Secure Coprocessor-based Intrusion Detection

Xiaolan Zhang Leendert van Doorn Trent Jaeger Ronald Perez Reiner Sailer
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
Email: {cxzhang,leendert,jaegert,ronpz,sailer}@us.ibm.com

1 Introduction

The goal of an intrusion detection system (IDS) is to recognize attacks such that their exploitation can be prevented. Since computer systems are complex, there are a variety of places where detection is possible. For example, analysis of network traffic may indicate an attack in progress [11], a compromised daemon may be detected by its abnormal behavior [14, 12, 5, 10, 15], and subsequent attacks may be prevented by the detection of backdoors and stepping stones [16, 17].

The most popular architecture for IDSs is *host-based intrusion detection*, where the IDS runs as a monitor on its host and collects information used to identify possible intrusions on that host. Since the compromise of any system service generally results in the compromise of the operating system's trusted computing base (TCB), the IDS is also susceptible to compromise, and thus cannot be trusted.

In this paper, we examine the effectiveness of *secure-coprocessor-based intrusion detection*. In this case, the IDS is run on a coprocessor rather than on the host. Thus, a compromise of the host does not affect the coprocessor, and self-protection of the IDS monitor is achieved. Since a coprocessor can see the memory of the host, a coprocessor IDS can verify that the host's state is correct. However, a coprocessor IDS cannot interpose the host's execution the way that a host IDS can. Therefore, we need to identify a new approach that enables effective detection given the external nature of the coprocessor.

The remainder of the paper is structured as follows. In Section 2, we define coprocessor-based intrusion detection. In Section 3, we discuss a series of applications possible with this approach, and describe experiments that show the kinds of attacks that can be successfully detected by a coprocessor. In Section 4, we discuss limitations of this approach and how it can be extended to take preventive steps when anomalies are detected. Section 5 concludes.

2 Coprocessor-based Intrusion Detection

Coprocessor-based intrusion detection means that host data is collected and processed by software running on a coprocessor rather than the host itself [4]. Typically, a coprocessor shares interfaces with the host processor that enables it to examine and perhaps modify the state of the host. The number and choice of interfaces determines the degree to which intrusion detection is possible. We use a secure coprocessor because it offers additional security features that are desirable for an IDS (see Section 2.2).

2.1 Secure Coprocessors

A *secure coprocessor* is a tamper-resistant computing device designed to perform critical tasks in an environment in which physical attacks are possible. Such a device can be used to securely boot the host system into a known state.

In the context of this paper, we examine use of the IBM 4758 PCI Cryptographic Coprocessor [1, 7, 13]. The 4758 consists of a CPU, volatile and non-volatile memory, and cryptographic accelerators. It is wrapped inside a tamper-responding secure boundary. The device communicates with the host via the PCI bus, whereby it can issue commands to operate on system memory.

The software architecture of the IBM 4758 device is designed to support generic security applications [8]. The software insures that the device boots securely, and that only authorized programs can execute on the device. The device also comes with factory-installed certificates that allow it to authenticate itself to external entities.

2.2 Advantages

Compared to host-based intrusion detection, the use of a secure coprocessor for intrusion detection has the following advantages:

1. **Independence from the host OS.** The secure coprocessor is an autonomous subsystem that has its own

operating system and application software. Its tamper-resistance also provides strong integrity protection for the IDS.

2. **Narrow interface.** The interface used for communicating between the host and the secure coprocessor is simplistic and well-defined. It is therefore much more difficult to exploit the interface and launch attacks.

3. **Secure boot.** The secure coprocessor can be used to boot the host into a known state in which invariants can be defined.

4. **Trusted observer.** Since the secure coprocessor is designed to protect its authentication keys against almost any attack, any authenticated statements made by the secure coprocessor can be fully trusted. This is very useful in a scenario where multiple coprocessors collaborate on a task or the IDS data is consumed by a remote entity.

2.3 Monitoring

In addition to self-protection, an effective IDS must be able to monitor the controlled operations that may lead to an intrusion. In a host-based IDS, system calls and key kernel operations are often interposed such that IDS data can be collected and analyzed. In a coprocessor-based IDS, monitoring cannot be done by interposition. The host will continue to execute, so the IDS paradigm must be altered to suit this environment.

Instead of interposing operations, we propose that the coprocessor IDS base its analysis on system invariants. The host system as a whole maintains certain integrity properties (invariants) when it is functioning correctly (i.e. it has not yet been compromised). Since it can boot the host into a known state, the coprocessor IDS can be expected to know certain host OS state, such as the location and value of key data structures. Given key data structures and invariants on their values and the way in which values are allowed to change, the coprocessor can sample the host OS to verify that the invariants are still held.

3 Applications

In this section we discuss several monitoring applications that can be implemented on the secure coprocessor.

3.1 Checking Kernel Data Structures Invariants

The first set of invariants we will examine concern in-memory kernel data structures. We can view the OS as a state machine whose states are stored in a collection of internal data structures. Examples of such data structures include *task_struct*, the data structure that abstracts the notion of process, and *inode*, the data structure for representing a file [1]. The operating system reacts to external events (such as system calls or network packet arrivals) by performing appropriate modifications on these kernel data structures. Assuming that, when the system is in a secure state, the values of these kernel data structures are consistent and exhibit a set of invariants, but when the system is compromised these invariants no longer hold, the monitoring system can detect break-ins (or break-in attempts) by continuously checking for consistencies of crucial kernel data structures.

Our approach is to be distinguished from previous approaches [12, 14], which focus on the events that cause the kernel to enter an illegal state, rather than on the states themselves. For example, in Forrest's approach, one first profiles the target application and collects a database of legitimate system call sequences (signatures) made by the application. In the production system, the OS monitors system calls executed by these applications, and issues a warning if a sequence does not match any in the database. Since events are program dependent, e.g., different programs typically have different signatures, one thus needs to maintain an extensive database of signatures covering all programs. Frequent software upgrades further complicate this problem. Our approach, on the other hand, tries to abstract the validity of states into invariants. Because we use abstractions, we keep less information. In addition, the invariants describe properties of the kernel only, and thus are much more stable.

3.1.1 Determining Kernel Invariants

To find out what the invariants are, we implemented a kernel module that intercepts system calls and records the values of crucial data structures at the entry of each system call. We then compare the value trace of a correct OS with that of a compromised OS for a given attack taken from an database of known exploit programs, and search for systematic differences between the pair of values. The systematic differences potentially highlight the invariants that are violated.

We define two types of invariants: *global invariants* and *application-specific invariants*. Global invariants are invariants that apply across the entire operating system, independent of the programs running on top of the OS. Examples of global invariants include immutability of the kernel image and images of crucial system programs, and immutability of kernel data structures such as system call tables. Application-specific invariants, on the other hand, depend on the specific nature of the application represented by the kernel data structures in question. For example, a normal user program's *uid* should never change to root. This

[1]Unless otherwise explicitly stated, we base our discussion on the Linux operating system.

invariant certainly does not apply to programs such as *su*.

3.1.2 Detecting Violations

Once the invariants are determined, it is relatively easy to detect violations against these invariants. For global invariants such as immutability of kernel image, the monitor computes a checksum over the image at (secure) boot time, and periodically recomputes the checksum and compares it with the stored value. For application-specific invariants, the monitor determines the type of application by its name (i.e. the command line field of the task structure), and loads appropriate invariants according to the application type. It then periodically samples the values of relevant data structures and checks the values against the invariants.

3.1.3 An Example

Let's look at an example invariant that we derive by running a local-root exploit program [2] and comparing the changes in the *task_struct* values between a successful attempt and those of a normal user program. The attack program, *ptrace24*, works by exploiting the race between `ptrace` and `execve` and injecting arbitrary code into a setuid program. Table 3.1.3 shows the fields of the *task_struct* data structure that exhibit different change patterns depending on whether the attack succeeds or not.

We derive the following invariants for normal user-programs from the above data.

1. *uid* should remain the same throughout execution.

2. *euid, suid and fsuid* should not be different from the original *uid* for an extended period of time.

To check the invariants, the monitor periodically scans the values of the relevant fields of the task structure for active processes, and validates the values against the invariants. Note that the monitor needs to store the old value of *uid* for each process.

This simple example illustrates that it is possible to infer invariants by profiling known exploits and use the invariants to detect ill-behaved processes.

To test the generality of this invariant, we examined another exploit program [3] that uses a different technique to gain control of the system. A bug in the `traceroute` program causes buffer overflow and allows arbitrary code to be executed on the stack. The invariants are essentially the same as the *ptrace* exploit. This is not particularly surprising because both exploits attack the system by becoming the root, which requires a change of the *uid* field to *root*. However, it demonstrates that the invariants are applicable to exploits of the same nature (in this case, local root exploit through setuid programs), and thus only one set of invariants are needed for these exploits even though they differ dramatically in the methodology of attacking.

3.2 File Integrity Checking

Another important correctness property of a system is the integrity of system files on disk. The monitoring system can independently scan the disk, compute checksums of system files, and compares the results against those stored in a database. This is similar to the Tripwire [9] commercial product. The difference being that the monitoring system and the checksum database reside on the secure coprocessor, instead of on the host system, and are thus not vulnerable to attacks.

3.3 Virus Detection

The monitoring system can also scan the entire memory for known viruses. Again because the monitor resides on the coprocessor, it is much less intrusive than a tool like Norton Utilities.

4 Discussion

Since the monitoring system is based on sampling, there is no guarantee that the attack is detected in time. However, we can reduce this likelihood through control of the placement of samples, the number of samples, the sampling period, and the period distribution. Sample placement is driven by the number of attacks that can be detected and the accuracy of the detection. Obviously, a single point that detects all errors with perfect accuracy would be the best case. Since this is unlikely, we can increase the number of samples until we have sufficient coverage. However, the number of samples is limited by the computing speed of the coprocessor.

One way to reduce this cost is to identify dependencies between sampling points. Only when one sample is triggered are its dependents samples, and others are delayed or removed temporarily. Another way is to adaptively change the sampling frequency based on the actual state of the system. For instance, we could raise the sampling rate when suspicious events are detected, such as when a process is running with root or setuid privilege, and slow down the rate once suspicious events cease to exist. This way, the time window for an undetected attack is smaller at times of higher risk. Finally, we can vary the periodic distribution of the sampling to reduce its predictability.

Another limitation of our current monitor is that it is based on the PCI bus which provides only limited control over the host. Ideally, we would like to be able to use the host JTAG bus [6]. The JTAG bus is a hardware debug facility that can be used to control peripherals in the host. That

Field	Sample Sequence of A Successful Attack				Sample Sequence of A Normal Process			
	1	2	3	4	1	2	3	4
flags	64	0	256	256	64	0	0	0
uid	500	500	**0**	**0**	500	500	500	500
euid	500	0	**0**	**0**	500	0	500	500
suid	500	0	**0**	**0**	500	0	500	500
fsuid	500	0	**0**	**0**	500	0	500	500
cap_effective	0	x	x	x	0	x	0	0
cap_permitted	0	x	x	x	0	x	0	0
user	x	x	y	y	x	x	x	x

Table 1. Sampled values of fields of *task_struct* for a successful attack and a normal user process. Shown above are a sequence of 4 sample points taken at 4 different system call entry points. For simplicity reasons, only a subset of sample points are presented here. For the flags field, 64 means forked but not exec, 256 means used privileges. For the cap_effective, cap_permitted, and user fields, x means non-zero value, and y means a non-zero value other than x.

is, stop the CPU, inspect its state, and resume execution, or inspect/change the state of memory or any other controller attached to the bus. The JTAG approach can thus take preventative/remedy steps when an anomaly is detected. There is however a tradeoff between cost and effectiveness. JTAG is chip-dependent and thus much more expensive then the generic PCI-based solution. However, if the PCI approach is promising we may explore the JTAG approach so we can assert greater control over the host.

5 Conclusion

In this paper we proposed building intrusion detection systems using external secure coprocessors. Because the coprocessor runs independent of the host, a compromise of the host does not affect the functionality of the IDS. The additional security features of the coprocessor ensure that the host starts from a secure state, and that messages sent by the coprocessor can be authenticated and trusted. We discussed a series of possible monitoring applications, and our early results demonstrated the viability of this approach.

References

[1] IBM PCI Cryptographic Coprocessor General Information Manual, May 2002. Available at http://www.ibm.com/security/cryptocards.

[2] Ptrace2.4. Available at http://packetstormsecurity.org/0203-exploits/ptrace-dark.c.

[3] Traceroute exploit + story. Available at http://security-archive.merton.ox.ac.uk/bugtraq-200010/0084.html.

[4] J. M. A. Mishra and W. Arbaugh. The coprocessor as an independent auditor. Available at http://www.missl.cs.umd.edu/komoku/documents/coauditor.ps.

[5] S. N. Chari and P. Cheng. Bluebox: A policy driven, host-based intrusion detection system. In *Proceedings of the 2002 Network and Distributed System Security*, February 2002.

[6] IEEE. IEEE standard test access port and boundary-scan architecture, IEEE std 1149.1b-1994.

[7] R. P. R. S. L. v. D. S. W. S. J. Dyer, M. Lindemann and S. Weingart. Building the ibm 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[8] S. W. S. J. Dyer, R. Perez and M. Lindemann. Application support architecture for a high-performane, programmable secure coprocessor. In *22nd National Information Systems Security Conference (NISSC)*, October 1999.

[9] G. H. Kim and E. H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. In *System Administration, Networking and Security Conference III*, 1994.

[10] E. G. M. Bernaschi and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 174–183, 2000.

[11] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[12] A. S. S. Forrest, S. Hofmeyr and T. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996.

[13] S. W. S. Smith, R. Perez and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference (NISSC)*, October 1999.

[14] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[15] D. Zamboni. Using internal sensors for computer intrusion detection, 2001. CERIAS Technical Report 2001-42, CERIAS, Purdue University.

[16] Y. Zhang and V. Paxson. Detecting backdoors. In *Proceedings of 9th USENIX Security Symposium*, August 2000.

[17] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of 9th USENIX Security Symposium*, August 2000.