Instrumentation of Open-Source Software For Intrusion Detection

William Mahoney
University of Nebraska at Omaha
282F PKI, 6001 Dodge Street
Omaha, Nebraska 68182-0500
1-402-554-3975
wmahoney@unomaha.edu

William L. Sousan
Computer Science Department
University of Nebraska at Omaha
Omaha, Nebraska
1-402-554-2423
wsousan@unomaha.edu

ABSTRACT

A significant number of cyber assaults are attempted against open source internet support software written in C, C++, or Java. Examples of these software packages include the Apache web server, open source DHCP servers, and network share software such as Samba. These attacks attempt to take advantage of inadvertent flaws left in software systems due to a lack of complete testing, inexperienced developers, intentional backdoors into the system, and other reasons. Detecting all of the flaws in a large system is still a daunting, unrealistic task. If it is not possible to completely secure a system, there is a desire to at least detect intrusion attempts in order to stop them while in progress, or repair the damage at a later date.

The information assurance area of expertise known as "intrusion detection" attempts to sense unauthorized attempts to obtain access to or manipulate information, or to deny the information to other legitimate users. There are several traditional methods used for intrusion detection, which can be categorized into two broad classes: Anomaly Detection, and Misuse Detection.

Anomaly Detection uses statistical approaches and pattern prediction techniques to generate profiles of "typical" user interaction with a system. For example, a certain percentage of the page accesses on a web site may be to a log-in page, and a certain percentage may refer to a page showing the users "shopping cart". Occasionally the user will mistype their password and the log in will fail; for this and other reasons it is likely that more references would be made to the login page than the shopping cart page. If, though, certain pages are suddenly referenced far more frequently, this is an unusual activity and may indicate an intrusion attempt. The advantages of this technique include the capability to detect intrusions which other methods miss, and the fact that the systems are generally adaptable to change over time. But anomaly detection via statistical approaches suffers from a few drawbacks. For example, a nefarious user who knows that the system is adaptable can gradually change the probability for future events until the behavior is considered to be normal. At that point the attacker can penetrate the system without triggering any of the detection alarms. As a counter to these approaches, many anomaly intrusion detection systems also incorporate some form of neural network, which learns to predict a user's next activity and signals an alarm when this prediction is not met.

Misuse Detection systems are typified by expert system software which has knowledge of many known attack scenarios and can monitor user behavior searching for these patterns. A misuse detection system can be thought of as more similar to anti-virus software, which continually searches files and memory for known attack patterns, and alerts the user if any are matched. Misuse systems include a state-based component called an "anticipator", which tries to predict the next activity that will occur on a system. A knowledge base contains the scenarios which the expert system uses to make this prediction, and the audit trail in the system is examined by the expert system to locate partial matches to these patterns. A wildly differing "next event" in a pattern could be an indicator that an intrusion attempt is in progress.

Both types of intrusion detection systems can rely on a variety of data sources in order to build an accurate picture of the normal versus abnormal system activity. However these data sources are almost exclusively comprised of two types: network traffic, and audit logs [1].

This research presents a new approach to generating records for intrusion detection by means of enhancements to the GCC compiler suite. These modifications automatically insert instrumentation calls into the compiled code; the intent of the instrumentation is to generate trace data for intrusion detection systems. Open source code such as a web server can be compiled in this manner, and the execution path of the server can be observed externally in near real-time. (We claim only "near" real-time since the instrumentation is typically queued for a short period between the producing instrumented program and the intrusion detection software.) This method thus creates a completely new source of intrusion detection data which can be incorporated into a detection system.

This "instrumentation compiler" is used for software which is run in a controlled environment in order to gather typical usage patterns. These patterns are ideal for an "anticipator" module in a misuse detection system, as they are made up of the actual execution path of the software under typical usage scenarios. The data included in the instrumentation tracks each procedure entry

and exit point in the software as well as the entry to each basic block in the compiled code.

In a sense the tool appears similar to the Linux utility "gcov" and similar software engineering programs which are used for verifying that each line of code has been executed and tested. However "gcov" and similar tools operate in a batch mode where they first collect statistics, and then later display the program coverage. Our modifications create trace information as each block of the original code is executed. The data generated includes the currently executing function name, the line number in the original software, and the basic block number (for debugging our system) within the function itself. The data could obviously be saved to a file for later analysis, similar to "gcov". But the data is readily available as the program executes and thus can serve as an immediate data feed to our misuse detection system. In addition, our system can change the coverage dynamically during runtime by indicating which functions are to be monitored without restarting the system.

This research paper outlines our intrusion detection scheme and includes two main foci.

First, we discuss the techniques used to modify the internal representations used by the GCC compilers to allow this instrumentation. The compiler uses an internal representation called RTX. Additional calls to the instrumentation functions are automatically generated in RTX just prior to emitting assembly language output. The research paper addresses the techniques for locating the instrumentation points and avoiding problems when software is compiled with optimization. We also present figures addressing the slowdown due to the instrumentation overhead and the additional memory requirements that result by including our instrumentation. The slowdown in compute-bound programs is significant, but our focus is typified by heavily I/O bound processes such as web servers.

Secondly, we have designed and describe a simple a priori domain specific language which we use in order to test for intrusion attempts. Since we are implementing a proof of concept system to determine the feasibility of this method for intrusion detection, our system does not currently encompass any learning modes; instead we manually enter rules based on the past known good observed behavior of the software we are compiling for instrumentation.

Our domain language is a way in which we can specify possible sequences of events which are expected from the instrumentation output, along with the probability of each successor to that event. In this way, potential state transitions create a DFA-like automaton. There is one automaton structure for each possible sequence, and these automatons are traversed in parallel according to the instrumentation output of the program being observed. Final states in the DFA correspond to acceptable sequences of events, while a sufficient number of invalid transitions may be an indicator of an intrusion attempt. Reaching a final state causes all automatons to reset to their initial state. Our language is thus compiled from a human readable format into this set of automatons, which the intrusion detection system then matches against the instrumentation coming from the server program in near real-time.

Our paper lastly outlines the results of this research in general and the issues we have raised but have not yet addressed.

[1] See for example: DARPA Intrusion Detection Evaluation Data Sets, Lincoln Laboratory, Massachusetts Institute of Technology. Available at http://www.ll.mit.edu/IST/ideval/index.html.