IBM Software
Rational

December 2009

The path to a secure application

A source code security review checklist

Contents

I. Introduction

The ongoing epidemic of data breach notifications forced by today's data breach disclosure laws has painfully highlighted the insecurity of many of today's applications. How can organizations ensure that their applications are secure, avoid the cost and public relations fallout, and stock price downturn from issuing numerous security patches? How do you avoid explaining to consumers and regulators that code defects allowed attackers to steal people's sensitive and perhaps regulated information?

The path to creating a secure application begins by rigorously testing source code for all vulnerabilities and ensuring that use of the application does not compromise or allow others to compromise data privacy and integrity.

For companies using custom-built, outsourced, or open source applications in-house, ensuring that all current and legacy code is secure, however, will be no small challenge. Detecting and eradicating security vulnerabilities have historically been extremely difficult. Many organizations relied on manual code review, which is costly and labor-intensive, as well as ethical hacking, which examines a subset of potential application vulnerabilities in an application.

It is known that by detecting and correcting vulnerabilities early in the software development lifecycle, organizations can substantially reduce both risk and costs.

While both of these approaches are beneficial, companies can use automatic software vulnerability scanning tools to approach secure code development in a more systematic, automated, and successful manner. These automatic vulnerability scanning tools greatly improve the speed and accuracy of code review, and can be integrated seamlessly into the development life cycle. In fact, the best tools can pinpoint vulnerabilities at the precise line of code and provide detailed information about the type of flaw, the risk it poses, and how to fix it.

II. Cost concerns drive companies to secure code development

The imperative for creating secure code has never been greater given the rapid rise in new technologies, including Web services and Rich Internet Applications, and the need to ensure the integrity of existing, legacy, and mid-development applications in a network-oriented world. Companies continue to integrate their systems with business partners to speed the exchange of information. In these conditions, companies must ensure that code is secure to protect data privacy, preserve customer loyalty, safeguard sensitive information, and maintain operational integrity.

One software flaw can lead to a data breach. One of the worst educational data breaches ever disclosed was the 2006 attack of the University of California, Los Angeles' (UCLA) database containing personal information of 800,000 people. In news accounts, Jim Davis, UCLA's associate vice chancellor of information technology (IT), revealed that the attacker had exploited a single software flaw to gain access. Furthermore, the attacker covered his or her attacks well, because the exploits might have begun up to a year before UCLA detected them. The inadvertent disclosure of a company's sensitive information or private and regulated information can lead to fines, lower stock prices, and damage to company's reputation.

Numerous studies have found that catching and fixing code flaws earlier in the software development life-cycle costs significantly less. Research the cost of just one bug that ends up in released code that leads to a data breach and it is readily noticeable the compounding cost of missing that single vulnerability. Studies support this concern. A survey of 31 companies that suffered data breaches found the average breach cost $4.8 million, related to IT clean-up, legal fees, notifications, customer loss, credit monitoring services for affected consumers, and the increased customer service load. The survey, by the Ponemon Institute ©, also discovered customer turnover related to the data breach averaged two percent, but was as high as seven percent in some cases.[1]

III. The path to secure code development practices

What is the best way to ensure that code is secure? The path to effective secure software development requires source code review processes to accomplish three things:

- **Create consistency:** When developing code, developers must create consistent processes, policies, and a culture of improved security.
- **Provide the whole security picture:** When it comes to dangerous vulnerabilities, large-scale design flaws typically exceed individual coding errors. Fixing individual vulnerabilities have little effect if data is not encrypted, authentication is weak, or open backdoors exist in an application.
- **Prioritize remediation:** When reviewing existing code, developers must identify all vulnerabilities in the code and remediate the greatest risks first.

A. The secure development path: Where to look for vulnerabilities

Ensuring code is secure requires examining all of the places that vulnerabilities might exist. Even when using automated tools, developers must understand that the path to creating a secure application might involve reviewing implementation and design practices, including external code and code-reuse practices, which they might not initially consider as vulnerable. Therefore, companies must be diligent with secure code development, and ensure that they analyze the myriad places where software vulnerabilities can exist.

To effectively measure the risk posed by any given application, security analysts or developers must watch for two types of errors:

- **Implementation errors:**
  These quality-style defects in code are fairly atomic, and typically stand alone when identified, and remediation is applied. Implementation errors are caused by bad programming practices. Examples include buffer overflows, which result from mismanagement of memory and race conditions, which result from call-timing mismatches.
- **Design errors:**
  These errors include the failure to use or adequately implement security-related functions. For example, authentication, encryption, the use of insecure external code types, and validation of data input and application output.

While implementation errors are the most common, it is actually design flaws that pose the greatest risk in today's Web-enabled applications. Is there access control? Is encryption present and is it strong enough? Is database access done securely and according to policy? These design-level security issues must be reviewed and addressed during secure application development.

B. The secure development path: How to look for vulnerabilities

The process for spotting errors is not simply to better define the need for security in the development process, but to look at all the places in the code where design flaws might, or do, exist. These places are typically far more extensive than even advanced developers realize. Properly analyzing source code might take testers to places they do not expect.

Ensuring that new and existing code is securely developed requires processes and procedures for hunting vulnerabilities, and tools to help. Which tools are most effective for building secure software?

Commonly used approaches include manual code reviews and ethical hacking. While these approaches are both useful, neither is sufficient to cope with the breadth of existing and potential design errors, and therefore cannot ensure that the code is secure. For example, manual code review is very time-consuming and expensive. Spotting which lines of code contain flaws or might lead to operational errors, is extremely difficult. Ethical hacking can discern only a small subset of errors an application might contain. While this approach is useful for highlighting such errors, it provides an incomplete picture of the overall application security.

Given the breadth of existing design errors, or potential errors, experts say companies must employ automated software vulnerability detection tools to spot all potential flaws. They agree the most effective approach for companies developing their own software is to integrate source code vulnerability scanners into the application development, integration, and test processes. Only advanced source code vulnerability testing tools and the related software development life-cycle practices can efficiently and effectively ensure that code is secure.

III. Steps to secure code: What to examine

The most efficient and effective technique for creating secure source code is to evaluate existing applications as well as code under development against five classes of code vulnerabilities:

1. Security-related functions
2. Input/Output validation and encoding errors
3. Error handling and logging vulnerabilities
4. Insecure components
5. Coding errors

Following security-related issues through the source code of an application dramatically reduces the vulnerability of the application and the critical data it processes and protects.

More detailed information about each class of error follows:

1. Security-related functions

Applications are the sum of many discrete features, which often seem harmless. Yet an incautious combination of these features, or lack of resulting documentation about how these features are implemented, can easily create a security risk and lead to breaches of privacy, confidentiality, or system integrity.

This risk is compounded by the widespread use of higher-level programming languages, and in particular prebuilt modules and precompiled libraries. With these tools, developers can rapidly deploy full-featured applications with access to a number of services and data sources. The tools also help prevent many security problems, for example, by abstracting memory management and resource control issues away from developers using high-level interfaces.

These tools do not demand a more detailed understanding of how to tap services and data in a secure manner, or whether doing so might conflict with an organization's already existing business processes or infrastructure security. As a result, implementing these libraries or modules in an insecure way is actually responsible for the majority of security problems in today's applications. Furthermore, these kinds of security design errors are often much more dangerous than previous types of coding problems, because today's applications often interface with both an organization's back office, as well as the Internet, creating a potential conduit for the loss of sensitive data.

A comprehensive determination of the security state of an application must include an analysis for the following critical design flaws:

a. Weak or nonstandard cryptography

One of the fundamental components of application security is encrypting data. Private or sensitive data gets scrambled to protect it. If attackers can break encryption algorithms, they can steal sensitive data. The use of weak random number generators and nonstandard cryptographic algorithms are two widespread encryption vulnerabilities that lead to attackers stealing sensitive data.

For encryption to be effective, the underlying cryptography must be based on randomness sufficient to ensure that an attacker cannot easily guess or reproduce the keys used to enable data sharing. Weak random numbers are insufficiently random. When the resulting less-random numbers are used as key seeds, this encryption algorithm exposes the encrypted data to sequence number prediction and session spoofing. Developers must avoid weak random number generators.

Nonexistent Encryption

When assessing code security, pay special attention to presence and correct encryption implementation as well as its inappropriate absence. Failing to encrypt sensitive information has led to numerous data breaches. For example, look at the United States Department of Veterans Affairs (VA) incident that was reported in May 2006. One of the VA applications stored the social security numbers and home addresses of all retired veterans. An employee's laptop containing this information was stolen, putting at risk information on 38.6 million veterans Having a code review can help find all the sensitive information stored in an insecure manner, and likely result in the question of why this data was even available on a laptop. Why not make secure calls to a centralized database, which is protected by rigorous access controls and monitoring? The probable answer is that creating a laptop-based application that stored information in an insecure format was simpler.

Beware of nonstandard cryptographic algorithms. Cryptographic algorithms scramble data and only a handful of truly secure algorithms exist, which have been thoroughly evaluated by cryptography experts. Even for these experts, producing a truly secure and acceptable algorithm is extremely difficult, which accounts for the continuing use of triple DES, Blowfish, and other well-worn algorithms. Even these algorithms are sometimes later found to be breakable. Regardless, defer to the experts' recommendations. Other algorithms are or at least might be of insufficient strength to stop an attacker from decoding encrypted data.

b. Nonsecure network communications

Using legitimate methods to send or receive data, but not documenting or protecting these processes, might expose critical data while in transit, allowing someone to easily intercept and read it. Developers must employ secure network communication protocols, such as Secure Sockets Layer (SSL), whenever possible, and thoroughly document these processes.

c. Application configuration vulnerabilities

Developers must ensure that the configurations files or options controlling their applications are also secured. Otherwise, an attacker might be able to access these unprotected files or options, manipulate them, and adjust software properties or access controls to access sensitive data.

d. Access control vulnerabilities

Without access controls, attackers with network access, including malicious insiders can easily tap into confidential data and resources. Organizations must implement a strong technique for identifying users, map identified users to data, and ensure that users can access only appropriate data.

The related vulnerability is when an application grants greater than necessary access rights, to either a user, or an application. Depending on the level of access granted, a user might be able to access confidential information, or even take control of the system. Therefore, for any application handling sensitive data ensure that the principle of least privilege reigns: Grant only the minimum level of access needed for a user or application to function. This approach involves identifying the different permissions that an application or user of that application needs to perform their actions and restricting all appropriate modules and objects to these privileges.

When evaluating access controls, also watch for these other security vulnerabilities:

- **Unprotected database and file system use:** You must ensure that application security tools carefully review calls to databases and file systems within the application source code. Otherwise, attackers can manipulate these calls to expose sensitive data.
- **Dynamic code vulnerabilities:** Does the application load dynamic code? Attackers can insert malicious commands into applications that load dynamic code. You must ensure that the applications are first examined to see if this potential exists.
- **External code loading:** Attackers can change, expose, or destroy data by manipulating improperly validated system-level calls.
- **Data storage vulnerabilities:** Why attack an application if data is left unsecured? Attackers might be able to access servers to steal sensitive data if it is not stored securely.
- **Authentication errors:** By using legitimate user credentials, or tricking a system into thinking legitimate credentials are being used, an attacker can steal or manipulate data. In the data breach incident at ChoicePoint, Inc © for example, attackers were able to acquire actual user credentials, which helped disguise malevolent activities.

Access control exploit: Forced browsing

When reviewing code, pay close attention to forced browsing. Here is how forced browsing works: Attackers issue a request directly to a Web page that they might not be authorized to access. If improper access controls are in place, they might be able to access the Web page, or the back-end resources, and possibly steal or corrupt those resources.
To prevent such attacks, developers must ensure that no Web page containing sensitive information is cached either on servers, or users' local personal computers. All such data is restricted to requests that are accompanied by an active and authenticated session token, and associated with a user who has the required permissions to access that page or information.

2. Input/Output validation and encoding errors

Most applications require input to be dynamic. User responses and selections in an application drive and tailor their experience with the application. Every input has the potential to introduce vulnerability and must be validated to ensure that its form or size does not cause the application to behave unpredictably. All sources of input and especially those provided through user interactions must be checked at some point after they enter the system, and before they reach the place where they are used. Developers must ensure that all inputs are validated, and that they reconcile with expectations, or else an application must prevent the malformed input from proceeding through the application.

To better understand when and where to utilize input/output validation, and where it is especially needed, it helps to understand some of the leading, related attacks, and how to stop them:

a. SQL injection vulnerabilities

One very common input validation vulnerability is an SQL injection. Attackers have used this technique to steal large amounts of credit card information from databases. As the name suggests, attackers make inappropriate SQL queries to a database to illicitly access data or cause unstable database behavior. The most effective method of stopping SQL injection attacks is to use only stored procedures or parameterized database calls that do not allow the injection of code. This simple technique stops SQL injection attacks.

b. Cross-site scripting vulnerabilities

Cross-site scripting (XSS) attacks refer to the use of insufficiently protected output mechanisms in applications. Attackers can use these mechanisms to cause unsuspecting users to run or access malicious code.

XSS attacks can generally fall into two categories:

- **Stored attacks:** Injected code is permanently stored on the target server database, message forum, or visitor log.
- **Reflected attacks:** Injected code reaches a victim, perhaps through an e-mail message, or by living on another server. After a user is tricked into clicking a link or submitting a form, the injected code travels to the vulnerable Web server, which reflects the attack back to the user's browser. The browser then runs the code because the code comes from a trusted server.

The most severe XSS attacks result in disclosure of a user's session cookie, which allows an attacker to hijack the user's session and take over their account. Other damaging attacks include the disclosure of end-user files, installation of Trojan horse applications, redirecting a user to some other page or site that contains a phishing attack and modifying content. To defend against scripting attacks, protect all user-supplied output to the client and log files with HTML Entity Encoding. This encoding organizes all nonalphanumeric characters into a special character sequence that cannot be interpreted by HTML-enabled viewers.

Using XSS to launch attacks at other sites

Cross-site scripting attacks are a particular risk for consumer-focused online sites, and in particular social networking sites, where the medium of communication skews heavily to cutting-edge multimedia, and a modicum of security.
For example, take a worm that targeted MySpace.com© in 2006. Dubbed amazingly virulent by one security analyst, the cross-site scripting attack exploited an Apple© QuickTime vulnerability (though Apple did not initially regard it as such) to inject JavaScript™ from an external source without warning into a user's profile, ultimately stealing a user's MySpace log-in credentials and launching spam attacks through MySpace. Vulnerability in a multimedia player helped facilitate a widespread attack.

c. Operating system (OS) injection vulnerabilities
Applications sometimes require access to operating system-level commands. Be careful when such access also involves user input because similar to SQL injections, attackers can use malformed inputs to affect OS behavior, leading to potential data breaches or the compromise of an entire system.

d. Custom cookie or hidden field manipulation
Cookies are a useful and popular means to maintain user information during and across sessions. The creation of custom cookies with custom names and values is a dangerous practice, often leading developers to rely too heavily on these unsafe cookies because attackers can easily modify cookies. Unless developers validate cookies, they can create an opportunity for attackers to create SQL injections or successful cross-site scripting attacks.

3. Error handling and logging vulnerabilities

Does your application fail gracefully? Error handling and logging are especially difficult for developers: Who can predict all the ways in which an application might fail, or the repercussions?

Web 2.0: Exposing insecurity to the Web

The Web-centric nature of business today is providing Web-enabled access to more data than ever before. Technologies such as Ajax facilitate easier access to information for software developers, and can make the end user experience much more positive. Still, these technologies do little to address security. Organizations must be especially vigilant about code quality and security vulnerabilities as they adopt new tools and development techniques.
One Web 2.0 innovation, for example, is the mashup, which mixes content from a host server with publicly available feeds. Yet with new capabilities comes the potential for new forms of cross-site abuse. When Web 2.0 mashups are not done securely huge gaps leave room for new forms of phishing and other attacks.

Tracing application error handling and logging, however, is of paramount importance because so many attacks today succeed by feeding bad data to an application, and exploiting the resulting application misbehavior.

When grading an application on its error handling, first evaluate these two aspects:

a. Insecure error handling
Poor error handling can provide attackers with crucial information for launching attacks. For example, indiscrete error routines can provide valuable insights into how an application processes inputs, especially if the error routine is custom-writing to detail-specific errors and data elements. Developers must limit the amount of detail they reveal to application users. Avoid having any applications that integrate with a Web server displaying errors to the server. An outside attacker might be able to use the resulting information to gain access to systems.

b. Insecure or inadequate logging
Log files might be essential for tracking application behavior, but they are also a rich resource for attackers. Do not make log files accessible. Consider logging application behavior for applications trafficking in sensitive data, to ensure that attackers cannot hide their tracks.

4. Insecure components

Vulnerable code might become part of an application, either through malice, or inadvertent poor code-writing practices. For example, an attacker can insert malicious code into an application to circumvent existing security measures. Unfortunately, malicious code often looks identical to nonmalicious code. Both types of code can provide access to networks and data, and both typically function like other parts of an application.

As a result, automatically identifying malicious code is extremely difficult if developers are assessing only for functionality. Rather, developers must focus on location. Start by identifying specific types of functionality (such as network communications, timed events, and privilege changes) and then map each with the application module in which it operates. Watch for mismatches, such as a graphic library conducting outbound network operations, or hard-coded timed events in a largely real-time application. Such mismatches are warnings for malicious intent.

Today's application development practices are heavily componentized. While this componentization helps create secure applications more quickly, two types of component use pose significant risk of vulnerability:

a. Unsafe Java Native Interface methods

If developers employ unsafe Java™ Native Interface (JNI™) methods in their code, attackers might be able to easily access critical resources, such as system or environment memory.

What are JNI methods? Higher-level languages that provide interfaces to verify any access to resources. By contrast, JNI is a more basic means of accessing these resources, which side-step such interfaces, resulting in improved performance. Because the code is written without the interface-level checks, the risk of miscoding is extremely high. In general, avoid using any JNI methods, except in exceptional cases where performance concerns are paramount. Always locate and thoroughly test any JNI methods that are used. Also, be aware of the potential that JNI libraries have built-in errors, and especially when programming in C or C++, which are already more susceptible to buffer overflow or race-condition problems.

b. Unsupported methods

Developers sometimes take shortcuts, relying on unsupported methods or calls when building applications. Using undocumented functions or routines can produce hidden insecurities, allowing attackers to exploit an application. For any software project, ensure all contracts specify that only supported methods are used. For existing applications, find any unsupported methods and remove and implement them with supported ones.

5. Coding errors

Coding errors, also known as implementation errors, might be caused by developers with insufficient training, compressed project schedules, improper project requirements prioritization, or by reusing code of questionable or unknown quality.

When coding errors exist in applications, they might cause such unexpected behaviors as yielding control of a system or process to an attacker, to shutting down an application. Because of the security risks inherent to coding errors, these errors must be removed from code, whether it is in development or already in production.

When evaluating code, be aware of these coding-related vulnerabilities:

a. Buffer overflow vulnerabilities

Buffer overflows occur when more data is copied into a buffer than the buffer can hold. Even though buffer overflows have been well understood for more than 20 years, they are still quite a common problem, and pose an extremely high risk.

Suspicious behavior

Signs an application might contain malicious code

Sign	Suggests
Raw socket access	Possible backdoors
Timer or get time function	Trigger
Privilege changes	Unauthorized access levels

Attackers can use this type of memory mismanagement to load and run exploit code in computer memory, which effectively allows an attacker to gain full control of a system.

In structured languages, such as C and C++, literally hundreds of calls and call combinations exist in which it is possible to misallocate memory, or insufficiently understand the range of application behavior. This complexity creates ideal conditions for buffer overflow attacks.

In higher-level languages, such as Java, JavaServer™ Pages (JSP™), C#, .Net, vulnerabilities are much less prevalent, because the language and runtime interpretation handle all lower-level memory management, protected from any influence by the programmer. Even in these higher-level languages, certain, and sometimes undocumented, calls might exist that create a buffer overflow vulnerability unbeknownst to the programmer. Be especially alert for buffer overflows in all higher-level languages.

b. Format string vulnerabilities

Format string vulnerabilities illustrate the relationship between implementation errors, and the overall quality of a code base. Many regard format string vulnerabilities as a type of buffer overflow, but this comparison is not exactly true. While a format string vulnerability might produce an overflowed buffer, the vulnerability can also lead to information exposure, without an overflow.

When building code, incomplete use of some calls can lead to format string vulnerabilities. Good coding practices dictate consistent use of certain arguments, such as field specifiers. If developers build code using incomplete calls instead, such calls are insufficiently bounded, which allows attackers to opportunistically embed additional data, arguments, or requests for information in those calls. Review all code for any incomplete calls and address them.

c. Denial-of-service errors

Any application that provides access to critical data or services can only serve that purpose when it is running. Denial-of-service (DoS) attacks compromise applications and affect the delivery of critical data or services.

A DoS vulnerability refers to implementation errors that cause either consumption of scarce, limited, or nonrenewable resources, or the destruction or alteration of configuration information. To avoid these failure conditions, developers must design their applications to run, even in worst-case scenarios.

d. Privilege escalation vulnerabilities

In many cases, the ultimate goal of an attack is privilege escalation. A user with insufficient credentials gains privileged access, allowing the attacker to access confidential data and resources, and even take control of and destroy an entire system, all while being classified as a trusted user.

An attacker typically elevates his or her privileges by exploiting sections of programs in which an application grants or receives higher-level privileges. Applications sometimes need to create processes as different users, or with different access levels. The burden of proof falls on developers to ensure that the privilege escalation process cannot be exploited by rogue programs. Similarly, the deescalation process must also be thoroughly tested.

For example, when a lower priority process returns control to another program with higher level privileges, the application must consistently check for valid return codes, error conditions, and privilege-lowering operations triggered by errors. If any of these operations fail, the programs can be left running at a different level or privilege than was intended or necessary.

e. Race conditions

Two processes might share control or data. Race conditions is the term applied to compromising this sharing, which typically results from synchronization errors, when the potential exists for process conflicts, and a resulting vulnerability. A typical exploit interrupts a pair of sequential calls that are meant to be performed automatically without interruption by another thread or process on the machine with a third process.

One example is the combined checking of access rights to a file, followed by a subsequent call to write or read that file. By interrupting the process between the two calls, an attacker can rewrite or modify the file because this behavior is expected. The attacker can place inappropriate information into a file, or perhaps access an inappropriate file.

Common denials-of-service errors

Application shutdown DoS: How does the application shut down? Some application writers, when implementing termination functionality for an application, do so too broadly. For example, if an application closes itself automatically because of input errors by using a system exit function, an attacker can cause a similar set of events that unnaturally and unnecessarily cause the application to stop.
Database connections not closed DoS: How cleanly does the application connect to databases? To grant a process access to data, the application and data source must first form a connection. If this process is improperly coded, the queue of requests for collection might become cluttered, then overload from failed database connection attempts.

IV. Applying the source code review checklist

A. Applications guilty until proven innocent

The five broad types of code vulnerabilities described previously all represent the likeliest and most dangerous risks contained in current and legacy code. Business customers, software development project managers, and developers must ensure that all code is reviewed for these five classes of vulnerabilities.

Given the myriad risks that are posed by these vulnerabilities, and their likely presence in many applications, project teams must treat every application they commission, create, or reassess, with security skepticism. This attitude represents a marked shift away from the traditional development approach, which is to analyze an application based on its speed, feature set, or ease of use.

Now, development teams treat every existing and underdevelopment application as a security risk, until it is proven otherwise. Teams have to react that way because of the risk such vulnerabilities pose to the business. Indeed, as an Institute of Electrical and Electronics Engineers (IEEE) working group investigating the new role of security in the software development life cycle found, applications now "can have a far greater effect on consumer approval, business stability and profitability in the long run."[2]

As a result, the IEEE group recommends that "project teams traditionally focused entirely on responding to customer requests must now learn to communicate security risk upfront to sponsors, so the appropriate budget and project needs will be justified." While such changes might be unpopular, today's security threats and mandate to protect sensitive and personal information demands a dedicated focus on security by all project team members.

B. Pursuing source code vulnerability testing

Source code vulnerability testing tools alone do not make software secure. Such tools help developers and code reviewers assess applications, even those with many millions of lines of code, to identify the most potentially damaging vulnerabilities. With this testing development and remediation teams can prioritize their efforts, and take a risk-based approach to improving the code base, starting with the most critical problems first.

Creating secure applications demands that organizations make secure code and schedule ongoing vulnerability testing. Make secure code an exit requirement for any application, before allowing the code to be released. The swift upsurge of targeted threats should make requiring mandatory application security vulnerability testing a primary focus within all enterprises.

This work is exacting and complicated by the fact that complex applications might contain extremely complex and difficult to identify coding errors and vulnerabilities. Code review teams demand the need for sophisticated code analysis. Indeed, the technical nature and well-understood roots of implementation and coding errors leads many companies to make code reviews their logical starting point for improving overall application security.

C. Selecting the right tool

When selecting an automated source code vulnerability testing tool to use throughout the software development life cycle, organizations must first assess their existing code development resources. These resources include in-house security expertise, technologies, and service partners, as well as their software development life-cycle project-improvement objectives. These objectives might involve decreasing the number of security patches released, reducing the potential for costly data breaches, lowering software development life-cycle costs, more stringent meeting regulations, and ensuring a competitive advantage. Companies must determine that the tool they select works best with their existing code resources, and helps the organization meet its software development life-cycle project objectives. They must also ensure that the tool they select covers the entire breadth of the coding errors and design flaws that must be identified and eliminated to create a secure application.

The numerous and well-publicized data breaches to date, many the result of code flaws, highlight how important eradicating vulnerabilities are in preventing the inadvertent or malicious disclosure of sensitive or regulated information. Even for organizations not currently subject to such regulations, the rapid growth in more connected tools, technologies, and programming languages, including Web 2.0 applications makes any organization using such technology vulnerable. Given the substantial costs in money and reputation that result from a data breach, companies are increasingly concerned with finding the most efficient and effective path to ensuring source code security, developing secure applications, and eliminating bugs as early as possible in the development process, and well before code is shipped.

Companies that can efficiently and effectively integrate source code vulnerability testing into their software development life-cycle practices can avoid the negative impact of security flaws. These improved practices result in substantial savings internally as well as for the customers, partners, and other stakeholders that rely on their software. This savings is the true sign of an effective source code security program.

V. APPENDIX: Source code security review checklist

Verifying that applications are secure begins by watching for these vulnerabilities to mitigate the risks they pose to application and data integrity:

Category	Vulnerability	Risk
Security-related functions	Weak or nonstandard cryptography	Attackers can break algorithms to steal sensitive data
	Nonsecure network communications	Legitimate methods of sending information are not documented or protected, exposing critical data
	Application configuration vulnerabilities	Access to unprotected configuration files or options allows manipulation of software properties or data
	Access control vulnerabilities	Unauthorized access to confidential data and resources
	Unprotected database and file system use	Hijacking and manipulating calls to databases and file systems exposes data
	Dynamic code vulnerabilities	Successfully inserting malicious commands into applications that load dynamic code without proper validation
	Local code loading	Manipulating these system-level calls allows data manipulation, exposure, or destruction
	Data storage vulnerability	Data stored insecurely can easily be stolen
	Authentication errors	Attackers use legitimate users' credentials to steal or manipulate data
Input/Output validation and encoding errors	SQL injection vulnerabilities	Sending SQL commands directly to databases to steal or manipulate data
	Cross-site scripting vulnerabilities	Users unknowingly have sessions hijacked, download Trojans, or fall for phishing scams
	OS injection vulnerabilities	Attackers modify or misuse operating system commands to control data and resources
	Custom cookie or hidden field manipulation	Creates a level of trust attackers can manipulate to launch attacks, such as SQL injection or cross-site scripting

Category	Vulnerability	Risk
Error handling and logging vulnerabilities	Insecure error handling	Furnishes attackers with information they can use for attacks
	Insecure or inadequate logging	Accessible log files divulge information useful for attacks, while inadequate logging allows attacker to hide tracks
Insecure components	Malicious code	Seemingly legitimate code inserted into software can allow attackers to circumvent security measures
	Unsafe local methods	Unchecked use of local methods provides entry for attackers to access critical resources such as system or environment memory
	Unsupported methods	Undocumented functions or routines are a hidden source of insecurity for potential exploitation
Coding errors	Buffer overflow vulnerabilities	Attackers can hijack system resources.
	Format string vulnerabilities	Leads to buffer overflows or data exposure
	Denial-of-service errors	Prevents software from functioning
	Privilege escalation vulnerabilities	Attackers can access confidential data and resources
	Race conditions	Circumventing an application process to manipulate operations
	Unsafe local method use	Might sacrifice security for performance, allowing unsafe access to system or environment memory
	Unsupported method	Legitimate operations might unknowingly invoke calls to vulnerable code

Notes

For more information

To learn more about the IBM Rational® AppScan® Source Edition, please contact your IBM marketing representative or IBM Business Partner, or visit the following Web site: **ibm.com**/software/rational/products/appscan/source/

About the authors

Ryan Berg is a Senior Security Architect at IBM. Ryan is a popular speaker, instructor, and author in the fields of security, risk management, and secure development processes. He holds patents and has patents pending in multilanguage security assessment, kernel-level security, intermediary security assessment language, and secure remote communication protocols.

[1] Ponemon Institute, "2006 Annual Study: Cost of a Data Breach," October 2006.

[2] Biscick-Lockwood, Bar, "The Benefits of Adopting IEEE P1074-2005," April 2, 2006.