# Dissecting and digging application source code for vulnerabilities

*by Shreeraj Shah*

## *Abstract*

Application source code scanning for vulnerability detection is an interesting challenge and relatively complex problem as well. There are several security issues which are difficult to identify using blackbox testing and these issues can be identified by using whitebox source code testing methodlogy. Application layer security issues may be residing at logical layer and it is very important to have source code audit done to unearth these categories of bugs. This paper is going to address following areas:

1. How to build simple rules using method and class signatures to identify possible weak links in the source code.
2. How to do source code walking across the entire source base to perform impact analysis.
3. How to use simple tool like AppCodeScan[1] or similar utility to perform effective source code analysis to detect possible vulnerability residing in your source base.

## *Keywords*

Source code audit, Code analysis, Code walking and digging, SQL injection, Source code audit rules, AppCodeScan, WebGoat, Regex

## *Author*

Shreeraj Shah, B.E., MSCS, MBA, is the founder of Blueinfy, a company that provides application security services. Prior to founding Blueinfy, he was founder and board member at Net Square. He also worked with Foundstone (McAfee), Chase Manhattan Bank and IBM in security space. He is also the author of popular books like Web 2.0 Security: Defending Ajax, RIA and SOA, Hacking Web Services (Thomson 06) and Web Hacking: Attacks and Defense (Addison-Wesley 03). In addition, he has published several advisories, tools, and whitepapers, and has presented at numerous conferences including RSA, AusCERT, InfosecWorld (Misti), HackInTheBox, Blackhat, OSCON, Bellua, Syscan, ISACA etc. His articles are regularly published on Securityfocus, InformIT, DevX, O'reilly, HNS. His work has been quoted on BBC, Dark Reading, Bank Technology as an expert. [shreeraj@blueinfy.com (e)] [http://shreeraj.blogspot.com/ (b)]

[1] AppCodeScan – http://www.blueinfy.com/tools.html

## Approach

In whitebox testing we are assuming to have full access to source code and we use this source as our target to determine possible vulnerability. We are going to use OWASP's WebGoat[2] as sample application to understand methodology. We will use AppCodeScan to scan Java based source code of WebGoat. AppCodeScan is simple tool to assist in dissecting and digging the source code; it is not an automated source code scanner.

## Scanning the Source

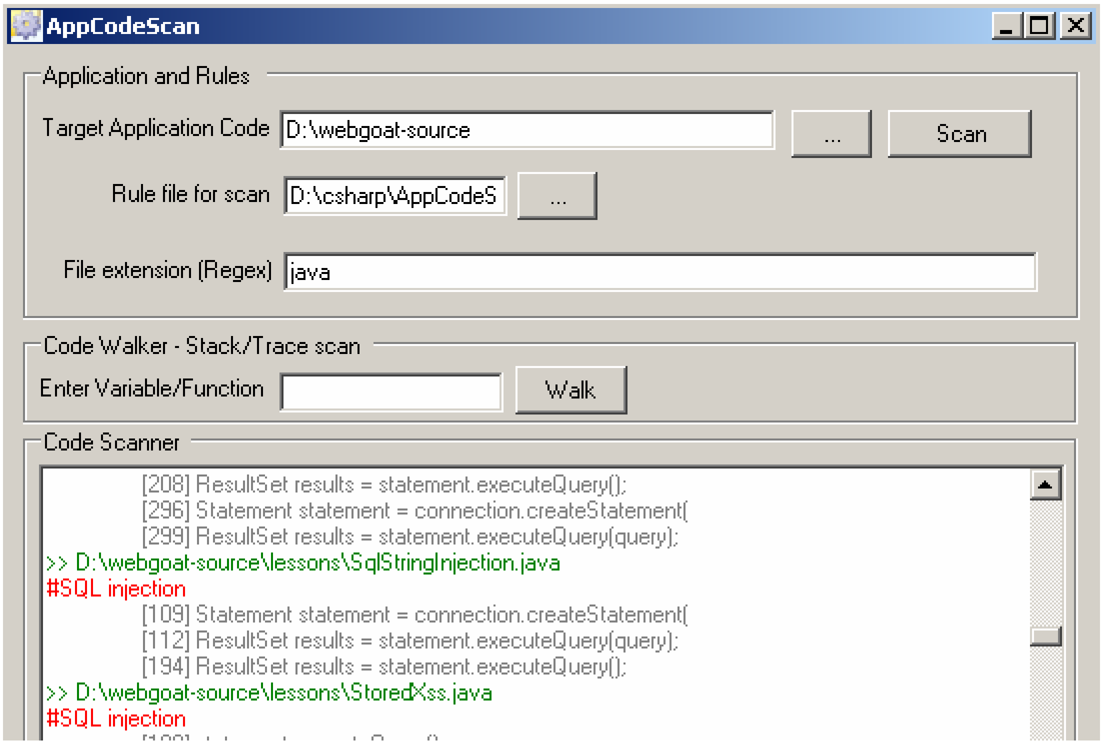
Source code of an application may be residing in multiple files with numerous lines of code. It is very important to define a start point for scanning. Application source may be vulnerable to several issues like SQL injection or Cross Site Scripting. First step is to identify possible line of code from which we want to start. Once we have that line of code we can do both forward and reverse tracking on it to unearth vulnerability or signing it off as secure call or code.

As shown in Exhibit 1 we have several regex patterns for thorough analysis of source code. We will look at these patterns at the end of this paper in detail but at this point let's focus on one type of vulnerability to understand both approach and tool.
For example, we are looking for SQL injection points in WebGoat application which is written in Java. We need to find high value call to analyze the source code and feed the rule to the AppCodeScan. Here is a simple method which we are looking at to see how SQL queries are implemented.

```
#SQL injection
.*\.createStatement|executeQuery.*
```

We are looking for how "executeQuery" method of Statement object is implemented in the source code. Above regex will find all possible matches in the source code and display on the tool's output. As shown in figure 1, we are supplying source code base along with rules file. At this point we just have one rule for SQL injection. We are scanning all Java files residing in the target folder.

[2] WebGoat - http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project



Target Application Code: D:\webgoat-source

Rule file for scan: D:\csharp\AppCodeS

File extension (Regex): java

Enter Variable/Function:

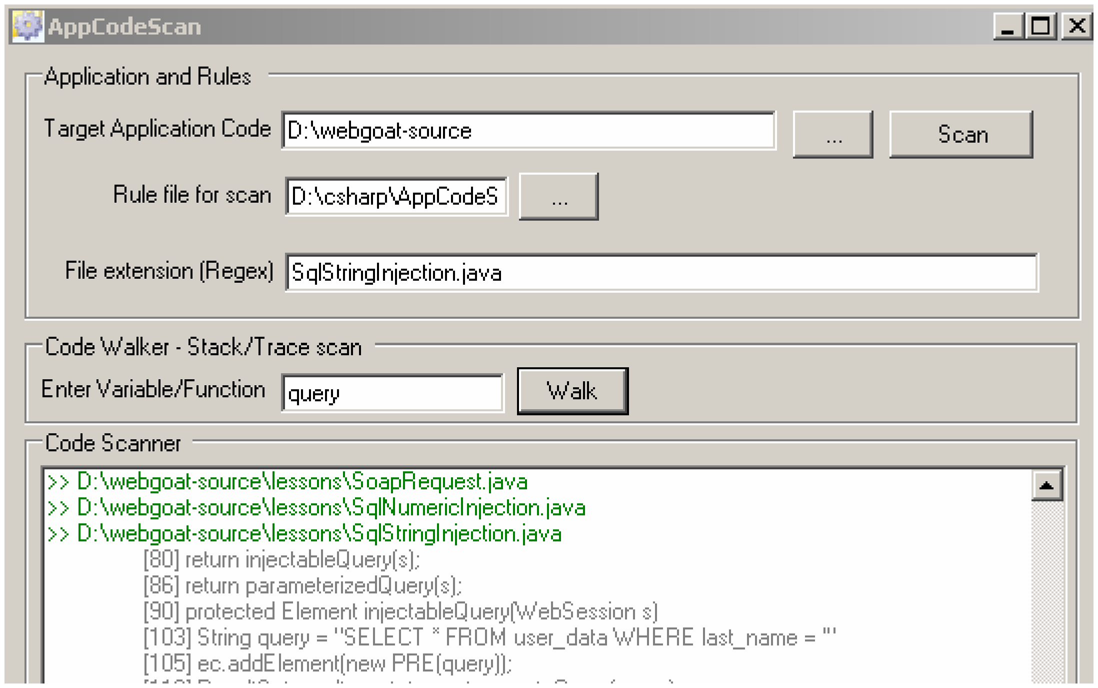
```
            [208] ResultSet results = statement.executeQuery();
            [296] Statement statement = connection.createStatement(
            [299] ResultSet results = statement.executeQuery(query);
>> D:\webgoat-source\lessons\SqlStringInjection.java
#SQL injection
            [109] Statement statement = connection.createStatement(
            [112] ResultSet results = statement.executeQuery(query);
            [194] ResultSet results = statement.executeQuery();
>> D:\webgoat-source\lessons\StoredXss.java
#SQL injection
```

***Figure 1 – Scanning for SQL injection points of high value calls***

At this point we have several files where this target call is fetched and we can start analyzing them in detail. Let’s take SqlStringInjection.java as our first target and try to identify its state with respect to vulnerability. Following line is interesting for us.

```
        [112] ResultSet results = statement.executeQuery(query);
```

Now we want to track “query” variable. We can narrow our scope to that particular file and variable by using File extension input and Code Walker functionality as shown in figure 2.



Target Application Code: D:\webgoat-source
Rule file for scan: D:\csharp\AppCodeS
File extension (Regex): SqlStringInjection.java
Enter Variable/Function: query

```
>> D:\webgoat-source\lessons\SoapRequest.java
>> D:\webgoat-source\lessons\SqlNumericInjection.java
>> D:\webgoat-source\lessons\SqlStringInjection.java
        [80] return injectableQuery(s);
        [86] return parameterizedQuery(s);
        [90] protected Element injectableQuery(WebSession s)
        [103] String query = "SELECT * FROM user_data WHERE last_name = '"
        [105] ec.addElement(new PRE(query));
```

*Figure 2 – Code Walking with "query"*

In this case we have SELECT query which is get executed with the "executeQuery" method. For us important line is 103 as below

```
[103] String query = "SELECT * FROM user_data WHERE last_name = '"
```

Line 103 is unfinished so we need to fetch line 104 and which looks like below.

```
[104] + accountName + "'";
```

So entire query would look like below in this case:

```
"SELECT * FROM user_data WHERE last_name = '"+ accountName + "'";
```

Now we need to track "accountName" to identify its origin by using Code Walker. The result is as below.

```
>> D:\webgoat-source\lessons\SqlStringInjection.java
        [63] private String accountName;
        [104] + accountName + "'";
        [193] statement.setString(1, accountName);
        [234] accountName = s.getParser().getRawParameter(ACCT_NAME, "Your Name");
        [235] Input input = new Input(Input.TEXT, ACCT_NAME, accountName.toString());
        [269] + "\"SELECT * FROM user_data WHERE last_name = \" + accountName ");
```

Here line 234 is of our interest,

```
[234] accountName = s.getParser().getRawParameter(ACCT_NAME, "Your Name");
```

Now we want to track ACCT_NAME and see from where it is coming, here is a result for it.

```
>> D:\webgoat-source\lessons\SqlStringInjection.java
        [57] private final static String ACCT_NAME = "account_name";
        [167] if (s.getParser().getRawParameter(ACCT_NAME, "YOUR_NAME").equals(
        [234] accountName = s.getParser().getRawParameter(ACCT_NAME, "Your Name");
```

Here its value is static as "account_name". Its trail ends here. Now let's focus on following:

```
s.getParser().getRawParameter
```

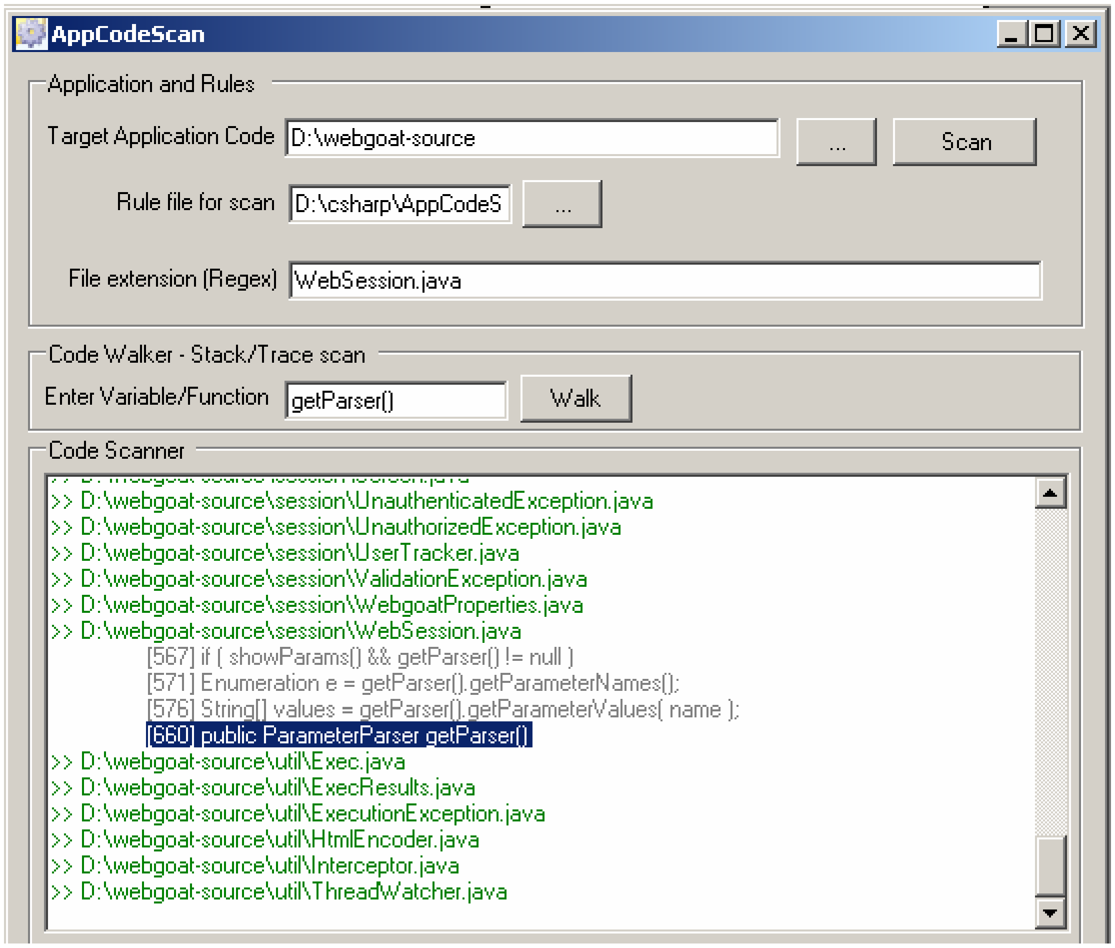
We need to find what is "s" here and we can fetch that information by doing walk for "\bs\b" or just "s" variable. The result is as follows.

```
>> D:\webgoat-source\lessons\SqlStringInjection.java
        [69] * @param  s  Description of the Parameter
        [72] protected Element createContent(WebSession s)
        [74] return super.createStagedContent(s);
        [78] protected Element doStage1(WebSession s) throws Exception
        [80] return injectableQuery(s);
```

It is instance of "WebSession" object. Hence, now we need to dig into this object and its "getParser" method.

```
>> D:\webgoat-source\lessons\SqlStringInjection.java
        [20] import org.owasp.webgoat.session.WebSession;
        [72] protected Element createContent(WebSession s)
```

WebSession is part of "session" so now we move to that file and look for "getParser" as shown in figure 3.



Target Application Code: D:\webgoat-source

Rule file for scan: D:\csharp\AppCodeS

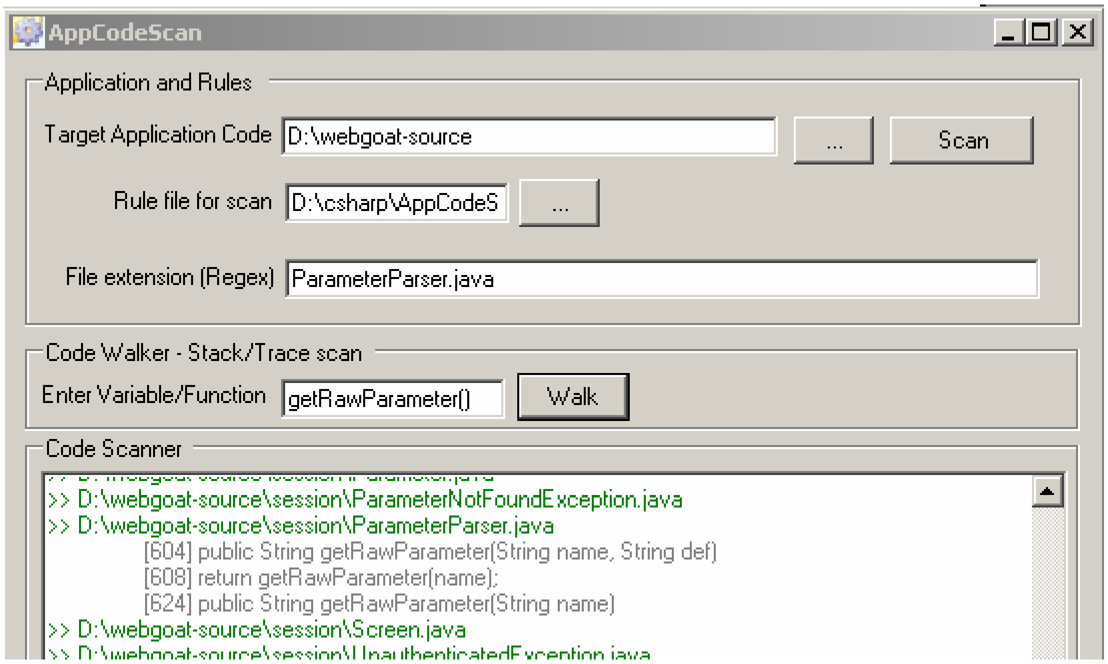
File extension (Regex): WebSession.java

Enter Variable/Function: getParser()

```
>> D:\webgoat-source\session\UnauthenticatedException.java
>> D:\webgoat-source\session\UnauthorizedException.java
>> D:\webgoat-source\session\UserTracker.java
>> D:\webgoat-source\session\ValidationException.java
>> D:\webgoat-source\session\WebgoatProperties.java
>> D:\webgoat-source\session\WebSession.java
          [567] if ( showParams() && getParser() != null )
          [571] Enumeration e = getParser().getParameterNames();
          [576] String[] values = getParser().getParameterValues( name );
          [660] public ParameterParser getParser()
>> D:\webgoat-source\util\Exec.java
>> D:\webgoat-source\util\ExecResults.java
>> D:\webgoat-source\util\ExecutionException.java
>> D:\webgoat-source\util\HtmlEncoder.java
>> D:\webgoat-source\util\Interceptor.java
>> D:\webgoat-source\util\ThreadWatcher.java
```

***Figure 3 – Looking for getParser() in WebSession.java***

“getParser()” is returning ParameterParser and we need to dig into getRawParameter() method of this class. We now search for it as shown in figure 4.



Target Application Code: D:\webgoat-source
Rule file for scan: D:\csharp\AppCodeS
File extension (Regex): ParameterParser.java
Enter Variable/Function: getRawParameter()

```
>> D:\webgoat-source\session\ParameterNotFoundException.java
>> D:\webgoat-source\session\ParameterParser.java
            [604] public String getRawParameter(String name, String def)
            [608] return getRawParameter(name);
            [624] public String getRawParameter(String name)
>> D:\webgoat-source\session\Screen.java
```

***Figure 4 Looking for getRawParameter method***

```
[604] public String getRawParameter(String name, String def)
[608] return getRawParameter(name);
[624] public String getRawParameter(String name)
```

Here in line 604 it has method with two arguments and in 608 it is calling same method name with one argument. This call is represented by line 624. It is clear now we need to track “name” and we can walk for that variable in the code, we get following result:

```
[624] public String getRawParameter(String name)
[627] String[] values = request.getParameterValues(name);
[631] throw new ParameterNotFoundException(name + " not found");
```

We are interested in value after line 624 here and we got the usage of name in line 627 which going to “request.getParameterValues” call. Now we need to see type of “request”. We can dig for request and get following results.

```
>> D:\webgoat-source\session\ParameterParser.java
            [8] import javax.servlet.ServletRequest;
            [48] private ServletRequest request;
```

At this point trail ends into native Java class and that is ServletRequest. Hence, value comes in the form of HTTP request as part of “account_name” and consumed in SELECT query without any sort of validations. We have confirmed candidate for SQL injection over here. We have touched minimum files and lines to dissect the actual call. This way it is possible to analyze source code very effectively with minimum effort. If we hit upon any validations then we can evaluate it as well and check its strength. This way it is possible to negate any false positives as well.

## *Scanning for other vulnerabilities*

In above section we covered SQL injection vulnerability in the source code. One can cook up many different rules as shown in Exhibit 1. They are simple regex patterns with different signatures. By using these rules we can identify entry points and some high value targets for evaluation. For example:

Identifying entry points to the application using servlet request calls.

```
# Entry points to Applications
.*HttpServletRequest.*
```

Tracing the responses by following calls can lead to XSS

```
# HTTP servlet response
.*HttpServletResponse.*
.*\.getWriter.*
.*\.println.*
.*Redirect.*
```

Remote command execution tracing

```
#Command injection points
.*\.getRuntime\(.*
.*Runtime.exec\(.*
```

File disclosure or traversal can be identify by following

```
# File access and path injections
.*File|FileInputStream|FileOutputStream|InputStreamReader|OutputStreamWriter|R
andomAccessFile.*
```

LDAP calls and injection

```
#LDAP injection points
.*LDAPConnection|LDAPSearchResults|InitialDirContext.*
```

This is a small set of rules but one can build more on top of this. Source code analysis is customized effort and one needs to add rules on the basis of specific code base.

## *Conclusion*

In this paper we discussed method of source code analysis and tracing for different set of vulnerabilities. The rules described in this paper are for Java source code and it is possible to build rules for several other languages as well. The simple tools like AppCodeScan or any other script which look for right patterns can help in source code analysis process and it can help in vulnerability detection process. Lately author used this methodology to dissect large code base for analysis and able to identify some of the vulnerabilities which were missed during the blackbox testing on the same application.

**Exhibit 1**

```
# Capturing public class
.*public.*class.*
# Capturing public methods and variables
.*public.*\(.*
# Entry points to Applications
.*HttpServletRequest.*
# Fetching get calls
.*\.getParameter*\(.*
.*getQuesyString|getCookies|getHeaders.*
# HTTP servlet response
.*HttpServletResponse.*
.*\.getWriter.*
.*\.println.*
.*Redirect.*
#Command injection points
.*\.getRuntime\(.*
.*Runtime.exec\(.*
# File access and path injections
.*File|FileInputStream|FileOutputStream|InputStreamReader|OutputStreamWriter|R
andomAccessFile.*
#LDAP injection points
.*LDAPConnection|LDAPSearchResults|InitialDirContext.*
#SQL injection
.*\.createStatement|executeQuery.*
.*select|update|delete|insert.*
#Good SQL usage
.*\.prepareStatement.*
#Buffer overflow from loading native libs
.*\.loadLibrary|\.load.*
#try/catch/leaks calls
.*try|catch|Finally.*
.*\.printStackTrace.*
# Session usage
.*HttpSession.*
#Response splitting
.*addCookie|addDateHeader|addHeader.*
.*writeHeader.*
.*setDateHeader|setHeader.*
# Log/Console injections
.*\.log.*
.*println.*
# Backdoors and Socket access
.*ServerSocket|Socket.*
# Crypto usage
.*Random.*
# Password usage/access
.*Properties|getProperty.*
.*password.*
# Driver manager usage
.*DriverManager.*
#Path to DoS
.*\.exit.*
```