A Practical Framework for The Dataflow Pointcut in AspectJ

Amine Boukhtouta, Dima Alhadidi and Mourad Debbabi
Computer Security Laboratory (CSL)
Concordia Institute for Information Systems Engineering
Concordia University
Email: {a_boukh, dm_alhad, debbabi}@encs.concordia.ca

Abstract—In this paper, we present the design and the implementation of the dataflow pointcut in AspectJ compiler ajc 1.5.0. Some security concerns are sensitive to flow of information in a program execution. The dataflow pointcut has been proposed by Masuhara and Kawauchi in order to easily implement such security concerns in aspect-oriented programming languages. The pointcut identifies join points based on the origins of values. The dataflow pointcut can detect and fix a lot of vulnerabilities that result from not validating input effectively, e.g., web application vulnerabilities, process injection, log forging, and path injection. AspectJ extends the Java programming language to implement crosscutting concerns modularly in general. The implementation methodology of the dataflow pointcut which depends in define-use analysis is described in detail together with case studies that demonstrate how the implemented dataflow pointcut can detect a considerable number of vulnerabilities.

I. INTRODUCTION

Security is taking an increasingly predominant role in today’s computing world. The industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the box, even on programs that have not been designed with security in mind. The challenge is even greater when legacy systems must be adapted to high-risk environments, while they are not originally designed to fit into such ones. In some cases, little can be done to improve the situation, especially for Commercial-Off-The-Shelf (COTS) software products that are no longer supported, or their source code is lost. However, whenever the source code is available, as it is the case for Free and Open-Source Software (FOSS), a wide range of security improvements could be applied once a focus on security is decided. As a result, integrating security into software becomes a very challenging and interesting domain of research.

A security of an application is an attribute that permeates the whole system. As such, any attempt to address security concerns must be global in nature. Besides, security solutions must be applied consistently at every relevant location. One way of achieving these objectives is by separating out security concerns from the rest of the application concerns, such that they can be addressed independently and applied globally. A methodology that would encompass separation of security concerns and consistent implementation of security solutions, would pave the road towards secure applications, enable a security expert to specify security properties, and facilitate the correctness verification of security solutions.

More recently, several proposals have been advanced for the injection of security code into an application using Aspect-oriented Programming (AOP) [6] which appears to be a very promising paradigm for software security hardening since it allows separation of security concerns [7]. The most prominent AOP approach is the pointcut-advice model. It is adopted in this paper because it appears to be the most appropriate one to harden security into applications [3]. The fundamental concepts of this model are: join points, pointcuts, and advices.

However, AOP has not been initially engineered with security in mind which resulted in many shortcomings in this technology. There are some contributions that suggest new pointcuts for security purposes such as the one of Masuhara and Kawauchi [9]. They have presented the design of the dataflow pointcut with a web-application example, and its prototype implementation. It has been implemented as an extension to Aspect SandBox (ASB) [10], which is a modeling framework for AOP mechanisms based on the pointcut-advice model. In this paper, we design and implement the dataflow pointcut as an extension to AspectJ compiler ajc-1.5.0. AspectJ is a seamless aspect-oriented extension to the Java programming language that enables clean modularization of crosscutting concerns. This pointcut identifies join points based on the origins of values. The dataflow pointcut can detect and fix a lot of vulnerabilities that result from not validating input effectively, e.g., process injection, log forging, path injection, and web application vulnerabilities. There’s no doubt that web applications have become the attackers’ target of choice. Web vulnerabilities are “the new buffer overflow” plaguing the information security community. Almost every day, Bugtraq and other sites record new flaws in commercial or open source web applications. The security of web applications has become increasingly important in the last decade. More and more web applications deal with sensitive financial and medical data, which if compromised, in addition to downtime can mean millions of dollars in damages. This highlights the need for the design and the implementation of the dataflow pointcut.

The rest of the paper is organized as follows. In Section II, an overview of the current literature is presented on the subjects that are related to the contribution of this paper: AOP and AOP for securing software. Some of input validation vul-
nerabilities together with their threats are presented in Section III. A high-level picture for the ajc compiler together with the design and the implementation methodology are summarized in Section IV. We illustrate in Section V case studies to demonstrate how the implemented dataflow pointcut can detect and fix cross-site scripting and SQL injection vulnerabilities. Concluding remarks as well as a discussion of future work are represented in Section VI.

II. BACKGROUND AND RELATED WORK

We present in the sequel an overview of the current literature on the subjects that are related to the contribution of this paper.

A. Aspect-oriented Programming

AOP depends on the principle of "Separation of Concerns", where issues that crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed. We distinguish from them AspectU built on top of the Java programming language, AspectC built on top of the C programming language, AspectC++ built on top of the C# programming language, and the AOP version addressed for Smalltalk programming language. AspectU and AspectC++ are the most prominent AOP languages. The approach adopted by most of these languages, is called the pointcut-advice model. The fundamental concepts of this model are: join points, pointcuts, and advices.

A join point is a principled point in the execution of a program. A pointcut is a concept that classifies join points in the same way a type classifies values. The pointcut expressions typically allow to pick out function calls, function executions, join points on the control flow anterior to a given join point, etc. An advice is a code fragment executed when join points satisfying a particular pointcut are reached. This execution can be done before, after, or around a specific join point. At the heart of this model, is the concept of an aspect, which embodies all these elements. Fig. 1 shows a tracing aspect written in AspectJ where the pointcut `ptrace()` picks out any call to any method. A before and an after advice are used to display the start time and the end time respectively. AspectJ has a comprehensive and expressive pointcut specification language that allows to specify particular points in the control flow of the program where advices are to be applied.

Finally, the aspect is composed and merged with the core functionality modules into one single program. This process of merging and composition is called weaving, and the tools that perform such process are called weavers.

B. AOP Approaches for Security Hardening

Most of the contributions that explore the usability of AOP for injecting security code into applications are presented as case studies that show the relevance of AOP languages for application security. We present in the following an overview on these contributions.

By means of an example of access control, De Win et al. [4] have investigated how well AOP can deal with the separation of security concerns from an application. In order to construct a more generic solution, they have suggested to abstract relevant pointcuts out of the aspect implementation. Besides, the feasibility to build a security aspect framework has been discussed. Viega et al. [12] have developed a general-purpose aspect-oriented extension to the C programming language that can be used to separate security policies from the code. The authors have pointed out that building in security into an application is preferable to the "penetrate-and-patch" approach where problems are addressed in an ad-hoc manner when flaws emerge. The aspect language specifies structured transformations that are inserted in the code at well-defined points. Points of interest for such insertion could be calls to functions, function definitions, and pieces of functions.

A case study has been described [11] by Ramachandran et al. to incorporate multilevel security system (MLS) using aspects. A MLS has two goals: First, it is intended to prevent unauthorized personnel from accessing information at a classification they are not authorized to. Second, it is intended to prevent personnel from changing the classification of information they do have access to. AspectJ is used to intercept Java library calls in order to enforce MLS policy. The authors have described how AspectJ can actually go further than conventional object-oriented approach to achieve stronger enforcement of MLS.

Masuhrara and Kawachi [9] have defined a dataflow pointcut for security purposes but this pointcut has not been implemented yet. The pointcut identifies join points based on the origins of values. Cross-site scripting (XSS) problem in web-applications is an example presented by them to clarify the need for such a pointcut. Local variables set and get pointcuts [3] have been introduced to protect the privacy and integrity of sensitive data. They allow to track the values of local variables inside a method. Many security hardening practices require the injection of code around a set of join points or possible execution paths. Examples of such cases would be the injection of security library initialization/deinitialization, privilege change, and logging. M-A. Laverdière et al. [8] have proposed new pointcuts that enable the identification of a join point common to a set of other join points where code can be injected efficiently once for all of them.
III. INPUT VALIDATION VULNERABILITIES

Input validation vulnerabilities are basically the problem of insecure information flows. Errors in validation of user inputs lead to serious security vulnerabilities that break the integrity and confidentiality properties of any system. In the sequel, some of these vulnerabilities, which can be detected and fixed easily by the dataflow pointcut, are presented to highlight the importance of the implementation of this pointcut.

- **SQL injection**: SQL injections put the data in the database at risk. Since user input is used invalidated in the SQL statement, the attacker can inject any SQL statement they wish to execute. This includes deletion, update, or creation of data. It may also be possible to retrieve sensitive data from the database with this type of vulnerability.

- **Cross-site scripting (XSS)**: An XSS vulnerability is the result of using unchecked user input in information used in the web interface. This data can contain arbitrary content such as HTML and SCRIPT tags. If this information is displayed for other users of the system, these scripts can be used to manipulate the user’s session, direct them to other web sites, perform phishing attacks or run other malicious script content.

- **Tainted index used for array access**: When invalidated user input is used as an index of an array, this can then be used to cause denial of service or to manipulate the state object to create an alternate control flow which could be used to the advantage of the attacker.

- **Unchecked e-mail**: E-mail addresses like all other user input is vulnerable to attack via malicious content. For example, an application may send e-mail to a user by invoking the sendmail command directly using the e-mail string stored for the user. This string if used unchecked, can contain arbitrary commands that would be run along with sendmail on the application host. Also, malicious users can use your web server to send spam e-mails or e-mails with unwanted contents.

- **Process injection**: In general, process creation or execution of external commands within an application is a security concern. There is a serious vulnerability if user input is used in any part of the command string used for execution. Attackers can inject additional commands and have them executed on the application server leading to a process or a command injection condition. The ability to run arbitrary commands can lead to denial of service, data corruption, data security violations, and other risks.

- **Log forging**: Log entries allow you to create automatic alert tools, gain statistics, or trace an attacker when there is a security breakdown. Log forging allows an attacker to manipulate log records by adding false entries which may confuse automatic scanning utilities and allow attacks to happen unnoticed or allow attackers to misdirect the reviewer after an attack has happened.

- **Path and file name injection**: There is a security vulnerability if user input, unchecked, is used in any part of a file or a path string used for execution. Attackers can manipulate files and paths to write data or to access data on the host system. A typical attack could be to manipulate a file name to access the `/etc/passwd` file from the host system. In general, path injections can compromise the security of the file system on the host server.

IV. DATAFLOW DESIGN AND IMPLEMENTATION IN ASPECTJ

We have designed and implemented the dataflow pointcut as an extension to AspectJ compiler ajc 1.5.0. The implemented dataflow pointcut tracks data dependencies inside methods. The implementation task is not an easy task because it requires digging into the actual code of the ajc compiler. The design and the implementation of the dataflow pointcuts depends primarily on matching. The matching of the dataflow pointcut has two levels. The first level corresponds to the matching of the pointcut p enclosed by the dataflow pointcut \( df\text{flow}(p) \) where \( p \) is any of the defined AspectJ pointcuts. The second level resides in checking if a join point has a define-use relationship with the join point that is matched by the enclosed pointcut \( p \). Accordingly, a join point \( jp \) matches a dataflow pointcut \( df\text{flow}(p) \) either:

- If this joint point \( jp \) matches the enclosed pointcut \( p \) of the dataflow pointcut and at the same time defines a value in the operand stack of the Java Virtual Machine (JVM) [2] or,
- If this join point has a define-use relationship with the join point \( j'p \) where \( j'p \) matches the enclosed pointcut of the dataflow pointcut \( p \) and defines a value in the operand stack of the JVM.

To facilitate the implementation task, a high-level description for the ajc compiler is required. After that, the design and the implementation methodology are detailed.

A. A High-level Description of AspectJ Compiler ajc

A high-level picture of ajc is presented in this subsection. It is composed, as appears in Figure 2, of the following three primary modules:

- **Package org.aspectj.ajdt.core**: This is the compiler front-end that extends the eclipse Java compiler from
org.eclipse.jdt.core. It compiles both AspectJ and pure Java source code into pure Java bytecode annotated with additional attributes representing any non-java forms such as advice and pointcut declarations.

- **Weaver**: This is the compiler back-end that provides the bytecode weaving functionality. It implements the transformations encoded in the attributes that represent any non-java forms as advice and pointcut declarations to produce woven class files.

- **Runtime**: These are the classes that are used by the generated code at runtime and must be redistributed with any system built using AspectJ.

To compile a pointcut, it should go through the phases of its life cycle which are:

- **Parsing**: It corresponds to build an abstract syntax tree where its nodes represent different declarations and designators provided in AspectJ syntax. Once nodes are set by the front-end AspectJ compiler, they can be formatted as patterns. A pattern is a syntactic representation that holds information about types, fields, methods, pointcuts, advices and aspects.

- **Resolution**: It corresponds to the association of a parameter declared in an advice to an identifier enclosed by a pointcut.

- **Serialization**: The defined pointcuts are serialized into attributes in the class file.

- **Deserialization**: The weaver unpacks serialized attributes when a class file is loaded by reading input data stream and recording declared aspects. Once aspects are added, the crosscutting members are extracted as shadow mungers. A shadow munger is a representation of declared advices that enclose pointcut designators where each one has its own signature.

- **Concretizing**: The weaver concretizes a shadow munger by replacing its named pointcuts by their declarations. Declared pointcuts that are held in shadow mungers can be either named or anonymous. A named pointcut has a name and can be referred to by another pointcut. An anonymous pointcut has no name and cannot be referred to.

These phases are followed by weaving. The weaving concept is implemented on bytecode [2] rather than on the Java source code. The weaving process matches join points against defined shadow mungers. Every join point has a corresponding static shadow in the bytecode of the program. Shadows are either a single bytecode statement or a bounded region of bytecode. The matching is based on fuzzy boolean operations. A fuzzy boolean can take one of the following values: maybe, never, no, or yes. This kind of boolean is used since a matching decision cannot be taken directly in the case of some pointcuts, i.e., args and cflow pointcuts. Tests are needed in such cases. Finally, if a join point matches a shadow munger, code is inserted at these join points to modify the dynamic behavior of the program.

### B. Implementation Methodology

In this subsection, we discuss the implementation methodology that is followed to implement the dataflow pointcut. It depends on define-use analysis of JVM [2] and consists of the following steps:

- The ajc parser is extended to recognize and parse the syntax of the dataflow pointcut. The general form of the dataflow pointcut is \( d\text{flow}(p) \) where \( p \) is any of the defined AspectJ pointcuts.

- All the aforementioned stages of any pointcut life cycle are went through by the dataflow pointcut.

- A new matching stage has been added to match the pointcut \( p \) of the dataflow pointcut \( d\text{flow}(p) \). If \( p \) matches a bytecode instruction, all the subsequent instructions in the same method are visited including the bytecode instruction that matches \( p \) itself. Once a bytecode instruction has been visited, its behavior is simulated by a defined stack as follows. If the bytecode instruction defines a value on the operand stack of the Java Virtual Machine (JVM), it will be pushed on the defined stack. On the other hand, if the bytecode instruction uses a value from the operand stack, the first bytecode instruction on the defined stack will be popped and a dependency relationship between the two instructions is created in a special structure represented as a hash table. A hash table is used to represent the dependency relationships by having the bytecode instruction that uses a value from the operand stack as an entry connected to a dependency list which contains the bytecode instructions that this instruction pops from the defined stack. In the case of a bytecode instruction that can be reached from a branch instruction, the branch instruction is also added to its dependency list.

- An analysis of the dependency relationships in a hash table has been done to track data dependencies between instructions with the help of the Byte Code Engineering Library (BCEL) API [1]. The BCEL API is intended to give users a convenient possibility to analyze, create, and manipulate binary Java class files. Analyzing the dependency relationships takes into consideration transitivity relationships and branch instructions.

```java
public int field;
private test() {
  boolean flag = true;
  int k=9;
  if(flag)
    k=9;
  else
    j=k;
  field = k;
}
public int inc(int i) {
  return i+1;
}
```

Fig. 3. Dataflow Analysis Example
To clarify the idea, suppose that the code which is presented in Fig. 3 is compiled with the following dataflow pointcut:

dflow(call(int test.inc(..))). The generated bytecode for the method test is as follows:

ICONST_1  
ISTORE_1  
BIPUSH 9  
ISTORE_2  
ALOAD_0  
ILOAD_2  
INVOKESPECIAL test.inc (I)I  
ISTORE_3  
ILOAD_1  
IFEQ L0  
ILOAD_3  
ISTORE_2  
GOTO L1  
L0:  
ILOAD_2  
ISTORE_3  
L1:  
ALOAD_0  
ILOAD_2  
PUTFIELD test.field I  
RETURN

To demonstrate how the hash table of the dependency relationships is populated, let us see the following cases. The bytecode instruction

INVOKESPECIAL test.inc (I)I

matches the pointcut

call(int test.inc(..))
of the dataflow pointcut. It loads a value in the operand stack which is stored in the local variable j by the subsequent bytecode instruction ISTORE_3. As a result we create a dependency relationship between them in the hash table where the bytecode instruction ISTORE_3 is connected to a dependency list that contains the bytecode instruction INVOKESPECIAL test.inc (I)I. Regarding the bytecode instruction IFEQ L0, it has two execution paths. Consequently, two dependency relationships are created where entries in the hash table are the two targets of the branch instruction. The dependency lists of the both entries contain the bytecode instruction IFEQ L0. Tracking branch instructions consists of finding branch bytecode instructions in the dependency list of a given entry in a hash table. If a branch instruction is found, the last bytecode instruction which is before the branch instruction and is within the dataflow analysis is kept in mind. Then a dataflow analysis is continued in each execution path. In the provided example, field is data-dependent on the call of the function inc if and only if the first path of the branch instruction is executed. In order to deal with the propagation of values in such a case, two flags are added to the blocks of the first and the second paths of any branch instruction. The flag is set to true if the path is executed at run time. Afterwards, for each join point in the dataflow analysis of a propagated value, we match this join point. Otherwise, we do not match it. At the end of the method and after the analysis is done, the hash table is cleared.

V. CASE STUDIES

Unvalidated input is a number one security issue for web-applications because it can lead to many security vulnerabilities such as SQL injection and XSS. We present in this section two Java code examples that are vulnerable to these attacks respectively and see how the implemented dataflow pointcut can detect such vulnerabilities.
One of the most important tools released with AspectJ is a graphical structure browser that edits program source files, compiles programs with ajc, runs programs, and navigates crosscutting concerns. This browser is used to demonstrate how the implemented dataflow pointcut picks out the vulnerable join points in a specific code. Fig. 4 contains a code that is vulnerable to XSS together with a dataflow pointcut to detect such a vulnerability. Fig. 6 shows how the defined dataflow pointcut catches calls to `println` methods that their parameter string originates from a return value of a `getParameter` method in a past join point. On the other hand, Fig. 5 contains a code that is vulnerable to SQL injection together with a dataflow pointcut to detect such a vulnerability. Fig. 7 shows how the dataflow pointcut catches calls to `executeQuery` methods that their parameter string originates from a return value of a `getParameter` method in a past join point. These vulnerabilities, in general, can be prevented by validating all input from outside the application using a before or an around advice. Validation should include length and content. Typically only alphanumeric characters are needed. Any other accepted characters should be escaped. For simplification the advice is represented here as a warning message.

VI. CONCLUSION

This paper has introduced the design and the implementation of the dataflow pointcut in AspectJ compiler ajc 1.5.0. The dataflow pointcut has been proposed to implement security concerns that are sensitive to flow of information in aspect-oriented programming languages. The dataflow pointcut can detect and fix a lot of vulnerabilities that result from not validating input effectively, e.g., web application vulnerabilities, process injection, log forging, and path injection. The dataflow matching approach is based on tracking dependencies between bytecode instructions. However, this approach has a limitation, which resides in making only an intra-procedural analysis for data dependencies to match the dataflow pointcut. Regarding future work, we will target inter-procedural analysis for dataflow pointcut matching. Besides, we are intending to propose, design and implement more security hardening constructs as extensions to AspectJ and other AOP code production compilers.

REFERENCES