RC24554 (W0805-065) May 12, 2008
Computer Science

IBM Research Report

SWORD4J:
Security WORkbench Development Environment 4 Java

Ted Habeck, Larry Koved, Marco Pistoia
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

SWORD4J: Security WORkbench Development environment 4 Java

Ted Habeck
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, New York
10598
habeck@us.ibm.com

Larry Koved
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, New York
10598
koved@us.ibm.com

Marco Pistoia
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, New York
10598
pistoia@us.ibm.com

ABSTRACT

Creating secure software systems remains a challenge for most developers, even for those who are conscientious about following security best practices. Software development has evolved to incorporate complex software frameworks, middleware and components developed by multiple parties. We have seen the rise of tools for testing the security of applications, including so called "black box" testing and "white box" testing. Some of these include static analysis technologies, and run-time testing to verify specific security properties, as well as conformance to "best practices" The lack of integration of these security tools creates a significant burden on most developers, many of whom lack formal training in secure software development and deployment practices. They are often less motivated to secure their software than security professionals.

To address the challenges of creating secure Java applications we created a tool called SWORD4J that integrates a suite of security analysis tools into the Java Developement Tool in the Eclipse Integrated Developement Environment. We believe that SWORD4J is more usable than standalone security tools because it greatly simplifies many time consuming tasks required to develop secure software components. significantly reducing the time to perform security analysis tasks. In this paper we also argue that secure Open Services Gateway initiative (OSGi) component development has characteristics that are common to many software environments, including Web application developement.

General Terms

TBD

Keywords

security analysis, static analysis, integrated developement environment, development tools, ease of use, software development

Symposium on Usable Privacy and Security (SOUPS) 2008, July 23–25, 2008, Pittsburgh, PA USA
.

1. INTRODUCTION

Software developers have long been challenged to create reliable applications, from requirements, through testing and deployment [11]. Reimer, et al. [36] observed that large Web applications are difficult to reliably create, partly as a result of the use of large and growing number of application frameworks to be used in their development. Creation of secure applications and systems adds an additional set of responsibilities for developers that go above and beyond the basic functional requirements of these systems.

Many of the security issues facing software developers are well known, although each programming language and operating environment has its own unique security challenges. For example, C and C++ developers must address buffer overflow and string formatting vulnerabilities, among other security vulnerabilities. PHP and other scripting language developers are concerned about file inclusion vulnerabilities and command injections. Java developers must be vigilant about checking for a variety of forms of tainted data vulnerabilities. Even when the applications are correctly developed, the systems need to be properly configured when the applications are deployed. In addition, the run-time platforms and application code must perform appropriate authentication and authorization when providing access to protected resources. These vulnerabilities, among many others, have been well documented (e.g., [5, 13], and there are many sources of security "best practices"). For most software developers, finding, understanding, and following security best practices can be an overwhelming task. This is especially true when multiple tools must be acquired and used.

The main contribution of this paper is that a set of well integrated security analysis technologies into software development and testing environment can substantially reduce the difficulty and cost (time) to create secure software, thus making security analysis tools more usable. We argue that, because of the complexity of the tasks involved, using a set of security analysis tools that are not well integrated into the development environment will be insufficient to get adequate developer productivity on security analysis tasks.

While we will be discussing the development of secure Java applications based on the OSGi framework for component-based application development and deployment [6, 2] written in Java, most of the principles outlined in this paper apply to many other commonly used programming languages and run-time environments such as the Web.

The remainder of this paper is organized as follows. Section 2 will describe specific security challenges facing Java

component developers. We will motivate the intuition as to why many of these security tasks are currently very challenging for software developers. Section 3 describes our earlier work on security analysis tools. Section 4 describes how our security tooling, SWORD4J[1], addresses the challenges outlined in the previous section in a fashion that reduces the number of steps required to address the security related development tasks, as well as reduce the cognitive load placed on the software developer. Section 5 discusses how SWORD4J addresses security issues outlined in section 2. Section 6 relates how the development tasks for OSGi are common to secure Web application development. Section 7 reviews related work. We conclude with some remarks on future work.

2. SECURITY CHALLENGES

We now describe a set of representative tasks developers need to perform to secure Java applications or software components when developing for the OSGi framework [6]. The development scenario is as follows, broken down into subtasks, and labelled (in parenthesis) for subsequent reference in this paper.

Consider a programmer who has been given an assignment to develop a software library (or component) for use in embedded devices. After completing the development and unit testing of the software, the developer is subsequently asked to make their component reusable. Now this component needs to be made "secure" from potentially malicious attackers.

(A) Being somewhat new to Java and OSGi development, the programmer needs to learn Java and OSGi security. This knowledge can be acquired through books, searching the Web for "best practices", and asking colleagues to teach them the relevant information. There are many security Java topics: code-based access control and permissions, cryptography, bytecode verifier, type safety, class loading, component-based software (e.g., OSGi, Java EE), etc. Searches of the Web will turn up other security best practices that ought to be followed (e.g., [7]).

The challenge for the programmer is in identifying the subset of Java security and best practices that are specific for the target programming model and run-time (Java and OSGi). Much of the information available is more general, which can result in wasted time and frustration for the developer. For example, application developers will not need to know much about bytecode verifiers and cryptography.

(B) Since this project is built to be used with the OSGi framework, the programmer will need to fully understand Java SE permissions and its stack based access control algorithm [23, 31]. The programmer needs to be able to create Java authorization policies based on Java permissions, and understand how to assign authorization policies based on a `CodeSource`.[2] Best security pactices dictate that an authorization policy not be overly permissive, otherwise it would be a violation of the *Principle of Least Privilege* [38]. This activity will also require understanding how code signing works, where the signatures are stored, and the operational implications of modifying the JAR files after the code has been signed.[3]

The developer will also need to learn how to debug the Java authorization policy. This includes learning how to inspect the results of an uncaught `SecurityException`, which generates a run-time stack trace. The privileged operation is typically composed of three parts: (1) The `Permission` subclass that represents the privileged operation that failed and caused the `SecurityException`, (2) the target of the operation (the protected resource), and (3) the operation on the target[4]. The developer then needs to identify which stack frame (i.e., the method, its class and the `CodeSource` for that class) not authorized for a privileged operation. Sometimes, source code inspection is needed to fully appreciate the nature and extend of the privileged operation in order to properly define an appropriate authorization policy.

(C) The developer must figure out how to enable Java SE authorization in the run-time platform. This step is unique to each run-time environment, requiring further investigation on the part of the developer. For example, for standalone Java, the `SecurityManager` can be installed via a command-line option or done programmatically. For other environments, there many be command-line options that are different from those for a standalone Java runtime, enabled through configuration file entries, or performed programmatically. Also, there may be a specific `SecurityManager` subclass that is required for the platform. The OSGi implementation for the Eclipse platform has its own `SecurityManager` that is enabled by launching with a command-line option to use the OSGi `SecurityManager`.

(D) Once Java authorization is enabled in the Java runtime, test cases are needed to cover most paths through the application in order to determine the Java authorization policy. Test case generation and code coverage tools may be needed to ensure a sufficient set of test cases.

(E) The test cases are run until they all succeed without failing due to `SecurityException`s. As described above, when a `SecurityException` occurs, the developer inspects the stack trace to determine which codebase was missing an authorization. The source code then needs to be consulted to determine whether a Java authorization needs to be added to the security policy.

(F) Adding appropriate Java authorization policy entries can be challenging for the programmer, especially when the policy is stored and its format (syntax) varies by platform. Policy may be stored in a flat file (as in the reference implementation of the Java SDK) or other repository, such as the component JAR files (OSGi). While the semantics of the policies are the same in both cases, the syntax differs.

(G) The above testing and policy update process is repeated until there is an authorization policy that will enable the test cases to run without `SecurityException`s being thrown at run-time.

(H) Similar to updating the Java authorization policy, as described above, there is a need to make some of the code "privileged" [23, 31]. Privileged code is needed when a component is to perform an operation that requires authorization, but the component developer does not want the code calling the component to also require that authoriza-

[1]http://www.alphaworks.ibm.com/tech/sword4j

[2]The location, or origin, of the code, typically represented as a URL, and/or the digital signature(s) associated with the code.

[3]The code will run, but may not have authorizations that were specified in the authorization policy.

[4]E.g., `java.io.FilePermission "/tmp/log.txt","read,write"`

tion.[5] An example is a file-based logging routine used to audit calls to a privileged operation. The caller should not require authorization to access the file, although the caller should be allowed to call the logging routine, which in turn is able to write the audit records to the file.

(I) Because untrusted software components will call the developer's component, there is a need to check for tainted data being passed into it from the untrusted components. The developer must trace the tainted data by manually perform a control- and data-flow analysis, including figuring out program slices [45] to identify which data values need to be sanitized before being used by privileged operations. The developer must discriminate the various types of privileged operations (commands, file inclusions, SQL calls, etc.) and identify the target operation-specific sanitizing functions that need to be called on the tainted data before it is used. The rules for sanitizing data can be relatively complex, depending on target privileged operation. The developer will need to find best practices that describe how to appropriately sanitize the data before it is used. As was described above for authorizations, the code will need to be retested after code is made privileged and sanitizing functions are added. New test cases may be needed to verify that the sanitizing functions are working as expected.

(J) Refactoring of the code is required to add a call to `AccessController.doPrivileged()` once the developer has determined that adding privileged code is reasonable and safe. The privileged operation needs to be wrapped in a `PrivilegedAction` instance. Often this is done by creating an anonymous inner class that is a subclass of `PrivilegedAction`. The refactoring of the code can be tedious, especially if there are multiple values that need to be passed into the privileged operation.

(K) Because the component under development is to be deployed in a hostile operating environment, additional security coding rules must be followed. For example, in Java, two concurrently running applications in the same JVM are typically isolated by loading them via different Java `ClassLoader`s. This allows each application to execute in separate name spaces, thus providing code isolation. However, because of the way that the Java runtime is constructed, there remain multiple ways to share state. From a security perspective, shared state needs to either be immutable, or have appropriate access control to restrict access or changes to the state. When the components are executed in the same name space (same `ClassLoader`), the immutability rules also apply. Determining immutability of an object proceeds as follows.

First, accessability rules for class members (fields and methods) need to be checked.[6] Next, for each type (e.g., a class) used by the component, determine whether the fields are *value-immutable* or *state immutable*. Fields are value immutable if the field is declared to be `final`. However, if the field is inaccessible to other components, and its value can not be updated after it is initialized, then it may be value immutable.

[5]Privileged operations include, among other things, networking, file and database operations, all of which require elevated privileges.

[6]While the limited set of access modifiers, `public, private` and default, are seemingly simple, the accessability rules turn out not to be obvious. Details are outside the scope of this paper.

The programmer needs to determine whether a type (class) is state mutable (the state of an instance is mutable). Unlike C++, Java does not have a `const` declaration. So the developer must determine whether the state of an object referenced by a class or instance field can be changed, irrespective of whether the field is declared to be `final`. For example, `final StringBuffer s = new StringBuffer();` declares a value immutable field `s` since it is declared to be `final`, and this restriction is enforced by the Java runtime. However, the `StringBuffer` object referenced by `s` is mutable since instances of `StringBuffer` are mutable – they can be updated by several methods, including the `append()` method. In contrast, instances of the class `String` are immutable since there are no mutating operations on the state of `String` instances. The mutability analysis process repeats for each class and instance field in the developer's component, as well as for all classes the component is dependent upon.

Computing mutability is a complex and arduous process involving computing data flows and program slices. A more detailed description of mutability analysis for Java can be found in [34].

(L) The developer also needs to check their code for compliance with other security best practices. Examples include:

- Searching for all uses of `ClassLoader` instances to make sure that secure Java loading rules are not violated (e.g., [17]). All security sensitive methods in `ClassLoader`s must be inaccessible to untrustworthy code.
- Security-sensitive fields in all classes (passwords, crytographic material, fields with personal identifying information, etc.) must be inaccessible to untrusted code.
- Making sure that security sensitive fields cannot be serialized without being encrypted in a way that would allow untrustworthy code to decrypt the values.

(M) Security-sensitive information needs to be protected by calls to the Java authorization system, using Java 2 permissions that are approriate for the resources. Complete mediation [38] must be validated to be consistent on all paths to the protected accesses (e.g., [46]).

(N) Once the code has been checked for security security vulnerabilities, and follows best practices, it is time to distribute the code. For Java, this involves code signing [31]. As a result, a number of questions arise for the developer:

- Is code signing necessary?
- Must a certificate for code signing be purchased from a certificate authority?
- Can self-signed certificates be used? What are their limitations?
- Once a certificate is acquired (or generated), where are they stored? How are they managed?
- How do you go about signing code? Where are the signatures stored? When/how are they verified?

Once these issues are resolved, the developer can sign the code and prepare it for distribution.

(O) Most of the above steps need to be repeated as the component, and the code it depends upon, are updated, or if security best practices change.

The above steps are representative of the work involved in securing a software component. The work is time consuming, repetitive, and very detail oriented, with a large number of opportunities for error. We have been developing tools to handle these security tasks in a more reliable, and less time consuming fashion.

3. EARLIER EXPERIENCES

We have tried a number of different strategies and developed tools to address many of the issues outlined in the previous section. Each time we investigated a new Java security challenge, we created new tools and techniques to reduce the time required for the analysis as well as the possibility of making errors.

3.1 Early tooling attempts

Early analyses was performed on the Java Development Kit, starting with JDK 1.1.4, and performing multiple analyses on the run-time classes, through JDK 1.2.0. Initially these analyses were done get a better understanding of gaps in the Java isolation model when running concurrent applications (Web browsers) in a single Java run-time. Much of the analysis was performed through manual code inspection of the source code using a text editor. Later analyses used simple text-based reports that identified non-`final` fields. But most of the analysis work was still performed with a text editor and manually keeping notes on mutable and immutable classes and fields. Analysis of each subsequent JDK release would require about three weeks of work.

To analyze larger code bases, we needed to create tools to reduce the burden on the programmers. Our next round of Java security analysis automated mutability analysis [34]. This work was done in the context of high performance transaction processing [18]. The program model for this work required strict isolation, with the exception of sharing common code and immutable state. Updates to static fields would result in significant performance degradation, security faults (transaction isolation) and violate programming model restrictions (no shared state across transactions except through the database). Our analysis searched for cases where static fields were mutable – both value and state mutable. The tool generated HTML reports, but investigation of the issues still required separate navigating throught the source code.

Several products had a need to run with Java SE `CodeSource` authorization enabled. The typical pattern of development was to write the code, enable a `SecurityManager`, and then observe the stack traces from the `SecurityExceptions` to see which code or policies needed to be updated. For larger products, this approach was not practical due to the quantity of code. We created an access rights analysis tool [26] and a secure code placement tool [30] to make security policy and identify privileged code refactoring opportunities. Text and HTML reports were generated, but investigation of the issues still required separate navigating throught the source code. Code refactoring for privileged code placement was also performed manually.

The analysis results from our tools were conservative. In particular, the underlying static analysis was path and flow insensitive, so there were false positives. In addition, since some of the code being analyzed was written to be used for multiple purposes (e.g., client and server), there were some paths through the code that were known to be unused by the target applications being analyzed. As a result, the developers wanted to see control flow paths through the code that resulted in the tools' recommendations. For these, we generated HTML reports that would allow source code naviation through a Web browser. HTML navigation through the source code greatly improved programmer productivity. However, updates to the code still needed to be done through a separate editor.

Our next round of security analysis tooling was done in the context of Eclipse plug-in developers. One of the authors (Ted) used the aforementioned tooling to add privileged code and construct authorization policies for one Eclipse Rich Client Platform (Eclipse RCP) [2] plug-in. This activity took six weeks of dedicated work, which was, coincidentally, the same amount of time between milestone for the Eclipse project.

Eclipse RCP contains many plug-ins, and some of the security analysis would need to be serialized because of plug-in dependencies (plug-in one depends on plug-in two, etc.). To make the process repeatable and affordable, we needed to drastically reduce the time needed to do the security analyses. Our goal was to reduce the analysis and remediation time by at least one order of magnitude compared to prior techniques we had tried.

The Eclipse community is comprised of Java development experts who are not necessarily security savy. From the Eclipse developers perspective, any tooling needed to be fully integrated into their development environment — the Eclipse Java Developement Toolkit (JDT) [1] — as well as integrated into their build process.

4. SWORD4J

SWORD4J was designed and developed to address the challenge of making security readily consumable by ordinary programmers without requiring them to become security experts. SWORD4J simplifies the process of security enabling Java code by providing both guided development [12] and expert use modes of operation. All security enablement tasks are performed completely within the context of the Eclipse [2] integrated develpment environment (IDE). The integration of the security tools within the IDE reduce the cognitive load on the developer that would otherwise be introduced by constantly switching contexts between a wide variety of outputs, commands, graphical user interfaces (GUIs), and external documentation.

4.1 Guided development

As described in Section 2, there are many security-related tasks to be addressed by a programmer. It is often difficult for programmers to know which security topics to learn and how to apply that knowledge to their code. Those who are new to security can easily feel overwhelmed by the whole process. Even for those developers who have previously been through the process of securing their application, the guide provides a reminder of the set of tasks that need to be performed, and can automatically initiate the automated code analysis needed for some of the security tasks.

Guided development is supported by *Cheat Sheets*, an educate-as-you-go approach to security enablement. The Cheat Sheets provide an ordered set of tasks to follow based upon the type of application they are developing (e.g., plain Java, OSGi). Each Cheat Sheet provides both textual documentation, and automation that invokes the appropriate

analysis at each step in the security enablement process. In addition, SWORD4J has video screen captures of demonstrations of key features of the tool.

A developer typically chooses the Cheat Sheet which best matches the projects' stage in the development lifecycle. For example:

Table 1: Cheat sheet selection matrix

Project type	*Cheat Sheet to select*
New	Developing a security enabled Eclipse plug-in
Pre-existing	Security-enabling a Java application using source code analysis
Third Party Component	security enabling a Java application using deep static analysis

There are differences in the sets of tasks to be performed based upon the type of analysis being performed and the type of project being analysis. The following gives a generalized overview of the guided security enablement process:

Code authorization tasks

- Review and update the required authorization (for OSGi and Eclipse plug-in projects)
- Review privileged code placement suggestions. Those locations in the code should be trusted libraries, whose inputs (parameters and objects/fields that are to be part of the trusted function) are not tainted, or have been sanitized.
- Refactor the code to eliminate unsanitized parameters and other input values to privileged code.
- Review and refactor nested privileged operations, eliminating redundant calls to `AccessController.doPrivileged()`.

Best practices

- Review mutable classes. If any listed mutable classes were intended to be immutable, refactor the code as necessary.
- Review native methods to follow best practices.
- Review member (fields, methods) accessibility to make it as restrictive as possible.

Debug authorization policy

- Select the project you would like to debug.
- Launch the debugger
- Review run-time security exceptions generated during the debug session
- Update authorization policy based on the information from the debugger

Export and digitally sign code

- Select the Java project to export and sign
- Sign the selected Java Archive (JAR) file

Each of these tasks has a number of subtasks, which we will review later in this paper.

Once a developer has reached a level of proficiency, where they are comfortable with the secure development process, SWORD4J continues to provide contextual help. Additionally, some analyses are performed automatically while editing or generating code. Examples include Java SE incremental permission analysis and warning generated for deviations from security best practices.

4.2 Authorization policies

As was described in Section 2, determining authorization policies is a complex multi-step process. This involves determining the Java authorization policies that need to be granted to each codebase (e.g., JAR file), potentially refactoring some of code to make it "privileged", making sure that no tainted data is inappropriately propagated to privileged operations, test cases are written and run against the code, and `SecurityException`s are investigated and addressed (e.g., policy changes and/or privileged code additions) so that the code will not unexpectedly throw `SecurityException`s when it executes.

We addressed the time to perform authorization analysis and privileged code placement along a number of dimensions. In our earlier tooling, as described in section 3, our user community accepted the analysis results as being useful. However, they were dissatisfied with the time required to run the static analysis of the entire project under development.

4.2.1 Static analysis time reduction

The deep static analysis we were using in SWORD4J would build control flow and data flow graphs of the entire project, starting with a set of entry points (e.g., all of the `public` and `protected` methods) and the control flow graph represented execution paths through all of the reachable code. The typical time to execute this deep static analysis, and perform the policy and privileged code placement algorithms, would be anywhere from 2 to 15 minutes for small to moderate sized projects[7]. The deep static analysis would also consume substantial amounts of RAM[8] What developers were requesting was immediate feedback in the IDE while they were entering new code or modifying existing code — no waiting for the deep static analysis.

We added the ability to perform the authorization and privileged code analysis incrementally in addition to our previous static analysis of the code by analyzing the entire component / application. The incremental analysis is performed by precomputing and caching the authorizations required for the all of the code upon which the developer's projects depends. As the developer is entering new code, SWORD4J locates method invocations and consults the authorization cache to see if there are any authorization requirements for newly entered method invocations. The developer can then update athorization policy, create privileged code, and inspect the code for tainted data flows, as will be described below. While this incremental analysis would work for new code being entered into the IDE, it can also be used to analyze entire projects, thus reducing the amount of time to analyze a previously developed project that is being loaded into the IDE workspace.

4.2.2 Further time reductions

The time to perform the deep static analysis phase was clearly an area of concern, although it might not been one of the biggest contributors to the time needed to perform

[7]Most of the processing time is in the computation of control flow and detailed data flows of the code, upon which the security analyses are run. Control / data flow analysis computation time depends on the complexity of the code being analyzed, as well as the processing speed of the computer.

[8]We typically would run SWORD4J with a 756MB heap in order to perform this deep static analysis.

the authorization policy task. Subtasks that were time consuming for the developer include many of the *very* repetitive editing tasks, including updating of the authorization policy and refactoring of the source code to enable privileged code. Reducing the number of false positives (particularly for common coding patterns[9]), understanding why the authorization requirements are being recommended (navigating the control flow path from the component's entry point to the authorization test), and understanding how tainted data could reach a program point that is being recommended as becoming privileged can also be very time consuming. The goal we had for SWORD4J was to take each of these time consuming developer subtasks and create a suite of automated techniques that would minimize the amount of time a programmer would need to spend on the subtasks. Where possible, provide *quick fixes* that would make appropriate policy updates, code refactorings, and source code navigation / exploring. These quick fixes are often one, or a few, mouse clicks, and provide source code navigation when appropriate.

The goal for SWORD4J is to provide appropriate information in the context in which it is to be used to minimize context swapping that would increase the amount of time (e.g., keystrokes, mouse clicks) and the cognitive load on the developer [28]. Once the static analysis phase is complete (deep analysis or incremental analysis), SWORD4J generates a set of markers and lists of problems in the task list (see Figures 1 and 2). This is consistent with the way developers expect to manage the set of actions that need to be addressed, including compiler errors and warnings. By selecting a problem in the task list, the IDE opens the source code file and navigates to the source statement being referenced by the problem in the task list.

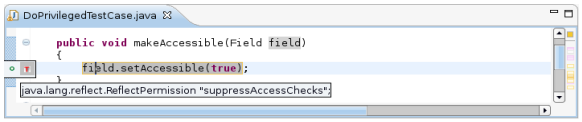

Figure 1: Generated markers displayed in the Eclipse Java Editor

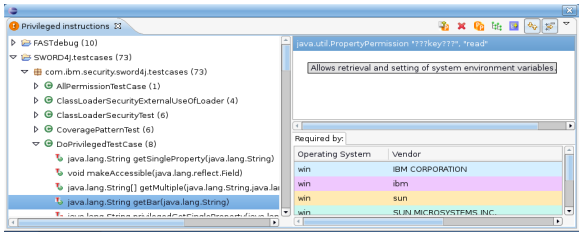

Figure 2: Problem task list

The source code line has marker (see Figure 10) with a set of quick fixes, including:

[9]Consider `if (sm==null) e1 else e2`, where expression `e1` typically does not have a set of authorization policy requirements that `e2` requires. This is a very common pattern in Java code. Filtering out these false positives can save a developer a substantial amount of time.

- Adding a suggested authorization (that can be edited) and added to the authorization policy entry (see Figures 3 and 4). Without this feature, the developer would need to copy and paste (or retype) the policy into a policy file or editor. Depending on the environment (e.g., Java SE, OSGi), the authorization policy has different syntax. SWORD4J correctly formats the policy for the developer.

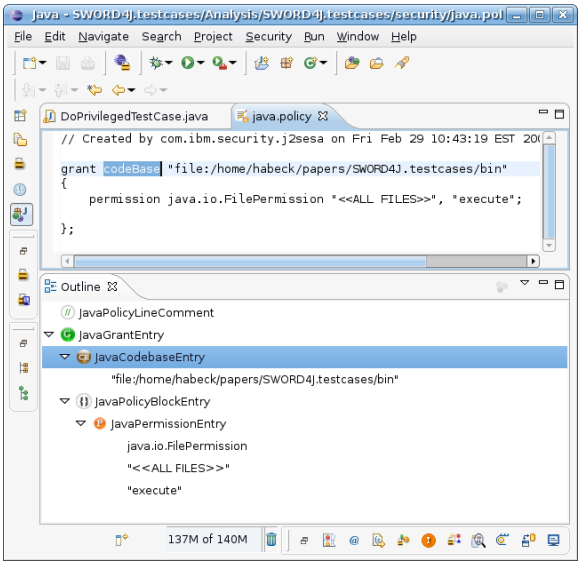

Figure 3: Java Policy Editor

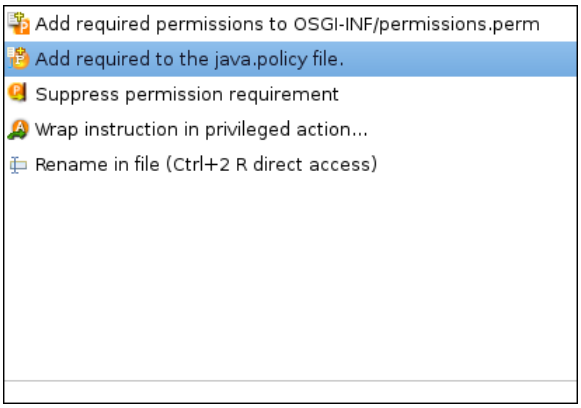

Figure 4: Context menu action to update authorization policy

- Refactoring options to modify the source code so it no longer requires the authorization (and/or make the code privileged — see Figure 10). This may be as simple as removing a statement or expression. However, if the developer wants to wrap the privileged operation into a `PrivilegedAction` instance (to pass to `AccessController.doPrivileged()`), then SWORD4J will initiate Ecipse's refactoring features, thus removing the tedious (and error-prone) process of creating the code

for `PrivilegedAction` instance creation and the call to doPrivileged().

- Bring up a task pane to show tainted data flows to the privileged operations (see Figure 5 and 6). Manually computing the tainted data flows is tedious and time consuming. SWORD4J shows the interprocedural tainted data flows through a combination of tree viewers, to show the control flow paths, and highlighted text to illustrate the tainted data flows from the tainted data source to the privilged operation.

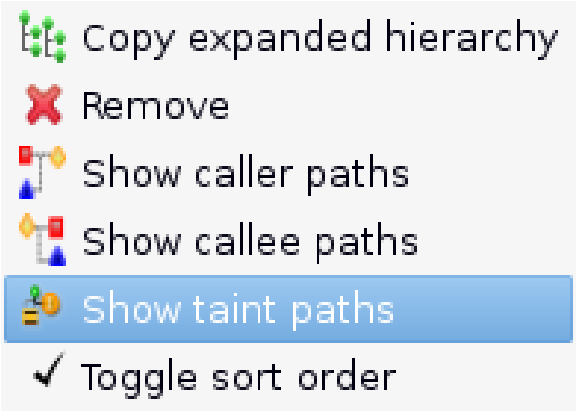

Figure 5: Context menu for tasks list to analyze taint flow

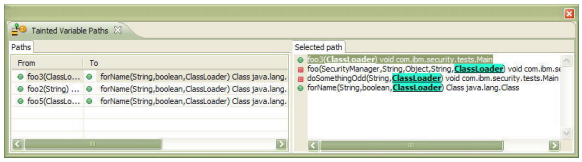

Figure 6: SWORD4J Tainted variable paths data flow viewer

- Task panes to show the control flow paths from the entry points into the component (see Figure 7) to a point in the program that is requiring an authorization. This can be very time consuming for most programs and components used in large applications.

- The control flow paths to the authorization points (e.g., calls to the `SecurityManager`) (see Figure 8). Programmers want to verify that an authorization is really required and not a false positive. Often it is not obvious as to why an authorization is required, so the control flow path helps the developer understand why SWORD4J is making a recommendation.

- One of the many frustrations of working with the Java authorization frameworks is that it offers few clues as to why authorizations fail. One reason is that the policy file is not correctly formatted (syntax error). SWORD4J provides a syntax directed editor to assist the developer in entering syntactically correct policy.

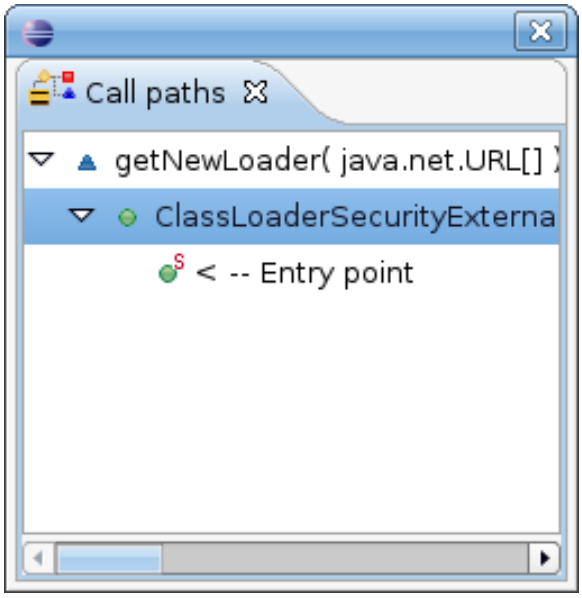

Figure 7: Control flow paths from the entry points into the component

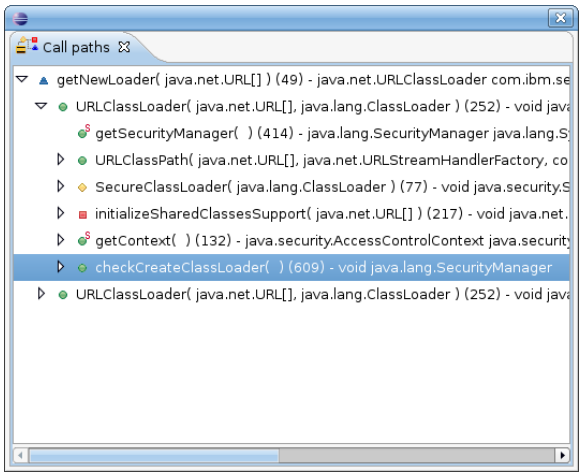

Figure 8: The control flow paths to the authorization points

- Context sensitive help. As the developer processes the various analysis results, assistance is provided in the form of hover text, context menu details, and coding examples. Providing these details, when and where they are needed, rather than in-bulk reduces the context switching by the developer and improves task completion rates (See figure 9).

4.2.3 *Policy debugging*

After the developer has created an initial authorization policy, they are able to launch the code in the IDE debugger to verify that the authorization policy is sufficient to run the code without `SecurityExceptions`. While other tools capture `SecurityExceptions` to infer authorization policies [3], SWORD4J goes further by capturing the `SecurityEx-`

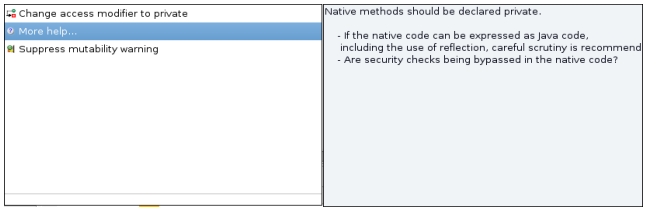

Figure 9: Context sensitive help

ception, including the run-time stack. SWORD4J is able to infer which CodeSource(s) are missing the authorization policy and allow the developer to update the authoriation policy. Quick fixes in the IDE are then provided to allow the developer to update the authorization policy on-the-fly (see Figure XXX).

4.2.4 *Isolation analysis*

As noted earlier, we are interested in identifying cases where (shared) state should be immutable to protect the state of a component. SWORD4J has a task viewer that shows the set of mutable classes and fields that it discovers. By selecting a mutable class in the viewer, a table of reasons show why a class, or one of its static fields, is mutable (see Figure XXX). Some of the reasons for mutability include: a field (static or instance) is accessible and non-final, an mutable object or array referenced by a inaccessible field or object is returned by an accessible method, etc.[10].

The mutable classes are listed in a task viewer. Once a class is selected, the mutable fields are displayed, and selecting a field will result in a display of reasons the field is mutable. For example, if a field is accessible and non-final, there are quick fix refactoring options to make the field final, or make it private and generate getter/setter methods. Returning arrays or mutable objects via an accessible method could be addressed by returning a cloned array/object. While not implemented, it is straightforward to implement such a quick fix.

4.3 Best practices

Just as the JDT is able to identify syntax errors (missing punctuation, undeclared fields and methods, etc.) while the developer is entering code, SWORD4J flags the violations of security best practices. These best practices include limiting the scope of members (fields, methods) in classes, considering public fields to be constants, limiting the extensibility of classes and methods, checking uses of ClassLoaders, SecurityManagers, reporting native declarations that are not private, uses of Java's reflection methods, verifying that security sensitive fields are protected, and identify access to the Java security Policy object.

Part of user interface is a set of "quick fixes" that allow the developer to rapidly address the best practice violations. For example, if a field is public and not final, then one of the quick fixes is to make the field private. One of the advantages of working within the IDE is that it is possible for the editor to search the project (or workspace) to see if there are any uses of the field outside the class. If the value of the field is used elsewhere in the code, one of the quick fix options is to generate getter/setter methods. This saves the developer quite a bit of effort in manually performing a search, or recompiling the project, o see if making the field private would break other code.

Another quick fix option is to allow the developer to tell SWORD4J to ignore the security issue. The quick fix is performed by selecting the marker on the vertical rule bar on the left side of the text pane on the screen. A pop-up context menu then allows the developer to choose to ignore the security violation (see figure 10). In other contexts, a

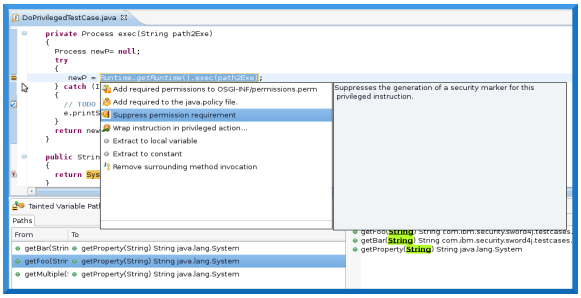

Figure 10: Marker suppression GUI

right mouse click on the highlighted violation results a pop-up menu with an option to ignore the security violation. In either case, a line or block comment of the form $NON-SEC added to the source code, thus directing SWORD4J to ignore the best practice violation.

For security sensitive fields a $SEC-SENSITIVE comment can be added to the code as part of the field declaration. This comment can be added by right mouse-click on the field and selecting the **Mark as security sensitive** action from the pop-up context menu. If there is an expression in the code, such as writing the sensitive field to an OutputStream, SWORD4J will flag that expression. The quick fixes include deleting the expression, suppressing the warning (as described above), or getting addition context sensitive help to explain why the expression is a violation of a best practice.

The other best pracitices are handled in a similar context sensitive manner: markers are generated, quick fixes are provided, and context help is available.

4.4 Code signing

SWORD4J provides a Java KeyStore editor for the Eclipse IDE. This editor facilitates the management of digital certificates. It supports viewing and editing of keystore entries such as: changing certificate aliases, removing certificates, copying certificates between certificate stores, and importing certificates from the file system.

SWORD4J provides a signing dialog to facilitate the signing of JARs from within the Eclipse IDE. This signing utility works together with the KeyStore editor to select which certificates will be used during the signing process.

4.5 Code distribution

Once a software component has been security enabled, it can be signed and queued for deployment to an update site. The update site administrator receiving updates, or components from various third party oranizations will want to know what level of code authorization a given component will require (i.e. what system and network resources will it

[10]See [34] for further discussion about mutability.

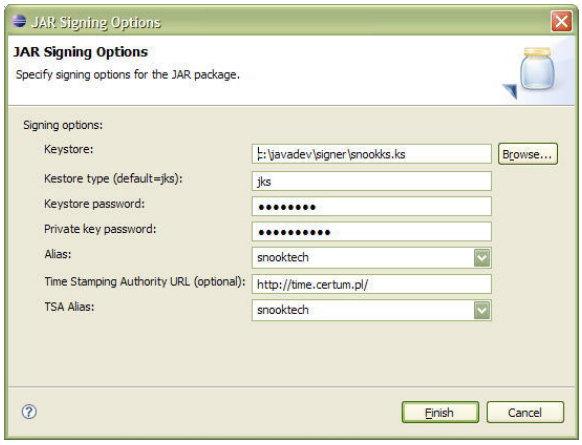

Figure 11: SWORD4J JAR Signing Dialog

have access to based upon its included authorization policy). Additionally, an important aspect of deciding to allow the execution, or hosting of third party-provided component is its associated digital signature. SWORD4J addresses this need by providing a JAR inspection utility. The JAR inspector analyzes a selected JAR within the Eclipse workspace and provides detailed information about the JAR architecture (package sealing), OSGi permissions (Code Authorization policy), and digital certificates (signing information).

4.6 How to keep it secure

Once the components are secured, the organization needs to ensure that they remain secure. SWORD4J has the option of running *headless* builds, whereby security reports are generated via batch process and the reports are stored in a repository for subsequent distribution. These reports can then be imported into Eclipse IDEs, where developers can view the analysis results. SWORD4J also generates summary reports which can be used by the build administrators and the quality assurance team.

5. DISCUSSION

Section 2 outlines a set of representative development tasks for securing Java code, and section 4 described how SWORD4J addresses these tasks. In this section, we return to the set of tasks of section 2 and describe how SWORD4J is able to make improvements on the development process.

(A) SWORD4J addresses the security learning process, to a limited degree, by providing online material that is relevant to the security enabling. There could be better integration with existing online resources. How to integrate educational reasources into the development process is a research opportunity.

(B) Similar to (A) with respect to integrated resources for learning while using the IDE. However, with respect to learning how to debug the authorization policy, SWORD4J greatly reduces the learning curve since the SWORD4J policy debugging is integrated into the IDE (see section 4.2.3, Policy debugging). The policy debugging support also provides a clear advantage over the traditional approach since SWORD4J figures out which `CodeSource`s need an authorization, and a quick fix will update the policy directly (no editing required).

(C) SWORD4J supports run-time authorization through a menu selection, or is automatic when using the Cheat Sheets. While not a big time savings for the developer, it is one less topic of concern.

(D) SWORD4J does not provide any support for test case generation. SWORD4J focuses on determining authorization policies through the use of static analysis. To the extent that policy creation cannot be fully automated (unsoundness due to all `native` methods not being modelled by the static analysis engine, as well as the `String` or literal values to the Permission class constructors), some test cases will still be needed

(E) and (G) Similar to (B). SWORD4J provides debugging support that allows for quick fixes to update authorization policy entries that are missing or are incomplete. This is clearly an advantage over the traditional approach to policy debugging.

(F) SWORD4J has quick fixes that allow for updating of the policy without having to know the location of format of the policy description. In addition, there is a policy editor which verifies the policy syntax (e.g., for the Java `PolicyFile` format), as well as the run-time policy debugging as noted in (E).

(H), (I) and (J). SWORD4J provides refactoring automation for creating privileged code that greatly reduces the amount of code editing needed for creating `PrivilegedActions`. Also, SWORD4J has tainted data detection, that shows the tainted data flow path. These features can save the programmer a considerable amount of time since these are common operations.

(K) and (L) Secure coding guidelines and mutability analysis are partly automated by SWORD4J. What is clearly valuable is that it can find the locations in the code where the guidelines are being violated, as well as some limited refactoring support. For mutability issues, SWORD4J clearly has an advantage over other techniques since it can automate the mutability computations that are tedious and error prone when performed manually. For large code bases, mutability analysis may not be practical due to the amount of time that would be required.

(M) Complete mediation is currently not implemented in SWORD4J, although it was implemented for C code using the same static analysis infrastructure [47]. This can be a very time consuming tasks in the absence of automation. Our experience was that automation can reduce a task through the use of several minutes of computation.

(N) Code signing and keystore management integrated into the IDE does not provide a big time savings, but does result in a better developer experience by not having to exit the developement envioronment to perform this task.

(O) SWORD4J has support for being integrated into the build process, as was described above in section 4.6 “How to keep it secure”.

5.1 Early emperical results

As previously noted, without SWORD4J, securing one Eclipse RCP plug-in took six weeks. Using a subset of SWORD4J that included privileged code placement with call path reporting, and authorization policy analysis, a new professional programmer was able to secure six Eclipse RCP plug-ins in six weeks. This included testing the plug-ins to ensure that they would continue to function as expected (us-

ing existing regression tests). However, this developer did not have the tainted data analysis feature. This missing feature was one of the reasons that limited his productivity in securing the plug-ins.

Even with the limited security analysis support, this programmer was six times more productive than before. We found these early results very encouraging, leading us to believe that we could improve security enablement by one order of magnitude.

6. WEB APPLICATIONS

Securing Web applications shares many of the same characteristics as securing Java components, although the details are quite different.

Web programmers, like Java programmers, need to learn the basics of secure programming. As with Java, there are many security best practices that are described in books and can be found on the Web. For the Web, however, there are additional challenges due to programming-model differences between the client and the server, as well as security concerns related to the network protocol, which can introduce security vulnerabilities.

Web programmers need to make authorization decision both at the server and at the client. Server-side authorization is usually based on a client's identity (e.g., user id and password). However, authorization must still be correctly configured for all of the applictions' entry points. Secure client side cross-domain communication is challenging (e.g., [21]).

Test cases need to be created for the Web applications, including testing for common Web vulnerabilities, such as cross-site scripting, cross-site request forging, and SQL and command injection. Black box testing appears to be a popular technique for detecting these vulnerabilities (e.g., [4]).

Once Web applications are created, they need to be maintained, resulting in the need to include security in the maintenance and build process. We hypothesize that adding integrated security analysis to Web development tools will be beneficial to Web programmers.

7. RELATED WORK

Static and dynamic analysis techniques have both been used for modelling authorization algorithms. Much of the work has focused on performance optimizations or on providing alternatives to the existing approaches employed by Java 2 [33, 32] and CLR [22]. Pottier, Skalka, and Smith [35] extend and formalize Wallach's security passing style [44] via type theory using a λ-calculus, called λ_{sec}. However, their approach does not model all of Java's authorization characteristics, including multi-threaded code and analysis of incomplete programs [37], nor does it compute the authorization object, which often includes identifying the `String` parameters to the `Permission` object's constructor. These requirements are all addressed by SWORD4J.

The static analysis algorithms for authorization and privileged-code analysis included in SWORD4J were described in two conference papers [30, 27]. The analysis described in those papers was not modular, was not optimized for efficiency and scalability, and did not include any IDE support. The work described in this paper greatly extends those preliminary implementations and adds support for multability and accessibility analysis.

Bartoletti, Degano, and Ferrari [9] are interested in optimizing performance of run-time authorization testing. This is done by eliminating redundant tests and relocating others as is needed. Additionally [10], they investigate ways in which program transformations can preserve security properties in existing code, particularly in the context of Java. Specifically, the transformations they study include redundant authorization tests elimination, dead code elimination, method inlining, and an eager evaluation strategy for stack inspection. While their model takes privileged code into account, they assume that privileged code has already been inserted, and do not solve the problem of detecting which portions of library code should be made privileged.

Felten, Wallach, Dean, and Balfanz have studied a number of security problems related to mobile code [42, 17, 44, 14, 43, 16, 15]. In particular, they present a formalization of stack introspection, which examines authorization based on the principals currently active in a thread stack at run time (*security state*). An authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [44]. Each method is modified so that it passes a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. By running the application and encoding the security state, the security passing style explores subgraphs of the comparable invocation graph, and discovers the associated security states and authorizations. The purpose of their work is to optimize the authorization performance, while ours is to discover which portions of library code should be made privileged. Our approach analyzes all the possible execution paths, even those that may not be discovered by a limited number of test cases.

Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [19] describe a system that inlines reference monitors into the code to enforce specific security policies. Their objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. Koved, Pistoia, and Kershenbaum [27] describe an algorithm and system for computing Java 2 security authorization requirements for existing Java code. Their algorithm, which is the starting point for this paper, covers many of the subtle aspects of Java 2 security, including authorization requirements for multi-threaded applications and analysis of incomplete programs [37], for the computation of a call graph.

Privileged code has historic roots in the 1970's. The Digital Equipment Corporation (DEC) Virtual Address Extension/Virtual Memory System (VAX/VMS) operating system had a feature similar to the `doPrivileged()` method in Java 2 and the `Assert()` method in CLR. The VAX/VMS feature was called *privileged images*. Privileged images were similar to UNIX `setuid` programs, except that privileged images ran in the same process as all the user's other unprivileged programs. This meant that they were considerably easier to attack than UNIX `setuid` programs because they lacked the usual separate process/separate address space protections. One example of an attack on privileged images is demonstrated in a paper by Koegel, Koegel, Li, and Miruke [25].[11]

[11]In a private communication with Dr. Paul A. Karger [24],

The notion of tainted variables became known with the Perl language. In Perl, using the `-T` option allows detecting tainted variables [41]. Shankar, Talwar, Foster, and Wagner present a tainted-variable analysis for CQual using constraint graphs [39]. To find format string bugs, CQual uses a type-qualifier system [20] with two qualifiers: *tainted* and *untainted*. The types of values that can be controlled by an untrusted adversary are qualified as being tainted, and the rest of the variables are qualified as untainted. A constraint graph is constructed for a CQual program. If there is a path from a tainted node to an untainted node in the graph, an error is flagged. Newsome and Song [29] propose a dynamic tainted-variable analysis that catches errors by monitoring tainted variables at run time. Data originating or arithmetically derived from untrusted sources, such as the network, are marked as tainted. Tainted variables are tracked at run time, and when they are used in a dangerous way an attack is detected. Volpano, Irvine, and Smith [40] relate tainted-variable analysis to enforcing information flow policies through typing. Ashcraft and Engler [8] also use tainted-variable analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables.

None of the works cited here discuss the integration of security analysis tooling into an IDE, or address the developer productivity impact of such tooling.

8. CONCLUSIONS

This paper described a number of challenges in creating secure Java applications that are both challenging and time consuming. A set of technologies were created that greatly improved the productivity of programmers. By analyzing the set of representative tasks for securing software components, we were able to identify those tasks that were most time consuming and technically challenging. We also considered the set of affordances of the environment in which our tools were to be integrated. These included task markers, tasks lists, quick fixes, Cheat Sheets, and other technologies with which the developers were already familar. The result was a set of integrate analysis technologies to target these security tasks, resulting in significant productivity gains.

We also argue that many of the security analysis tasks involved in Java component development are similar to the challenges for Web application development. We postulate that similar technologies for Web applications will have comparable productivity gains for Web developers.

9. ACKNOWLEDGMENTS

We would like to thank the following people.

Eclipse and the JDT developers for creating a great development tool. Marina Biberstein, Bilha Mendelson, and Sara Porat for their contributions to the JaBA static analysis framework, as well as implementations of mutability, permission (authorization policy) and privileged code placement based on this framework. Our colleagues across IBM, including Jeff McAffer, Oleg Besedin, Huey V Chung, Matthew Flaherty, Thomas Watson, Eric Li, Anthony Nadalin, Nataraj Nagaratnam, Jay Rosenthal, Pascal Rapicault, Mary Ellen Zurko, and others whom we have (embarassingly) omitted, who helped us to refine the requirements that led to SWORD4J. Aaron Kershenbaum with whom we had many Java security, static analysis and graph algorithm conversations, and who contributed to the design and implementation of the deep static analysis engine. Jeff McAffer with whom we collaborated on the idea of incremental permission (authorization policy) analysis.

he indicated that privileged images had been a very significant source of security attacks in the VAX/VMS operating system, and required many patches and updates over the years. He did extensive work on resolving those problems at DEC in the 1979-1980 timeframe.

10. REFERENCES

[1] Eclipse^{TM} Foundation. The eclipse platform project at http://www.eclipse.org.

[2] Eclipse^{TM} Foundation. The equinox project at http://www.eclipse.org.

[3] http://www.jchains.org.

[4] Appscan suite for web application security testing, 2007.

[5] Category:owasp top ten project, http://www.owasp.org/index.php/OWASP_Top_Ten_Project, 2007.

[6] OSGi^{TM} *Alliance.OSGi*^{TM} Service Platform Core Specification Release 4, Version 4.1 http://www.osgi.org/Release4/HomePage, April 2007.

[7] Secure coding guidelines for the java programming language, version 2.0 http://java.sun.com/security/seccodeguide.html, 2007.

[8] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, USA, May 2002. IEEE Computer Society.

[9] M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. In *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science*, volume 54, Amsterdam, The Netherlands, 2001. Elsevier.

[10] M. Bartoletti, P. Degano, and G. L. Ferrari. Stack Inspection and Secure Program Transformations. *International Journal of Information Security*, 2(3):187–217, Aug 2004.

[11] F. P. Brooks Jr. The mythical man-month: After 20 years. *IEEE Software*, 12(5):57–60, 1995.

[12] J. M. Carroll and C. Carrithers. Training wheels in a user interface. *Commun. ACM*, 27(8):800–806, 1984.

[13] Cve - common vulnerabilities and exposures (cve). http://cve.mitre.org/.

[14] D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, 1997. ACM Press.

[15] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Silver Spring, MD, USA, 1996. IEEE Computer Society Press.

[16] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java Security: Web Browsers and Beyond. Technical Report 566-597, Princeton University, Princeton, NJ, USA, February 1997.

[17] R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, Jan. 1999.

[18] D. Dillenberger, R. Bordawekar, I. C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. S. John. Building a java virtual machine for server applications: the jvm on 0s/390. *IBM Syst. J.*, 39(1):194–210, 2000.

[19] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, CA, USA, May 2000. IEEE Computer Society.

[20] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[21] M. S. S. C. S. Y. Frederik De Keukelaere, Sumeer Bhola. Smash: Secure component model for cross-domain mashups on unmodified browsers. In *17th International World Wide Web Conference*, 2008.

[22] A. Freeman and A. Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, Jun 2003.

[23] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[24] Paul A. Karger, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. Private communication, 17 December 2004.

[25] J. F. Koegel, R. M. Koegel, Z. Li, and D. T. Miruke. A Security Analysis of VAX VMS. In *ACM '85: Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, pages 381–386. ACM Press, 1985.

[26] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java, 2002.

[27] L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.

[28] L. Koved and B. Scneiderman. Embedded menus: selecting items in context. *Commun. ACM*, 29(4):312–318, 1986.

[29] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, Feb 2005. IEEE Computer Society.

[30] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 362–386, Glasgow, Scotland, UK, July 2005. Springer-Verlag.

[31] M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin. *Enterprise Java 2 Security: Building Secure and Robust J2EE Applications*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[32] M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.

[33] M. Pistoia, D. Reller, D. Gupta, M. Nagnur, and A. K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.

[34] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in java, 2000.

[35] F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.

[36] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. Saber: smart analysis based error reduction. In *ISSTA*, pages 243–251, 2004.

[37] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.

[38] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Sept. 1975.

[39] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, USA, Aug. 2001.

[40] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.

[41] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition, July 2000.

[42] D. S. Wallach. *A New Approach to Mobile-Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, Jan. 1999.

[43] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, 1997. ACM Press.

[44] D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.

[45] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[46] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.

[47] X. Zhang, L. Koved, M. Pistoia, S. Weber, T. Jaeger, G. Marceau, and L. Zeng. The case for analysis preserving language transformation. In *ISSTA '06:*

Proceedings of the 2006 international symposium on Software testing and analysis, pages 191–202, New York, NY, USA, 2006. ACM.
Bibliography in this case