IDEA: A New Intrusion Detection Data Source

William Mahoney, William Sousan
Peter Kiewit Institute, University of Nebraska at Omaha
6001 Dodge Street, Omaha, Nebraska 68182-0500
{wmahoney,wsousan}@unomaha.edu

Abstract

In the context of computer systems, an intrusion is generally considered to be a harmful endeavor to prevent others from legitimate use of that system, to obtain data which is not normally available to the intruder, or to plant data or disrupt data already existent on the machines. Traditionally intrusion detection has relied on two data sources: various log files which record user's activity, and network traffic which contains potential threats. This research presents a system which we call IDEA; the Intrusion DEtection Automata system. We utilize a third source of data for intrusion detection in the form of an instrumented process. Open source software is recompiled using a modified compiler we have created, and the resulting executable program generates the data as it runs. An external monitoring facility then checks the behavior of the program against known good execution paths. These paths are specified either using a domain specific language and hand-written rules, or by running the software in a learning mode and capturing the normal behavior for later comparison.

1. Introduction

Intrusion Detection (ID) is an area of computer security which deals with monitoring the events occurring within a computer system in an attempt to determine whether accesses to the system are legitimate or nefarious[13][2][4][15]. The events which are monitored are generally one of two types: system logs which are created by the operating system and applications, and network packets containing addressing and control information[3]. Our research in this area uses an additional source of ID data in the form of process monitoring. While this method has been used in the past, these existing systems base their detection on the system or function call level. Examples of this type of detection systems include work done by Hofmeyr[7], Maniatty[9], Peisert[14], and Locasto and Keromytis[8]. The methods frequently monitor a process by watching the system calls which that process makes, and comparing these to typical sequences of calls.

Our work inserts intrusion detection monitoring into the source code automatically when the system is compiled. It is thus a finer granularity of monitoring and also can monitor a process whether it is currently executing operating system requests or not. Our project is called "IDEA" – the Intrusion DEtection Automata system, and the research makes the following contributions:

- We have added an instrumentation insertion system to the popular GCC compiler suite. Programs compiled in this mode can have instrumentation enabled or disabled at function boundaries and export block by block data as the program runs. The location of our additions makes the tool applicable to any language supported by GCC.
- We have measured the slowdown due to instrumentation and determined that it is reasonable for the types of programs which are typically subject to intrusion attempts (e.g. web servers).

- We have proven that our program instrumentation works in the general case by recompiling and testing several open source projects as well as the SPEC CPU2006 benchmarks[16].
- We have created a domain specific language (DSL) in order to provide specifications for "normal" behavior of the program in question.
- We have created monitoring software which learns typical execution paths as the program is running.
- We have demonstrated both the DSL and the learned patterns using a common open-source browser called "thttpd"[17].

The principle difference between our ID system and other ID systems referenced above lies in the fact that we are monitoring an executing program in real-time at a block level, not at a system call or function call level.

This paper presents the motivations in section two, and brief descriptions of the processes involved in adding instrumentation to open source code, in section three. Section four contains the specifics of how the monitoring facility is constructed from either a set of rules written in a new language, or by learning typical execution paths from the actual software. Our conclusions and notes concerning future work are in the last section, section five.

2. Motivation

One critical aspect of computer security is the proliferation of open-source software. As an example, as of this writing the open Apache web server is running in the neighborhood of 51% of the web servers on the internet[11]. A vulnerability in Apache would have a profound impact on the e-commerce, and thus the economy, of many countries. Apache is but one example; others include open-source code for DHCP servers, NFS, open-source which is subsequently included in embedded platforms such as routers, SAMBA, FTP servers, etc. An attack against any one of these might be possible; if we concede this, than intrusion detection becomes a more critical concern. Thus the motivation for this research is the desire to leverage what might be considered a disadvantage (open-source) and instead use it as an advantage to assist in the detection of malicious users. Our system, IDEA, does this.

At the same time, one can generate either "synthetic normal paths" by running a program in a laboratory environment and executing as many paths as possible through the source code, or alternatively one can operate the program in a real environment and thus capture "real normal paths" of execution. A tool such as "gcov" [6] might be useful for synthetic paths, and the user can capture the runtime data by providing a suitable set of tests against the program. But we consider capturing synthetic paths in this way to be a senseless idea. Consider that we require the open source code for the original program, and we have created a modified compiler. Capturing all possible execution paths is a relatively simple matter – dump the control flow graph out of the compiler on a function-by-function basis. This captures all possible execution flows through the program. But what we wish to note in an ID system is not all paths through the code, but an unusual path through the code. There may be many of these unusual paths, all of which are permissible synthetic paths, but many of which are indicative of strange behavior on the part of the user. It is a considerably better approach to build an ID system based upon paths that are normally seen during the execution of the source code, and to then notify the user on instances where the process varies from this norm.

3. Adding instrumentation to open-source

The IDEA system uses modifications which we have made to the popular GCC compilers. This tool chain was selected because the compiler suite itself is open-source and thus facilitates the necessary modifications. In addition, the alterations were made in such a way that they are source language independent; we have tested web browsers written in C, C++, and Java, as well as a few lingering Fortran programs. The implementation currently runs on Linux, due to the large proportion of open-source code which executes on this platform; the compiler and the remainder of the system could be easily moved to a different architecture. This section outlines the changes which we have made to the GCC collection to support IDEA.

3.1. GCC Modifications

The GCC compiler suite maintains several internal representations for a program as it is compiled. The source language is first parsed and converted over to "generic trees". These tree data structures are then processed into "gimple trees" (the name is a GNU modification of "simple") which are then converted into a representation called RTL. Lastly, the RTL is translated into assembly language which is then passed to a separate program.

The IDEA system operates on the RTL rep-resentation of the program, in a block-by-block manner. Each (basic) block has additional code inserted which, when the RTL is translated to assembly language, creates a call to our instrumentation function. The function signature is:

```
void __cyg_block_enter( void *func, unsigned int line, unsigned int block);
```

Thus, the instrumentation function has ready access to the address of the function currently being executed at run time, as well as the basic block number and the line number for that block. As the line number and block number are at least partially redundant we use only the block number in the IDEA monitoring software but retain the line numbers for debugging purposes.

3.2. Performance Impact

The instrumentation naturally causes a slowdown in the program when it runs. This slowdown is dependent to a large extent on what the supplied instrumentation function actually does; we can set a lower bound on the performance impact by running code which includes the instrumentation calls to a function which does nothing. The slowdown in this case was measured using the CPU2006 integer tests [16] and is outlined in table 1:

Table 1. Instrumentation slowdown

	Time (seconds)		
Benchmark	Original	Instrumentation	Factor
401.bzip2	3890	5720	1.470
403.gcc	2280	3250	1.425
429.mcf	2910	3560	1.223
445.gobmk	2020	3330	1.649
456.hmmer	5650	7500	1.327
458.sjeng	2790	4900	1.756
464.h264ref	5450	6820	1.251
473.astar	2590	3910	1.510
		Average:	1.452

As the table shows, adding instrumentation calls to these benchmarks adds about 45% additional overhead to the process when the instrumentation function performs no activity. This is the lower bound for the slowdown on the benchmark programs. We also note that adding instrumentation to an optimized program causes a greater slowdown, although the optimization still generates a faster program, due to the necessity of saving and restoring registers which are live at the point of the added code.

3.3. Open-source demonstrations

In addition to testing the CPU2006 benchmarks, as a means for verifying that the instrumentation does not modify the program behavior, we have recompiled several open-source projects. In particular we have tested the following web servers, which typify the point of attack for many ID systems: Apache Web Server[1], C source, Fizmez Web Server[2], Java source, Thttpd Web Server[17], C source, and Monkey Web Server[11], C Source.

Because of the overall complexity of the Apache system, for the majority of our research in the IDEA system we have principally tested with the other three servers listed above. Note though that the original source code language is irrelevant as long as it is one of the languages supported by GCC.

4. The IDEA monitor

The run time data generated by the executing process is monitored in real time by an external program. This program alerts the user when the expected run time behavior of the program does not match certain specifications. In order to create these specifications the user writes the rule set using a domain specific language (DSL) which is then converted into the automata used by the IDEA system. Each individual rule is saved as a deterministic finite automata (DFA), and there can be many DFAs used. Alternatively, the system can learn the automata by observing the execution paths of the executing process. In both cases, multiple DFAs are then used by the external process, which observes the original program and makes the intrusion detection determination.

We first briefly describe the language based method for constructing the monitor, then describe how the rules are used to create the monitoring program and how the trace data is enabled and disabled at points within the process. We then describe the method for learning the DFAs from a live environment.

4.1. The domain specific language

The grammar for the DSL, in extended BNF, is as follows:

```
program        -> program rule | ε
rule           -> ‘within’ INT ‘max’ INT function
function       -> '(' IDENTIFIER not_empty ')'
not_empty      -> item { item }
item           -> elist | identifier | INT
elist          -> elist { ‘|’ expression } |  ε
expression     -> '(' not_empty ')' optmod
optmod         -> '+' | '*' | '?' | ε
```

where IDENTIFIER follows the same rules as function names within the source language, and INT is an integer. Informally the grammar denotes rules. Each rule first sets up a “within” time and defines a list of items starting at a function boundary. The optional modifiers are used as in regular expressions; “+” and “*” are positive and Kleene closure, and “?” indicates an optional component. The “|” is used for alternation. Semantically, a rule contains a timing section followed by an execution path section. For example:

```
within 30 max 5
(func1 4 5 6 (func2 (10 20))?)
```

This example states that the function “func1” must be called at least every 30 seconds, and when it is invoked, it must complete within five seconds. The function starts by executing blocks four, five, and six, and then it possibly calls “func2”, executing blocks 10 and 20. Execution of the code within “func2” is optional due to the presence of the “?” following that portion of the rule. Similar operators ‘*’, and ‘+’ behave as they do for regular expressions.

4.2. The monitoring facility

To create the monitoring program requires several steps. First, the rules are compiled into the DFAs. These machines have all of the function names replaced with their addresses in the executable program; this is accomplished by examining the symbol table of the program after it is compiled. The DFAs are then used as the input for further processing. The unique strings, now addresses, in each machine are used to create code for a perfect hash function. The DFA and the hash function are then combined to create state tables which represent the legal transitions specified in the original rule set. The state tables are then sourced into the monitoring facility when it is built. Lastly the monitor process is run simultaneously with the original source program, and it alerts the user if the execution path of the program does not match the specified paths in the original rule set.

4.3. Specifying function traces

The rule set specified in the DSL might conceivably require one to indicate all frequent execution paths through the software. However an ancillary program called “joinPoint”, named after another project dealing with aspect oriented software[9], is used to indicate the starting points and stopping points for the generation of trace data. In this way one can start off small, writing rules for only a few functions, and then enlarge the coverage over time. For an illustration, as the open source program is started, and using the previous rule as an example, one would indicate the start and stop point for the trace data via:

```
./joinPoint ./prog func1 trace
./joinPoint ./prog -func1 notrace
```

The first of these lines turns on the trace at the entry point to “func1” and the second disables trace data at the corresponding exit point. It is necessary to coordinate the rule set with the startup procedure so that the areas traced correspond to the rules used. We typically enable monitoring at “main” and then gather data looking for good candidates for join points. We then write rules based on these candidates and enable the trace facility only where it corresponds to our rules.

4.4. Automatically learning typical behavior

The domain specific language used by the IDEA system was initially set up as a proof of concept method to verify our technique. However, writing the rules, by hand is cumbersome to say the least. It became apparent that the ID system would work, but that it was a labor intensive project to set up all of the rules. Thus a preferred method for setting up the monitoring program is to watch a live system and first build the DFA from the activities of the system, and then to use this machine in the same manner as before. The DFAs constructed from either the DSL or the learning methods are identical, so the monitoring program is similar regardless of which approach is used.

Construction of a DFA by learning proceeds by first running the program with the monitoring data flowing to the learning system. Each executed block sends the function, line, and block number as data. The external process uses the unique combination of function address and block number from this data as the differentiating name of a unique state in the DFA. This new state is added to the machine if it did not already exist. Transitions are tracked from state to state, new states are added as needed, and the DFA is built over time as the program runs. As we noted above, these are not synthetic paths but real normal paths through the code which (assuming a normal workload for the program being monitored) are thus representative of the actual execution of the software being watched.

After a time, the program is stopped. Sensing the end of the data, the monitoring process writes the DFA data and the machines are processed in the same manner as if they were constructed from the DSL. The timings for “within” and “max” are determined by tracking the number of starts of each state machine as well as the total execution time and the total of the duration of each DFA. Settings for “within” are factored up by a reasonable amount, and settings for “max” are not allowed to be less than a certain threshold.

5. Conclusions

This paper has introduced the IDEA Intrusion Detection system. IDEA uses instrumented programs as a new source of data for ID, and we have created a proof-of-concept system using several open-source web applications. Modifications at the RTL level in the GCC compiler have enabled us to automatically add the instrumentation to the compiled program, and the modifications are language independent; we support whatever languages GCC supports. We have created patterns which match up to execution paths within the “thttpd” server, and have monitored the application. Furthermore, we have created simulated intrusions and verified that the data set from the process does in fact trigger the IDEA tool and indicates an alert. Slowdown was measured and is a possible factor in a compute-bound program, but the target of our intrusion detection effort are various web servers, which are typically I/O bound and not compute bound.

Our research goal, determining whether this new data source is viable for intrusion detection, has been met. To test that the instrumentation is applicable to larger servers and programs, we have instrumented Apache and other open source projects and verified that they still function as before. Thus they are candidates for our next round of testing for the IDEA intrusion detection system.

Additionally we are in the process of setting up a dedicated server on the public internet which will further describe our project and also act as a live test. We anticipate receiving some true intrusion attempts at that time, which would trigger the IDEA system.

6. References

[1] Apache, at http://httpd.apache.org/

[2] Bace, Rebecca Gurley: “Intrusion Detection”, Macmillan Technical Publishing, 2000.

[3] DARPA Intrusion Detection Evaluation Data Sets, Lincoln Laboratory, Massachusetts Institute of Technology. www.ll.mit.edu/IST/ideval/index.html.

[4] Endorf, Carl, Schultz, Dr. Eugene, Mellander, Jim: “Intrusion Detection and Prevention”, McGraw-Hill/Osborne, 2004.

[5] Fizmez, at http://fizmez.com/

[6] Gcov, http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[7] Hoffmeyr, Stephen A., Stephanie Forrest, Anil Somayaji, “Intrusion detection using sequences of system calls” Journal of Computer Security, Vol. 6, Issue 3 (August 1998) 151–180.

[8] Locasto, Michael E., Keromytis, Angelos D.: “Binary-level function profiling for intrusion detection and smart error virtualization”. Columbia University Technical Report, http://mice.cs.columbia.edu/getTechreport.php?techreportID=381.

[9] Mahoney, William and Sousan, William: “Using Common Off-The-Shelf Tools To Implement Dynamic Aspects” SIGPLAN Notices, vol 42 (2), February 2007.

[10] Manniatty, William A., Adnan Baykul, Vikas Aggarwal, Joshua Brooks, Aleksandr Krymer, and Samuel Maura, “A Linux kernel auditing tool for host-based intrusion detection”, 21st Annual Computer Security Applications Conference, December 5-9, 2005, Tucson, Arizona.

[11] Monkey, at http://monkeyd.sourceforge.net/

[12] Netcraft, “Web Server Survey”, http://news.netcraft.com/archives/web_server_survey.html

[13] Northcutt, Stephen, and Novak, Judy: “Network Intrusion Detection”, New Riders, 3rd ed. 2003.

[14] Peisert, Sean, Matt Bishop, Sidney Karin, Keith Marzullo, “Analysis of computer intrusions using sequences of function calls”, IEEE Transactions on Dependable and Secure Computing, Vol. 4 No. 3, April-June 2007.

[15] Proctor, Paul E.: “The Practical Intrusion Detection Handbook” (Prentice Hall, 2001)

[16] Spec; http://www.spec.org/cpu2006/

[17] Thttpd, at http://www.acme.com/software/thttpd/

Authors

William R. Mahoney received his B.A. and B.S. degrees from Southern Illinois University, and his M.A. and Ph.D. degrees from the University of Nebraska. He is a Research Fellow and Graduate Faculty at the University of Nebraska at Omaha Peter Kiewit Institute. His primary research interests include language compilers, hardware and instruction set design, and code generation and optimization. Prior to the Kiewit Institute Dr. Mahoney worked for 20+ years in the computer design industry, specifically in the areas of embedded computing and real-time operating systems. During this time he was also on the part time faculty of the University of Nebraska at Omaha. His outside interests include bicycling, photography, and more bicycling.

William Sousan is a PhD student at the University of Nebraska at Omaha, where he also received his bachelors and masters degrees in Computer Science. His research interests include software engineering with dynamic aspects, agent based internet search technologies, and image and pattern processing, with emphasis on orientation-independent object recognition. He also has an extensive background in embedded systems and is a principal in a local embedded systems engineering firm.