Web Application and Web Service Security: Avoiding Internal Application Vulnerabilities

By Adam Kolawa

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 can make a 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.

A multi-tier strategy can help identify, correct, and prevent security vulnerabilities that stem from the application.

- At Tier 1, analyze the source code in the source code repository and use static analysis and unit testing to expose coding issues that make the application vulnerable to security attacks.
- At Tier 2, exercise the deployed Web application or Web service to determine whether test inputs can result in security breaches and—for Web applications only—whether the Web site pages are constructed in a way that makes the application vulnerable to security attacks. This level of analysis—with limited visibility into the application's construction—is essentially black-box security testing.
- At Tier 3, perform runtime error detection on a deployed Web application or Web service to expose potentials for SQL injection, tainted input reaching vulnerable functions, buffer overflows, and denial of service attacks. This level of analysis—with full visibility into the application's construction—could be considered white-box security testing.

This multi-tiered strategy is used because each approach informs the other. That is, many problems can be discovered with one technique and explored further with another. With a single-tiered code scanning solution, it is not possible to automatically validate findings.

TIER 1: SOURCE CODE ANALYSIS

At Tier 1, source code is scanned to expose coding issues that make the application vulnerable to security attacks. The analysis includes two steps. The first step is static analysis, which verifies whether code complies with a set of security rules that identify positive or possible security vulnerabilities. The second step is unit testing, which involves testing software code starting with its smallest functional point, which is typically a single class or method. Unit testing can be extended to span through units and sub-modules, on to modules and applications. If each individual unit is thoroughly tested in isolation, then with other units as part of a sub-module, module, or application, most of the errors that might otherwise surface over the software's lifecycle will be detected or prevented. The objective of unit testing should be to verify the code's functionality and construction/robustness (e.g., the code's ability to handle unexpected or exceptional situations). This becomes increasingly important with new technologies such as Web services, where internal system interfaces are exposed to the outside world. If code in these systems is not tested appropriately, unpredictable behavior can not only lead to functionality and performance problems, but also provide hackers a way to enter and manipulate the system.

Static analysis exposes different types of security problems than unit testing. Essentially, there are two levels of static analysis:

- Scanning for legitimate mistakes that can lead to buffer overflows, SQL injection, and other security breaches.

- Scanning for malicious code which is purposely added to create a backdoor entry—for instance, by calling a random number generator in a strange way, having every n transactions behave differently, or allowing dynamic linking of other libraries that can overload classes.

Although many security issues can be positively identified through automated testing, manual inspection is required to evaluate some potential security issues. For example, backdoor entries often look like valid application code, and valid application code may sometimes resemble backdoor entries; tools cannot make the subjective analysis required to determine whether suspicious-looking code is actually a backdoor entry. However, tools can pinpoint the suspicious areas of code that require human inspection, making the evaluation process as rapid and accurate as possible. Automating the code inspection process typically reduces the time required for code reviews by a factor of ten.

Tier 1 analysis can and should be performed as soon as a developer completes a piece of code. To ensure that code modifications do not introduce security problems, Tier 1 tests should be added to the team's regression test suite and be executed as part of the team's nightly build and testing process.

TIER 2: DEPLOYED WEB APPLICATION AND WEB SERVICE APPLICATION RUNTIME ANALYSIS

Although static analysis is an essential tool for verifying security, it cannot detect all problems. For instance, only some memory corruption problems can be discovered statically; most cannot be found unless the integrated operational application is exercised dynamically.

Tier 2 runtime analysis involves dynamically exercising and scanning the deployed application, which can be located on a staging server or the production server, from the client side. The general analysis procedure is this: for any data that an attacker can modify (including user input, referrer ID's, cookies, and hidden fields), a set of malicious attack-like inputs designed to expose potential security vulnerabilities is submitted. These inputs can be generated automatically, then the application's response can be checked by parsing the HTML or XML code and HTTP response returned after that submission and verify whether it matches a pattern that indicates potential security problems.

This general strategy can be applied to any deployed and integrated Web application or Web service application—no matter what language it is built with, what platform it is sitting on, or what operating system it is running on. It diminishes the need to develop and execute complete attack scenarios—a task which requires considerable time and effort—because it can verify how the application responds to potential attacks by submitting attack-like inputs and verifying the application's response. This analysis cannot begin until after integration has started and the application can be exercised by an actual or test client. As a result, it is typically performed by the QA team. However, a fully-deployed application is not required. For tips on deploying and testing Web applications piece by piece—a process Parasoft calls "Web-Box Testing"—see *Bulletproofing Web Applications* by Kolawa, Hicken, and Dunlop (John Wiley and Sons, 2001). Regardless of how early Tier 2 analysis is started, the related tests should be added to the team's regression test suite and executed nightly to ensure that code modifications do not introduce Tier 2 security errors.

The specific analysis procedure varies for Web applications and Web services. The following sections will discuss the details of each application.

WEB APPLICATIONS

Web application runtime analysis is a two-step process. First, each Web page is scanned to determine whether it is constructed in a way that makes the application vulnerable to attacks. For example, this technique can be used to identify whether the page is vulnerable to attacks launched via:

- Malicious code (through source such as query strings, URLs and pieces of ULs, posted data, cookies, and persistent data supplied by users then retrieved at a later date [for instance, from a database]).
- Elements of Web integration (for instance, attacks via Flash or embedded files or objects).

It is also helpful to verify whether code follows appropriate security best practices that protect the application against HTTP-based attacks. For instance, the following best practice rules can be checked via static analysis:

- All comments and parts of the HTML that are commented out should be stripped out from Web pages.
- Ensure that page information is not cached locally on shared hosts/proxy servers, the meta "no-cache" lines should be inserted in the HTML HEAD of pages.

The following best practice rules can be checked via dynamic analysis:

- When client input is required from web-based forms, avoid using the "GET" method to submit data; GET methods should not be used because they leave a telltale signature in proxy caches.
- Limit application responses and limit server responses.
- Encode HTML-entity encode output that is generated from user modifiable data to avoid cross-site scripting and similar attack.
- Validate input parameters and accept only a specific set of characters.
- Filter output based upon input parameters for special characters.
- Secure authentication procedures.
- All data should be re-validated and sanitized at the receiving server to ensure the data is correct and has not been tampered with.
- When data is submitted to a server, always limit the type of acceptable data as much as possible by using strict validation rules. Programmatically, always ensure that the default data processing rule is "fail" - only accept the data if it is of the correct type, falls within the specified bounds (minimum and maximum lengths) and contains expected content.
- Any security checks should be completed after the data has been decoded to its simplest form (through canonicalization) and validated as acceptable content (e.g. maximum and minimum lengths, correct data type, does not contain any encoded data, textual data only contains the characters a-z and A-Z etc.).

The second phase of Tier 2, which could be called automated penetration testing, is where it is possible for black-box testing to uncover problems such as SQL Injection, Blind SQL Injection, XSS, Command Injection, LDAP Injection, XPath Injection, Access Control, Session Management, and so on. This phase involves checking how the application responds to attack-like inputs. All opportunities for an attacker to modify data (including user input, referrer ID's, cookies, and hidden fields) are identified and "safe" values for each instance of user-modifiable data are determined. Next, for each data modification opportunity found, a set of malicious inputs are submitted. The application response is verified by parsing the HTML code and HTTP response returned after each input submission and verifying whether it matches a pattern that indicates potential security problems. This pattern can be checked by running a static analysis rule that cross-references the actual response with the submitted input. Using this process, it is possible to expose security vulnerabilities that occur when input is not validated and the input is output unencoded (for example, cross-site scripting and code insertion).

WEB SERVICES

The application's response to attack-like inputs is also checked for Tier 2 analysis of Web services. All input opportunities through XML variables are identified. Next, for each input opportunity found, a set of malicious inputs is submitted. The application response is verified by parsing the XML response returned after each input submission and verifying whether it matches a pattern that indicates potential security problems. This pattern is checked by running a static analysis rule that cross-references the actual response with the submitted input.

TIER 3: RUNTIME ERROR DETECTION

In Tier 3, the application is exercised from its front end and the following types of analyses are applied:

- SQL injection detection
- Data flow analysis
- Memory corruption and memory leak detection

All three of these analyses cannot be started until the application is fully integrated and development has passed the application on to QA. Consequently, these analyses are typically performed by the QA team. As with all other tiers of analysis, these tests should be added to the team's regression test suite and continued throughout the remainder of the project's lifecycle.

For SQL injection detection, the application is exercised through its front end and an SQL proxy sits between the middleware and the back end. During this process, the messages passing through the application are monitored at the proxy level. For instance, SQL injections can be discovered by exercising the application with specific scenarios while the proxy monitors SQL statements and identifies suspicious ones.

Data flow analysis is designed to identify when tainted inputs are passed to vulnerable functions. First, functions are divided into three groups:

- Functions that can bring tainted inputs
- Functions that are used to clean tainted inputs
- Functions that are vulnerable to tainted inputs

Next, the data returned from tainted input calls is traced until either 1) it is passed through a security policy function that can clean it, or 2) it is passed to a function that is vulnerable to tainted inputs. If #1 occurs, the tracing stops. If #2 occurs, an error is reported.

If an application is built with C/C++, memory corruption and memory leak detection should also be performed during Tier 3. Memory corruption (especially memory corruption on the stack) indicates a potential for buffer overflows, which could allow serious security attacks, and memory leaks make the application more vulnerable to denial of service attacks.

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.

NaSPA member Dr. Adam Kolawa is co-founder and CEO of Parasoft, a leading provider of Automated Error Prevention (EAP) software solutions.