Findings Security Errors in Java Applications Using Lightweight Static Analysis

V. Benjamin Livshits

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

livshits@cs.stanford.edu

Application security is becoming increasingly important in Java. In this paper, we focus on security issues that frequently occur in enterprise Java components. We describe two commonly violated security patterns and show how such violations can be prevented with static analysis of the application source. We describe our techniques and experimentally evaluate them on a set of 10 large open-source Java applications totalling over 130,000 lines of code. Our current approach allows us to successfully find 22 real security errors. Finally, we outline limitations of the current approach as well future work that will allow us to increase our analysis coverage and detect more errors.

Keywords: *Database security, Electronic commerce security, Enterprise security, Middleware and distributed systems security, Software safety and program correctness.*

1. OVERVIEW

With the increased adoption of Java and J2EE technologies for critical enterprise applications, their security becomes increasingly important. Many Java and J2EE application get deployed on Web servers as *servlets* and as such are easily exposed to hackers. Since many e-commerce sites depend on servlets for continuous operation, it is important to come up with automatic approaches to improve Web application security.

Various dynamic approaches exist that allow the execution of servlets to be monitored to disallow malicious activity, such as returning sensitive data back to the user. However, analysis of the application code is in many ways advantageous to dynamic monitoring.

Static analysis allows potential problems in the programs to be found before they have a chance to manifest themselves in an execution without inducing a runtime overhead. Sound static analysis generally also allows providing *security guarantees*. That is, if a program is deemed safe by a static analyzer, than it must indeed be safe for all possible inputs. In this paper we demonstrate how a lightweight syntactic source-level analysis can be used to expose a number of real application errors.

2. SECURITY ISSUES

We address the following two common security issues in this paper: "bad session stores" and SQL injections. Both occur in the context of servlets, Java components deployed within application servers such as Tomcat.

2.1 Bad session stores

The "bad session store" problem arises when objects stored in an attribute of a `javax.servlet.http.HttpSession` are not subclasses of `java.io.Serializable`. This can cause correctness issues because `HttpSession` objects may be written out to disk, thereby requiring all objects stored as attributes to be serializable, which is indicated by implementing interface `java.io.Serializable`. Failure to do so may cause exceptions or data corruption. The former may be exploited by a malicious user to mount a denial-of-service attack; the latter may cause intermittent problems that are hard to isolate because `HttpSession` objects are written out only under high load, which does not happen in the test environment.

2.2 SQL injections

SQL injections arise from allowing user-controlled strings to be used as part of SQL statements passed to a database. Allowing the user to do so may allow a malicious user to get access to unauthorized data or even delete information from the underlying database.

This problem falls into the category of so-called "taint" problems; a taint problem involves tracking the flow of data between values in the set of sources and the set of sinks. In the case of SQL injection using servlets, methods that read user-provided data such as

`javax.servlet.HttpRequest.getParameter(String name)`

are the sources and methods that pass SQL strings to the database for execution such as

`java.sql.Statement.executeQuery(String query)`

are the sinks. A static violation a sequence of values of the form $v_0, v_1, \dots v_n$, such that v_0 is a source and v_n is a sink.

3. ANALYSIS

We have implemented our static analyses as Eclipse plugins. Eclipse, an open-source Java IDE, allows easy access to ASTs for Java programs while allowing us to show to the user precisely where errors occur. We have evaluated our tools for both bad sessions stores and SQL injections on 10 open-source Java blogger and bulletin board applications that are widely deployed on the Web. A summary of information about the benchmarks is given in Figure 1.

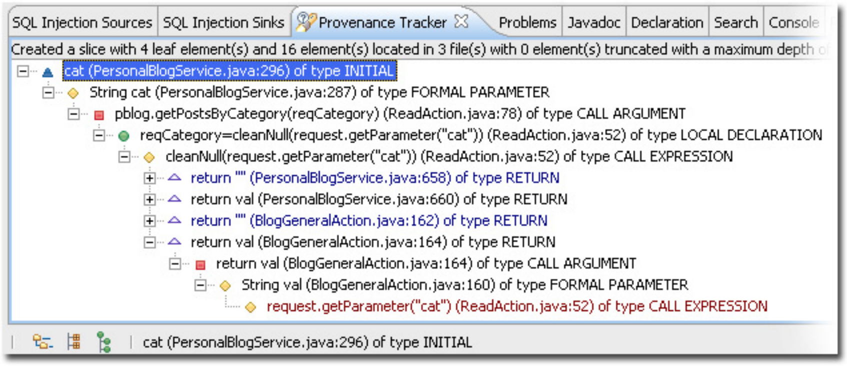
3.1 Bad Sessions Stores

To detect bad session stores, our tool looks at the type of parameters of `HttpSession.setAttribute` calls that are used to store objects withing a session. Checking the type of a parameter of a call to `setAttribute` and checking for subclassing can be done easily using Eclipse compiler APIs.

Figure 1 shows the total number of calls to `setAttribute` in column 4 and the number of calls with non-serializable types in column 5. We have manually examined each of the reported warnings to see if it is truly an error or a false positives; the numbers of real errors and false positives are summarized in columns 6 and 7, respectively. As can be seen from the Table, we find 14 real bad store errors with an average false positive rate of about 37%. Most false positives are due to the fact that runtime types are different from declared types; for instance, an object declared as a `Map` is

Benchmark	LOC	Classes	Session stores all	bad	errors	false pos.	SQL injections sources	all sinks	unsafe sinks	errors
mapleblog	2,156	36	5	5	3	2	8	16	16	1
personalblog	2,317	38	2	0	0	0	29	35	27	3
blueblog	4,142	38	0	0	0	0	6	1	1	0
blogwelder	4,901	33	3	3	3	0	115	24	24	0
javablog	5,184	79	10	0	0	0	12	42	38	0
snipsnap	9,671	1,331	28	12	7	5	195	33	33	1
blojsom	14,382	30	0	0	0	0	12	1	1	0
jboard	17,368	138	1	0	0	0	3	18	17	3
pebble	30,319	169	2	1	1	0	109	1	1	0
roller	47,044	267	24	1	0	1	81	45	30	0
Total	137,484	2,159	75	22	14	8	570	216	188	8

Figure 1: Summary of benchmark data and experimental results.



Created a slice with 4 leaf element(s) and 16 element(s) located in 3 file(s) with 0 element(s) truncated with a maximum depth

```
cat (PersonalBlogService.java:296) of type INITIAL
  String cat (PersonalBlogService.java:287) of type FORMAL PARAMETER
    pblog.getPostsByCategory(reqCategory) (ReadAction.java:78) of type CALL ARGUMENT
      reqCategory=cleanNull(request.getParameter("cat")) (ReadAction.java:52) of type LOCAL DECLARATION
        cleanNull(request.getParameter("cat")) (ReadAction.java:52) of type CALL EXPRESSION
          return "" (PersonalBlogService.java:658) of type RETURN
          return val (PersonalBlogService.java:660) of type RETURN
          return "" (BlogGeneralAction.java:162) of type RETURN
          return val (BlogGeneralAction.java:164) of type RETURN
            return val (BlogGeneralAction.java:164) of type CALL ARGUMENT
              String val (BlogGeneralAction.java:160) of type FORMAL PARAMETER
                request.getParameter("cat") (ReadAction.java:52) of type CALL EXPRESSION
```

cat (PersonalBlogService.java:296) of type INITIAL

Figure 2: Tracking a value in Eclipse GUI using the provenance tracker view. The topmost line is the sink being explored. Blue lines indicate string constants and red lines indicate dangerous sources.

in fact a `HashMap`. While type `HashMap` is serializable, `Map` is not, leading to a false positive.

3.2 SQL Injections

We address the problem of SQL injection checking by providing the following three separate custom views in Eclipse:

1. The *SQL injection source view* allows the user to easily navigate between the sources of user data.
2. The *SQL injection sink view* allows the user to easily navigate between the calls that pass strings to be executed on the database.
3. Finally, the *Provenance tracker view* allows the user to find origins of a particular value in the source by tracking the flow of data backwards.

Using a combination of these views, we have managed to find a number of real SQL injection errors in out benchmarks. Table 1 summarizes the number of SQL injection sources and sinks in columns 8 and 9, respectively. Column 10 shows the number of unsafe sinks, i.e. calls that take non-constant string parameters. Finally, the number of real errors we detected is shown in column 11.

A typical error detection session would start by choosing an injection sink and then using the provenance tracker view to see if data flow ever reaches an injection source. An example of an error we detected in `personalblog` is shown in Figure 2. In that example, we start with variable `cat`, which is part of the argument of a sink method call

```
session.find("..." + cat + "...")
```

and use our provenance tracker view to propagate data back to the source, which happens to be the return result of `request.getParameter("cat")`.

The provenance tracker view is based on Eclipse's information about variable declarations. It is also important to point out that this approach requires manual effort and is generally incomplete, because for arguments of functions with many callers, the number of possible origins for values we are tracking may quickly become prohibitively large, thus calling for a more automatic technique.

4. CONCLUSIONS AND FUTURE WORK

In this section we summarize our preliminary findings and formulate future plans. First, we believe that our tools, while not fully automatic, are still quite practical to be used by the average programmer. With the push of a button in Eclipse IDE, the programmer gets reports about potential security violations in her code. Computing and displaying bad session stores, SQL injection sources, and SQL injection sinks for all 10 benchmarks takes 27, 120, and 73 seconds, respectively on an Athlon 2500+ machine. Using the provenance tracker view to find origins of values is also fast in most cases.

The results we obtain for the bad session store problem are encouraging: we find a total of 14 errors with 8 false positives. The number of false positives can be reduced further by using a pointer analysis. Pointer analysis requires a stronger semantic understanding of the program and may generally have higher runtimes, however. In fact, we have experimented with a cloning-based context-sensitive pointer analysis to successfully eliminate most of false positives.

While we find a total of 8 errors in 10 applications, our overall experience with SQL injections is somewhat less encouraging. Looking for injections is a difficult manual process that requires automation. Moreover, while generally precise in practice, the results are potentially both *unsound* and *incomplete*. That is, while tracking where a particular value originates, some of the results may not be correct and some may be missing.

We are currently working on a precise static analysis for Java that would allow us to track the flow of data in the program between the sources and sinks precisely and efficiently. Such a tool will allow us to either find more errors and prove that they cannot exist.