Parasoft Proposal For Security Assessment:
How Can Parasoft Help You Make Your Applications Secure?

PARASOFT®

When most people in the software industry refer to "security", they mean the security of the network, operating system, and server. Organizations that want to protect their systems against security attacks invest a lot of time, effort, and money ensuring that these three components are secure. Without this secure foundation, the system cannot operate securely.

However, even if the network, server, and operating system are 100% secure, vulnerabilities in the application itself make your system just as prone to dangerous attacks as unprotected networks, operating systems, and servers would. In fact, if an application has security vulnerabilities, it can allow an attacker to access privileged data, delete critical data, and even break into the system and operate at the same priority level as the application, which is essentially giving the attacker the power to destroy the entire system. Consequently, the security of the application is even more important than the security of the system on which it is running. Building an insecure application on top of a secure network, OS, and server is akin to building an elaborate fortress, but leaving the main entryway wide open and unguarded.

The ideal application security is an application that is completely invulnerable to security attacks. Achieving this goal requires a detailed and specific understanding of the system in question, as well as the flexibility to address unusual or unknown circumstances. When tested systems fail, it's because their testing didn't cover every component and consider every possible condition. An untested or lightly tested piece of code gets executed in a way that the developers never intended, causing a security problem. Why doesn't this code get tested properly? Because of inefficiencies inherent in the most common methods of verifying application security: penetration testing and/or static analysis designed to expose general security vulnerabilities.

Penetration Testing

Penetration testing involves manually or automatically trying to mimic an attacker's actions and checking if any tested scenarios result in security breaches. Although it may sound helpful in theory, in practice, this approach is a very inefficient way of protecting the application and exposing its security problems. Basically, to expose a security vulnerability with penetration testing, you must be lucky enough to design (or generate) an attack scenario which is going to expose a security vulnerability in the application. Considering that there are thousands, if not millions, of possible scenarios for even a basic application, there is a small chance that you will happen to test scenarios that expose *any* potential vulnerabilities, let alone identify *all* potential vulnerabilities. As a result, penetration testing consumes a lot of time, energy, and money, but in the end, it doesn't deliver much assurance that the application is protected.

Furthermore, penetration testing can fail to catch the most dangerous types of problems. Let's assume that you have a web application to test, and this application has a backdoor that gives admin privileges to anyone who knows to supply a secret argument, like h4x0rzRgr8 = true. A typical penetration test against a web application uses known exploits and sends modified requests to exploit common coding problems. It would take years for this test to find this kind of vulnerability through penetration testing. Even an expert security analyst would have a tough time trying to exploit this. What about a difficult to reach section of code in the error handling routine that performs an unsafe database query? Or the lack of an effective audit trail for monitoring security functions? These kinds of problems are often entirely overlooked by even a diligent penetration test.

Bug-Finding Static Analysis

After realizing the problems inherent in relying on penetration testing, security specialists started to think about checking application security from the perspective of the code, through static analysis. The static analysis engine looks for potentially dangerous function call patterns and tries to infer if they represent security vulnerabilities (for instance, to determine if code has unvalidated inputs, and if unvalidated inputs are passed to specific functions that can be vulnerable to attack).

This is an improvement over penetration testing because it has a greater chance of exposing vulnerabilities, but there are still several problems with this method. One is that these patterns don't consider the nuances of actual operation; they don't factor in business rules, or general security principles. If you have a web application that allows your customer to see their competitor's account by adding one to the session ID, this is a very serious problem. However,

this kind of problem escapes static analysis because it doesn't involve a dangerous function call. Security assessment, in this sense, isn't always a bug to find, but a design problem to verify. Another problem is false positives. Static analysis cannot actually exploit vulnerabilities; it can only report potential problems. Consequently, the developer or tester must review every reported error and then determine if it indicates a true problem, or a false positive. Sophisticated static analysis methods can improve accuracy, but ultimately, a significant amount of time and resources must be spent reviewing and investigating reported problems and determining which actually need to be corrected.

Security Policies

An alternative, more efficient and effective strategy for securing the application approaches security from the perspective of prevention. The crux of this approach is to define, concentrate, and verify security policies. A security policy is a document that defines how code should operate to maintain the degree of security that the organization requires for that system. Security policies are required for Sarbanes-Oxley compliance, so defining and enforcing a security policy not only improves application security, but also brings you one step closer to Sarbanes-Oxley compliance.

What does a security policy involve? First, you define how the code needs to be written so that it is not vulnerable to attacks. This policy should be designed to prevent both possible types of security bugs: bugs in the code that cause security mechanisms to malfunction, and security mechanisms that aren't implemented correctly. The first case tends to be a problem when critical security tasks such as input validation or authentication are handled differently in different parts of the code. Not only is this bad for maintainability, it is bad for security because it introduces more attack surfaces where vulnerabilities can hide.

When implemented, all security-related operations specified in the security policy should be concentrated in one segment of the application. You can then focus your resources on verifying and maintaining the security of that one critical module. This centralized security policy acts like a drawbridge for a castle: it isolates the area attackers can exploit and allows for a more focused defensive strategy.

If the security policy is implemented in this manner, you can use static analysis tools to enforce this policy by scanning code and checking whether it matches patterns that indicate violations of the defined security requirements. This scanning can be performed for traditional programming languages such as C/C++, Java, and .NET languages (including C#, VB.NET, and Managed C++), as well as scripting languages, such as PHP, Perl, and ColdFusion. In many cases, the static analysis tools can automatically correct the code so that it complies with the security policy. Identifying and removing security vulnerabilities in this manner takes just seconds, which provides an opportunity to make significant security improvements with minimal resources.

After the policy is verified and enforced through static analysis, penetration testing can be used to confirm that the security policy is implemented correctly and operates properly. When penetration testing is performed in this manner (for confirmation, rather than as the main security bug-finding method), it can provide reasonable assurance of the application's security after it has verified just several paths through each security-related function.

This paper describes how Parasoft and its suite of AEP (Automated Error Prevention) products allow you to create and enforce a security policy that is customized and extensible, and adds a transparent layer of security testing and verification throughout the development process.

Creating and Enforcing an Effective Security Policy

Step 1: Review the Existing Security Policy

The first step in securing application code is to determine the current application security policy—the guidelines defined for ensuring that the proper security mechanisms are in place to protect application resources. If no application security policy is currently defined, Parasoft works with your organization to develop one. This policy should describe what types of resources require privileged access, what kind of actions should be logged, what kind of inputs should be validated, and other security concerns specific to the application.

Step 2: Extend/Customize the Parasoft Security Policy

Next, the existing security policy is used to customize and extend the Parasoft security policy, which provides a framework for automating the process of enforcing security constraints. The security policy consists of rules that are based on the range of categories pertaining to best practices in application security. This core set of rules is supplemented with application-specific rules that address your organization's specific security concerns. This process results in the creation of a document which enforces the centralization of security mechanisms, prevents coding problems that can lead to security vulnerabilities, and details custom security requirements.

Step 3: Create Static Analysis Rules to Enforce the Policy

To ensure that all team code complies with the security policy consistently, the static analysis rule set is customized to detect policy violations. By customizing the static analysis, specific security best practices such as sensitive function logging and proper access control can be enforced according to business rules. This reduces the amount of false positives and provides a more sophisticated level of policy enforcement than standard source code analysis.

Step 4: Configure Automated Source Code Scanning

Once these static analysis rules have been implemented to enforce the security policy, they are included in the static analysis configuration and can be run unobtrusively as part of the automated nightly build to ensure that violations are identified as soon as they are introduced. By referring to the security policy, developers can understand why these violations are security vulnerabilities, as well as how to fix the problem and secure the code.

Because Java is a strongly-typed, sandboxed platform, it is a common choice for applications that require a high level of security. However, C or C++ code is often necessary and needs to be analyzed for security concerns as well. Using a similar approach, policies can be customized to enforce secure coding standards for preventing problems unique to C-style languages (for example, buffer overflow, format string attacks, and memory leaks).

Step 5: Review the Implementation of Centralized Security Functions

The Parasoft security policy enforces the centralization of certain security functions to ensure that there is only a single point of failure in critical actions such as authentication, input validation, and encryption. It is important to ensure that these important functions are centralized, but it is also critical to ensure that they are implemented correctly. Best practices such as unit testing should be used to verify that these functions operate as expected and do not contain inherent vulnerabilities.

Step 6: Penetration Test with WebKing

Once the application is up and running, the next step is to penetration test the system. Analyzing how the application responds to attacks simulated during penetration testing can help you determine the effectiveness of the overall security policy.

One key advantage of performing penetration testing with Parasoft WebKing, a comprehensive automated web testing product, is that it allows testers to perform a security assessment for applications written in scripting languages that are difficult to analyze statically. PHP, Perl, and ColdFusion are languages that are primarily used for simple or internal web applications without focusing on security, but can leave gaping holes in a network. Because WebKing interacts through the interface of the running application, it can test the security of any web-enabled application, regardless of the server-side implementation.

Step 7: Address Vulnerabilities Found in Penetration Testing

Generally, a penetration test that reveals a single security vulnerability often leads to the specific vulnerability being fixed, but similar vulnerabilities scattered throughout the code are left untouched. Because of the centralization enforced by the security policy, any security vulnerabilities that WebKing finds will point to a single flaw in the source code. Fixing this single flaw will remove the vulnerability for all cases.

Step 8: Configure Regression Tests

Individual penetration tests that identify security vulnerabilities, as well as the passed tests, can be added to a regression suite that alerts the team immediately if code modifications reintroduce previously-corrected security vulnerabilities or introduce new ones. The static analysis security rules are also added to the nightly regression tests so that the source code repository is automatically scanned in the background each night to verify whether all new code complies with existing security policies. This automated testing allows for the effective mitigation of security vulnerabilities and provides documented proof (such as that required for Sarbanes-Oxley compliance) that the application security policy has been enforced.

Final Thoughts

There are several ways to address application security; some are more effective than others. In general, there are no silver bullets or easy answers to the multitude of security problems found in custom applications. Instead, it is important to understand the security issues before, during, and after development, so that failures can be identified immediately and remediated quickly. By adopting the Parasoft security policy, a software development organization can regulate the security of its applications and efficiently deliver hardened, secure, and robust applications.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (0)1923 898005
Germany: Tel: +49 7805 956 960
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Appendix A: Sample Security Policy Enforcement Rules

The following table shows a sample of the Parasoft static analysis rules that may be used to enforce a security policy for a Java-based application.

<table>
<tr>
<td>Use JAAS in a single, centralized authentication mechanism.

Instead of doing access control checking in each servlet or JSP, use a "front-door" servlet to do the access control checking.</td>
<td>The following code is not allowed:

<pre>Class ISOGenericServlet extends Servlet {

  Void doPost(HttpServletRequest req,
HttpServletResponse resp) {

   lc.login();

}</pre>
JAAS Authentication can not take place outside of the ISOAuthentication

The following code must be implemented:

<pre>class ISOAuthentication {

  void authenticate(String user, String pass) throw
     LoginException {

      //do authentication

   }

}</pre>
This code is called every time a JAAS authorization operation takes place</td>
</tr>
<tr>
<td>Do not cause deadlocks by calling a "synchronized" method from a "synchronized" method.

Avoid potential deadlock conditions that can cause denial of service.</td>
<td>The following code is not allowed:

<pre>public synchronized void method1() {
  method2();
}

public synchronized void method2() {
}</pre>
Synchronized methods that call other synchronized methods are deadlock prone. Try to write the code so that a thread doesn't try to get a lock on a monitor while already holding a lock. One possibility is to use a "synchronized" statement to only synchronize the part of the method that really needs to be synchronized. Alternatively, before locking a second object, ensure that it is not already locked.</td>
</tr>
</table>

Use only strong cryptographic algorithms.
 	
Some encryption algorithms are not as strong and therefore not as secure as others. It is recommended to use the strongest practical cryptography and avoid weak cryptographic algorithms.
	The following code is not allowed:
	
	`Cipher.getInstance("MD5");`
	
	Do not use weak cryptographic algorithms.
	
	The following code shows the correct behavior:
	
	`Cipher.getInstance("SHA-1");`
	
	Only use strong cryptographic algorithms.

Validate 'HttpServletRequest' object when extracting data from it.
 	
HttpServletRequest objects contain user-modifiable data that, if left unvalidated and passed to sensitive methods, could cause serious security problems such as SQL injection and XSS.
	The following code is not allowed:
	
	`String name = req.getParameter("name");`
	
	Unvalidated user data could be passed on to sensitive methods.
	
	The following code shows the correct behavior:
	`try {`
	`  String name =`
	`ISOValidator.validate(req.getParameter("name"));`
	`} catch (ISOValidationException e) {`
	`  ISOStandardLogger.log(e);`
	`}`
	
	Immediately validate user data and log validation exceptions.

Session tokens should expire.	**The following code shows the correct behavior:**
	`long sessionAge = System.getCurrentTimeMillis() -`
	`session.getCreationTime();`
	`if (sessionAge > maxAge) {`
	`  session.invalidate();`
	`}`