Proceedings of the 2003 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY June 2003

HIDDEN PROCESSES: The Implication for Intrusion Detection

James Butler, Jeffrey L. Undercoffer and John Pinkston

Abstract—**In this paper we introduce a novel class of intrusion: the hidden process, a type of intrusion that will not be detected by an intrusion detection system operating under the assumption that the underlying computing architecture is functioning as specified. A hidden process executes in a manner that is unobservable by many of the operating system's accounting and reporting functions. In this paper we present a mechanism to hide processes. Additionally, we show how a hidden process may communicate with an external entity by piggybacking onto a legitimate network connection. We have implemented a mechanism that detects hidden processes and make recommendations calling for the separation of *critical* operating system functions from more general operating system functions.**

I. INTRODUCTION

In this paper we introduce a novel class of intrusion – the *hidden process*, a type of intrusion that will not be detected by an intrusion detection system that assumes that the underlying computing architecture is functioning as specified.

According to the Computer Emergency Response Team/ Coordination Center (CERT/CC) of Carnegie Mellon University's Software Engineering Institute, the number of reported adverse computer incidents continues to increase significantly [1]. For example, in 1990 there were 252 reported adverse incidents, in 2000 that number grew to 21,756 and by the third quarter of 2002 there have been 73,359 reported adverse incidents. An anecdotal explanation of this dramatic increase may be attributed to Kargle, et. al [2] who have observed that computer attack tools are becoming increasing more effective and available.

The most significant and far reaching consequence of a computer attack is the intruder gaining *root* (administrative) access to the target computer [3]. This type of access opens the targeted system to continued misuse and exploitation. According to the CERT/CC advisories, root access is the most frequent consequence of an attack that successfully exploits some system or software vulnerability [4]. Criscuolo [5] and Toxin [6] have observed that once an attacker has gained unauthorized administrative access to a computer he, or she, will often install a *Root-Kit* as a means of continued access and compromise of the targeted system.

The authors of root-kits and other attack tools often go by pseudo names and are probably somewhat on the fringe. Although one is tempted to minimize their credibility and not take them seriously, let alone cite their work, they are to a large part the very reason that the Internet security software market continues to grow. According to Sutterfield [7] revenues from computer security products and services are expected to reach $28 billion by the year 2006 [7].

We believe that knowledge of the capabilities and intentions of one's adversaries is central to the mission of securing computer resources. Consequently, we have surveyed the repertoire of *hacking tools* that are readily available on the Internet, analyzing those that appear to pose the greatest potential threat. Our survey and analysis serves as a means of determining the direction of interest and the level of expertise of those who attempt to attack and disrupt our systems.

Chief among the hacking tools are root-kit like programs that grant an intruder a continued and unfettered presence on a system. Root kits, which first appeared in 1993, are a collection of tools and *trojaned* replacements of core system utilities. These core utilities include binaries such as *top, ps, ls, du* and *netstat* which are used to manage a system and ensure that it is operating properly. Rootkits replace these core system utilities with modified versions in order to hide the presence of the intruder and her tools. Our survey of freely available attack tools has indicated that there is an increase in the level of sophistication exhibited on the part of the authors of some of these tools. Although complex and difficult to implement, these tools have become easier to use due to their inclusion in automated attack scripts.

We have also observed that loadable kernel modules (LKMs) are being employed as an advanced form of root-kit. LKM rootkits have modules that may replace the system call table or force the kernel to hide specific processes from the */proc* file system. The consequence of hiding process from the /proc file system is that even legitimate copies of system utilities will not accurately list information about the processes executing on the system. The inherent danger of hidden processes (we make the assumption that any process that is hidden from the user is intrusive) is that a *Host Based Intrusion Detection System* that functions under the assumption that the underlying system is operating according to specification, will never see the intrusive process and will give the system owner a false sense of confidence that the system is normal and hence secure.

To explore the feasibility of hiding process on a system without making detectable modifications to the system call table or system utilities, we have written our own program that hides a process on the Windows (NT, 2000 and XP) operating system. As there exists similar tools for hiding processes that execute on the Linux operating system, we have examined and report on

James Butler, Jeffrey Undercoffer and John Pinkston: Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD

ISBN 0-7803-7808-3/03/$17.00

one such tool. Accordingly, we are able to hide processes in the Windows environment without affecting the system call table or by making any detectable modification to the operating system kernel. We have also designed a mechanism that enables a hidden process to *piggyback* onto an existing network connection giving the intruder remote access to the targeted system.

The remainder of this paper is as follows: Section II discusses related research in the area of detecting root-kits and other mechanisms for detecting hidden processes. Section III presents methods for hiding processes. Section IV presents our method for detecting hidden processes. We conclude by presenting our plans for future work.

II. RELATED WORK

The intrusion detection research community specializing in host based intrusion detection is primarily focused on detecting anomalous behavior. Typically they concentrate on the behavior of a specific user or a process. The research of Forrest et. al, [8], [9], [10] which profiles system call usage to detect anomalous behavior exhibited by a process is promising, however they measure and baseline known processes. Consequently, their methodology may be ineffective in detecting hidden processes. Apap et. al [11] have conducted research into anomalous accesses of the Windows *Registry*. Their method detects processes that access Registry key values in opposition to a statistical norm that is derived from presumably attack free data from the specific host. If a hidden process were installed prior to establishing the baseline, or if the hidden process did not access the Registry, its behavior would be considered normal.

There are tools and utilities that detect the presence of known root-kits and LKMs, most notably, *chkrootkit* [12] and *KSTAT* [13]. The chkrootkit tool is a collection of utilities that checks for the presence of a root-kit's configuration files, tests system binaries for malicious content and determines if the network interface is in promiscuous mode. KSTAT is a kernel memory tool that provides kernel statistics. Here too, these utilities check for the presence of rootkits and LKMs that are known *a posteriori* to be malicious.

III. HIDING PROCESSES

As stated in the introduction, we believe that the trend in attack tools is the continued development of the means to hide the presence of intrusive processes. We have examined the feasibility of hiding process on both the Windows (NT, 2000 and XP) and Linux operating systems. Due to Window's prevalence, its subversion offers the greatest profitability to the attacker. Linux is also an excellent target since it is widely deployed and provides many enterprise infrastructure services.

A. Windows

We have observed that when queried about the processes currently running on the system, the Windows operating system presents a list of active processes that is obtained by traversing a doubly link list referenced in the *EPROCESS* structure (process descriptor) of each process. Specifically, a process EPROCESS structure contains a *LIST-ENTRY* structure that has members *FLINK* and *BLINK*. FLINK and BLINK are pointers to the processes forward and behind the current process descriptor. Figure 1 illustrates the EPROCESS block of the Windows kernel.

To hide a process in Windows we first locate the Kernel's Processor Control Block (KPRCB), which is unique and located at 0xffdff120. We then follow the *CurrentThread* pointer to the *ETHREAD* block. From the ETHREAD block we follow the pointer from the *KTHREAD* structure to the EPROCESS block of the current process. We then traverse the doubly linked list of EPROCESS blocks until we locate the process that we wish to hide. Once located, we change the FLINK and BLINK pointer values of the forward and rearward EPROCESS blocks to point around the process to be hidden. Referring to Figure 1, the BLINK contained in the forward EPROCESS block is set to the value of the BLINK contained in the EPROCESS block of the process to hide and the FLINK of the process to the FLINK contained in the EPROCESS block of the rearward process is set to the value of the FLINK contained in the EPROCESS block of the process that we are hiding.

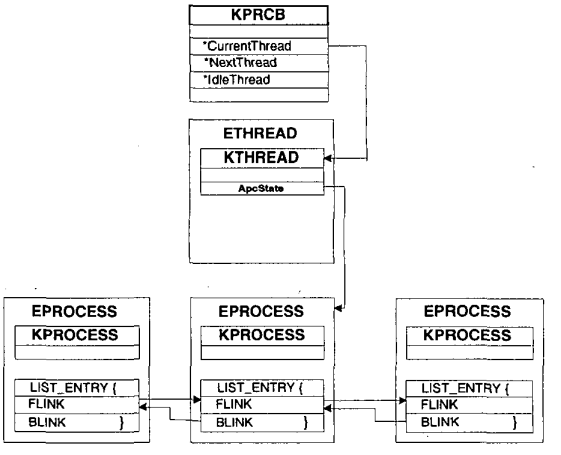

Fig. 1. Windows Kernel Data Structures

Intuitively, one would think that hiding a process by removing its process descriptor from the doubly linked list of EPROCESS blocks would prevent the process from being allocated a time slot in which to execute. We have observed that this is not the case. The Windows scheduling algorithm is highly complex, is done at thread granularity, is priority based and pre-emptive. Accordingly, a thread is scheduled to run for a *quantum* of time, which is the length of time before Windows interrupts the thread to check for other threads of the same or higher priority or to reduce the priority level of the current thread. A process may have multiple threads of execution and each thread is represented by an ETHREAD structure that contains pointers to its siblings. Although, we have been unable to precisely determine why "un-linking" a process' EPROCESS block from the doubly

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

linked list does not adversely affect execution of the process we strongly suspect that the Windows scheduler references those threads from some other linked list, not the EPROCESS block.

We have implemented our method by writing a device driver (.sys), which is similar to a LKM in Linux, and by writing a Dynamically Loadable Library (DLL) that provides an interface to the device driver. To hide a process, we load the device driver and invoke it by passing the name and the unique *PID* (Process Identifier) of the process. This procedure hides the process and unloads the device driver. Since it takes but a few milliseconds to hide a process and unload the device driver we minimize the chance that an intrusion detection system would notice our activity. If this process were to be executed as a system booted, and before any IDS software is executed, it would be virtually impossible to detect.

As demonstrated by our procedure, we do not make any changes to the system call table. Once a process was hidden we have used Microsoft's *Windbg* to attempt to locate the hidden process by running the Windbg as master on one machine and slaving the machine with the hidden process to it via a serial connection. Windbg could not detect the presence of the hidden process.

B. Linux

In our survey of *attack tools* we have focused on two tools that employ LKMs to hide processes, specifically *Adore* [14] and *Phantasmagoria* [15]. The following is an overview of the Phantasmagoria LKM.

The Linux operating system employs a structure of type *task_struct* as a process descriptor. There is a strong one to one correspondence between a process and its process descriptor where the individual process descriptors contain pointers for the *run queue* of runnable processes. Figure 2 provides an overview illustration of the Linux process descriptor.

As depicted, the *task_array* contains a pointer to each process descriptor. Additionally, the first entry in the task_array is to *process 0* which is synonymous with *init_task* or *swapper*. Process 0 is the first process started by the Linux kernel and is at the head of the doubly linked list that is referenced by *next_task* and *prev_task* pointers of each process descriptor. The run queue, a data structure that points to those processes whose state is *runnable* is also maintained via a linked list formed by the *next_run* and *previous_run* pointers of the process descriptor. Finally, each process descriptor contains pointers to its parent, sibling, and child processes. It is important to note that Process 0 does not have a parent process.

To hide a process, Phantasmagoria unlinks the process descriptor from the *task_array*, removes any referencing links from corresponding parent, sibling and child process descriptors, and removes the *next_task* and *prev_task* links of any referencing process descriptors. To maintain a reference to the hidden process Phantasmagoria sets the parent pointer (*p_pptr*) of Process 0 to point to the hidden process. Consequently, the *p_pptr* of Process 0 serves as the root of a list of hidden processes as additional hidden process may be added to the list.

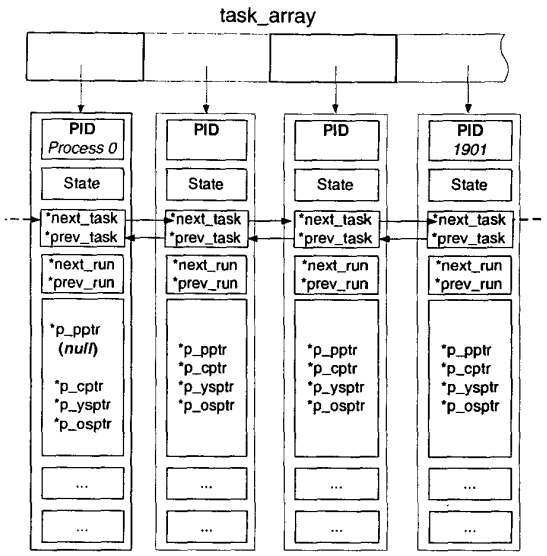

Fig. 2. Linux Process Descriptors

Linux, unlike Windows, exclusively relies upon pointers contained within the process descriptor for scheduling CPU time to a process. Consequently, Phantasmagoria includes code that modifies the *scheduler* function while it is executing. Under normal execution the Linux scheduler, at the start of each *epoch*, traverses the doubly linked list of process descriptors assigning each process *quantums* or CPU time slices. This modification causes the scheduler, to also traverse the list of hidden processes and add runnable processes to the run queue. Figure 3 illustrates the concept of removing PID 1901 from the normal list and hiding it by making it the parent of Process 0.

C. Network Connections

An attacker has little to gain if a she is not able to interact with the computer on which she has placed some intrusive process. If the hidden process were to open a connection and listen on a specific port, utilities such as *netstat* would indicate their presence. Additionally, if netstat were to be replaced with a modified copy, the presence of the modified version is readily detectable by programs such as *Tripwire* [16], a file system integrity checker. Consequently, to establish a *hidden connection* our hidden process *piggy backs* onto an existing applications network connection.

To enable a hidden connection it is feasible to insert a software *"shim"* between the hardware driver and the IP layer of the TCP/IP protocol stack. This notion is illustrated in Figure 4. In order to establish a connection, the hidden process scans its host system searching for an opened port. Once an open port is found the hidden process connects to a predetermined IP address by sending IP packets using *raw sockets* [17]. This informs the attacker of IP address and port number to which the hidden

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

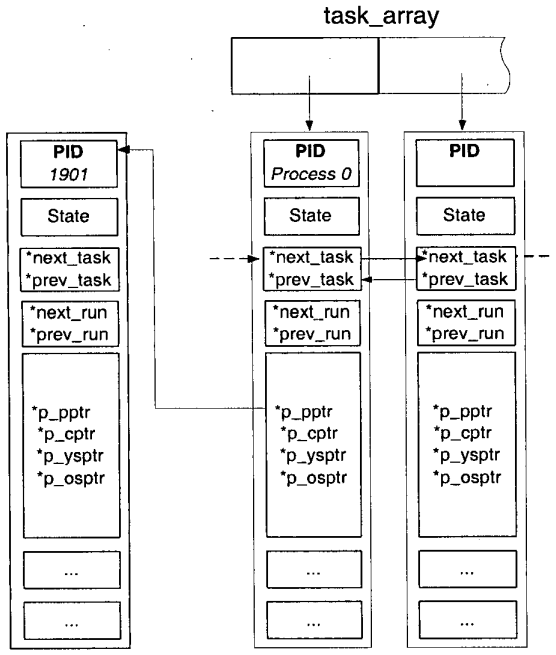

Fig. 3. A Hidden Linux Process Descriptor

process i is "listening". To communicate with the process, the attacker crafts a packet that differentiates it from communications intended for the host and those intended for the *hidden process*. This differentiation is carried out by setting a flag in the IP header. Accordingly, the three precedence bits of the *TOS* field of the IP header are ignored by current implementations of TCP/IP, consequently setting them to a predetermined value will serve as such a flag. Figure 4 illustrates the concept of the shim.

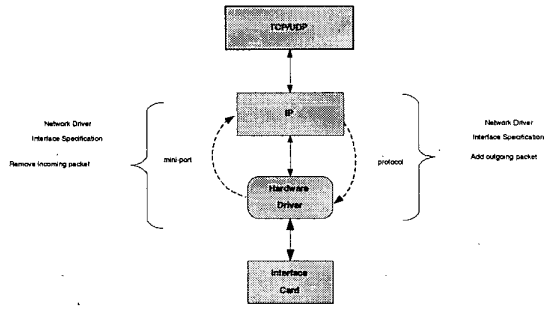

Fig. 4. Insertion and Extraction Points for Hidden Network Traffic

Upon initial observation, one may think that a network shim would be easily detectable and a clear indication that the host has been compromised. However, finding the shim and determining that it is not part of an already complicated network stack is difficult at best. To further exacerbate the situation we are also able to hide device drivers with a technique similar to that employed to hide processes.

If the symbols for the kernel are loaded, *PsLoadedModuleResource* and *PsLoadedModuleList* are easily locatable. PsLoadedModuleResource is similar to a mutex and is used for obtaining exclusive access to the loaded driver list, PsLoadedModuleList. There are corresponding symbols for processes called *PspActiveProcessMutex* and *PsActiveProcessHead*. By removing a driver from the PsLoadedModuleList list, it is hidden from the kernel in many respects. As is the case with hidden processes the *Windbg* debugger no longer lists the driver. The hidden driver continues to work because *IOCTL*'s sent to it run in the context of the thread that issued the IOCTL. If the driver creates its own threads, they become part of System.

One difficulty in hiding device drivers on the Windows operating systems is that the attacker does not have the symbols to the particular version of the OS. However the operating system exports the function *ZwQuerySystemInformation*, which in turn calls *NtQuerySystemInformation*. By following this execution path, the attacker will find a call to *ExpQueryModuleInformation*. The first parameter to this function is the address of PsLoadedModuleList. With a bit of effort, an attacker now has everything she needs to hide her malicious process and her duplicitous driver.

IV. DETECTING HIDDEN PROCESSES

Our method for detecting hidden processes, whether running on the Windows or Linux operating system, requires the examination of each thread to ensure that its corresponding process descriptor is appropriately linked. Accordingly, both require added functionality to the operating system.

Hunt and Brubacher introduce *Detours* [18], a library for intercepting arbitrary Win32 binary functions, as a means of instrumenting and extending application and operating system functionality in the Windows environment. Detours replaces the first few instructions of a *target function* with an unconditional jump to a user provided *detour function*. The instructions from the target (original) function are preserved in a *trampoline function*. Figure 5 shows the temporal ordering of the detour, trampoline and target functions.

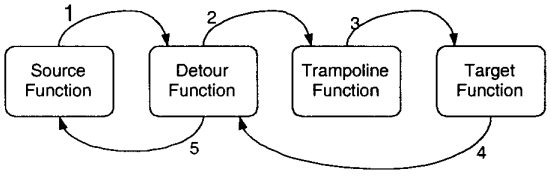

Fig. 5. Temporal Ordering of a Detour Function

It is important to note that Detours are inserted at execution time, where the code of the target function is modified in memory – not on disk.

In Windows, context switching is carried out by a call to *SwapContext* which is a function in *ntoskrnl.exe*. The first seven

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

bytes of SwapContext are:

```
OR CL, CL
MOV byte ptr ES:[ESI+2D], 02
```

Accordingly, when SwapContext has been called, the value contained in the *EDI* register is a pointer to the next thread to be swapped in and the value contained in the *ESI* register is a pointer to the current thread, which is be swapped out. In our detection method, we replace the seven bytes of SwapContext with a five byte unconditional jump to our Detour Function. The Detour Function verifies that the KTHREAD of the thread to be swapped in (referenced by the EDI register) points to a EPROCESS block that is appropriately linked to the doubly linked list of EPROCESS blocks.

Our method for detecting hidden process in Linux is similar to that for Windows. The *Injectso* [19] library provides functionality similar to Detours, but in the Linux environment. Specifically, *Injectso* is a tool that can be used to inject shared libraries into running processes on Linux and Solaris. In Linux, the kernel needs to be modified so that when a process' state is set to *Task_Running* and it is placed in the run queue by setting the *next_run* and *prev_run* pointers of its process descriptor, a test of its *next_task* and *prev_task* pointers is conducted to ensure that they are correctly linked.

Although our detection methods are sound and functional, they raise concerns and issues. The computer security situation today is akin to an *"arms race"* between *"good guys"* and *"bad guys"* with the hackers playing to role of bad guy. This arms race is made possible because neither Windows or Linux protects memory in kernel mode from other processes also running in kernel mode. Consequently, no matter what defense is constructed, it is generally only a matter of time before the other side constructs a counter measure. It should be underscored that the **Detours** and the **Injectso** libraries may be equally used for benign as well as malicious purposes, which only contributes to the *arms race* between attacker and defender.

William Wulf[20] equates our present computer security model to that of the the "Maginot Line" [1], emphasizing that this model does not work. Dr. Wulf further states that we need to abandon this model and establish a new one. Perhaps a new model should be developed that uses the *x86* architecture to its fullest potential.

The *x86* microprocessor architecture provides hardware support for four privilege modes of execution. Both Windows and Linux only make use of two of these modes. If operating system designers were to make use of all four modes, critical functionality could reside at the inner most layer and be protected from device drivers and LKMs [21]. An alternative is proposed by Arbaugh, et. al [22]. They present the *AEGIS* bootstrap protocol which systematically loads the BIOS, the operating system, and all device drivers from a trusted source. Although *AEGIS* is only concerned with loading trusted software, not testing for modification while it is in process, it could be extended to periodically test to ensure that operating system that is resident in memory has not been modified and remains consistent with the image that was loaded.

[1] The Maginot Line, named after Andre Maginot, the French Minister of War 1928 - 1932, was a series of defensive fortifications built by France along her border with Germany and Italy. It failed however, in 1940 the German army by passed the Maginot Line, entering France through a "neutral" third country and swiftly defeating her.

V. CONCLUSIONS AND FUTURE WORK

We have presented a novel class of attack, the hidden process, which executes in a manner that is transparent to a system's management and audit functions. We have demonstrated the feasibility of this class of intrusion by implementing a device driver to hide processes on the Windows operating system. Additionally, we have presented an analysis of LKMs with similar functionality for the Linux operating system. We have detailed a mechanism for a hidden process to surreptitiously piggyback onto an existing network connection, which enables communications between an attacker and a hidden process.

We have detailed and implemented a mechanism to detect hidden processes in both the Linux and Windows environments. We use the *Detours* library for the Windows detection mechanism. Our mechanism, however, operates with root level privileges implying that a malicious program that also operates at root level could subvert our detection mechanism.

To alleviate the *"arms race"* between attackers and computer security practitioners we suggest using the *x86* architecture to its fullest potential. By using all four hardware privilege modes of the *x86* architecture and by placing critical operating system components within the most secure level will prevent LKMs and device drivers from modifying an operating system consequently ensuring that it continues to function as specified.

We are continuing our research into detecting hidden processes and the channels that they might use to communicate with an attacker. As many intrusion detection system employ *learning* algorithms to distinguish between intrusive and benign events we are exploring the possibility of employing *Analytical Learning* as an alternative to inductively derived rule sets as the basis for a host based anomaly detection engine. According to Mitchell [23] the inputs to an analytical learning algorithm are the same as with inductive learning however an additional input referred to as *domain theory* consisting of background knowledge is used to explain observed training examples. The output is a rule set consistent with both the training data and the domain theory. The notion of placing the responsibility for intrusion detection onto a trusted third party mitigates the situation caused by the previously mentioned *arms race*.

REFERENCES

[1] CERT/CC, "Cert/cc statistics 1988-2002," http://www.cert.org/, October 2002.

[2] Frank Kargle, Joerm Maier, and Michael Weber, "Protecting web servers from distributed denial of service attacks," *Proceeding of the Tenth International World Wide Web Conference*, 2001.

[3] Dorothy E. Denning, *Information Warfare and Security*, Addison Wesley, 2001.

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

[4] Jeffrey Undercoffer and John Pinkston, "Modeling computer attacks: A target-centric ontology for intrusion detection," Tech. Rep. TR CS 02-169, University of Maryland, Baltimore County, December 2002.
[5] Paul J. Criscuolo, "Distributed denial of service," Tech. Rep. CIAC-2319, Lawrence Livermore National Laboratory, February 2000.
[6] Bob Toxen, *Real World Linux Security*, Prentice Hall, 1999.
[7] Lee Sutterfield, "Annual computer security conference and exhibition," http://www.press.securelogic.com/events.html, Speech by Lee Sutterfield to the Computer Security Institute.
[8] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas Longstaff, "A sense of self for unix processes," in *1996 IEEE Symposium on Security and Privacy*. 1996, pp. 120 – 128, IEEE Computer Society Press.
[9] Anil Somayaji and Stephanie Forrest, "Automated response using system-call delays," in *Proccedings of the Ninth USENIX Security Symposium*, August 2000, pp. 120 – 128.
[10] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, "Detecting intrusion using system calls: Alternative data models," in *1999 IEEE Symposium on Security and Privacy*. 1999, pp. 133 – 145, IEEE Computer Society Press.
[11] Frank Apap, Andrew Honig, Shlomo Hershkop, Eleazar Eskin, and Sal Stolfo, "Detecting malicious software by monitoring anomalous windows registry accesses," in *Fifth International Symposium - Recent Advances in Intrusion Detection*, A. Wespi, G. Vigna, and L. Deri, Eds. 2002, number 2516 in LNCS, pp. 37 – 53, Springer.
[12] Nelson Murilo and Klaus Steding-Jessen, "chkrootkit version 0.37," www.chkrootkit.org, September 2002.
[13] FuSyS, "Kstat," http://s0ftpj.org/en/site.html, KSTAT for Linux.
[14] Stealth, "Adore," http://teso.scene.at/releases/adore-0.42.tgz, The Adore Root Kit.
[15] Dark-Angel, "Phantasmagoria," http://online.securityfocus.com/archive/1/290724.
[16] Gene H. Kim and Eugene H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *ACM Conference on Computer and Communications Security*, 1994, pp. 18–29.
[17] W. Richard Steves, *Unix Network Programming: Networking APIs: Sockets and XTI*, vol. One, Prentice Hall,, second edition, 1998.
[18] Galen Hunt and Doug Brubacher, "Detours: Binary interception of win32 fucntions," in *Proceedings of the Third USENIX Windows NT Symposium*, 1999.
[19] Shaun Clowes, "Injectso 0.2," http://www.securereality.com.au, 2002, Tool for injecting shared libraries into running processes.
[20] William Wulf, "Statement before the House Science Committee, U.S. House of Representatives," October 10 2001.
[21] James R. Butler, "Detecting compromises of core subsystems and kernel fucntions in windows nt/2000/xp," M.S. thesis, University of Maryland, Baltimore County, 2002.
[22] William A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy*, 1997, pp. 65 – 71.
[23] Tom Mitchell, *Machine Learning*, WCB McGraw-Hill, 1997.

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE