Java Security Vulnerabilities Detection With Static Analysis

Tulasi Veera Prasad N
Student (M.Tech) ,CSE,
Gokul Institue of Technology and Science,
Visakhapatnam, India.

A.Achutharao
Asst. Prof, CSE,
Gokul Institue of Technology and Science,
Visakhapatnam, India.

Abstract:

Security in software plays an important role in todays society as computer networking is getting more and more important. Security measures are taken to protect private information, but bad programming practices can still cause security vulnerabilities in software systems. Source code analysis tools can be used to detect such security vulnerabilities automatically. The use of these tools helps to improve the quality and security of software systems and could prevent future problems.

The class of security vulnerabilities called input validation vulnerabilities can be detected using static taint analysis. The design and implementation of such a tool are the subject of this paper. This tool detects input validation vulnerabilities in source code written in the Java programming language. This paper also describes in detail how to deal with complexities related to the object oriented nature of Java.

The tool first derives a graph structured model from the source code. This graph structured model captures data dependency relations between important program elements. This graph model is then analyzed using taint analysis to detect potential input validation vulnerabilities.

Keywords:

SQL Injection, Software Analysis Toolkit (SAT), Cross site scripting (XSS).

I. INTRODUCTION:

In today's world where computer networking plays an important role in everyday life, computer criminals cause havoc in critical or important network environments. Com- mon criminal activities include: tapping network traffic, tampering databases, modify- ing websites, disabling services and information theft [26]. These activities can cause bad publicity, data-loss and privacy problems, which could result in significant (financial) damages to companies.

Systems that are secure enough to resist such attacks are therefore essential. Security breaches are often the result of bad programming practices during development.Some of these security vulnerabilities are easily detected and fixed when the program crashes or unexpected output is given. Other security vulnerabilities will never be noticed during normal use. Automatic source code analyzers can help detect- ing these security vulnerabilities before deployment of a software system.

1.1. Style Conventions:

This paper follows a style convention for clarity. The following style conventions are used:

• Relevant large program parts are displayed as code fragments, which are listed on its own index page. The code of a simple class is given in code fragment 1.1. The keywords that belong to the programming language are bold. The lines are numbered for easy referencing in from the text.

1.2. Development Environment:

To get an impression of how the security tool is developed at the office of the Software Improvement Group (SIG), the development environment is described. To confirm to the existing software standard used by SIG, the development environment influences the way the tool is developed. The workstation is an Apple iMac running Mac OS X as Operating System, which is also connected to the Internet.

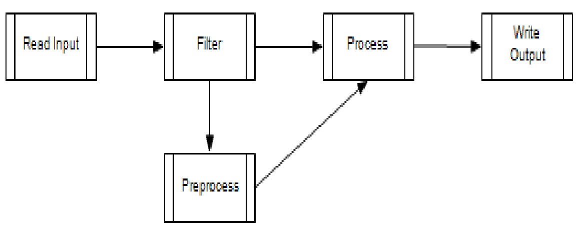


Figure 1.1: Flow diagram

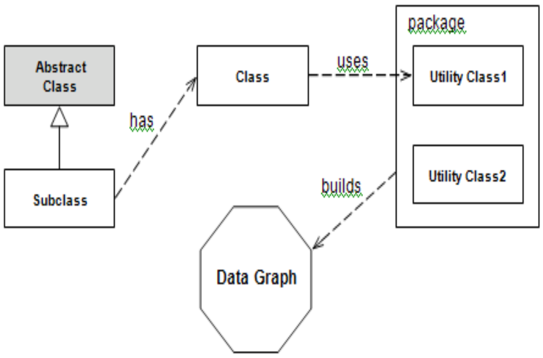


Figure 1.2: Modules diagram.

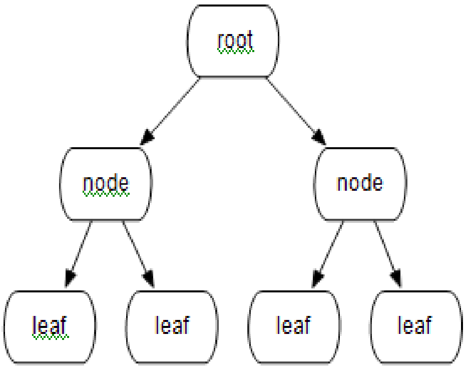


Figure 1.3: Graph diagram.

II. JAVA SECURITY VULNERABILITIES:

2.1 SQL Injection:

A popular application of the Java programming language is the use of Java servlets to handle web server requests. The underlying pattern in the architecture of a potential vulnerable system is depicted below in figure 3.1 as an example. The architecture consists of three modules or components, which are interacting with each other.

The web server component is responsible for handling requests initiated by the user. When a HTTP request is received from the user, the web server delegates the request to the Java servlet. The Java servlet may interact with a SQL database by querying, which depends on the user input. In short, a system may be vulnerable to SQL injection attacks, when SQL input by an application depends on the user input.

SQL injection [43, 31, 32, 13, 19, 38] occurs when the semantics of a SQL query that is embedded in the source code is changed due to specially crafted user input. The bad query can do things not allowed or intended by the application. The syntax of the embedded may be correct, but the semantics is changed.

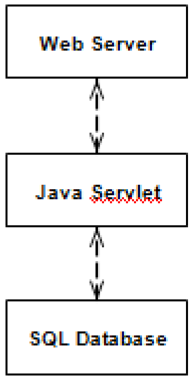


Figure 2.1: Web server Java SQL architecture.

2.2 Cross Site Scripting:

In cross site scripting (XSS) [43, 31, 32, 19, 38] a vulnerability exists in a web ap- plication that makes it possible to trick users of the website to execute arbitrary code using the website as a relay. The code appears to be originated from the website that may be a website that is trusted by the users. The trust of the user in a website with a XSS vulnerability and the website itself are abused to trick the computer/browser of the user to execute arbitrary code, which steals private information from the user. No entry is gained in the website itself.

The attacker who wants to abuse an XSS vulnerable website, first crafts a special hyperlink that has hidden code embedded. The code is meant to steal information from the user. The second problem is to get the user to click on this link. One way to do it is to post it on a forum that is known to be regularly visited by users of the XSS vulner- able website, another way is to email it directly to the users. If an user has clicked on the link, the code in the link is relayed and echoed back to the user by the website. The browser of the user starts executing the code, which can do things like stealing cookie information that contains login information. Stealing cookie information can be done by letting the code dump this information at a specially installed drop site.

2.3 Command Injection:

Command Injection [43, 31, 32, 38] tricks the application into executing another pro- gram. This can be used by an outsider to gain entry into a web server or to execute something with the same privileges as the application. It can also be used by an user of a stripped down computer. A stripped down computer is intentionally restricted in accessibility, so the only use of the computer is through a particular program or interface.

2.4 Input Validation Vulnerabilities Detection:

SQL injection, cross site scripting, command injection and path traversal vulnerabil- ities have fundamental properties in common. The commonality is that user input is trusted and not validated before it is used in a subsystem. A subsystem is an in- dependent system that is used by the application. This subsystem is accessed by its interface, which is commonly a collection of methods. The subsystems of the vulner- able programs discussed earlier, are the SQL database for applications vulnerable to SQL injections, the browser for cross site scripting vulnerable websites and the un- derlying operating system for the last two vulnerabilities. These vulnerabilities would not exist if user input is properly checked and sanitized before being used, eliminating dangerous input. The vulnerability is exploited when the attacker tricks the subsystem into doing something not intended by the application.

III. SOFTWARE ANALYSIS TOOLKIT (SAT):

This section describes the Software Analysis Toolkit (SAT) framework used by the SIG to perform Software Risk Assessments (SRAs). The SAT is a collection of software analysis programs used to analyze all kinds of software systems. Each program does so by looking at the source code of the system, which is better known as static analysis. Implementing a new analysis for the SAT requires the use of standard classes and inter- faces provided by the SAT software framework. SAT makes it possible to perform the analysis in a standardized way and it prevents source code duplication. The standard classes that are described below form the basis of SAT. To understand the existence of these classes and why they are standardized, a typical source code analysis will be described.

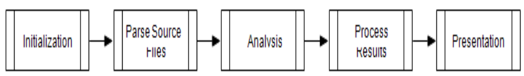
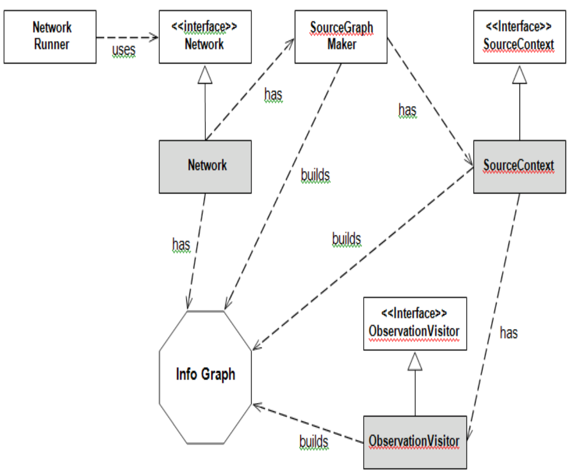


Figure 3.1: Simple analysis.



IV. JAVA SECURITY ANALYSIS:

4.1 The Security Analysis Architecture:

The architecture of the Java Security Analysis is illustrated in figure 8.1. Interface classes and several other supertypes that belong to the SAT framework are omitted to avoid cluttering in the figure. The architecture includes the classes discussed in earlier chapters. The heart of the architecture is the Security Fact Graph, which role is twofold. First, the graph is constructed while the Java source files are analyzed. Second, the graph is analyzed to dectect input validation vulnerabilities.

The Java classes in the Java Security Analysis are divided into Java packages. Related classes belong to the same package, often packages contain classes that inherit from the same superclass. There are 5 packages in total.

• typeinformation: This package contains classes that are used to resolve pro- gram entities like (super) classes, variables and methods.

• obsvisitors: This package contains subclasses of the ObservationVisitors class, which are used to traverse source files. An AST is constructed for every source file, which is then analyzed by using the TreeWalker class.

• astvisitors: This package contains subclasses of the AbstractActionVisitor class, which are used by the TreeWalker class to traverse the ASTs.

• sfgvisitors: This package contains subclasses of the AbstractLinkVisitor class, which are used to traverse the Security Fact Graph.

• javasecurityanalysis: This package contains all other classes.

4.2 Experimental Results:

A guestbook web application is created to show the workings of the analysis on a real Java web application. The source code of the guestbook can be found in ap- pendix D. The guestbook has two basic functionalities.

One is adding a new guest- book entry to the database and the other is retrieving all the entries from the database for display. A MySQL [49] database is used to store guestbook entries. The guest- book contains several SQL injection vulnerabilities.

The untrusted method is speci- fied as javax.servlet. ServletRequest.getParameter(), which returns the user supplied parameter. In order to make the security vulnerability complete, the criti- cal method java. sql.Statement.executeQuery() is used, which executes a SQL query..

The stripped Security Fact Graph that corresponds to the guestbook can be found in figure 8.2. All the nodes of the stripped Security Fact Graph are tainted. Redundant nodes are removed, without influencing the outcome of the analysis. The names of the edges are not displayed. Like expected, several dangerous paths from an untrusted method to a critical method are found. In total, there are three paths found. The paths originate from the getParameter() method call, which is used to retrieve the user input. The paths end with the executeQuery() method call on the Statement object. The specific tainted paths can be found in appendix E, which contains the literal output file content.

The analysis does not recognize methods or algorithms used to validate input, which means that dangerous tainted paths are also found if the input is validated cor- rectly. This is the reason why tainted paths found by the Java Security Analysis have to be verified manually.

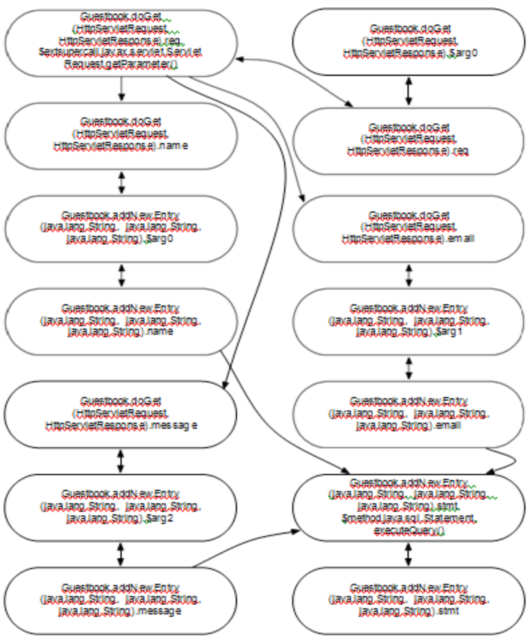


Figure 4.1: Security Fact Graph of Guestbook

CONCLUSION:

The Java Security Analysis allows SIG to detect security vulnerabilities in software of clients. The list of services to clients can be extended by a security check or assessment service, which makes it interesting for clients who want their Java web application to be checked for security vulnerabilities. The other contributions are actually side products, which are essential components of the analysis. The first one is the extended type inference framework, which now supports object types. This is in contrast with the former type inference framework, which can only deal with integer types. The second is the BCEL wrapper that is used to resolve library types. This wrapper is proven to be useful and it can be used to increase accuracy of the existing analyses used by SIG.

The analysis is strongly data flow oriented, which means that it can easily be mod- ified for other purposes than detecting security vulnerabilities, for instance, to identify all locations where a certain value is used. This way, dependencies of classes or mod- ules on that value can be identified to separate software architecture modules. This information can be used to improve program understanding, which is in line with the SWERL research area.

This project shows a way how type inference can be used to capture data depen- dency relationships between variables. These relationships are then used to perform taint analysis in order to detect input validation vulnerabilities. It also shows a way to deal with objects, which can be defined in libraries. In addition to normal source code analysis, the use of byte code analysis is described to improve accuracy.

REFERENCES:

[1] The Byte Code Engineering Library (BCEL).http://jakarta.apache.org/bcel,2006.

[2] Oracle TopLink Developer's Guide (v10.1.3.1.0). Oracle, 2006.

[3] Website Apache Ibatis. http://ibatis.apache.org, 2006.

[4] Website Checkstyle Plugin for Eclipse. http://eclipse-cs.sourceforge.net, 2006.

[5] Website Graphiz. http://www.graphviz.org, 2006.

[6] Website Jakarta Apache Project. http://jakarta.apache.org, 2006.

[7] Website JavaServer Faces. http://java.sun.com/javaee/javaserverfaces, 2006.

[8]Website Software Evolution Research Lab (SWERL). http://swerl.tudelft.nl, 2006.

[9] Website Software Improvement Group (SIG). http://www.sig.nl, 2006.

[10] Website Spring Framework. http://www.spring-framework.org, 2006.

[11] Website Clover. http://www.cenqua.com/clover, 2007.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, tech- niques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[13] Chris Anley. Advanced SQL Injection in SQL Server Applications. NGSSoftware Insight Security Research (NISR), 2002.

[14] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi- threaded java programs. aswec, 00:0068, 2001.

[15] Thomas Ball. The concept of dynamic analysis. In ESEC / SIGSOFT FSE, pages 216–234, 1999.

[16] Scott Stender Brad Arkin and Gary McGraw. Software penetration testing. IEEE Security and Privacy, 03(1):84–87, 2005.

[17] Chuck Cavaness. Programming Jakarta Struts. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[18] Brian Chess and Gary McGraw. Static analysis for security. IEEE Security & Privacy, 2(6):76–79, 2004.

[19] Paul Dullaart. Secure coding. Informatie, 16:10–12, 2005.

[20] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. Lecture Notes in Computer Sci- ence, 707:406–431, 1993.

[21] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In SIGPLAN Conference on Programming Language Design and Im- plementation, pages 192–203, 1999.

[22] Michael Frumkin. Data Flow Pattern Analysis of Scientific Applications. Intel, 2005.

[23] Daniel Geer and John Harthorne. Penetration testing: A duet. In ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference, page 185, Washington, DC, USA, 2002. IEEE Computer Society.

[24] David Greenfieldboyce and Jeffrey S. Foster. Type Qualifiers for Java. University of Maryland, College Park, 2005.

[25] Vivek Haldar. Verifying Data Flow Optimizations for Just-In-Time Compilation. Sun Microsystems Laboratories, 2002.

[26] John Douglas Howard. An analysis of security incidents on the Internet 1989-1995. PhD thesis, Pittsburgh, PA, USA, 1998.

[27] Andrew Hurst. Analysis of Perl's Taint Mode. 2004.

[28] Will Iverson. Hibernate: A J2EE(TM) Developer's Guide. Addison-Wesley Professional, 2004.

[29] Guy Steele James Gosling, Bill Joy and Gilad Bracha. Java(TM) Language Spec- ification, The (3rd Edition) (Java Series). Addison-Wesley Professional, July 2005.

[30] Ralph E. Johnson. Components, frameworks, patterns. In ACM SIGSOFT Sym- posium on Software Reusability, pages 10–17, 1997.

[31] V. Benjamin Livshits. Finding Security Errors in Java Applications Using Lightweight Static Analysis. Computer Systems Laboratory, Stanford University, 2004.

[32] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java pro- grams with static analysis. In Proceedings of the 14th Usenix Security Symposium, pages 271–286, August 2005.

[33] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis, 2005.

[34] Panagiotis Louridas. Version control. IEEE Software, 23(1):104–107, 2006.

[35] Steven Myers Markus Jakobsson. Phishing and Countermeasures : Understand- ing the Increasing Problem of Electronic Identity Theft. Addison-Wesley Long- man Publishing Co., Inc., Boston, MA, USA, 2006.

[36] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, pages 365–383, 2005.

[37] Ravi Mendis. WebObjects Developer's Guide. Sams, Indianapolis, IN, USA,2002.

[38] David LeBlanc Michael Howard and John Viega. 19 Deadly Sins of SoftwareSecurity. McGraw-Hill/Osborne, 2005. ISBN 0-07-226085-8.

[39] Christopher Kruegel Nenad Jovanovic and Engin Kirda. Pixy: A Static Anal- ysis Tool for Detecting Web Application Vulnerabilities. Secure Systems Lab, Technical University of Vienna, 2005.

[40] James Newsome and Dawn Song. Dynamic taint analysis for automatic detec- tion, analysis and signature generation of exploits on commodity software. In NDSS 05: Proceedings of the 12th Annual Network and Distributed System Se- curity Symposium, San Diego, California, USA, 2005. Internet Society.

[41] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), volume 26, New York, NY, 1991. ACM Press.

[42] David Scott and Richard Sharp. Abstracting application-level web security. In WWW '02: Proceedings of the 11th international conference on World Wide Web, pages 396–407, New York, NY, USA, 2002. ACM Press.

[43] Mike Shema. HackNotes Web Security Portable Reference. McGrawHill/Os- borne, 2003. ISBN 0-07-222784-2.

[44] Jim D'Anjou Sherry Shavor, John Kellerman Pat McCarthy, and Scott Fair- brother. The Java Developer's Guide to Eclipse. Pearson Education, 2003.

[45] Jeffrey S. Foster Umesh Shankar, Kunal Talwar and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. University of California at Berkeley, 2001.

[46] Larry Wall. Programming Perl. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[47] Brad Calder Weihaw Chuang, Satish Narayanasamy and Ranjit Jhala. Bounds checking with taint-based analysis. In HiPEAC 2007, 2007. cations. In Technical Report SECLAB-05-04, 2005.

[48] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation, pages 131–144, 2004.

[49] Michael Widenius and Davis Axmark. Mysql Reference Manual. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[50] Wei Xu, Sandeep Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web appli [51] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In 15th USENIX Security Symposium, Vancouver, BC, Canada, August 2006.

[52] Christian Hang Chung-Hung Tsai Der-Tsai Lee Yao-Wen Huang, Fang Yu and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In WWW '04: Proceedings of the 13th international conference on World Wide Web, pages 40–52, New York, NY, USA, 2004. ACM Press.

[53] Benjamin Grgoire Yves Bertot and Xavier Leroy. A structured approach to prov- ing compiler optimizations based on dataflow analysis. In Types for Proofs and Programs, Workshop TYPES 2004, volume 3839 of Lecture Notes in Computer Science, pages 66–81. Springer-Verlag, 2006.