Countering The Faults Of Web Scanners Through Byte-code Injection

Introduction

Penetration testing, web application scanning, black box security testing – these terms all refer to a common technique of probing a running application with various inputs in order to simulate attacks and identify potential vulnerabilities in an application. While effective at finding some vulnerabilities, these tests lack transparency and their thoroughness is often debated. In this whitepaper, we will first take a look at what penetration testers thought about the thoroughness of their tests and second, the results of conducting automated web scanning against a number of open-source applications. We will then explore the shortcomings that become obvious as we probed deeper into those results and the application itself. We will then go through a proposal that attempts to address these shortcomings in web scanners.

The Process

We started our analysis by conducting a survey of several different security experts and developers to see how effective they thought their penetration tests were. After collecting the results from the survey, we analyzed an expert penetration tester as he conducted automated and manual penetration tests on five test applications ranging in size and scope. We used both a general code coverage tool called EMMA and a specialized tool called Fortify Tracer, which sits inside the application being tested and examines what's being tested.

Findings

Between our survey and the extensive analysis we generated three key findings:

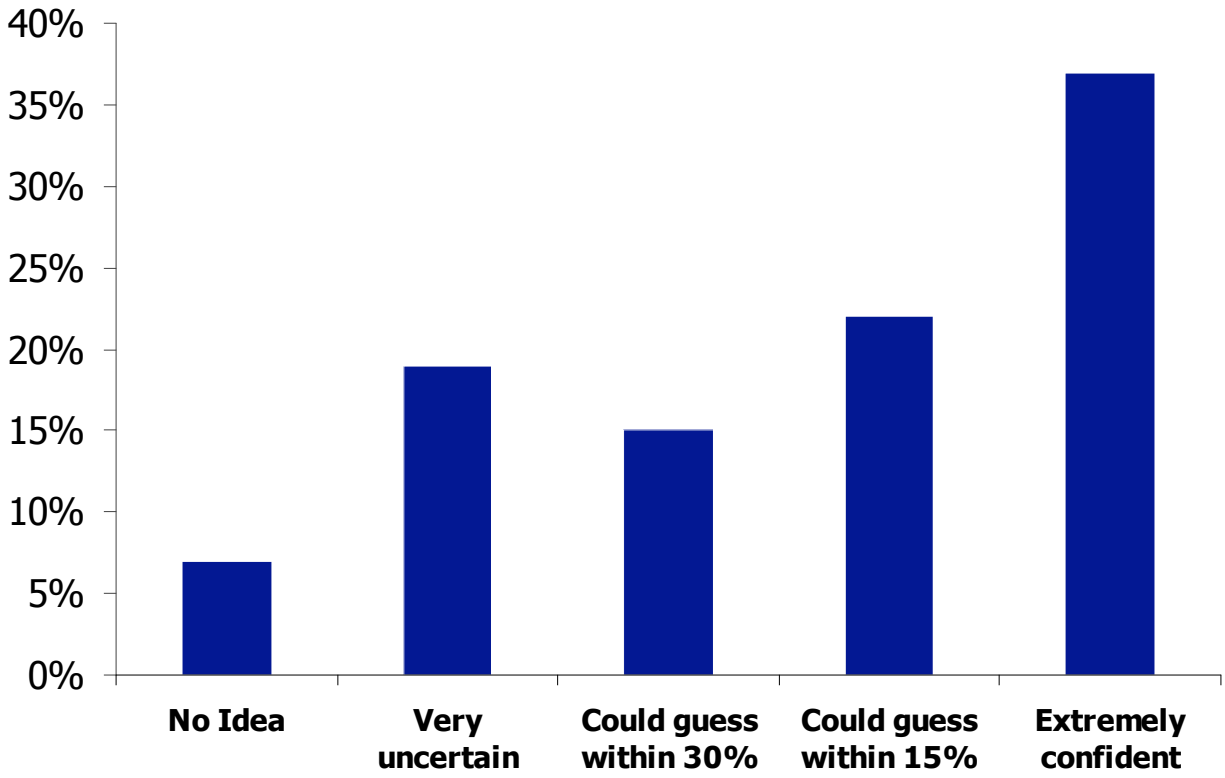
1) Penetration testers admit a low level of awareness in terms of thoroughness but estimate high coverage
2) In our analysis, automated and manual penetration testing yielded very low and inconsistent coverage numbers, averaging only 25% of security related APIs

3) Even when a penetration test reached a portion of an application, it often failed to identify a critical vulnerability

Finding 1: Penetration Testers admit a low level of awareness in terms of thoroughness but estimate high coverage

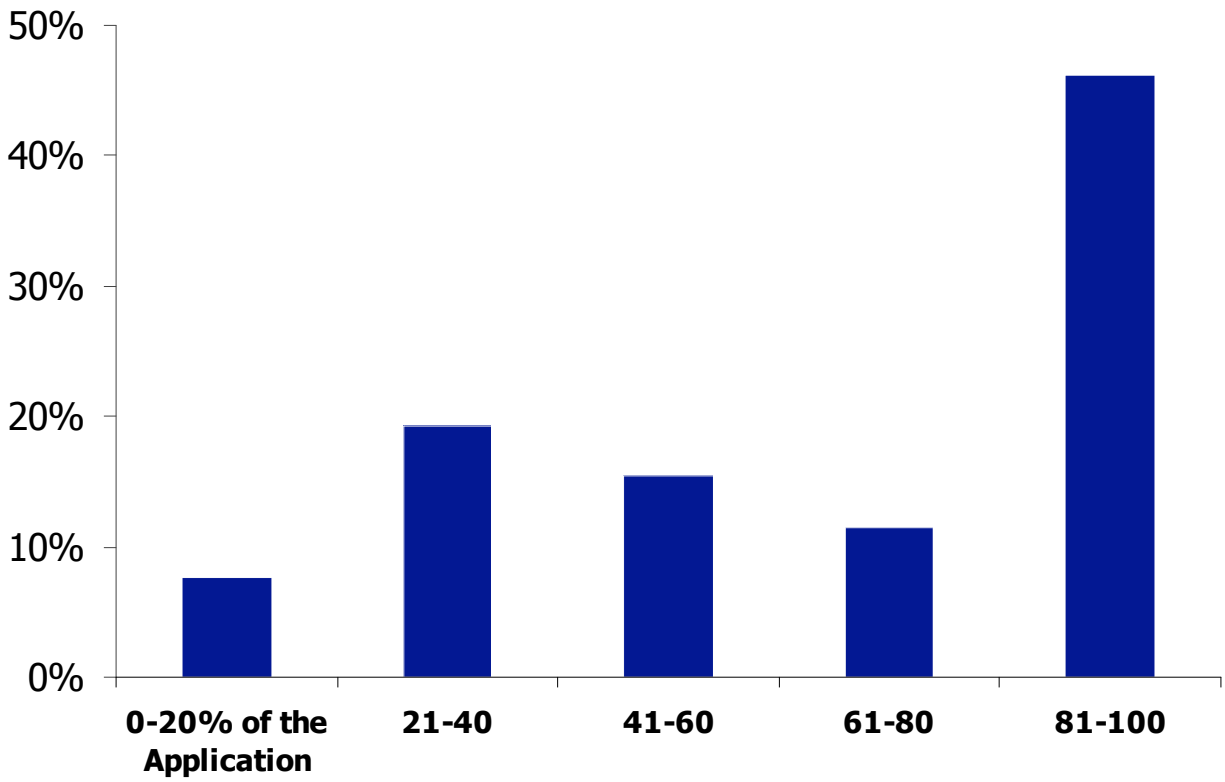
The nature of black box security tests is such that the test operates outside the application, with no knowledge of the internal code structure or how data flows throughout the application. As a result, there is an inherent lack of information about what is taking place and how thorough the test really is. We wanted to get a sense for how confident security experts were in the tests they conducted. We surveyed experts at various organizations and asked them several questions about the penetration tests they conducted on their applications. The questions that generated the most insightful answers were:

Do you know how thorough your penetration test is?



The answers were fairly split with only 37% saying they were extremely confident, the remainder acknowledging they were not confident to a reasonable degree, and over a quarter saying they were very unaware. The next question forced the respondents to estimate the percentage of their application they were able to test when conducting a black box test.

If you had to estimate, what percentage of the security APIs do you think your test hit?



Almost 50% of the respondents estimated that their penetration tests were able to reach at least 80% of their application's security related APIs, and the majority believed they could hit over 61%. We were surprised by how many people believed strongly in the thoroughness of their penetration tests and wanted to investigate these claims. While nobody actually currently tracks these statistics on individual application, we conducted penetration tests using two of the most common commercial web application vulnerability scanners on a number of open source applications by using a standard code coverage tool called EMMA, and a specialized tool called Fortify Tracer that analyzes coverage of security critical functions within an application.

Finding 2: **In our analysis, automated and manual penetration testing yielded low and inconsistent coverage numbers, averaging only 25% of security related APIs**

After instrumenting our test applications with EMMA & Fortify Tracer, we performed multiple penetration test against them using two of the market leading penetration testing tools. While these tools claim to provide thorough scanning capability, we found a noticeable difference in their effectiveness. Below we show the percentage of security critical points that these two penetration testing tools were able to generate on our five test applications. We pointed the tools at our applications and let them run, with no customization or manual work to tailor the test to our application other than specifying login credentials.

Coverage of Security Critical Areas

Test Applications	Coverage % Penetration Testing Tool 1	Coverage % Penetration Testing Tool 2
Test Application 1	42%	27%
Test Application 2	46%	10%
Test Application 3	13%	8%
Test Application 4	28%	27%
Test Application 5	18%	29%
Average	**29%**	**20%**

As you can see, not only are the coverage numbers inconsistent between the two tools (leading one to question their effectiveness), they are extremely low. To some extent this is to be expected, as most companies augment automate testing efforts with manual testing. However, after manual testing the applications, we were only able to increase the coverage by – on average - a small amount.

Coverage of Security Critical Areas

With Additional Manual Efforts

Test Applications	Coverage % Penetration Testing Tool 1	Coverage % Additional Manual Efforts
Test Application 1	42%	55%
Test Application 2	46%	49%
Test Application 3	13%	29%
Test Application 4	28%	73%
Test Application 5	18%	42%
Average	**29%**	**50%**

Test Applications	Coverage % Penetration Testing Tool 2	Coverage % Additional Manual Efforts
Test Application 1	27%	49%
Test Application 2	10%	5%
Test Application 3	8%	30%
Test Application 4	27%	68%
Test Application 5	29%	32%
Average	**20%**	**37%**

Finding 3: **Even when a penetration test reached a portion of an application, it often failed to identify a critical vulnerability**

False negatives are a failure to identify a legitimate security issue. We can think of three possibilities why this could happen. The first possibility is pretty obvious: the vulnerabilities that are not found because the penetration test failed to test that area of the application. The second possibility is very interesting – the vulnerabilities that web scanners fail to catch even when they actually reached that area of the application. How could this happen? Some vulnerabilities may be difficult to find using traditional web scanners because they may not display an obvious HTTP response indicating the existence of such vulnerabilities. Examples of this include "blind" injection vulnerabilities that offer very little clue about the existence of the vulnerability. Very often, if a web application configures its error page properly, these tools have difficulty finding these vulnerabilities because they rely on matching the HTTP response with a set known signature – if a general error page is shown instead, the tools cannot determine the existence of such vulnerabilities. Finally, the third possibility is that web scanners are not capable of finding certain vulnerabilities. Examples include vulnerabilities like privacy violation that may occur when an application needlessly writes social security numbers or credit card numbers to a log file that a regular web administrator may view. Since these types of

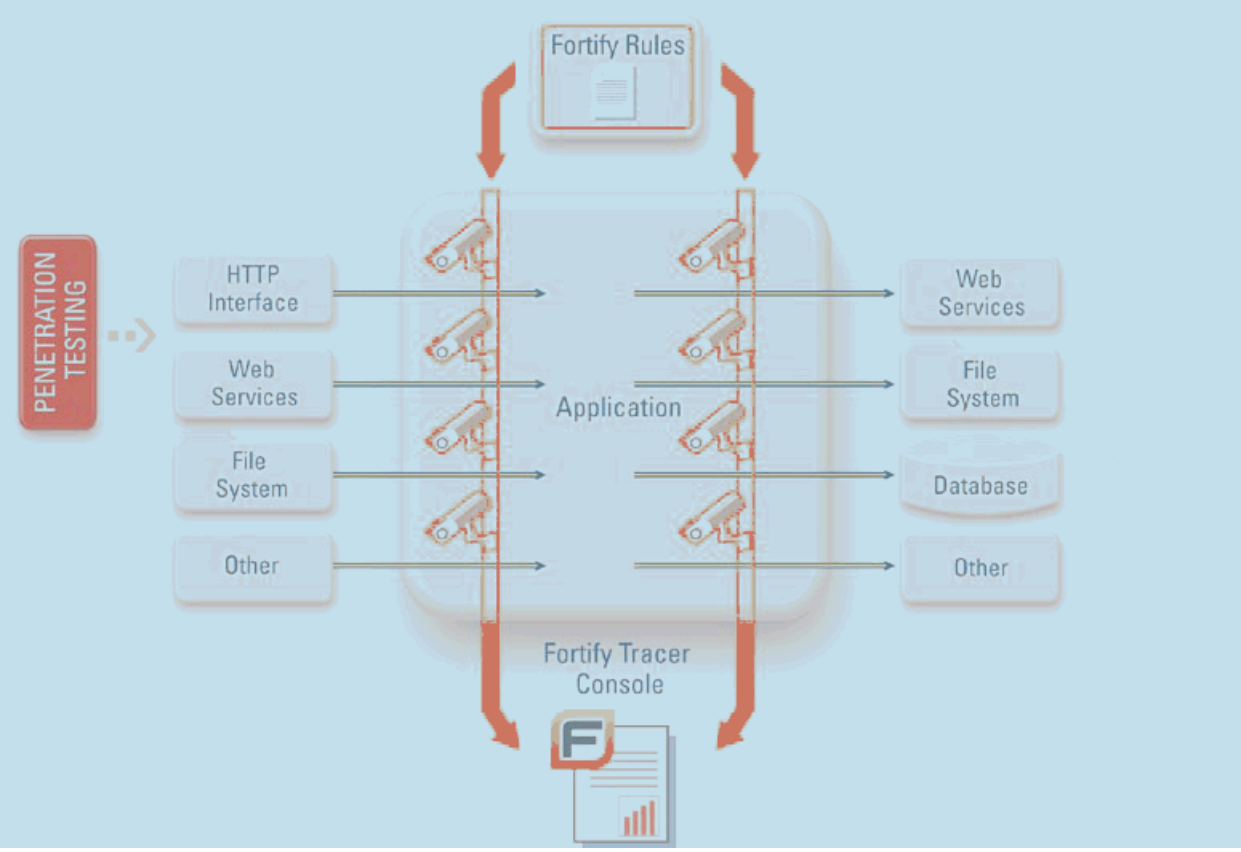
vulnerabilities are not possible to determine through the HTTP interface, web scanners simply cannot catch these vulnerabilities.

The first two findings – lack of coverage information & resulting low coverage numbers – are related. Since the testers don't know the coverage of their test, they cannot improve on the test to yield high coverage. The key to solving the lack of coverage information is to gain insight into the application while the tests are running – to understand which areas of the application are being exercised and which parts aren't. Solving the false negative fault of the web scanners also seem to indicate a similar solution – to gain insight into the application while the tests are running – specifically to monitor security related APIs as they are being exercised. In the next section, we try to solve the faults of web scanners by doing exactly that – putting monitors inside the application to gain insight into the application while it is running.

Proposal: Injecting Monitors Through Byte-code Injection

The fault and findings that we have uncovered in regard to using these web scanners all seem to share a common theme – lack of information about what is happening inside the application while the tests are running.

Our proposal is this: put "monitors" inside the application to observe the penetration testing. The below is a conceptual model of what this process looks like:



Here, we have monitors looking at both incoming & outgoing traffic through the application, as well as the execution of security critical functions. The monitors are controlled through a centralized 'rules' file for modularity, and the results of what the monitor sees is fed into a 'console'.

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. The 2 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'.

Proof Of Concept

In order to demonstrate this concept, we will go through an actual example.
The following is an example of a simple SQL Injection monitor:

```
aspect SQLInjection {

        pointcut sqlExec(String sql):
                call(ResultSet Statement.executeQuery(String))
                && args(sql);

        before(String sql) : sqlExec(sql) {
                checkInjection(sql, thisJoinPoint.getSignature());
        }
        void checkInjection(String sql, Signature sig){
                if(Config.isInIgnoreMode()){
                        return;
                }
                // If there is an odd # of single quotes
                if (sql.split("'").length % 2 == 0) {
                        System.out.println("SQL Injection exists!!!");
                }
        }

}
```

The above SQL Injection aspect does a very simple check – if the sql statement about to be executed has an odd number of single quote, flag it as a SQL Injection vulnerability. The

location of the monitor is at "call(ResultSet Statement.executeQuery(String))". Right before "executeQuery" method is called, call the "checkInjection" code that looks for SQL injection vulnerability.

To find all input areas to a web application, we have the following aspect:

```
aspect CoverageWebSource
{
    pointcut coverageServletRequest() :
        call(String ServletRequest+.getParameter(..)) ||
        call(Enumeration ServletRequest+.getParameterNames(..)) ||
        call(String[] ServletRequest+.getParameterValues(..)) ||
        call(Map ServletRequest+.getParameterMap(..));

    /*struts*/
    //ActionForm
    pointcut coverageActionForm() :
        call(* ActionForm+.get*(..));

        before() : coverageServletRequest() || coverageActionForm()
        {
          System.out.println("HIT:" + fileName + " " + thisJoinPoint.getSignature() + " " +
thisJoinPoint.getSourceLocation().getLine();
        }
}
```

The above aspect is for illustrative purposes – we should ideally exhaustively put all known APIs to read web input into the above aspect file.

For this proof of concept, we took a simple struts based application called SPLC. When we applied our CoverageWebSource aspect to the sample application, it yielded 45 web input sources. When we ran one of the two most popular web scanners, we were immediately able to tell that it only hit 28 of the web input sources – about 62%. Furthermore, it has missed to identify a SQL Injection vulnerability in the application that the monitors had detected. When

we went back into the application and disabled a default error page in the web.xml file of the application, the web scanner was able to find 5 SQL Injection vulnerabilities in the same application.

Conclusion

While penetration testing has become a popular approach to testing applications for security vulnerabilities, it has significant shortcomings that need to be addressed. Due to the "black box" nature of these tests, not only do web scanners have difficulties reaching all areas of the application but also fail to report any information that aids the users find this flaw. In the areas that these tests actually reached, they often failed to identify key vulnerabilities. Using byte-code injection, users now can gain visibility into the coverage of these web scanners as well as detect vulnerabilities that these web scanners have previously failed to uncover. Running a web scanner and testing a running application is a crucial step – but it needs to be augmented with insight into the running application and its code.