Intrusion Detection Based On Dynamic Information Flow Analysis

Wes Masri
Computer Science Dept.
American University of Beirut
Beirut, Lebanon 1107 2020
wm13@aub.edu.lb

Andy Podgurski
Electrical Engineering & Computer Science Dept.
Case Western Reserve University
Cleveland, OH 44106
podgurski@case.edu

ABSTRACT
This paper presents a new approach to detecting intrusions against application software, whose primary goal is facilitating identification and repair of security vulnerabilities rather than permitting online response to attacks. The approach employs fine-grained dynamic information flow analysis in conjunction with policy enforcement, signature matching, and anomaly detection. Program executions are captured in the field and are replayed, profiled, and analyzed offline. Suspicious ones are audited manually. The approach is capable of revealing a wide variety of attacks against software, including both ones that violate confidentiality or integrity requirements and ones that do not. A prototype tool implementing the approach has been developed for Java byte code programs. To illustrate the applicability of the approach, it is applied to reveal security vulnerabilities in several open-source programs.

Categories and Subject Descriptors

General Terms
Computer security, Experimentation.

Keywords
Dynamic information flow analysis, intrusion detection, information flow signature matching, information flow anomaly detection, information flow policies.

1. INTRODUCTION
Information flow control, which deals with restricting the flow of information between objects manipulated by a program, is a classical subject of computer security research [2][8]. It is necessary because controlling access to individual objects is not sufficient to prevent indirect propagation of information resulting in either leakage of information from sensitive objects to untrusted recipients or tampering with sensitive objects by untrusted agents. For example, a program that legitimately accesses confidential information associated with one user may, under certain conditions, disclose it to another user who is not authorized to access it.

Despite its long history, research on information flow control has until recently had relatively little impact on the field of intrusion detection, which deals with detecting and responding to intrusions/attacks against systems and applications. This is surprising because the purpose of certain sophisticated attacks against software is precisely to induce insecure information flows to or from sensitive objects, and such flows may be the only indication of such attacks. Recently, Zimmerman et al proposed an intrusion detection model based on runtime enforcement of an information flow policy, which specifies which information flows are permissible in a given system [31][32]. This model is intended to address problems with the detection reliability and accuracy of traditional intrusion detection systems (IDSes) as well as their maintenance costs. Such systems are of two types: signature matching systems and anomaly detection systems. The former look for intrusion signatures, which are characteristic indications of known attacks. Note that an intrusion signature may be based on superficial aspects of an attack. Anomaly detection systems look for system/application behavior that is anomalous in the sense that it differs markedly from normal, safe behavior. Signature matching systems tend to issue fewer “false positive” alerts than anomaly detection systems, but the former cannot detect novel attacks, while the latter can. In practice, both kinds of systems suffer from problems of inaccuracy and imprecision. Zimmerman et al argued that their model detects confidentiality and integrity violations more reliably than either signature matching systems or anomaly detection systems do because it focuses on policy violations rather than on ancillary events.

Enforcement of information flow policies is not a complete solution to the problem of intrusion detection. First, many attacks do not cause confidentiality or integrity violations. Second, information flow policies may suffer from the same types of problems that affect other software specifications, such as incorrectness and incompleteness. In this paper, we present evidence that dynamic information flow analysis (DIFA) has important applications to intrusion detection besides the enforcement of information flows policies. These applications are based on the fact that patterns of information flow that occur during execution of a program characterize its computation to a high degree.
We hypothesize that many attacks have information flow signatures – distinctive flow patterns that can be specified once the attacks are diagnosed and subsequently can be recognized by a signature matching IDS. Such attacks need not involve violations of an information flow policy. Moreover, information flows that are critical to such attacks are preserved when the attacks are disguised in certain ways, such as by paddling the attack sequence with irrelevant operations. This means that an IDS that matches information flow signatures may be resistant to such evasion attempts. Of course information flow signatures are applicable to detecting only known attacks. However, we hypothesize that novel attacks often cause anomalous patterns of information flow, which can be detected by applying anomaly detection techniques to information flow profiles, that is, to execution profiles characterizing a program’s dynamic information flows. In Section 5.3, we describe how such information flow anomaly detection by adapting the observation-based testing technique cluster filtering [5].

There is a tradeoff involving the precision, granularity, and cost of DIFA. Course-grained DIFA is less precise than fine-grained DIFA, making the former more prone to false positives. Fine-grained DIFA has higher overhead, which precludes online application with processing intensive applications [17]. Zimmerman et al.’s policy-based model deals with information flows between entire objects. They assert that a finer-grained online analysis, which would involve analyzing (1) data flows involving object fields and local variables and (2) control dependences, is unrealistic on a large-scale operating system. In our approach, in Section 4, we describe how the added precision of fine-grained DIFA can be exploited in the context of corrective software maintenance by applying it offline with captured executions, to reveal attacks that were not detected in the field by users, administrators, IDSes, application firewalls, etc. We present an approach to detecting and correcting security vulnerabilities that is based on this idea.

In Section 5.2, we describe our implementation of information flow signature matching, information flow anomaly detection, and information flow policy enforcement, which is based on our fine-grained DIFA tool DynFlow. In Section 6 we illustrate the utility of combining these techniques, by applying them in several case studies involving security vulnerabilities in open source software.

2. ASSUMPTIONS

To limit the scope of our investigation, we focus on attacks against applications that do not compromise the underlying computing platform or violate the semantics of the implementation language. Thus, for example, we do not attempt to address buffer-overflow attacks or other attacks that exploit vulnerabilities in an operating system, runtime system, or virtual machine. Even with these constraints, an enormous variety of attacks are possible, namely, all attacks that cause an application to behave contrary to its requirements (whether they are specified or not) to the advantage of the attacker and to the detriment of the application’s stakeholders.

Naturally, information flows are not the only aspect of program execution that are relevant to the detection and prevention of such attacks against applications. We assume that a comprehensive approach to application-based intrusion detection would involve profiling and analyzing a number of other aspects of program execution, including (a) application-specific ones and (b) generic ones like code coverage, input and variable values, object states, event sequences, and event timing. However, these are not addressed here.

3. RELATED WORK

Haldar et al. extended the Java Virtual Machine (JVM) to add mandatory access control (MAC) that allows information flow policies to be enforced [10]. Their mechanism tracks information flows at the granularity of objects and does not consider flows involving user variables or control dependences, which makes it substantially less precise than DynFlow. Brown and Knight describe a trusted computing base supported by special hardware that enforces an information flow policy [1]. Lampson motivated research on information flow analysis by describing the problem and listing a number of possible information leaks [14]. Fenton proposed an abstract machine called the Data Mark Machine to support dynamic checking of information flows. Static analysis techniques for validating information flows are usually safer but less precise than dynamic mechanisms, because they treat all control flow paths as executable. Denning and Denning proposed a technique based on static control flow and data flow analysis for verifying a program’s compliance with an information flow policy [2][3]. A number of language-based static type checking systems have also been proposed [22]. In such systems every program expression is assigned a security type in addition to its ordinary type. In type checking a program, the compiler ensures that the program cannot exhibit illegal information flows at run-time.

Leon et al. evaluated the effectiveness of information flow analysis as a basis for filtering test cases, but they did not address security vulnerabilities [15]. The Perl scripting language [20] provides a special execution mode called taint mode, in which all user supplied input is treated as suspicious unless the programmer explicitly approves it. Taint mode is very effective at preventing flows from user supplied input data to potentially dangerous system calls, but is unable to detect or prevent any other type of flows, including ones resulting from control dependence.

4. CAPTURE/REPLAY, OFFLINE DIFA, AND CORRECTIVE MAINTENANCE

In a previous empirical study [18], we obtained empirical evidence suggesting that with processing-intensive applications, the overhead of fine-grained DIFA was too high to permit its use online. With non processing-intensive applications, online information flow analysis appears feasible. For example, this is the case with the Servlet and JSP based applications described in Section 6. To enable the precision of fine-grained DIFA to be exploited for detecting security vulnerabilities in processing-intensive applications, we propose exploiting lightweight capture/replay technology, such as that described in [21][26], to enable DIFA to be applied offline to detect attacks that were not detected in the field. In this approach, only application inputs are captured online, and they are later used by developers to replay executions offline so that DIFA can be done without affecting users unduly. This requires that the overhead of execution capture is relatively low. The ability to replay captured executions repeatedly makes it possible for suspicious executions to be more accurately identified.
identified by policy violations, signature matching, or anomaly detection can be audited manually to confirm that an attack really did occur. The availability of captured executions also facilitates debugging, permits statistical estimation of attack and failure rates, and provides a basis for characterizing "normal" software usage in the context of anomaly detection. Naturally, the proposed approach precludes online detection and interruption of attacks, but it permits vulnerabilities to be discovered and repaired so as to prevent future attacks.

5. IMPLEMENTING A GENERALIZED INFORMATION FLOW BASED IDS

In [17] we presented a new approach to DIFA that can be used to detect, prevent and debug insecure flows in programs, and we described a prototype tool implementing the approach for Java bytecode programs. We used this tool, which is called DynFlow in the sequel, to implement fine-grained DIFA based policy enforcement, signature matching, and anomaly detection. As originally designed, DynFlow can be used to prevent and detect violations of information flow policies and record information flow profiles of executions. An information flow profile records the number of times information from x flowed into y via dynamic data and control dependences, where x and y are local variables, global variables, or fields of a class instance. DynFlow easily handles language features that are difficult to handle in a static context, such as pointers, arrays, and dynamic binding. DynFlow handles both intra-procedural and inter-procedural dependences (though it does not currently address dependences resulting from exceptions) and it handles data flows between threads in multi-threaded programs.

DynFlow comprises two main components: the Instrumenter and the Profiler. The preliminary step in applying DynFlow is to instrument the target bytecode classes and/or jar files using the Instrumenter, which was implemented using the Byte Code Engineering Library, BCEL [28]. The Instrumenter inserts a number of method calls to the Profiler at given points of interest. At runtime, the instrumented application invokes the Profiler, passing it the information that enables it to monitor information flows. Note that when applying DynFlow to a program, one can instrument any subset of program classes (no source code required) and Java libraries. Generally, the more classes that are instrumented, the more accurate but costly the analysis is.

In the remainder of this section we describe how DynFlow was used to implement policy enforcement, signature matching, and anomaly detection. A detailed description of the DIFA algorithm used by DynFlow is presented in [17].

5.1 Information Flow Policy Enforcement

An information flow policy specifies the information flows between particular objects, variables, and methods that are permissible in an application. For example, an information flow policy may indicate that no information may flow from an object representing a credit card account to a socket connection with an untrusted client. DynFlow monitors information flows as they occur and determines if they violate a given policy. The original version of DynFlow [17] was capable of detecting and preventing attacks that violate a specified information flow policy as long as the attack involved illegal flows from sensitive objects to sink methods. For the current study, DynFlow was modified to also handle information flow policies that proscribe illegal flows from particular objects, fields, or variables to other ones. Recognizing that security requirements often change after deployment, DynFlow supports configurable information flow policies.

5.2 Information Flow Signature Matching

The information flows induced by an attack might not violate a specified information flow policy, e.g., because the policy is incomplete or because the attack does not involve any illegal flows. Even so, a subset of the induced flows may be characteristic of the attack, permitting an attack signature to be defined once the attack has been discovered and analyzed. Note that such a signature might comprise multiple flows that must occur in a specific sequence. A simple example of an information flow signature is $x \rightarrow y \rightarrow z$, which denotes an information flow from an object $x$ to an object $y$, followed by a flow from $y$ to object $z$. Another example, from the denial of service attack described in Section 6.4, is $r \rightarrow m$, where $r$ is a request object for a file named $con$ and $m$ is the createFileReply method located in HTTPFrame.java of Jigsaw 2.0.5. DynFlow is capable of recognizing such signatures given their programmatic specifications.

Due to the transitivity of information flows, using them to specify attack signatures provides some protection against evasion schemes that involve padding the sequence of essential operations in an attack with "semantic no-ops" [27] [30] in order to disguise the attack.

Note that an information flow signature is more specific than an information flow policy as it is associated with a particular attack. Also, defining a signature requires a clear understanding of the vulnerability at hand whereas this is not generally the case for a policy which may prevent unknown attacks. As a result of successfully debugging most of the vulnerabilities involved in our experiments, we were able to specify signatures for most of them as presented in Section 8. For each subject program a policy could also be specified by combining, and possibly abstracting, the associated information flow signatures.

5.3 Information Flow Anomaly Detection

Because signature-based intrusion detection systems are capable of detecting only known attacks, anomaly detection techniques [16] have been proposed to enable the detection of novel attacks. These techniques are based on the assumption that attacks often induce unusual execution behavior that can be distinguished from normal behavior. Most research on host-based and application-based anomaly detection techniques involves the analysis of system call sequences (e.g., [9] [16]). Because information flow patterns strongly characterize a program’s computation, information flow profiles are potentially very useful, in conjunction with anomaly detection algorithms, for detecting novel attacks against applications. The offline approach to employing DIFA, described in Section 4, is particularly suitable for use with anomaly detection techniques, because of their high false-positive rates. With this approach, a suspicious execution is audited to confirm that it corresponds to an attack before any action is taken.

The particular anomaly detection technique we are currently employing in conjunction with DynFlow is cluster filtering, which was originally proposed for use in observation-based testing of software for non-security defects [5] [6]. Cluster filtering is based on automatic cluster analysis, which is a multivariate analysis
method for finding groups or clusters in a population of objects. Cluster analysis algorithms use a dissimilarity metric such as Euclidean distance or Manhattan distance to partition the population into clusters. Objects placed together in a cluster are more similar to one another than to objects in other clusters. Cluster filtering uses cluster analysis to partition a set of executions into clusters based on the dissimilarity of their profiles. One or more executions are selected from each cluster and audited.

A cluster filtering procedure is defined by a choice of clustering algorithm, dissimilarity metric, cluster count, and sampling method. Our technique uses the agglomerative hierarchical clustering algorithm and the proportional-binary dissimilarity metric [5] to cluster executions. The sampling methods used are one-per-cluster (OPC) sampling [5] and failure-pursuit (FP) sampling [6]. OPC sampling calls for selecting exactly one execution from each cluster. It exercises each program behavior represented by a cluster, and it also favors the selection of unusual executions, which tend to be placed in isolated clusters.

FP sampling is an adaptive extension of cluster filtering that is based on the observation that executions that are attacks are often clustered together in small clusters. FP calls for selecting the k nearest neighbors of any attacks found by auditing the initial subset of executions. If any additional attacks are found, each of their k nearest neighbors is selected, and so on, until no additional attacks are found. (In the experiments reported in Section 6, k = 5 is used.)

6. EMPIRICAL RESULTS

To empirically verify the efficiency of our new intrusion detection techniques, we applied them to the following Java subject programs that exhibit documented vulnerabilities: Apache Tomcat 3.0 and 3.2.1 [29], JBoss 3.2.1 [10] and Jigsaw 2.0.5 [13]. Section 8 describes the vulnerabilities exhibited by these subject programs. It also presents information flow signatures for most of the attacks exploiting them. Such signatures, which were specified after the vulnerabilities were diagnosed and analyzed, could be combined and possibly abstracted to be included in information flow policies, although we do not do so here.

To assess how efficiently our anomaly-based techniques, one-per-cluster sampling and failure pursuit sampling, revealed vulnerabilities, we compared them to each other and to simple random sampling with respect to how many vulnerabilities each technique revealed for given numbers of tests selected. In the case of OPC sampling, our experiments involved the following steps for every subject program: (1) instrumenting the program using DynFlow; (2) submitting several hundred client requests that include a number of attacks; (3) collecting the information flow profiles; (4) choosing the number of clusters c; (5) clustering the executions based on their information flow profiles; (6) randomly selecting a single execution each of the k clusters; (7) auditing the selected executions; and (8) recording the number of attacks and vulnerabilities revealed. The process of selecting the executions was replicated 1000 times and the results are averaged, which explains why the results presented include fractions. To assess its effect on attack detection, the cluster count c was varied to correspond to the following percentages of the total number of executions: 1%, 2.5%, 5%, 10%, 25%, and 30%. In the case of FP sampling, steps (1)-(8) above were followed by the selection of additional executions as described in Section 5.3. Note that in order to prevent interference between client requests to any subject program, a new execution of the program was initiated for each submitted request. This means that the profile of one request represents the information flows that occurred during program initialization, request processing, and termination.

In the following sections we show that the attacks launched against the subject programs are detectable using one or more of our techniques.

6.1 Case Study: Tomcat 3.0

Apache Tomcat is an open source Servlet/JSP container [29]. Version 3.0 of Tomcat, running under Windows XP and jdk1.4, exhibits the five vulnerabilities described in Section 8.1: Vul-1 involves JSP source code disclosure; Vul-2 involves directory listing disclosure; Vul-3 involves illegal file access; Vul-4 and Vul-5 involve JSP-engine denial of service.

Exploits of Vul-1, Vul-2 and Vul-3 can be detected by matching the information flow signatures specified in Section 8.1. Note that an information flow policy could be specified by combining and generalizing those signatures. An example for such policy is one that in addition to disallowing requests containing null bytes or.jsp extensions in some contexts also disallows requests for any file not residing in some user specified directories.

Following our analysis of the Tomcat code we were not able to identify flow signatures for the attacks exploiting Vul-4 and Vul-5. But as we show next, Vul-4 and Vul-5 are efficiently detectable offline using information flow anomaly detection.
We instrumented the Tomcat 3.0 server and submitted 510 requests to it that included:

1) 190 normal Servlet requests, 190 normal JSP requests and 110 normal HTML and plain text requests.
2) 2 requests exploiting Vul-3, 3 requests exploiting Vul-5, and 5 requests exploiting each of the remaining 3 vulnerabilities.

Figure 1 shows a multidimensional scaling (MDS) display of information flow profiles induced by these requests. Each point in the display represents the information flow profile of one request. The distance between each pair of points approximates a measure of the dissimilarity between the corresponding profiles. The top leftmost rectangle encloses the profiles of the two attacks exploiting Vul-3. The top rightmost rectangle encloses the eight attacks exploiting Vul-4 or Vul-5 that caused Tomcat to crash. The isolated position of these attacks suggests that cluster filtering is likely to reveal them. The bottom right and left polygons enclose the attacks exploiting Vul-1 and Vul-2, respectively. The proximity of the latter attacks to normal executions suggests that cluster filtering is likely to reveal them only when a relatively fine-grained clustering is used.

Figure 2 shows the percentages of all vulnerabilities revealed by OPC sampling, FP sampling, and simple random sampling, respectively, as functions of the number of requests selected. This figure shows that for Tomcat 3.0, information flow anomaly detection techniques were considerably more efficient in revealing vulnerabilities than simple random sampling. For example, by selecting 75 requests, 100% of the vulnerabilities are detected using FP sampling, whereas only 45% are detected using simple random sampling. Also, by selecting 98 requests, 100% of the vulnerabilities are detected using FP sampling, whereas only 55% are detected using simple random sampling.

In summary, for Tomcat 3.0, all five vulnerabilities are detectable using information flow anomaly detection, with Vul-1 and Vul-2 requiring a relatively high number of selected tests. In addition, Vul-1, Vul-2 and Vul-3 are also detectable by matching information flow signatures.

6.2 Case Study: Tomcat 3.2.1
Version 3.2.1 of Apache Tomcat exhibits two vulnerabilities under Windows XP and jdk1.4.2, which are described in Section 8.2. Vul-1 is a JSP source-code disclosure vulnerability and exploits of it are detectable by matching the information flow signatures described in Section 8.2. Vul-2 is a JSP-engine denial-of-service vulnerability. Exploits of both vulnerabilities are efficiently detectable using our information flow anomaly detection as shown next.

We instrumented the Tomcat 3.2.1 server and submitted 500 requests to it that included:

1) 200 normal Servlet requests, 200 normal JSP requests and 90 normal HTML and plain text requests.
2) 5 requests exploiting Vul-1 and another 5 requests exploiting Vul-2.

Figure 3 shows an MDS display of information flow profiles induced by these requests. The top rectangle encloses the five attacks exploiting Vul-1 and the bottom polygon encloses the five attacks exploiting Vul-2. The latter are extreme outliers; the former are less extreme but are still outliers. This suggests that cluster filtering is likely to be effective if sufficient clusters are employed.

Figure 4 shows the percentages of all vulnerabilities revealed by one-per-cluster sampling and failure pursuit sampling, respectively, as functions of the number of requests selected. This figure shows that for Tomcat 3.2.1, our information flow anomaly detection techniques were considerably more efficient for revealing vulnerabilities than simple random sampling. Specifically, by selecting around 30 requests, 100% of the vulnerabilities are detected using OPC sampling and FP sampling, whereas only 25% are detected using simple random sampling.
In summary, for Tomcat 3.2.1, exploits of both vulnerabilities are efficiently detectable using one-per-cluster sampling and failure pursuit sampling. In addition, exploits of Vul-1 are detectable using our signature-based technique.

### 6.3 Case Study: JBoss 3.2.1

JBoss is an open source J2EE-based application server [10]; it uses Jetty [12] as its default Servlet/JSP container. Version 3.2.1 of JBoss, running under Windows XP and jdk1.4, exhibits the four vulnerabilities described in Section 8.3: Vul-1 involves directory/file disclosure; Vul-2 involves JSP source code disclosure; and Vul-3 and Vul-4 involve JSP-engine denial of service.

Exploits of Vul-1 and Vul-2 are detectable by matching the information flow signatures described in Section 8.3. We were not able to identify flow signatures for the denial of service attacks exploiting Vul-3 and Vul-4. But as we show next, Vul-3 and Vul-4 are efficiently detectable offline using information flow anomaly detection.

Note that in addition to instrumenting JBoss we also instrumented its Jetty component since three of the four exploits involve JSP requests. JBoss has a built in test suite framework that we used to submit 500 requests to it that included the following:

1) 200 normal Servlet requests, 200 normal JSP requests and 85 normal HTML and plain text requests.
2) 6 requests exploiting Vul-2, 5 requests exploiting Vul-1, 2 requests exploiting Vul-3 and another 2 requests
exploiting Vul-4.

Figure 5 shows an MDS display of information flow profiles induced by these requests. The two leftmost rectangles enclose the profiles of the four attacks exploiting Vul-3 and Vul-4. The bottom right rectangle encloses the profiles of the five attacks exploiting Vul-1. Since these attacks are all outliers, it is likely that cluster filtering will reveal them.

Figure 6 shows the percentages of all vulnerabilities revealed by one-per-cluster sampling and failure pursuit sampling, respectively, as functions of the number of requests selected. This figure shows that for JBoss 3.2.1, our information flow anomaly detection techniques were considerably more efficient in revealing vulnerabilities than simple random sampling. For example, by selecting 40 requests, 100% of the vulnerabilities were detected using OPC sampling, whereas only 25% were detected using simple random sampling. Also, by selecting 53 requests, 100% of the vulnerabilities were detected using FP sampling, whereas only 30% were detected using simple random sampling.

6.4 Case Study: Jigsaw 2.0.5

Jigsaw is an open source Java-based web server and Servlet container [13]. Version 2.0.5 of Jigsaw, running under Windows XP and jdk1.4, exhibits the four vulnerabilities described in Section 8.4: Vul-1 involves denial of service; Vul-2 involves path disclosure; Vul-3 involves directory-listing disclosure; and Vul-4 involves illegal file access. One of several JSP containers could be plugged in Jigsaw in order to service JSP requests, we chose GNUJSP 1.0.1 for our experiments. All four vulnerabilities are detectable by matching information flow signatures and by information flow anomaly detection as shown next.

When instrumenting Jigsaw 2.0.5, we opted not instrument the GNUJSP component, since none of the exploits involve JSP requests. We submitted 500 requests to Jigsaw that included:

3) 200 normal Servlet requests, 200 normal JSP requests and 90 normal HTML and plain text requests.

4) 2 requests exploiting Vul-1, 2 requests exploiting Vul-2, 3 requests exploiting Vul-3 and another 3 requests exploiting Vul-4.

Figure 7 shows an MDS display of information flow profiles induced by these requests. The two rightmost rectangles enclose the four attacks exploiting Vul-1 and Vul-2. The bottom rectangle encloses the three attacks exploiting Vul-3. The top leftmost rectangle encloses the three attacks exploiting Vul-4. All of these attacks are clearly outliers, so it is likely that cluster filtering will be effective for revealing them.

Figure 8 shows the percentages of all vulnerabilities revealed by OPC sampling and FP sampling, respectively, as functions of the number of requests selected. This figure shows that for Jigsaw 2.0.5, our information flow anomaly detection techniques were considerably more efficient in revealing vulnerabilities than simple random sampling. For example, by selecting 24 requests, 100% of the vulnerabilities are detected using FP, whereas only 10% are detected using simple random sampling. Also, by selecting 40 requests, 100% of the vulnerabilities are detected using OPC, whereas only 20% are detected using simple random sampling.

Comment [AP15]: Were they actually detected in experiments?
7. CONCLUSIONS

A new approach to detecting attacks against software was presented that employs the results of fine-grained dynamic information flow analysis in three ways to reveal (1) violations of information flow policies, (2) known attacks with information flow signatures, and (3) novel attacks resulting in anomalous patterns of information flows. A prototype tool implementing the approach was described, and the utility of the approach was illustrated by using the tool to reveal actual vulnerabilities in several open source programs. Further empirical studies with additional subject programs will be required to assess to generality of the approach.

We conclude with a word about the overhead of DIFA. Table 1 contrasts the execution times of the original programs and their instrumented counterparts. For the three programs for which we were able to measure the slowdown due to DIFA, it was no more than a factor of 3.67.

8. APPENDIX A

8.1 Tomcat 3.0 Vulnerabilities

Version 3.0 of Tomcat, running under Windows XP and jdk1.4.2, exhibits the following five vulnerabilities:

Vul-1) JSP Source Code Disclosure

Source- securitytracker.com/alerts/2001/Mar/1001207.html

Exploit- An attacker can view the JSP source code by sending a JSP request containing at least one uppercase letter in the .jsp extension. For example, in order to view the source code for the cal1.jsp, the following request can be used, respectively:

GET /examples/jsp/cal/cal1.jsp HTTP/1.0
GET /examples/jsp/cal/cal1.Jsp HTTP/1.0

Discussion- A file such as cal1.jsp is not recognized as a JSP file, therefore it will be directed to the DefaultServlet class to be served as a plain text file.

Signature- .jsp file passed as a parameter to the processFile(File ...) method in DefaultServlet.java (line 145). That is, SensitiveSources = {files with .jsp extension} and Sinks = {the processFile method located in org.apache.tomcat.core.DefaultServlet class}.

Vul-2) Directory Listing Disclosure

Source- SecurityFocus Vulnerability Database [23], id# 6721.

Exploit- An attacker can view directory contents by sending a JSP request containing null bytes (%00). For example, in order to view the directory listing for C:\tomcat-3.0 and C:\, the following requests can be used, respectively:

GET /%00.jsp HTTP/1.0
GET /%2e%2e/%00/file.jsp HTTP/1.0

Discussion- Since the string %2e%2e%00/file.jsp ends with .jsp the request is rightfully directed to the JSP component (Jasper), but because of the presence of the null byte (%00) the string actually represents /../windows/win.ini.

Signature- This attack is preventable by using the same information flow signature defined for Vul-2.

Vul-3) Illegal File Access

Source- www.securiteam.com/securitynews/5HP110A3PA.html

Exploit- An attacker can view any file on the server by sending a malformed JSP request containing null bytes (%00). For example, in order to view the content of c:\winnt\win.ini the following request can be used:

GET /../windows/win.ini%00/examples/jsp/junk.jsp HTTP/1.0

Discussion- Since the string /../windows/win.ini%00/examples/jsp/junk.jsp ends with .jsp the request is directed to the JSP component (Jasper), but because of the presence of the null byte (%00) the string actually represents /../windows/win.ini.

Signature- This attack is preventable by using the same information flow signature defined for Vul-2.

Vul-4) JSP Engine Denial of Service

Source- SecurityFocus Vulnerability Database [23], id# 4995.

Exploit- An attacker can crash Tomcat by requesting a JSP containing the following snippet of code:

new WPrinterJob().pageSetup(null, null);

Signature- No signature was identified. This attack is efficiently detectable using our anomaly-based technique.

Vul-5) JSP Engine Denial of Service

Source- SecurityFocus Vulnerability Database, id# 7109.

Exploit- An attacker can crash Tomcat by requesting a JSP containing the following snippet of code:

new CRC32().update(new byte[0], 4, Integer.MAX_VALUE-3);

Signature- No signature was identified. This attack is efficiently detectable using our anomaly-based technique.

8.2 Tomcat 3.2.1 Vulnerabilities

Version 3.2.1 Tomcat exhibits the following two vulnerabilities under Windows XP and jdk1.4.2:

Vul-1) JSP Source Code Disclosure

Source- Open Source Vulnerability Database [19], id# 5580.

---

### Table 1 – Average slowdown. T1 and T2 are the average times in milliseconds taken by a request using the original and instrumented programs respectively.

<table>
<thead>
<tr>
<th>Programs</th>
<th>T1</th>
<th>T2</th>
<th>T2/T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomcat 3.0</td>
<td>362</td>
<td>796</td>
<td>2.2</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>415</td>
<td>1524</td>
<td>3.67</td>
</tr>
<tr>
<td>JBoss 3.2.1*</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Jigsaw 2.0.5</td>
<td>347</td>
<td>904</td>
<td>2.6</td>
</tr>
</tbody>
</table>

*We were unable to accurately measure the slowdown for JBoss due to our use of its test suite framework containing '%00'; and Sinks = {the getResourceAsStream method invoked in the org.apache.jasper.compiler.JspReader class}.

---

Comment [WM18]: Let’s hope we get some nice reviewers

Comment [WM16]: of was missing

Comment [AP17]: Perhaps this should be placed elsewhere

Comment [AP19]: SensitiveSources and Sinks are not currently discussed in the body of the paper.
exhibits the following four vulnerabilities:

**Vul-1)**

**Version 3.2.1 of JBoss, running under Windows XP and jdk1.4,**

**Vul-2)**

requests can be used respectively:

GET /examples/jsp/cal/cal1.jsp

GET /examples/jsp/check/result.jsp

**Discussion-** The erroneous behavior of this version of Tomcat is summarized as follows: if the string following the GET is recognized as a file then Tomcat serves the file as plain text using the org.apache.tomcat.request.FileHandler class. Appending the HTTP protocol will understandably change this behavior. As a matter of fact appending any ‘meaningless’ substring to the above requests will cause Tomcat to behave normally, i.e. treat them as JSP requests.

**Signature-** .jsp file passed as a parameter to the java.io.FileInputStream constructor in FileHandler.java (line 365). That is, SensitiveSources = {files with .jsp extension} and Sinks = {the FileInputSteam constructor located in the org.apache.tomcat.request.FileHandler class].

**Vul-3)** JSP Engine Denial of Service (same as Vul-5 of §8.1)

**Vul-4)** JSP Engine Denial of Service (same as Vul-6 of §8.1)

8.3 JBoss 3.2.1 Vulnerabilities

Version 3.2.1 of JBoss, running under Windows XP and jdk1.4, exhibits the following four vulnerabilities:

**Vul-1)** Directory/File Disclosure

Source- www.illegalacess.org

Exploit- An attacker can view the content of any directory or file under the directory in which JBoss was started by sending a malformed request. For example, if JBoss was started in C:, an attacker can view the content of C:\windows\win.ini by submitting the following request:

GET /bosstests/%/windows/win.ini

**Discussion-** JBoss comes with a built in mini server defined by the org.jboss.web.WebServer class. WebServer listens on port 8083 and is meant to serve the byte code of classes loaded by a given class loader. When no class loader is specified, the thread context class loader is used which allows for such illegal access to resources and files. Also, in WebServer, data is sent to the client via the write methods of an object of type java.io.DataOutputStream named out.

**Signature-** Secure resource or file flowing to any of the write methods of out. That is, SensitiveSources = {secure resources or files} and Sinks = {the write methods of out invoked in the org.jboss.web.WebServer class].

**Vul-2)** JSP Source Code Disclosure

Source- securitytracker.com/alerts/2001/Mar/1001207.html

Exploit- An attacker can view the JSP source code by sending a JSP request containing at least one upper case letter in the .jsp extension. This vulnerability is identical in its exploit and symptoms to Tomcat’s 3.0 Vul-1.

**Discussion-** This vulnerability resides in the Jetty component. It is the result of method org.mortbay.util.StringMap to perform string matching and that StringMap contains a flag called _ignoreCase which is set to false by default. A fix to this vulnerability simply involves setting the _ignoreCase flag to true.

**Signature-** .jsp file passed as the resource parameter of the handleGet() method in Default.java (line 191). That is, SensitiveSources = {files with .jsp extension} and Sinks = {the handleGet method located in the org.mortbay.jetty.servlet.Default class].

**Vul-3)** JSP Engine Denial of Service (same as Vul-5 of §8.1)

**Vul-4)** JSP Engine Denial of Service (same as Vul-6 of §8.1)

8.4 Jigsaw 2.0.5 Vulnerabilities

Version 2.0.5 of Jigsaw, running under Windows XP and jdk1.4, exhibits the following four vulnerabilities:

**Vul-1)** Denial of Service

Source- www.securityspace.com/smysecure/catid.html?id=11047

Exploit- An attacker can submit multiple requests that will never time out, thus resulting in denial of service. The following is a known example of such requests:

GET /con HTTP/1.0

**Discussion-** In response to the above request Jigsaw instantiates a java.io.FileInputStream object associated with con which is a DOS device. When trying to read from the instantiated object Jigsaw blocks indefinitely.

**Signature-** Request for a file named “con” passed as a parameter to the createFileReply(Request request) method in HTTPFrame.java (line 1641). That is, SensitiveSources = {request object for a file named con} and Sinks = {the createFileReply method located in the org.w3c.jigsaw.frames.HTTPFrame class].

**Vul-2)** Path Disclosure


Exploit- An attacker can learn the installation path of Jigsaw by submitting the following request twice:

GET /aux HTTP/1.0

**Discussion-** The first “GET /aux HTTP/1.0” request behaves exactly like Vul-1, i.e. the request blocks indefinitely trying to read from the aux DOS device. During the second request, the attempt to instantiate a second FileInputStream associated with aux throws a java.io.IOException which is caught and handled by sending a reply to the client that discloses the installation path of Jigsaw.

**Signature-** A signature similar to the one defined for Vul-1 can be used to prevent any of the two requests from succeeding and consequently the whole attack.

**Vul-3)** Directory Listing Disclosure

Source- Undocumented

Exploit- An attacker can view the content of any directory by sending a malformed request. For example, in order to view the directory listing for C:, the following request can be used:

GET /%SC/ HTTP/1.0

Comment [WM20]: We’ve been using ‘Section’ so far
GET /servlet/HelloWorld.class%00 HTTP/1.0

Vul-4) Illegal File Access
Source- Undocumented
Exploit- The following request will return the content of the servlet (byte code) file as opposed to its output:
GET /servlet/HelloWorld.class%00 HTTP/1.0

Vul-4) Illegal File Access
Source- Undocumented
Exploit- The following request will return the content of the servlet (byte code) file as opposed to its output:
GET /servlet/HelloWorld.class%00 HTTP/1.0

9. REFERENCES
[31] Zimmermann, J., Mé, L., and Bidan, C. An improved reference flow control model for policy-based intrusion detection. 8th European Symposium on Research in Computer Security (Gjøvik, Norway, October, 2003), Lecture Notes in Computer Science 2808, Springer-Verlag.