11-2014

POSTER: Protecting Against Data Exfiltration Insider Attacks Through Application Programs

Asmaa Mohamed Sallama
Purdue University

Elisa Bertino
Purdue University, bertino@cs.purdue.edu

Sallama, Asmaa Mohamed and Bertino, Elisa, "POSTER: Protecting Against Data Exfiltration Insider Attacks Through Application Programs" (2014). *Cyber Center Publications.* Paper 623.
http://dx.doi.org/10.1145/2660267.2662384

POSTER: Protecting Against Data Exfiltration Insider Attacks Through Application Programs

Asmaa Sallam, Elisa Bertino
Computer Science Department
Purdue University
West Lafayette, IN 47906
asallam, bertino@purdue.edu

ABSTRACT

In this paper, we describe a system that distinguishes between legitimate and malicious database transactions performed by application programs. Our system is particularly useful for protecting against code-modification attacks performed by insiders who have access to and can change the programs' source code to make them execute different queries than those they are expected to execute. Our system works with any type of DBMS and requires minimum modification to application programs.

Categories and Subject Descriptors

H.2.m [**Information Systems**]: DATABASE MANAGEMENT—*Miscellaneous*

1. INTRODUCTION

Organizations ranging from government agencies (e.g., military, judiciary, etc.) and contractors, to commercial enterprises and research labs are witnessing an increasing amount of sophisticated insider attacks that are difficult to mitigate with existing security mechanisms and controls. Insider threats are staged either by disgruntled employees, or by employees engaged in malicious activities such as espionage. One of the most important objectives of insiders is to exfiltrate sensitive data.

Protection against data exfiltration from insiders requires combining different techniques [1]. One important technique is represented by anomaly detection tools which create data-access profiles of normal transactions; accesses to the database are monitored and checked upon these profiles to detect anomalous accesses [4]. Some access patterns may be indicative of insider attackers on a mission to steal data. For example, consider a clerk who, for his/her daily activity, only needs to access 10% of the records in a table in the database. An access by this clerk that retrieves all records from this table is certainly anomalous. We refer the reader to [2] for a discussion on anomalous access patterns.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-2957-6/14/11.
http://dx.doi.org/10.1145/2660267.2662384 .

A major limitation of previous work on anomaly detection for databases is to assume that queries are issued directly by users. However, database queries are not necessarily only issued by individuals but can also be issued by application programs. A malicious insider, such as a software engineer within the organization with the authorization to modify the application programs, can exfiltrate data from a database by modifying the source code of the program so that the program issues queries different from those it normally sends. One possible approach to address this problem would be to create profiles of queries issued by application programs and compare the actual accesses by the programs against these profiles. Current commercial Security Information and Event Management (SIEM) tools, that are able to capture and log queries issued by application programs together with relevant metadata, can be used for this purpose. However, creating complete and accurate profiles is a challenge in the case of application programs since programs issue different queries depending on input parameters and context information. Recording and saving all legitimate sequences of queries is expensive in the case of large programs, large input space, and high number of input parameters.

In this paper, we propose a different approach for supporting profiling and anomaly detection for application programs. During the profile-creation phase, our approach uses software testing techniques to build a profile for the program in which the control structure of the program and locations in the code where SQL queries are issued are recorded. During the detection phase, the system uses this profile and the input values to compose the query strings that are expected to be issued by the program. These expected queries are then compared to the actual ones sent by the application program to the DataBase Management System (DBMS) and differences are considered anomalous.

The rest of this paper is organized as follows. Section 2 defines relevant notions from the software testing area. Section 3 describes the architecture, algorithms and implementation of the proposed system. Section 4 concludes the paper.

2. PRELIMINARIES

The **Symbolic Execution** technique [3] is a program analysis technique that uses symbolic values as program inputs and symbolic expressions to represent the values of program variables. The **symbolic execution tree** is the tree representation of all the possible paths of the program; a node in the tree stores the program variables as a function of the symbolic inputs and the **path constraint (PC)** which

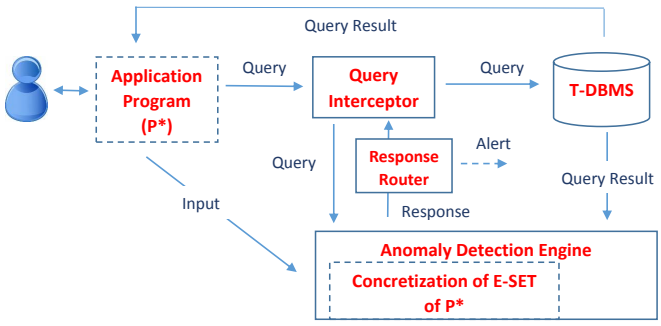

Figure 1: System Architecture

is a conjunction of conditions on the input to follow a path in the program. Symbolic execution is being used for software testing [3] [5].

Software Instrumentation refers to adding additional code to a program for monitoring some program behaviour. Instrumentation can be done either statically (i.e., at compile time) or dynamically (i.e., at runtime).

The **Backward Data Slice** of a variable in a program at a specific statement is the sub-program that affects the value of the variable at that point.

3. PROPOSED SYSTEM

3.1 Architecture

Our system consists of four components: a Target DBMS (T-DBMS), the Anomaly Detection Engine (ADE), the Query Interceptor, and the Response Router. These components interact in order to check the proper sequence of queries sent by an application program. T-DBMS is the DBMS that stores and manages the database to be protected against insider attacks. The ADE stores the application program profile and performs the anomaly detection task. The Query Interceptor intercepts the query, before it is sent to the T-DBMS, and forwards it to the ADE for anomaly detection. The Response Router checks the response policies in order to take appropriate response to anomalies detected by the ADE. Note that this architecture is designed so that no restrictions are imposed on the T-DBMS as a result of adding the anomaly detection functionality since the T-DBMS always receives SQL queries and responds with their result set which is the normal operation of a DBMS.

3.2 Phases of Operation

The proposed system operates in two phases: Profile-Creation phase and Detection phase. Details on each phase are given in what follows.

3.2.1 Profile-Creation phase

In this phase, the binary of the application program is given as input to a **profiler** that analyzes the program statically. The profiler first finds statements in the program that issue SQL queries to the DBMS. It then computes the combined backward data slices of the variables used to compose the query strings. The result of this operation is a sub-program of the original one for which the profiler constructs and outputs a variation of the Symbolic Execution Tree which we refer to as **Extended Symbolic Execution**

```
0.   input char: x
1.   input integer: y        <-- 1.1 Send input to ADE
2.    q1 = "SELECT salary FROM employees WHERE id = " + y
3.    send q1 to the DBMS
4.    salary  = extract salary from resultset of q1
5.    if (salary > 1000)
6.       salary = salary + 200
7.    else
8.       salary = salary * 1.2
9.    q2 = "UPDATE employees SET salary = " + salary
                          + "WHERE id = " + y ;
10.   send q2 to the DBMS
11.    ....
```

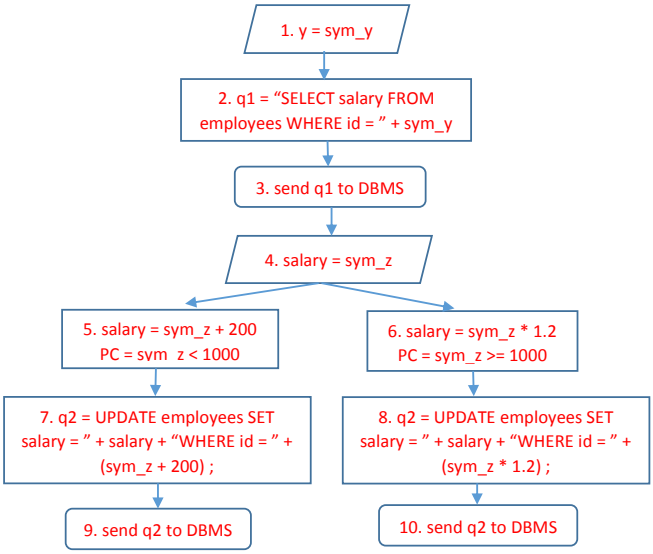

Figure 2: Example

```
1.  Monitor () {
2.    if (currNode.type == 'Wait-For-Query') {
3.      Expected = 'Q';
4.      ExpectedQuery
            = getExpectedQueryString(currNode);
5.      Wait();
6.    } else if (currNode.type == 'Wait-For-Input') {
7.      Expected = 'I'
8.      Wait();
9.      Save received input to correct variables;
10.   } else if (currNode.type == 'Computation') {
11.     Perform Computations in currNode;
12.   }
13.   currNode = currNode.next;
14.   Monitor();
15. }
```

```
1.  Signal() {
2.    if (Expected == 'Q' && receivedType = 'Q') {
3.      if(ExpectedQuery == receivedQuery) {
4.        return BENIGN;
5.      }
6.    } else if (Expected == 'I' && receivedType = 'I') {
7.      Use received as input;
8.      return BENIGN;
9.    }
10.   RAISE ANOMALY;
11. }
```

Figure 3: Detection Algorithm

Tree (E-SET). The profiler also instruments the program by adding statements to send input values to the ADE; the result is a modified version of the program which will be used in production instead of the original.

Unlike the normal Symbolic Execution Tree, the E-SET differentiates between three types of nodes: Computation, Wait-For-Query, and Wait-For-Input nodes. "Computation" nodes are nodes that contain expressions for variables in the program as functions of inputs. "Wait-For-Query" nodes indicate locations in the execution paths of the application program where input should be provided by the user. "Wait-For-Input" nodes are locations where the program sends queries to the DBMS. Figure 2 shows an example program, and its corresponding E-SET and instrumentation. During Profile-Creation, Statements 3 and 10 are identified by the profiler to be accessing the DBMS. The profiler then computes the backward data slices of the query strings: q1 and q2. The resulting sub-program (P') will contain all statements except 0 and 11. Since Statement 1 has a user input, a new statement (1.1) is added to P' to compose program P* that will be run by users.

3.2.2 *Detection Phase / Concrete Execution of the program*

At program run-time, whenever the program opens a new connection to the DBMS during a user session, the Query Interceptor, which is listening on the communication line between the DBMS and the program, notifies the ADE of the new connection. As a result, the ADE creates a new process that would be responsible for any further communication between the ADE and the Query Interceptor. Based on the user-input and the E-SET of the program, the process will know the path the program P* should follow and queries expected to be issued by the program. This operation is referred to as **concretizing** the Symbolic Execution Tree. The ADE compares queries actually sent by the program and those it generated as explained next.

The newly-created process runs the algorithms in Figure 3. It starts by setting the variable *currNode* to the root node of the E-SET and then calls the function *Monitor()*. *Monitor()* checks the type of node *currNode* is pointing to. If it is 'Wait-For-Query' or 'Wait-For-Input', the process sleeps waiting for external input from either the Query Interceptor or T-DBMS (lines 2:9). Otherwise (the node type should be a 'computation'), the process performs the computations indicated in the node, moves *currNode* to the next node in the tree and calls *Monitor()* again (lines 11, 13, 14).

The function *Signal()* is called when the process receives an external input. It checks that the type of input it is expecting is the same as what it received (lines 2 and 6). In case the process is waiting for a query, the query string it is expecting is compared to the one it received too; if they are similar, the process returns from *Signal()* (lines 2:5) and continues processing nodes. All other scenarios are rejected and an anomaly is raised. Note that additional synchronization between P* and the ADE process has to be performed for the algorithm to work properly. For instance, in case P* is done with a computation which has not yet been finished by the ADE process, input can be sent to the ADE process while it is not waiting for it; *Signal()* then has to check that P* is ahead of it and choose to defer the processing of the input accordingly.

3.3 Implementation

We have developed an initial solution based on tools which perform Symbolic Execution of the program. JPF[5] and CUTE[6] are well-known tools whose source codes are available for modification.

An important implementation issue to mention is that the program flow and values of variables may depend on the result sets of SQL queries as in line 4 in the example code in Figure 2. In our solution, we consider the result as input. However, this approach is problematic if the result set of a query is large and therefore needs long time to be processed. One solution to this problem, that we are currently investigating, is to instrument the program to directly send values of some variables to the ADE so that the ADE does not need to perform all the computation.

Another important issue concerns securing all the anomaly detection system components, such as the ADE and the Query Interceptor, as well as all the communications between these components and the T-DBMS and application programs. Currently available security tools can be combined and deployed to address this issue.

4. CONCLUSION

In this paper, we presented a system for protecting against data exfiltration attacks based on source-code modification. As part of future work, we will extend our system along several directions. For example, as currently the system only deals with desktop applications, we will investigate its application to web-based ones. We also believe that the technique described can be used for tracking user behaviour at the OS level and therefore it can be used for collecting and using data provenance. The idea of integrating the Profile-Creation phase with a testing technique that uses Symbolic Execution of the program is also another direction of future work.

5. REFERENCES

[1] E. Bertino. *Data Protection from Insider Threats. Synthesis Lectures on Data Management*. Morgan and Claypool Publishers, 2012.

[2] E. Bertino and G. Ghinita. Towards mechanisms for detection and prevention of data exfiltration by insiders: keynote talk paper. *ASIACCS*, pages 10–19, 2011.

[3] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *PLDI*, 2005.

[4] A. Kamra, E. Terzi, and E. Bertino. Detecting anomalous access patterns in relational databases. *VLDB*, 2008.

[5] C. S. Pasareanu, W. Visse, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis. *ASE*, 2010.

[6] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. *ESEC-FSE*, 2005.