Iron Chef Black Hat

Brian Chess, Jacob West
Sean Fay, Toshinari Kureha
{brian, jacob, sean, tkureha}@fortify.com

August 2, 2007

Iron Chef Black Hat is a competition between two tool-building titans: Sean Fay and Toshi Kureha. Sean and Toshi have one hour to use tools that they have built in order to find vulnerabilities in code they've never seen before. Toshi builds runtime analysis tools, while Sean builds static analysis tools. A panel of judges will decide who's hack-fu is superior.

This paper outlines the runtime and static analysis techniques that Toshi and Sean will use.

Finding Security Vulnerabilities with Runtime Analysis

There are many ways to perform run time analysis to discover security vulnerabilities in the application. This paper describes the workings of traditional vulnerability scanners and their limitations, and a companion tool that addresses those limitations.

Traditional Vulnerability Scanners

Traditional vulnerability scanning involves 2 steps:

1. Navigating Through The Application
2. Testing The Application

Navigating Through the Application

Before testing can begin, we need to find the test target. For web application, this means finding all the different web pages, the parameter that this web page takes, and the sequence of navigation that makes this all possible.

One common way of achieving the discovery of the test target is by automatic web crawling. In this scenario, the user enters a starting URL, and the vulnerability scanner tool will automatically navigate through all the various links, submit forms, and find all the web pages.

In simple web applications, this method works very well – within a matter of minutes, it finds all the different web pages and parameters for this web application.

However, for more complex web applications, this runs into several problems. The first problem is that there might be thousands of similar pages. In the case of e-commerce application, there might be one 'functionally unique' page, but several thousand products use that same page (say "productDetail" page). In this case, these redundant pages cause significant overhead in the tool making it unusable in many

situations.

In the other case, the web crawler simply does not have the ability to crawl through the application because the application requires the user to step through very specific steps – add an item to the shopping cart before checking out, buy a stock before selling a stock, etc. An automated crawler does not have the intelligence needed to crawl through the application using the required sequence.

The other method for discovering the test target is through manual crawl of the application. In this case, the tool instructs the user to use some kind of a browser to navigate through the web application manually while it records the navigation sequence, effectively telling the tool what that test target is and how to navigate through them. This often produces the desired result of finding the correct set of test target, but can be time-consuming. For tools that are not mature, AJAX-based applications often cause even manual crawling to be ineffective, where navigation parameters are highly dynamic.

A similar approach to the manual crawl is to import existing QA test cases into the vulnerability scanner. Since the QA test cases often already exists (for the development team to test the functionality of the web application), and that these QA tests are usually very comprehensive about testing all the feature sets of the application, they are prime sources of test target.

Testing the Application

Once the test targets have been identified, then testing can begin. We can break the testing phase into 2 phases:

1. Fuzzing
2. Analysis

Fuzzing is a technique that sends various different inputs into the test target. It may send malformed characters or special characters such as a single quote.

The result of fuzzing is usually analyzed using two different approaches:

- Signature Analysis
- Behavior Analysis

Signature analysis analyzes the response to see whether anything in the response matches any of the known signatures that have been known to be indications that a vulnerability exists. For example, if the response contains "ODBC Error" right after fuzzing, then we know that it most likely has a SQL Injection vulnerability because "ODBC Error" is a string that is in the signature database for SQL Injection vulnerabilities. This "vulnerability signature database" is often a compiled list of signatures based on extensive research and real world experience.

The other approach is behavior analysis. That is, rather than looking for a particular string to appear in the response, this approach looks at the behavior of the application. Are the results same as before? How has the response changed? As a specific example, to find a SQL Injection vulnerability based on behavior, this approach would send a normal query – say id=3. Store the result. Then send a second query that is semantically the same – say id=3 AND 1=1. If the response is the same, then it is very likely that there is a SQL Injection vulnerability. This approach is very effective when the application has good

error handling but has not fixed the actual vulnerability. This approach can have many different variants to make it effective, including sending false & null conditions.

Limitations

Despite the many different technique and advances made in traditional vulnerability scanning approaches, limitations exist.

- Unknown Coverage
- No Remediation Information
- False Positives
- False Negatives

Lack of Coverage

When you run a vulnerability scanner, you usually get a list of URLs that vulnerabilities were found. What you don't get is how comprehensive the scan was in actually testing all the security critical areas of the application.

No Remediation Information

Vulnerability scanners provide very generic remediation recommendations, such as "Do input validation, and use prepared statements and bind variables." It does not provide where in the code the problem needs to be fixed – file name, line number, etc. This is a limitation caused not by the tool themselves but the approach itself – it has no visibility into the application code.

False Positives

False positives are vulnerabilities that are reported but are not real vulnerabilities. This happens because of the guesswork involved in the analysis methods used by these tools. For example, if a 'vulnerability signature' appear in a valid page, this will cause the tool to think there is a vulnerability on that page, which is a problem if the tool is – for example – testing a search page for a list of common error messages for a database, where these common error messages are often part of the vulnerability signature database.

False Negatives

False negatives are vulnerabilities that the tool failed to find. False negatives can happen for 3 reasons. The first scenario is if the tool didn't test the page at all. If it didn't test, and if it has a vulnerability, then the tool will obviously not find the vulnerability. The second scenario is if the tool cannot conclude that there is a real vulnerability. Assume that tool uses only signature-based analysis method. If the application has a good error handling code, then error messages from the database is not often reported back to the user – instead, they will see a generic error message, causing the tool not to find a match in its vulnerability signature database. Using behavior analysis, if the page contains dynamic content that is different in every request, most behavior analysis techniques would fail because in this case the original response and the 'true response' would differ due to the dynamic content (this can be mitigated somewhat using more complex differential analysis where multiple requests are sent to determine the 'static' vs. 'dynamic' parts and conducting analysis only on the static part, but most tools currently do not do this). In the third scenario, the tool simply cannot find entire categories of vulnerabilities because it has no

outward manifestation. For example, logging a credit card number to a plaintext log file has no manifestation in the HTTP response. The tool will never be able to find such vulnerabilities.

Overcoming Limitations Through Bytecode Injection – A Companion Tool

To overcome these limitations, making small adjustments to the traditional vulnerability scanning approaches will not solve these issues. An entirely new approach is required. The approach taken in this paper is through bytecode injection.

Our proposal is this: put "monitors" inside the application to observe what the vulnerability scanners do. Here, we have monitors looking at both incoming & outgoing traffic through the application, as well as the execution of security critical functions.

In order to make this happen, we need to answer 3 basic questions:

1. How do I inject the monitors inside the application?
2. Where inside the application do I inject the monitors?
3. What should the monitors do?

How do I inject the monitors?

One way to inject the monitors is the simply put them into the source code of the application, recompile the code, and redeploy the application. But this can be challenging if you don't have the source code or if you don't have a compilable environment.

On the other hand, it is definitely more feasible to simply have access to a running application – whether that is a WAR file for a Java application or a MSIL dll for a .NET application. Assuming you have access to a running application, what we propose is to inject the monitors through byte-code weaving, using aspect tools using AspectJ for Java and AspectDNG for .NET.

Byte-code weaving takes classes/dlls and "aspects" and weaves them together to produce a binary-compatible .class/dll files that run in any normal Java / .NET environment. An informal description of an aspect is a file that describes a specific point in a program flow, and a body of code that gets executed at that point in time. For example, one aspect file may describe to print some statement to the console (body of code) when a SQL statement is being executed (specific point in a program flow).

Where inside the application do I inject the monitors?

Let's reflect back to what problems we are trying to solve. Two of the problems we are trying to solve are the following:

1. Gain coverage information
2. Reduce false negatives

Let's look at coverage. As a tester, you want to ensure that all attack surface exposed is tested. In a web application, you are interested in all areas that read input. In a Java application, they correspond to APIs like "javax.servlet.ServletRequest.getParameter" and its equivalent in various different web frameworks.

Now, let's look at reducing false negatives. Recall the 3 possibilities why web scanners fail to detect vulnerabilities – didn't reach, difficult to detect, impossible to detect. While the existence of monitors cannot directly help the 'didn't reach' situation, it can help the tester become aware of that situation so that they can improve the test to gain better coverage. To combat the "difficult to detect" and "simply cannot detect" causes – we propose to put the monitors directly at all the security related APIs. For "difficult to detect vulnerabilities" such as SQL Injection or Command Injection, the best place to monitor is right at the API call. For SQL Injection, it would be "java.sql.Statement.executeQuery"; for command injection, it would be "java.lang.Runtime.exec". If unfiltered user input reaches these APIs, you don't need to take a look at HTTP response to know that there is a vulnerability – the user is in fact directly manipulating these security sensitive APIs. For previously "impossible to detect" vulnerabilities, we propose the same – put the monitors directly on the relevant API call. For example to detect vulnerabilities such as logging credit card numbers to a log file, we will put monitors directly on the file writing APIs. If what is being written to a file using that API contains credit card numbers, then we flag it as an vulnerability.

What should the monitors do?

Once the monitors are in place, all we need to do to get coverage information is to signal when the monitor's code is called. This can be as simple as logging the name/location of the monitor that was hit.

To detect whether a vulnerability exists in the area monitored, the monitors need to do some real work, and this depends on the nature of vulnerability that you want to detect. Take the example of SQL Injection. A typical web scanner will send a single quote to attempt to generate an error in the application and look for error messages coming back in HTTP response. Instead, using injected monitors, we can examine the actual argument passed to the executeQuery API. If the sql statement being executed contains an odd number of single quotes, that means that the user's single quote reached into the sql statement without being validated. We can immediately determine that this is a SQL Injection vulnerability. While this is a very simple illustration of what monitors can do, it illustrates a very important point: it is much easier detecting vulnerabilities at the source rather than through the HTTP response.

The main advantage here over the web scanner is the ability to look at the sql statement being executed. If the application has proper error handling mechanism or simply has an error page configured, most web scanners will fail to detect this SQL Injection vulnerability. On the other hand, the monitor will have no trouble determining the existence of this SQL Injection vulnerability.

What about the kinds of vulnerabilities that web scanners simply cannot detect? What should the monitors do to detect these kinds of vulnerabilities? Let's take a concrete example – say that we want to detect instances of privacy violations where credit card numbers are being logged to a file. Since logging a credit card number has no manifestation in the HTTP response stream, there is no chance web scanners can find this vulnerability. The monitor, on the other hand, simply needs to run a regular expression on the argument to a file write API to see whether a credit card number is being written to a log file. Similar to SQL Injection case, the monitor's task is to examine the content of the argument to the API and compare it against its 'ruleset'

Many aspect technologies have the ability to determine the current location of the code being executed. This information – such as the class file name and the line number being executed, are valuable remediation information that can be fed back to the developers.

Combined Approach

With both of these tools in place, users now have the advantage of having a strong test driver – a traditional vulnerability scanner – with a strong analysis engine that works inside the application itself. Now, the users can – in addition to finding actual vulnerabilities that they were able to find before – improve their scans based on coverage information reported by the monitors, find more vulnerabilities, weed out false positives, and provide remediation information back to the developers.

Finding Security Vulnerabilities with Static Analysis

What is Static Analysis?

Static analysis refers to an automated inspection of properties of a software application that is performed without executing the application. Static analysis is typically used to reveal defects in software and assist in code reviews and development. Although the code review process can never be fully automated, static analyzers can make code review more efficient and more comprehensive by consistently applying checks across large bodies of code that would take thousands of hours to perform by hand.

Static analysis can take many forms, including:

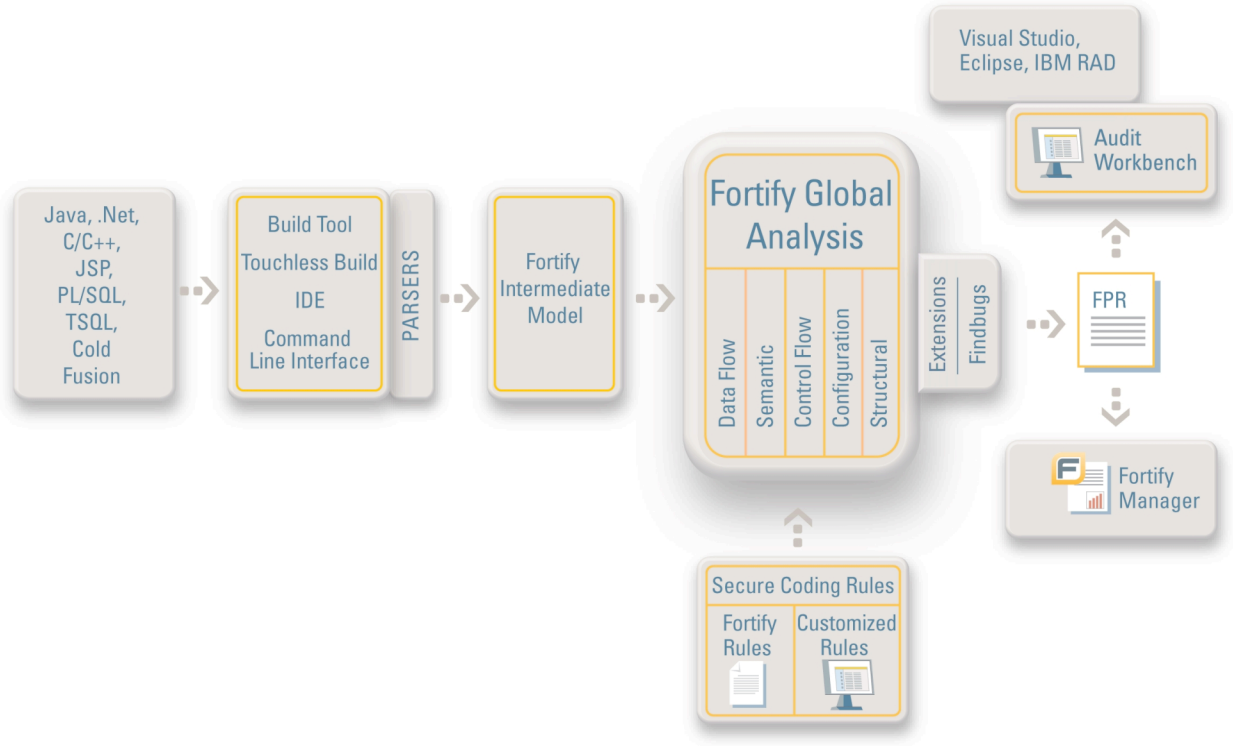
- Type checking
- Style checking
- Program understanding
- Program verification / Property checking
- Bug finding
- Security review

Developers will find some of these familiar: statically typed languages such as Java, C#, C and C++ have type checkers built into the compilers; a program will not compile if the type checker finds errors. Style checking and program understanding tools are also common in the development world. Most modern IDEs have these types of static analysis built-in to assist programmers. Bug finding tools such as FXCop (for .NET) and FindBugs (for Java) are gaining popularity.

SCA: A Static Analysis tool for Security

SCA is a static analysis tools designed to aid in the security review process. Its first step – referred to as translation – is to convert the application source code into an intermediate format appropriate for analysis. Once the intermediate model is built, SCA is ready to analyze the application. The input to analysis is the intermediate model and a set of secure coding rules that describe security-relevant properties of APIs used by the source code. SCA then uses a combination of different analysis techniques to identify potential

security issues in the code.



Some examples of code-level security problems that can be detected by SCA include:

- **Injection Vulnerabilities and Unvalidated Input**
  Interprocedural taint propagation analysis allows SCA to detect when unvalidated user input may be used in sensitive operations such as file operations or SQL queries.
- **Unsafe Memory Operations**
  By building a bit-level model of the operations performed by the source code, SCA can precisely check the safety of memory operations such as buffer copies.
- **Time of Check / Time of Use**
  Security issues often arise from unsafe sequences of operations. The SCA Controlflow Analyzer can detect paths through the code through which an unsafe sequence of operations can occur.

Many other types of issues can be detected as well, and the set of issues adding custom rules to the set up secure coding rules used during analysis.

Using SCA to find security problems

Running Analysis

The first step is running an analysis is gathering all the code. Though this may sound trivial, it can often be challenging to gather all the source code along with its library dependencies, build files and deployment configurations in one place and configure the build environment correctly. SCA is tolerant of missing dependencies and incomplete code, and analysis can be run outside of the build environment. But these missing elements will have an effect on the comprehensiveness and accuracy of the analysis, so

it is best to have them all in place. Often it is simplest to run the analysis on the build server for the application as part of the build process.

Reviewing Results

Before sitting down the review the results, you should already have thought about what "secure" means in the context of your application. The process of reviewing results will involve looking at both individual issues and categories of issues reported by the analyzer and making a determination about whether each issue represents a threat to the security of your application. There are many great resources available for learning about how to evaluate the security context of your application, so we won't go into any great detail here. Some important questions to have answered are:

- What resources does the application protect?
- What are the attack scenarios?
- Is the application's environment trusted?
- Are the users trusted?

The first step in tackling the results will be to eliminate or reprioritize categories of issues based on the security context of the application.

Finding Cross-Site Scripting: An Example

In this example, we'll use SCA to find Cross Site Scripting problems in an unnamed open-source application. We selected a J2EE web application of approximately 20k lines of code.

Step 1: Scan the Code

Since this project was a J2EE web app, we used a standard 3-step process to run the analysis:

- Translate all the Java files via the ant build script
- Translate all the JSP and configuration files
- Run the analysis

The whole process took about 10 minutes.

Step 2: Investigate the Issues

The analysis results contain dozens of categories of issues, including four which are ranked in the hotlist as high-severity security issues:

- Cross-Site Scripting
- HTTP Response Splitting
- SQL Injection
- System Information Leak

For the purposes of this example, we're only look for Cross-Site Scripting issues.

The first issue we look at has a very simple dataflow trace: Tainted user data is assigned to a variable in a JSP from a call to request.getParameter(), and a few lines later is written out via the <%= … %> tag:

```
String theme = request.getParameter("look");
…
```

```
<link rel="stylesheet" type="text/css" media="all"
        href="<%= request.getContextPath() %>/roller-ui/theme/<%= theme %>/colors.css" />
```

This is a pretty obvious bug and doesn't require too much thinking on our part.

The next issue is a little trickier. The dataflow trace passes through several function layers and files. In the end it boils down to a page printing out a list of usernames which come from the database. We decide that this is a potentially serious bug too; if a new user can register with the site and embed special characters in their username (we don't know what if any validation is performed on the username at registration), then they have a potential Cross-Site Scripting vector to what looks likely to be an administrative page.

There are many more issues in this vein reported by SCA. Digging a little deeper though, we uncover even more problems that could lead to Cross-Site Scripting vulnerabilities. By investigating the Trust Boundary Violation issues reported by SCA, we discover a number of places where unvalidated user input is stored in the attributes collection. Doing a little manual review quickly shows that there is rendering code that pulls data out of the attribute collection using the same keys and renders it unfiltered, creating yet another Cross-Site Scripting vector. Although this type of indirect path is difficult for the automated tool to follow, a little manual effort on top of the clues provided by the analysis yields many more serious issues.

Taking it Further

Customizing for Your Code

The greatest leverage from a static analysis tool is achieved through customization. A static analysis tool provides a powerful framework for checking properties of source code, but is dependent on a specification to understand external APIs and to understand what properties to check. By adding custom rules to the Secure Coding Ruleset, development teams can enable enforcement of custom coding standards, tune results for their specific project, and allow SCA to better understand any 3rd party APIs that may be in use.

For the codebase we analyzed for our example, we could imagine wanting to enforce some custom coding standards to solve the Cross-Site Scripting problems endemic in the code. For instance we might want to require that user input be validated before being stored in the database or in the attributes collection. If this were the case we would write a rule to flag any unvalidated user input being stored in the database, and would elevate Trust Boundary Violation issues to high-severity. Once we were certain that the proper validation was in place, we could also write a rule to tell SCA to treat data coming out of the database as validated, allowing the analyzer to know that rendering data from the database is a safe operation.

Conclusions

Our short example doesn't take us through the entire process of a tool-assisted security review of an application. There are many more important components to a security review, and many more important types of issues to check for. Our examples demonstrates that using a static analysis tool can greatly increase the breadth and efficiency of a security code review by making it easy for a human auditor to uncover complex code-level security issues that would take hours to track down manually.