entry and exit points (connectivity), and key security components and mechanisms. In so doing, you get a clear architectural system overview—and avoid the often knee-jerk impulse to simply analyze the customer interface. As the chief security officer of a major financial service company often puts it, "Would you drive a car that was only front-impact tested? Of course you wouldn't! So why only test the front impact of your Web applications?"[2]

## Defining vulnerabilities

Another thing to consider is vulnerability types. To this end, it's helpful to mentally partition security defects into two high-level categories: implementation bugs and design flaws. If you're like us, you might initially complain about this taxonomy's simplicity, but, assuming you use well-reasoned arguments, it eventually proves highly useful. An implementation bug example is a buffer overflow caused when a developer mistakenly uses the `strcpy` function incorrectly. In contrast, a design flaw might be an unprotected user credential store (a password file that is stored in the clear) or an authorization system that makes ineffective checks. Once you begin thinking about it, you quickly realize that all issues fall into one category or the other.

**A breadth of possible security vulnerabilities must be addressed, and no one tool can effectively analyze them all.**

### Creating a security framework

Application security vulnerabilities also fit a common issues taxonomy that you can use to build an ordered set of test cases. Such a security framework is helpful for analysis and reporting because it promotes consistency and order. It does this by ensuring that you're comparing apples to apples and oranges to oranges when you analyze vulnerability classes and applications. The following security framework also scales beyond Web applications to desktop and client-server type applications:

- *Configuration management*—infrastructure issues, including the configuration of the Web server, application server, and runtime technology (.NET Common Language Runtime or Java Virtual Machine).
- *Authentication*—user and entity authentication and process issues.
- *Authorization*—issues related to accurately and effectively making access control decisions.
- *Data protection*—issues related to how the system protects sensitive data when it's stored or transmitted between system components or across the Internet from the user to the system.
- *User and session management*—issues related to how the system manages users through such things as password reset mechanisms and transactions.
- *Data validation*—problems arising when the user or system components accept or result in malicious or incorrect data.
- *Error handling and exception management*—issues related to how the system handles security-related errors and exceptions.
- *Auditing and event logging*—issues resulting from auditing and event-logging mechanisms (or the lack thereof).

As this framework shows, a breadth of possible security vulnerabilities must be addressed, and no one tool can effectively analyze them all. Some issues, such as configuration management, are typically introduced at deployment and only visible during runtime or through technology with unabridged infrastructure access. Other issues, such as authorization, require a detailed understanding of the system's correct and incorrect behavior. Given this, it's extremely difficult to determine accurate and repeatable results in an automated manner. Although some tools might help you discover that your Web server lacks security patches (configuration management), they'd likely be unable to determine if that server is on a shared host that hosts a high-profile attack target. In England, the phrase "horses for courses" means that different racehorses are better on certain race courses. The same is true for selecting tools: you must match the tools to the Web site environment.

### Tool requirements

To decide what your best course is, consider the following questions.

*Do you own or have access to the code?* When you're testing an application, you should have access to its source code, if possible. Source-code access lets you look under the covers and truly discern how the application works. It also lets you employ tools that analyze the application's DNA and monitor its runtime environment. That said, many security consultants won't use or even ask for readily available source code, perhaps due to either pseudomasochism or a "cool hacker" complex. The sad fact is, however, that security experts and auditors in most information security organizations and consulting firms can't read or write code. They're more at ease with tools like nmap than with modern development languages like C# and Java. As Winston Churchill once said, however, "The short road to ruin is to emulate the methods of your adversary." Why not use the biggest advantage you have over an attacker?

*If the software is delivered or installed as a compiled runtime application, does the license explicitly allow reverse engineering?* If not, the Digital Copyright Millennium Act probably prohibits reverse engineering, and thus you might not be able to legally run binary or runtime analysis or use decompilers. Of course, laws rarely stop hackers or criminals.

*Can you influence the development process?* If so, find bugs early in the software development life cycle (SDLC) so you can influence the implementation (see the "When to test" sidebar). To do this, you should consider how your toolset integrates into the development life cycle. Don't be fooled by so-called developer editions of some tools, which claim to mesh with integrated development environments like Visual Studio and Eclipse. In reality, they're little more than a menu button to open the main application.

*Who will do the security testing?* Developers understand code and are familiar with development tools. Penetration testers typically understand protocols like HTTP. In our experience, giving developers tools such as protocol fuzzers doesn't yield good results, nor does giving penetration testers code review tools. People tackle testing from different mindsets. Empowering individuals with the right tools for their jobs yields the best results.

*What technologies and languages does your organization use?* Every application and framework has its idiosyncrasies and particular security best practices. When considering applications built and deployed using Microsoft .NET, for example, the security concerns surrounding trust levels are critical, whereas on J2EE-based solutions, one of the main considerations might be the `web.xml` file's contents and configuration.

## Tool categories and analysis

To help with your tool selection, we offer a high-level view of eight tool categories: source code analyzers, Web application (black-box) scanners, database scanners, binary analysis tools, runtime analysis tools, configuration management tools, HTTP proxies, and miscellaneous tools.

Table 1 shows example tools in each category, along with the SDLC phase they're best applied in and the skill levels needed to use them effectively. However, it's important to note that, as with all tools, effectiveness often depends on the operator's skill. Finally, be warned: our analysis is likely to differ significantly from vendors' marketing campaigns.

### Source-code analyzers

Generally, source-code analysis tools fall into two categories: static and dynamic. Basic *static analyzers* run simple text-based searches for strings and patterns in source-code files, recursively analyzing the code base for security defects and then generating a report. More modern static-analysis tools trace the data's path through code to provide a more complete and accurate analysis.

Static code analyzers have been around a long time; several established open-source tools now exist, including FlawFinder (www.dwheeler.com/flawfinder) and Rough Auditing Tool for Security (RATS; https://securesoftware.custhelp.com/cgi-bin/securesoftware.cfg/php/enduser/doc_serve.php?2=Security). Because they require very little processing, such tools are generally fast. However, they're severely limited in what they actually find and are prone to reporting false positives. Take, for example, the infamous `strcpy` function case. A basic static code analysis tool will flag all uses of `strcpy` and report each instance as a potential buffer overflow. So, while these simple tools can be quite useful for finding potential issues, they require significant manual intervention to determine actual risks. Most commercial static analysis tools perform dataflow analysis and thus partially reduce obvious false positives.

## When to test

In 1996, Capers Jones showed that, if the unit cost of finding a bug during development were US$1, failing to find the same bug until deployment would cost $16,000 (see Figure 1).[1]

In 2000, an IBM System Sciences Institute survey found similar results. Such results offer a clear and dramatic message: the earlier you conduct your security testing, the cheaper the cost. Nonetheless, companies today typically spend most of their security-testing dollars after all the application's code is written—sometimes waiting until just a few weeks before the go-live date. Indeed, many spend their money after they've deployed applications. In our experience, and as Figure A illustrates, most flaws are actually introduced earlier in the software development life cycle (SDLC), during the requirements and design phases.

Before considering a particular tool's merits, companies would thus be well advised to reconsider their testing and how it fits into their SDLC. Companies that test only after they've built the application will likely get more "bang for the buck" if they test earlier in the SDLC. The strategic approach of "design it out and build it in" pays significant dividends for companies both logically and economically.

**Reference**

1. C. Jones, *Applied Software Measurements*, McGraw-Hill, 1996.

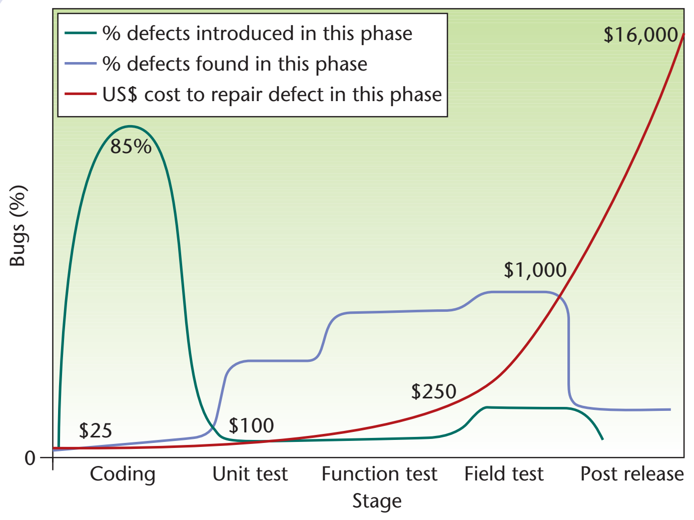

Figure A. The cost of locating bugs by software development life-cycle phase. Clearly, the earlier companies run security tests, the lower their security costs.

*Dynamic analyzers* attempt a far deeper analytical approach with source code. Essentially, they not only find instances of bad coding practice, but also attempt rich control and dataflow traversal. Dynamic analyzers first attempt to construct all possible runtime functional call stacks. Next, they attempt to determine if a call (to `strcpy`, for example) can be reached by data received as an input from the user or the environment. This helps

**Table 1. Tool overview.**

| TOOL TYPE | EXAMPLE TOOLS | LIFE-CYCLE PHASE | REQUIRED SKILL LEVEL |
|---|---|---|---|
| Source-code analyzers | **Commercial:**<br>Secure Software CodeAssure (www.securesoftware.com/products)<br>Ounce Labs Prexis (www.ouncelabs.com/prexis_engine.html)<br>Fortify Software Source Code Analysis Suite (www.fortifysoftware.com/products/sca.jsp)<br>Klocwork K7 (www.klocwork.com/products/klocworkk7.asp)<br>Coverity Prevent/Extend (www.coverity.com/products/index.html)<br>Compuware DevPartner SecurityChecker (www.compuware.com/products/devpartner/securitychecker.htm)<br>**Free/Open Source:**<br>Rough Auditing Tool for Security (RATS) (https://securesoftware.custhelp.com/cgi-bin/securesoftware.cfg/php/enduser/doc_serve.php?2=Security)<br>FlawFinder (www.dwheeler.com/flawfinder)<br>FindBugs http://findbugs.sourceforge.net/ | Development*<br>Testing<br>Predeployment | A senior developer or architect should lead the scanning effort and determine remediation strategies for any potential threats discovered in the code. |
| Web application (black-box) scanners | **Commercial:**<br>SPI Dynamics WebInspect (www.spidynamics.com/products/index.html)<br>Sanctum AppScan (www.watchfire.com/securityzone/product/appscansix.aspx)<br>**Free/open source:**<br>OWASP WebScarab (www.owasp.org/software/webscarab.html)<br>OWASP Berretta (www.devcafe.co.uk/beretta/index.htm)<br>Nikto (www.cirt.net/code/nikto.shtml)<br>Wikto (www.sensepost.com/research/wikto)<br>EOR (www.sensepost.com/research/eor)<br>Spike (www.immunitysec.com/resources-freesoftware.shtml) | Predeployment<br>Postdeployment | Typically most useful to security auditors rather than developers. While usually point-and-click, these tools often have intrusive checks that could crash or cause a denial of service. It's critical that operators tread lightly. |
| Database scanners | **Commercial:**<br>Application Security Inc. AppDetective (www.appsecinc.com/products/appdetective/index.shtml)<br>**Free/open source:**<br>MetaCortex (www.securityforest.com/wiki/index.php/Category:Enumeration) | Testing<br>Predeployment<br>Postdeployment | Typically require the database administrator or developer to run because many of their findings pertain to database configuration and hence require access beyond the data. |
| Binary analysis tools | **Commercial:**<br>BugScan with IDAPro (www.logiclibrary.com)<br>**Free/open source:**<br>FxCop (www.gotdotnet.com/team/fxcop/)<br>BugScam (www.sourceforge.net/projects/bugscam/) | Testing<br>Predeployment<br>Postdeployment | Although easy to use, understanding the tools' results require the highest skill level of any in our list; to be effective, they require a skilled hacker or security professional with in-depth knowledge of assembly programming and the underlying hardware. |

dramatically reduce the false-positive rate. However, these tools tend to be slow and performance intensive.

A relatively newer class of dynamic analysis tools integrates with debuggers and, like performance profilers, they can help detect potential security problems as they occur. These analysis tools are particularly useful in detecting memory-related problems—such as stack and heap overflows—and have been adapted to detect problems such as SQL injection. However, in the latter case, the analysis is contingent on specific API usage, and hence architectures that include such things as data-access layers often result in false negatives.

Table 1. Tool overview (cont.)

| TOOL TYPE | EXAMPLE TOOLS | LIFE-CYCLE PHASE | REQUIRED SKILL LEVEL |
|---|---|---|---|
| Runtime analysis tools | **Commercial:**<br>Compuware BoundsChecker (www.compuware.com/products/devpartner/studio.htm)<br>**Free/open source:**<br>Foundstone .NETMon (www.foundstone.com/resources/proddesc/dotnetmon.htm)<br>CLR Profiler (http://msdn.microsoft.com/netframework/downloads/tools/default.aspx)<br>NProf (http://nprof.sourceforge.net) | Development<br>Testing<br>Predeployment<br>Postdeployment | Because many of these tools aim to make it easy to find run-time flaws, most people can use use them. |
| Configuration analysis tools | **Commercial:**<br>Desaware CAS/Tester (www.desaware.com/products/castester/index.aspx)<br>**Free/Open Source:**<br>Foundstone SSLDigger (www.foundstone.com/resources/proddesc/ssldigger.htm)<br>PermCalc (http://msdn2.microsoft.com/ms165077.aspx) | Development<br>Testing<br>Predeployment<br>Postdeployment | While using these tools isn't difficult, analyzing their results and implementing mitigation strategies often require a developer and a system administrator |
| Proxies tools | **Free/Open Source:**<br>Paros (www.parosproxy.org/index.shtml)<br>Fiddler (www.fiddlertool.com)<br>OWASP WebScarab (www.owasp.org/software/webscarab.html)<br>BURP Proxy (www.portswigger.net/proxy/) | Testing<br>Predeployment<br>Postdeployment | These tools typically require minimal configuration and can be thrown into use almost immediately. However, effectively using Web proxies requires in-depth knowledge of Web application hacking techniques, making these tools more useful for security testers than for developers. |
| Miscellaneous tools | **Commercial:**<br>Visual Studio Team System (stress and Web UI testing tools) (http://msdn.microsoft.com/vstudio/teamsystem/default.aspx)<br>**Free/open source:**<br>Brutus (www.hoobie.net/brutus)<br>Cain & Abel (www.oxid.it/cain.html)<br>Firefox Toolbar (https://addons.mozilla.org/extensions/moreinfo.php?id=60)<br>Internet Explorer Developer Toolbar (www.microsoft.com/downloads/details.aspx?FamilyID=e59c3964-672d-4511-bb3e-2d5e1db91038&DisplayLang=en)<br>Foundstone WS-Digger (www.foundstone.com/resources/proddesc/wsdigger.htm)<br>Foundstone SiteDigger (www.foundstone.com/resources/proddesc/sitedigger.htm)<br>PREfast (www.microsoft.com/whdc/devtools/tools/PREfast.mspx)<br>NUnit (www.nunit.org)<br>JUnit (www.junit.org)<br>HTTPUnit (http://httpunit.sourceforge.net) | Development<br>Predeployment<br>Postdeployment | These tools vary in their ease of use. Some, such as Brutus, are point-and-click, while others, such as unit-testing frameworks, require significant development effort. |

* Several development organizations have found these tools to be most effective when integrated into the source-code control system, so that code is reviewed before it's checked in.

***Benefits.*** Code analysis tools operate on source code and are therefore familiar to developers. They integrate tightly into the development process and early in the construction process, including build processes and defect-tracking systems. The tools also pinpoint where in the code developers can fix the target issues.

***Drawbacks.*** In our experience, these tools offer reasonable value for traditional languages such as C and C++, but are less useful for managed languages such as .NET and Java. This is primarily because the technology surrounding C/C++ code analysis is relatively mature. Also, common C and C++ vulnerabilities such as buffer overflows are well understood and easy to scan for. In contrast, with managed languages, such "basic" threats are no longer a factor. In languages such as C# and Java, security problems—including complex authorization flaws, ineffective and insecure exception handling, and cross-site scripting—are far more difficult to scan for in an automated, accurate, and effective way.

**Because these tools are not aimed at developers, there's often a disconnect here, making mitigation much more complex.**

To analyze most commercial tools, you must compile the entire application. On large projects in today's distributed development environments, this can be difficult, if not impossible. As for the two main defect types—bugs and flaws—code analyzers (like other automated tools) are generally ineffective at finding the latter. Also, while they're generally highly useful at finding bugs on any given line of code, they're less effective at finding bugs that span multiple code lines and close to ineffective at finding issues instantiated across multiple functions or files.

### Web application (black-box) scanners

Web application scanners—also known as *black-box* scanners—use a Web browser to mimic an attack. There are two basic scanner types: those that look for specific URLs (such as administrative URLs) or known vulnerable CGIs, and those that follow all of a Web site's links and then run specific security test cases. Commercial scanners combine both methods. In any case, Web application scanners typically record a good HTTP transaction, and then attempt to inject malicious payloads into subsequent transactions and watch for indications of success in the resulting HTTP response.

***Benefits.*** Black-box scanners require little to no skill, and can catch low-hanging fruit, such as basic SQL injection and cross-site scripting. They're also effective at finding many configuration management issues, especially those related to Web servers.

***Drawbacks.*** Generally, these tools perform poorly. Our testing indicates that, on average, they find less than 5 percent of a typical Web site's vulnerabilities. There are several reasons for this. First, static Web sites (brochure sites built from HTML) are designed to be spidered; that is, they're designed so that search engines can easily index content. In contrast, most Web applications consist of complex forms—such as wizards or open text boxes—that require human users to enter contextually relevant information. So, the tools' first hurdle (which they fail) is site coverage, because client-side navigation technologies—such as JavaScript, AJAX, Flash, and image maps—make it difficult to determine where a user might go. Without coverage, tools can't build and execute suitable test cases.

Second, because the tools operate on HTTP streams, they can analyze only built and compiled applications deployed in test or production environments. As a result, the tools find problems far too late in the SDLC to be cost-effective. Also, the tools typically look only at the "front door," neglecting a given application's specifics in favor of building a generic test suite based on an "average" application.

Finally, if you find an issue using such a tool, you typically have no idea where to fix it in the code without further investigation by the developers. Because these tools are not aimed at developers, there's often a disconnect here, making mitigation much more complex.

### Database scanners

Database scanners typically act as an SQL client and perform various database queries to analyze the database's security configuration. Commercial tools let users operate in a penetration-testing mode, which doesn't require a valid administrative account, and an audit mode, which lets users run in the database administrator's context. The tools often scan the database configuration itself to determine common problems, such as loosely permissioned and powerful stored procedures, such as the infamous `xp_cmdshell`. They also verify database users and role memberships against known best practices.

***Benefits.*** Database scanners are easy to use, and they do a good job of assessing configuration management issues in databases.

***Drawbacks.*** Database scanners are usually completely divorced from an application's source code and therefore rarely find problems outside database configuration. The tools can look for vulnerabilities related to missing database patches and database user management, authentication, and authorization, but they won't, for example, review code within stored procedures.

In a sense, these tools are barking up the wrong tree. While they make sense from a defense-in-depth strategy, most networks maintain the database on a protected internal network that's not directly accessible. Hence, most database-related bugs, such as SQL injection, simply result from a lack of data validation in the application or data-access tier, rather than in the database itself. Furthermore, even when these bugs do exist in the application, their impact only grows in severity when the application operates in a high-privilege environment (such as connecting to the database as the database administrator). Thus, if you follow the least-privilege principles in your application, you'll often find that the database becomes a less attractive attack target.

Finally, these tools typically have little understanding of the database data's semantics. They therefore provide little value in identifying and recommending data protection techniques for sensitive data.

### Binary analysis tools

Binary analysis tools are also predominantly useful for applications and libraries created in C and C++. Essentially, these tools first determine the public interface of the application or library, and they're typically more adept at operating against APIs rather than user interfaces. The tools next attempt to "fuzz" the input parameters, looking for signs of an application crash, common vulnerability signatures, and other improper behavior. This fuzzing process basically involves feeding the application numerous input combinations, focusing on boundary conditions among other tests. Such tools have also been extended to support testing over network API and sockets using a similar approach.

Another class in this broad category includes those tools—such as Microsoft's FxCop—that use reflection and introspection to find security-related problems in application binaries. These tools are typically the most adept at checking for declarative security-related vulnerabilities, such as the lack of code access security demands or the use of potentially dangerous constructs, such as `Asserts` and `LinkDemands` in the .NET framework. Obviously, such tools are usually successful only against code written in managed languages.

***Benefits.*** Binary analysis tools can be fairly easy to use, requiring little configuration or in-depth knowledge. The tools can be effective at finding low-hanging fruit, such as high-level buffer overflow problems with no access to source code.

***Drawbacks.*** Although easy to use, these tools produce very complex results; a highly skilled analyst is required to understand and use the results effectively. The tools have an extremely high false-positive rate, and their numerous tests generate large amounts of data. Thus, to be of value, the tools require significant human intervention. Furthermore, binary analysis tools are performance intensive and can overwhelm both the application and the underlying machine. Moreover, they're not well suited to automated testing because they can crash applications, which then requires manual intervention to restart the application services (if not the whole computer). Given the drawbacks, technology in this category should really be considered research tools.

### Runtime analysis tools

The relatively new runtime analysis tools operate much like dynamic source analysis. However, unlike source analysis tools, runtime analysis tools don't attempt to determine what's wrong with an application, but rather act as an aid to manual code review and testing. Thus, instead of identifying application bugs, these tools give reviewers and testers a variety of critical information, then let them make the decision themselves. Among the information they return are calls to specific security-related authentication and authorization API- and framework-based data validation functions. They also provide information on application control flows and data flows.

Runtime analysis tools essentially act like profilers and intercept function calls as they occur. They can also be configured to log parameter values, as well as record two different sessions—say, one as an administrator and the other as regular user—and then compare the two.

***Benefits.*** Runtime analysis tools are useful for comparing sessions belonging to users at the same and different levels to find both horizontal and vertical privilege escalation. They can also be a tremendous aid to developers reviewing code and testers performing black-box component testing.

***Drawbacks.*** These tools are relatively new and thus immature for Web applications. Also, they're not very useful by themselves, and require experienced reviewers to leverage the information they provide during the testing phase.

### Configuration analysis tools

This class of tools usually performs static analysis. Rather than examining code, however, the tools typically operate against the application configuration files, host settings, or Web/application server configuration. Configuration analysis tools are therefore most useful for eval-

**The tools can be effective at finding low-hanging fruit, such as high-level buffer overflow problems with no access to source code.**

uating deployment security and ensuring that the application operates in its desired security context. For example, while Secure Sockets Layer (SSL) isn't extremely difficult to configure, many of the cipher suites it supports are known to be insecure and should therefore be disabled. However, many administrators and developers either don't know this, don't know the insecure suites are enabled, or don't know how to disable them.

Similarly, these tools can also review your `web.xml`, `machine.config`, and `web.config` for problems such as enabling verbose error messages or traces, code access security misconfigurations, and hard-coded credentials stored in the clear.

***Benefits.*** Configuration analysis tools are easy to configure and run. They can identify many areas of concern that analysts can easily tweak to the right settings with little to no impact.

***Drawbacks.*** These tools are disconnected from the code, and are typically useful in production environments rather than on a developer's machine. Configuration analysis tools also require that testing be replicated on every machine hosting the application. Unlike code fixes, configuration file fixes often must be manually replicated.

### Proxies

Security analysts typically use Web proxies to intercept Web traffic dynamically. Proxies sit between the tester's Web browser and the Web server hosting the application. Most Web proxies will let you trap the HTTP request (after it leaves the browser) and response (before it returns to the browser). Much like debugging breakpoints, when a request or response is trapped, the tools let testers view and modify different parts of the request, ranging from cookies, HTTP headers, GET and POST parameters, and HTML content. Obviously, proxies also let you effectively bypass any form of client-side validation or JavaScript-based enforcement. You can accomplish the latter by deleting the JavaScript code entirely from the response or modifying "validated" form field values after they leave the browser.

***Benefits.*** Proxies are useful for testing sites that operate both under SSL and without data protection. They're extremely useful for testing the effectiveness of server-side versus client-side security measures and for bypassing client-side JavaScript. Finally, proxies let you record and replay transactions to test the effectiveness of measures such as session expiry and replay protection.

***Drawbacks.*** Proxies can often be difficult to use, especially in situations in which the browser already sits behind a corporate HTTP proxy, thus requiring your tool to support proxy chaining (which some proxies don't). This problem can be further complicated if your organization uses proxy autoconfiguration scripts rather than a well-defined proxy address (although you can usually work around such problems).

Proxies can also have difficulty with non-HTTP applications, such as those using Windows-integrated authentication or Java applets and ActiveX controls. Finally, some proxies have difficulties dealing with SSL-protected sites or sites running off the local machine and the loopback address.

### Miscellaneous tools

Many tools don't directly fit under any category; they're basically a mix of

- *white-box tools* for performing security-related activities such as unit testing and Web application user interface testing; and
- *black-box tools* that attempt brute-force authentication against the Web application or check for search-engine information leakage (leading to what is often termed as "Google hacking").

Visual Studio Team System (VSTS) provides several tools that are useful for Web application testing, including a load/stress testing tool, a Web user-interface (UI) testing tool, and PREfix and PREfast, which combine static and dynamic analysis tools for C and C++ based code. Visual C++ 2005 also introduced a new feature, Source Code Annotation Language (SAL), which is useful in declaring pre- and postconditions as attributes on functions. PREfix and PREfast can consume these conditions and provide errors and warnings at build time if the conditions aren't met. For instance, SAL can be used to tie an integer parameter as the length of another buffer parameter.

Organizations can also adapt existing unit-testing tools to test for security. Because these tools typically support build integration, organizations can use them to set up quality gates and build verification tests as part of the nightly build process.

Although there are few testing tools for the increasingly popular Web services area, some have emerged. These tools typically test for problems such as XPath injection and the effectiveness of XML schema validation. You can also use these tools to automate more complex attacks, such as recursive parsing using document-type definition instructions.

Finally, most browsers now offer toolbars that support basic Web application tests. While not the most powerful tools, they do support basic testing, such as parameter recording, session replay, and other key test cases.

***Benefits.*** Developers can use many miscellaneous tools during the SDLC's quality assurance phase or even during unit testing, and can thus catch problems early, when they're cheaper and easier to fix. Also, because VSTS-based tools are closely tied to the developer's integrated development environment, they can automatically create task lists and record bugs in the defect database.

***Drawbacks.*** Most tools in this category are fairly new and lack maturity. The unit-testing tools, for example, are not built specifically for testing security; you must therefore invest significant time into building effective unit-testing libraries, which you can then reuse.

Before discussing how to choose the right tool, we offer a key piece of advice: *don't accept what tool vendors tell you*. Because the information security field is unregulated, many tool vendors attempt to create and play on significant fear, uncertainty, and doubt in their customers, often taking artistic license to new heights. Such vendors package marketing as research and biased opinions as fact. The same is sadly true of popular advertising-based periodicals. We've seen trade magazines give five-star reviews to products that have gone through our labs (at the vendor's expense) and earned a "not fit for purpose" rating.

So, it might sound obvious, but you must road-test your tools. Don't test the tools on the vendor's canned Web site or code. Any tool will perform well on a test suite designed specifically to produce good results. If a vendor won't let you test the tools on your own code or a site you select, you should deduce that it has something to hide.

The most effective way to test tools is to use a site or piece of code that you know and have analyzed. The most important thing to deduce when you're testing tools is not that the tools can find vulnerabilities, but rather that they can find the right vulnerabilities—those that are most critical to your business. Also, you must

- compare the tools against an expected set of results, not an unknown quantity; and
- understand the tools' limitations (as described earlier) and compare those limitations against your typical test case.

As we noted, for example, Web application scanners are generally poor at spidering complex Web sites with large amounts of client-side navigation code and form-based wizards. If your site is made up of such technology, you must test a tool's effectiveness to perform in that environment.

Before selecting a suitable test bed, we recommend that you also consider

- page count (the number of dynamic pages);
- navigation (client-side or dynamically generated, for example);
- client-side technology (such as Applets, flash, JavaScript, and AJAX); and
- your environment's complexity (such as using forms wizards or Captcha programs—completely automated public Turing tests to tell computers and humans apart).

Finally, remember that not all tools are created equal. The analogy clearly applies to Web application security as well. ▢

## References

1. M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed., Microsoft Press, 2002.
2. D. Verdon, "Teach Developers to Fish," keynote address, Open Web Application Security Project Conf., 2004; http://sourceforge.net/project/showfiles.php?group_id=64424&package_id=123286&release_id=251535.

***Mark Curphey*** *is vice president of professional services at Foundstone, now a McAfee company. He is also founder of the Open Web Application Security Project, which aims to help organizations understand and improve Web application security and publishes an annual list of the top 10 Web application security vulnerabilities. His research interests include information security business-process management and applying business management theory to information security. Curphey has an MS in information security from Royal Holloway, University of London, and is a Microsoft Visual Developer–Security MVP. Contact him at mark@curphey.com.*

***Rudolph Araujo*** *is a principal software security consultant at Foundstone, where he specializes in threat modeling, security code review, and content and training creation for the company's courses in building secure software and writing secure code. His research interests include Web services security, knowledge management, and helping SDLC stakeholders better analyze security. Araujo has an MS in information networking from Carnegie Mellon University and is a Microsoft Visual Developer–Security MVP. Contact him at rudolph@alumni.cmu.edu.*