ASM — Application Security Monitor

Micha Moffie and David Kaeli
Computer Architecture Research Laboratory
Northeastern University, Boston, MA
{mmoffie,kaeli}@ece.neu.edu

Abstract

Our Application Security Monitor *(ASM) is a run-time monitor that dynamically collects execution-related data. ASM is part of a security framework that will allow us to explore different security policies aimed at identifying malicious behavior such as Trojan horses and backdoors.*

In this paper, we show what type of data ASM can collect and illustrate how this data can be used to enforce a security policy. Using ASM we are able to explore different tradeoffs between security and performance.

1 Introduction

Criminal computer attacks grew at an alarming rate in 2004 and are expected to continue to rise in 2005. Zero-day attack exploits are being sold on the black market [11]. Trojan horses and backdoors may be a significant part of those exploits. They can cause significant damage, such as information loss and information tampering. Even worse, many Trojan horses and back-doors may go undetected.

We are currently developing a security framework that will uncover Trojan horses and backdoors, and also defend against harmful activity. In this paper we present ASM, a run-time monitor which is at the heart of our new security framework. We consider the different kinds of information that need to be collected by ASM. We discuss ASM's usefulness for security policies and analyze ASM performance impact.

In the next section we review examples of security exploits. We then analyze some malicious code examples and show what can be done to detect these exploits. Next, we analyze the impact of our solutions on performance. Finally, we discuss related work and conclude in section 6.

2 Security Exploits

To establish the basis and motivation for our work we present some real world examples of malicious code

1. W32/MyDoom.B virus is an executable file that can infect a Windows system. When executed, the virus attempts to generate files and add entries to the Windows registry. The virus modifies the registry to execute itself (at log in time) and to reference a backdoor component. In addition, the virus downloads and installs a backdoor. The backdoor component (ctfmon.dll) opens a TCP port, and can accept commands, execute additional code, or act as a TCP proxy. (US-CERT Alert TA04-028A) [14].

2. The CIH/Chernobyl Virus infects executables and is spread when an infected executable is executed. The CIH virus has several variants. Some are triggered every month on the 26th day, while other variants are triggered just on April 26th or June 26th. Once the CIH virus executes, it attempts to erase the entire hard drive and overwrite the system BIOS. CIH affects only Windows 95/98 machines (CERT IN-99-03) [15].

3. The Phatbot Trojan can be controlled by an attacker on a remote site (using a p2p protocol). The Trojan has a large set of commands which can be executed. A few of these commands include: stealing CD keys, running a command using *system(..)*, displaying system information, executing file from an ftp url and killing a process [5].

4. The Trojan Horse version of the Sendmail Distribution contains malicious code that is executed during the process of building the software. The Trojan forks a process that connects to a fixed remote server on port 6667. The forked process allows an intruder to open a shell running as the user who built the Sendmail software (CERT Advisory CA-2002-28) [15].

5. A Trojan horse version of TCP Wrappers can provide root access to intruders who are initiating connections with a source port of 421. Also, upon compilation of the program, this Trojan horse sends email to an external address. The email includes information which can identify the site and the account that compiled the program. Specifically, the Trojan sends the information obtained from running the commands *whoami* and *uname -a* (CERT Advisory CA-1999-01) [15].

Analyzing these viruses, Trojan horses, and backdoors reveals several distinct characteristics and behaviors:

1. Executables are downloaded and executed without user intervention.

2. The malicious code may create and/or update files in the file system (possibly Windows registry) with *fixed* (i.e., hard-coded) values.

3. The code initiates a connection to a *fixed* remote host. It then downloads executables/data or uploads private information (which may be obtained by executing *fixed* commands).

4. The malicious code may execute only under specific conditions. For example, execute only on a specific date or port number.

5. The code allows a remote user to initiate commands or control the execution on the local host.

An important observation is that many of the resources used by the malicious code, such as file names and network addresses, are hard-coded in the binary. This is not surprising because for the malicious code to be self-contained, it must execute without any guidance from the user or from the attacker.

3 ASM

Our goal is devise new security policies that can effectively counter attacks such as those described above. In this section we describe the data ASM gathers at run time. Next, we present a detailed example of C code that mimics malicious code and shows what needs to be done to collect information to detect its presence. We then discuss our implementation within ASM.

3.1 Dynamic Data Collection

We begin by describing the different data types we collect at run time.

1. Access to resources (file system/network) – This information enables us to track program behavior. It is used by most behavioral security mechanisms.

2. Data flowing between different resources, such as data read from a socket – the type of the data (executable, text, etc.) can be used by security policies to filter downloaded data.

3. Source and target of the data flow – source or target such as a network address or a filename. This data is used by most behavior based security mechanisms to implement different security policies.

4. Source of the access request – the origin (in the code) from where the request originated. This data can be used to detect intrusions (similar to *code origins*, as discussed by V. Kiriansky et al. [6]).

5. Source of the access request arguments – in particular the source of the file name or network address (e.g., embedded in the binary, user input, user file, or network). This is essential for finding any hard-coded information.

6. Code frequency – this type of information can be used to detect code segments that rarely execute (malicious code segments).

3.2 Mimicing Malicious Code

In this section we show an example of a simple C code that may be used as part of a malicious program. We show in detail how tracking the data can be useful.

The example in figure 1 shows a program that contains potentially malicious behavior. The program opens a file and writes to it. Both the file name and the content written to the file are hard-coded (i.e., embedded in the binary). In lines 7-9, a static buffer *script* is declared and initialized, and in line 13, the buffer *filename* is declared and initialized. Note, that *filename* is declared inside a function, and is therefore an automatic variable that will be allocated on the stack.

We are interested in detecting the following:

1. If there are accesses to a file.

2. If the source of the argument to the *open* call is hard-coded, or

3. If the source of the data flowing to the file is hard-coded.

In order to track file accesses, we monitor system calls. To detect the data sources of each of the buffers, we need to track the data.

To be able to understand the sources of data and the precise flow of data, we compile the program (we use gcc 3.4.2 (20041017) on Red Hat Linux, no flags). Figure 2 shows the assembly of the main function.

Lines 19 and 26 are calls to *open* and *write*, respectively. The open call expects two parameters: filename and flags. Moving up from line 19, we see two push instructions (lines 16 and 18) that push arguments onto the stack. In line 16 we push the value 0x440 – the O_CREAT | O_APPEND flags. In line 18 we push %eax – this register will point to the location where the filename *.csh* can be found.

Debugging the assembly further reveals that %eax holds an address on the stack that contains the filename string. Looking earlier in the code, we can see the that %eax is assigned the effective address of 0xffffffd8(%ebp) in line 17. In lines 11-14 the value of 0xffffffd8(%ebp) is set using %eax. Looking more closely at line 11, we see that %eax is assigned the value *.csh* located at address 0x80484d8. (A similar assignment happens with %al: null is loaded from the memory and stored just after *.csh*.)

To summarize, lines 11-14 load the string *.csh* into the stack starting at 0xffffffd8(%ebp). 0xffffffd8(%ebp) is later used as a parameter to the *open* syscall in lines 17-19.

Following the flow of the *write* call is much simpler. Line 24 pushes the address $0x8049600, which contains the script

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/types.h>
4.  #include <sys/stat.h>
5.  #include <fcntl.h>
6.
7.  static char script[] =
8.       "csh malicious code .. here \
9.         .. grep, fork and more ...";
10.
11. int main(void) {
12.   int fd;
13.   char filename[] = ".csh";
14.
15.   fd = open(filename,
16.             O_CREAT | O_APPEND);
17.   write(fd, script, sizeof(script));
18. }
```

Figure 1. A hard-coded C function.

code. In order to detect whether the arguments sent to an *open* or *write* call are hard-coded, we need to track the data flow in the program. We track the data being loaded, track their manipulation on the stack and any future use. Effectively, this means tracking all memory accesses, as well as ALU operations. In addition, we need to know that the both filename and script originate from the binary image. This can be detected when the binary is loaded: we can identify each of the binary sections being loaded and label those for future reference. Figure 3 shows two of the binary sections (.rodata and .data), where the filename string and script code are located, respectively.

Note that lines 19 and 26 are actually the calls to the *open* and *write* library functions. These routines are the functions that actually call the system (using linux int 80).

3.3 Our implementation of ASM

In order to gather the information types presented above, we need to track the following:

1. System Calls – used to track accesses to both the file system and the network.
2. BB Frequency – support for finding portions of the code that are rarely executed.
3. Program Data Flow – for tracking data flowing between system calls and for tracking data sources.

One option is to statically analyze the binary and compute the data flow. This method can potentially reduce overhead introduced during runtime, but has several shortcomings. First, there is no knowledge about the real execution path (e.g., basic block frequencies). Second, no real-time data from the user or the network can be monitored and analyzed. Third, a static tool may not be able to discover all the code being executed.

```
1.   push   %ebp
2.   mov    %esp,%ebp
3.   sub    $0x28,%esp
4.   and    $0xfffffff0,%esp
5.   mov    $0x0,%eax
6.   add    $0xf,%eax
7.   add    $0xf,%eax
8.   shr    $0x4,%eax
9.   shl    $0x4,%eax
10.  sub    %eax,%esp
11.  mov    0x80484d8,%eax
12.  mov    %eax,0xffffffd8(%ebp)
13.  mov    0x80484dc,%al
14.  mov    %al,0xffffffdc(%ebp)
15.  sub    $0x8,%esp
16.  push   $0x440
17.  lea    0xffffffd8(%ebp),%eax
18.  push   %eax
19.  call   0x80482e4 <_init+72>
20.  add    $0x10,%esp
21.  mov    %eax,0xfffffff4(%ebp)
22.  sub    $0x4,%esp
23.  push   $0x36
24.  push   $0x8049600
25.  pushl  0xfffffff4(%ebp)
26.  call   0x80482c4 <_init+40>
27.  add    $0x10,%esp
28.  leave
29.  ret
30.  nop
31.  nop
```

Figure 2. An assembly dump of a hard-coded C function.

On the other hand, while dynamic instrumentation may impose a performance penalty, it provides for richer data

```
...
Contents of section .rodata:
 80484d0 03000000 01000200 2e637368 00        .........csh.

...
Contents of section .data:
 80495e0 00000000 00000000 f0940408 00000000  ................
 80495f0 00000000 00000000 00000000 00000000  ................
 8049600 63736820 6d616c69 63696f75 7320636f  csh malicious co
 8049610 6465202e 2e206865 7265202e 2e206772  de .. here .. gr
 8049620 65702c20 666f726b 20616e64 206d6f72  ep, fork and mor
 8049630 65202e2e 2e000000                    e ......
...
```

Figure 3. A object dump of a hard-coded C function.

flow analysis. Using dynamic instrumentation, we can easily monitor the executable and track data flow as well as determine code frequency.

Tracking data flow has two tasks. The first is to track all memory locations that are loaded or mapped from different files or sockets. These memory locations are labeled with the appropriate data source for future reference. The second task is to inspect individual assembly instructions. Instructions that access memory (e.g., push, pop, mov), as well as ALU instructions that manipulate data, need to be tracked. This allows us to label a register with the appropriate data sources once the register is assigned. We will also be able to update memory locations that are assigned values with up-to-date data sources. For example, assume there are two registers that are assigned values from different memory locations (two different data sources). Xoring both registers will result in one register (or memory location) with a label corresponding to both data sources. Note that our second task imposes a significant overhead on execution time. We call the first aspect Direct Data Flow and the second one Indirect Data Flow.

Tracking the code frequency can be obtained by counting the number of basic block executed. Since we are usually interested in the frequency of code in the application itself, we distinguish between the application's binary versus dynamically loaded shared objects. In our experiments, we count the execution frequency of only application code. Since dynamic instrumentation is done 'on top of' the operating system, we can monitor system calls and inspect all the data associated with them.

We use Pin[9] as our method for dynamic accessing dynamic information. Pin is a tool developed by Intel for dynamic instrumentation toolset that can instrument Linux binary executables. Pin can instrument Linux binaries compiled for the Intel (R) Xscale (R), IA-32, IA-32E (64 bit x86), and Itanium (R) processors. We are utilizing the IA-32 features of the toolset.

We have implemented ASM to dynamically extract runtime information using Pin. Note, however, that the current implementation is incomplete. Although we track many system calls, we do not cover all of them. In addition, we monitor a most assembly instructions, but do not presently include infrequently used instructions.

4 Performance

Arming a system's security framework with ASM's abilities can increase our confidence in the system and reduce our exposure. Unfortunately, security in this form comes at a cost. The performance impact of dynamic instrumentation is not negligible, and the added overhead ASM imposes can be significant.

To explore the tradeoffs between security and performance, we define several configurations:

1. *ASM System Calls* – tracks system calls.
2. *ASM System Calls, Direct DataFlow* – collects both system calls and Direct Data Flow.
3. *ASM System Calls, Frequency, Direct DataFlow* – tracks the applications code frequency as well.
4. *ASM System Calls, Frequency, DataFlow* – tracks both direct and indirect data flow as well as system calls and frequency.

We compare the performance of different ASM configurations and include two reference points:

1. Stand Alone – The application itself, without any instrumentation or monitoring.
2. Profiling – The application running on top of Pin, instrumented without any monitoring (i.e., without any tool).

All experiments where run on an Intel Xeon, 2GHz machine with 512KB cache and 1GB of RAM running Linux. We executed three SPEC2000 applications, each running reference inputs: gcc - 200.i, gzip - input.graphic and mcf - inp.in. Figure 4 shows the execution time of the the three benchmarks directly and then on top of the four different ASM configurations.

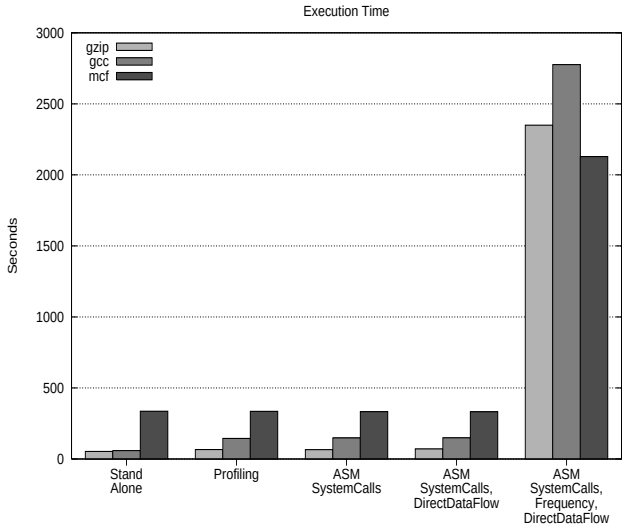

Figure 4. ASM configurations performance

4.1 Analysis

Both *ASM System Calls* and *ASM System Calls, Direct DataFlow* configurations have relatively good performance. The reason for the good performance is the relatively small numbers of events that are being monitored. For the *ASM System Calls* configuration, we only monitor system calls. These events are not very frequent and thus, the overhead is low. The *ASM System Calls, Direct DataFlow* has a similar performance. This configuration encounters additional overhead caused by tracking the memory that is mapped when data files or executables are loaded. Since these events are relatively uncommon, the overhead is low.

The *ASM System Calls, Frequency, Direct DataFlow* configuration exhibits a significant slowdown. This occurs for two reasons: First, we instrument each basic block in the application. This means that for every basic block executed, additional overhead is introduced (accessing a data structure). Second, we limit the traces inserted into Pin's code cache to one basic block in length. We do this so we can easily find newly inserted code. This has an additional impact on performance. Note that we cannot just count the frequency of system calls instead of basic blocks. This is because system calls are usually called from shared libraries. Malicious code executing for the first time may very well invoke a previously executed system call. This will leave us with no knowledge about the frequency of that application code segment which triggered the system call.

The *ASM System Calls, Frequency, DataFlow* configuration exhibits a dramatic slowdown and is not shown in the graph. Our tests show a slowdown ranging from 100x to 1000x. This slowdown in unacceptable. The slowdown is introduced by tracking the indirect data flow, which involves instrumenting numerous instructions. During each profile event, a data structure is dynamically updated with a newly computed label. Even though we encounter unacceptable over, we will study the security benefits of this configuration further given the level of security provided. In our future work we will explore hardware support to increase performance [16, 13].

5 Related Work

There have been several methodologies proposed to reduce the security risk from malicious code. Information flow security systems have focused on language-based and static analysis mechanisms [1, 10]. These systems only allow the programmer to specify the policy.

System call monitoring is often used to detect malicious code. Such monitoring can be used to differentiate between normal behavior, which was recorded beforehand, and anomalous behavior [8]. The history of access requests can be also be used to dynamically classify programs on-line and execute them with appropriate privileges [2]. Software wrappers are used to detect and remedy system intrusions [7]. These wrappers are software layers, dynamically inserted into the kernel, that can selectively intercept and analyze system calls performed by processes. Using software wrappers in the kernel can significantly reduce the performance penalty, but offer less help compared to the detailed information available in user space.

Runtime monitoring of untrusted helper applications was introduced by Goldberg et al. [4]. The authors proposed to create a secure environment to contain untrusted helper applications by restricting program access to operating system resources.

Scott et al. develop a portable extensible framework for constructing a safe virtual execution system [12]. The authors show how easily system calls can be monitored and how a simple policy can be constructed. They present several policies that can track specific malicious behavior.

Program Shepherding was introduced by Kiriansky et al. [6] and is used to enforce secure execution. They suggest to thwart attacks that change the control flow (such as buffer overflow attacks) by dynamically monitoring the flow.

Gap et al. [3] perform an analysis of many host-based anomaly detection systems. These systems monitor a process running a known program by tracking the system calls the process makes. They organize previously proposed solutions across three dimensions:

- Runtime information that the detector uses to check for anomalies. This includes system call number, as well as arguments and information extracted from the process address space, such as the program counter and return addresses.

- The Atomic unit that the detector monitors, i.e., a single system call or a variable-length sequence of system calls.

- The history - the number of atomic units the detector remembers.

RIFLE, an architectural framework for user-centric information flow security, can track information flow in every program. This equips the user (in contrast to the programmer) with a practical way of enforcing any information flow policy [16]. Information tracking can also be used to defeat malicious attacks by identifying spurious information flows and restricting their usage [13].

6 Conclusions

As part of our security framework, ASM will be used to study the tradeoffs between security and performance. We show that several ASM configurations have a different impact on performance and can be used to counter different security vulnerabilities such as Trojan Horses and backdoors.

Although complete data flow monitoring can have a significant impact on performance, mechanisms such as RIFLE [16] may be used to alleviate the problem. We plan to continue our work to increase the effectiveness of dataflow monitoring and explore solutions reduce the overhead.

References

[1] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[2] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, 1998.

[3] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, Aug. 9-13 2004.

[4] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[5] LURHQ Threat Intelligence Group. Lurhq. http://www.lurhq.com/phatbot.html.

[6] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Security '02: Proceeding of the 11th USENIX Security Symposium*, San Francisco, August 2002.

[7] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the USENIX Security Conference*, pages 145–156, Jan 2000.

[8] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[9] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, Jun. 2005. Chicago, IL.

[10] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[11] Bruce Schneier. Attack trends 2004 and 2005. In *ACM Queue vol. 3, no. 5*. ACM, Jun. 2005. http://acmqueue.com/.

[12] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 209, Washington, DC, USA, 2002. IEEE Computer Society.

[13] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[14] United States Computer Emergency Readiness Team. Us-cert. http://www.us-cert.gov/.

[15] Carnegie Mellon University. Cert. http://www.cert.org/.

[16] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.