Detecting and Countering System Intrusions Using Software Wrappers

Calvin Ko
Timothy Fraser
Lee Badger
Douglas Kilpatrick
NAI Labs, Network Associates, Inc.
{*calvin_ko, tfraser, lee_badger, douglas_kilpatrick*}*@nai.com*

Abstract

This paper introduces an approach that integrates intrusion detection (ID) techniques with software wrapping technology to enhance a system's ability to defend against intrusions. In particular, we employ the NAI Labs Generic Software Wrapper Toolkit to implement all or part of an intrusion detection system as ID wrappers. An ID wrapper is a software layer dynamically inserted into the kernel that can selectively intercept and analyze system calls performed by processes as well as respond to intrusive events. We have implemented several ID wrappers that employ three different major intrusion detection techniques. Also, we have combined different ID techniques by composing ID wrappers at run-time. We tested the individual and composed ID wrappers using several existing attacks and measured their impact on observed application performance. We conclude that intrusion detection algorithms can be easily encoded as wrappers that perform efficiently inside the kernel. Also, kernel-resident ID wrappers can be easily managed, allowing cooperation among multiple combined techniques to enforce a coherent global ID policy. In addition, intrusion detection algorithms can benefit from the extra data made accessible by wrappers.

1 Introduction

Intrusion detection is a retrofit approach to enhancing the security of computer systems. It utilizes various audit data to identify activities that could compromise the security of a system. Traditionally, intrusion detection systems (IDS) are user-space applications that utilize audit data generated by audit systems (e.g., Solaris Basic Security Module (BSM)) or network sniffers to detect intrusive activities. The capabilities of these user-space IDSs are restricted by the quality of the audit data and the services provided by the operating systems. For instance, audit systems do not provide all the data required by IDSs, thus limiting the attacks that can be detected by the IDSs. In addition, audit systems offer rudimentary methods for selecting data to be logged. In particular, most audit systems do not support selection of a particular program to audit. Also, as the data is generated in the kernel, every time a system action has to be logged or analyzed, the information has to be transferred from kernel space to user space, causing a context switch, and increasing the load imposed on the system by the IDS. Thus, user-space IDSs suffer from high overheads and low efficiency, as well as long delay (in CPU cycles) in detecting intrusions. Lastly, user-space IDSs are not sufficiently protected by operating systems and cannot completely protect themselves.

Our goal is to integrate ID functions into the kernel to remedy some of the problems arise in user-space intrusion detection. Specifically, we exploit the execution environment provided by Generic Software Wrappers [4] to enhance the intrusion detection and response capability of a system. An ID logic implemented as an ID wrapper can 1) selectively examine any parameters of system calls and the entire system state, 2) analyze a system call before or immediately after the call is executed, 3) analyze system calls inside the kernel, thus avoiding the overhead of transferring audit data from kernel space to user space, and 4) protect itself by denying intrusive operations.

*This research was supported by the Defense Advanced Research Projects Agency under contract F30602-96-C0333.

We have implemented several intrusion detection techniques, tested the ID wrappers using several existing attacks, and measured the performance of the ID wrappers. Our conclusion is that intrusion detection algorithms can be easily encoded as wrappers that perform efficiently inside the kernel. Also, ID wrappers can be configured and managed easily to support a coherent global intrusion detection and response policy. We envision that ID wrappers can be used individually to protect a system or as components of a large-scale intrusion detection system.

The rest of the paper is organized as follows. Section 2 presents an overview of ID wrappers, focusing on the capability of ID wrappers provided by the Generic Software Wrapper Toolkit and our extensions to the toolkit for supporting intrusion detection. In section 3, we present how we implement various ID techniques—specification-based, signature-based, and sequence-based techniques—using wrappers. In section 4, we present our experiments for testing ID wrappers with simulated attacks. We also describe a composition experiment in which two ID wrappers employing two different techniques cooperate with another abstract wrapper that combines the findings of the two ID wrappers. In addition, we present the performance results of the ID wrappers, showing that intrusion detection functions can be executed, managed, and coordinated in the kernel with a minimal observed application performance penalty. Section 5 discusses related work. In section 6, we discuss the pros and cons of the kernel-resident intrusion detection approach as well as our experience in realizing this approach using Generic Software Wrappers. Section 7 provides the conclusion and suggests future research.

2 Intrusion Detection Wrappers

This section presents the architecture of ID wrappers. It describes the capability of ID wrappers naturally provided by the Generic Software Wrapper Toolkit and our extensions to the toolkit for supporting intrusion detection.

Figure 1 gives a high-level view of an ID wrapper. An ID wrapper is a state machine that is bound dynamically to a program in execution and that gains control when system calls are invoked. Multiple ID wrappers may be bound concurrently to a single program in order to combine multiple ID techniques or to collaborate in the enforcement of a single policy. An ID wrapper is specified using the Wrapper Definition Language (WDL)[10], a superset of C language. WDL supports high-level specification of the events to be intercepted and accesses to parameters of the intercepted system call. WDL also hides specific details of different operating systems so that generic wrappers that run on multiple platforms can be written. An ID wrapper specified in WDL is compiled by the Wrapper Compiler (WrapC) into native object code of the destination platform for deployment. Currently, the wrapper toolkit supports FreeBSD, Solaris, Linux, and Windows NT[1]. ID wrapper capabilities, deriving from WDL features, fall naturally into two groupings:

Event Interception Criteria: An ID wrapper specifies events that it intercepts. Such events may be system calls or more "abstract" events defined and generated by other wrappers. An ID wrapper will listen to events that represent steps in attack specifications [5, 9], events defining (or deviating from) behavioral profiles [3, 8], events that attempt to subvert the intrusion detection system, or events that access system resources after a successful attack sequence. Events may contain parameters, and an ID wrapper may condition the interception of the events based on pre-established groupings (e.g., *open*, *close*, *read*, *write* are all "file" events), parameter value matching, global system state, and event sequence relationships (e.g., event e_1 that occur before event e_2 will be "listened for").

Actions: When an event is intercepted, an ID wrapper may take a variety of actions. In general, these actions serve to deny, transform, or augment the event, and perhaps also to generate new events that can be intercepted by other active wrappers. For intrusion detection and response purposes, an action will often be to update an intrusion detection model or fact base, to determine if any misuse rules have completed or if the current behavior exceeded the defined bounds in the normal profile, and to take countermeasures if an intrusion is imminent. Such countermeasures at least will protect the intrusion detection system from tampering, but also can include a variety of techniques that prevent damage, de-

[1]The NT prototype has a different architecture from Unix prototypes. It employs library hooking techniques to intercept the Win32 API calls performed by processes.

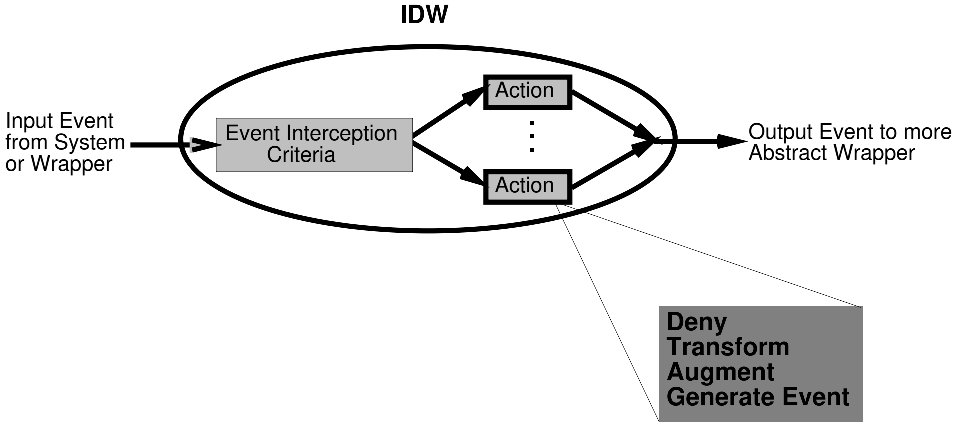

Figure 1: Intrusion Detection Wrapper Structure

ceive the intruder, or collect additional information for subsequent legal or military action. At the implementation level, ID wrapper capabilities derive from WDL facilities that support convenient access to (and modification of) event parameters, access to local environment variables and global system state, generation of new events, and access to lightweight DBMS services.

2.1 Management and Composition

ID wrappers need to be properly managed and configured to offer the best protection to a system. Depending on the overall ID policy, some ID wrappers should wrap every process while other ID wrapper should wrap only certain critical processes. The Wrapper Support Subsystem (WSS) provides support for configuration and management of ID wrappers. To use an ID wrapper, an administrator first registers the wrapper with the WSS through a loading process, which dynamically inserts the run-time image of the wrapper into the kernel. Selection of processes for wrapping is controlled by activation (or deactivation) criteria which specify when a loaded wrapper should begin (or cease) to wrap a process. The activation criteria language allows specifications based on the invoker, the program name, and attributes of the executable. The WSS tracks running processes and evaluates the activate criteria to activate wrappers to wrap processes that satisfy the criteria. Therefore, ID wrappers can be configured and administered easily in our framework to enforce a coherent ID policy.

The whole problem of intrusion detection is beyond the capability of any one intrusion detection system or ID technique [6]. Therefore, cooperation of different ID techniques is required to enhance the protection of a system. To combine multiple ID techniques, it is often convenient to implement each ID technique in a separate, independent ID wrapper and to run processes under the simultaneous control of multiple ID wrappers. Additionally, it is highly desirable to have ID wrappers that are aware of one another to support hierarchies of increasingly abstract wrappers. For example, one ID wrapper can listen to system calls to generate abstract system independent audit events to be consumed by a more abstract ID wrapper that analyzes the abstract audit events. Figure 2 shows the two fundamental forms of composition:

Layered Composition: Multiple ID wrappers intercept an event (e.g., a system call) and perform some actions. In this case, the actions of the wrappers will be executed in the order in which the wrappers were installed on the system. Figure 2a illustrates the ordering for layered composition. In layered composition, the wrappers involved in the composition might not be aware of the composition occurring. This type of layering could be compared to an onion, in which the user's request must travel down through the "layers" of wrappers to get to the system call; the return value must travel back out through the "layers" to reach the API again.

Active Composition: ID Wrappers generate events intercepted by other ID wrappers (out-

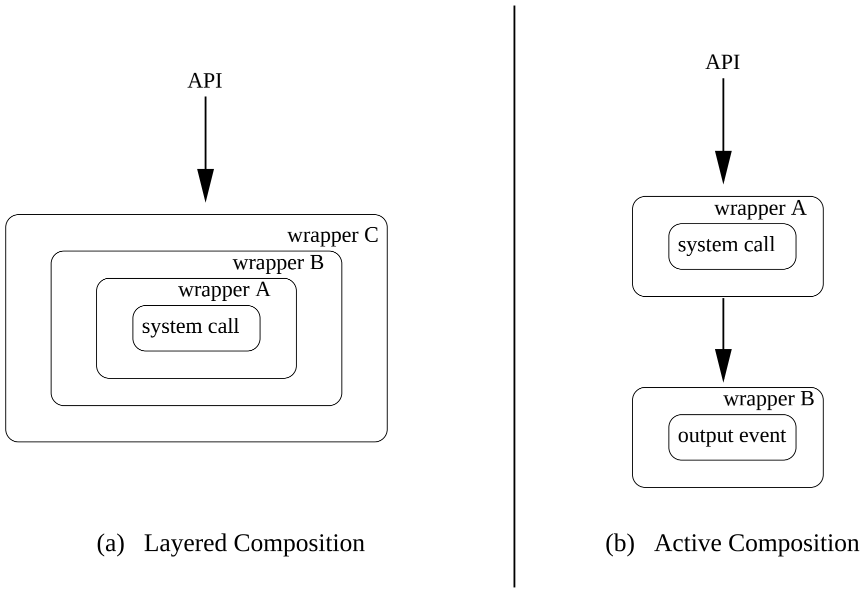

Figure 2: Wrapper Composition

put events), shown in figure 2b. Output events represent active composition, in which the wrappers generating the events are aware of the possible communication/coordination with other wrappers. In this instance, a ID wrapper generates an output event to be intercepted by another, usually more abstract, ID wrapper. The more abstract wrapper will return to the calling wrapper; control passes through the calling wrapper to the system call.

The two forms of composition are not mutually exclusive: a system event could be intercepted by layers of ID wrappers, some of which could generate output events to be intercepted by other ID wrappers. The composition facility is flexible enough to allows ID wrappers to cooperate in the manners (e.g., complement or reinforce each other's findings) described by in Common Intrusion Detection Framework [6].

2.2 Obtaining System State Information

In addition to the parameters of the intercepted system calls, ID wrappers may need to access system state to acquire additional data for its analysis. For example, the owner, group, and permission mode of a file being accessed may be required to determine whether this file access deviates from a specified valid behavior profile.

An ID support module has been added to the Generic Software Wrappers toolkit to provide a set of library functions for ID wrappers. Table 1 enumerates the library functions that have been implemented. Additional functions can be implemented and added to the system easily.

2.3 Dispatching Audit Data to User-Space IDSs

In a large-scale IDS, an ID wrapper may be used as a data-collection component that collects security-relevant data for intrusion analysis engines running in user space. Such scenario requires a very efficient mechanism for transferring a large amount of data from wrappers running in kernel space to user processes in a secure fashion. In addition, such mechanism should allow multiple intrusion detection systems to listen to the audit event data generated by possibly different ID wrappers.

An audit event handler providing support for dispatching audit data to user processes is incorporated into the basic wrapper toolkit. An intrusion detection engine cooperating with an ID wrapper

Name	Function
wr_stat	obtain status of the file specified by a path
wr_fstat	obtain status of the file specified by a file descriptor
wr_audit	delivery audit data to the audit event handler
wr_audit_printf	same as wr_audit, but with the printf interface
wr_get_addr	obtain the socket address of a socket specified by a file descriptor
wr_getpeername	get the name of the peer of a connection
wr_getsockname	get the name of the local entity of a connection

Table 1: Wrapper Library functions to support Intrusion Detection

can register with the audit event handler for the event queue to which it wants to listen. When the cooperating ID wrapper collects relevant audit data and sends it to the audit event queue, the audit event handler dispatches the data to the registered intrusion detection engine.

In this approach, the IDS thread calls a registered system call to register for some number of audit queues. The system call creates a pipe and returns the read end of the pipe. The IDS thread performs a select system call on the read end of the pipe, effectively blocking the process. The event handler writes the entire event structure for each audit event to the write end of the pipe. This method can promptly transfer events from the event handler to the waiting thread in a thread-safe manner and with little overhead.

3 Implementation

This section presents our experience in the implementation of various intrusion detection techniques using ID wrappers.

3.1 Specification-based Techniques

Specification-based intrusion detection systems employ specifications that describe the valid behavior of programs to detect intrusions. In particular, programs in execution are monitored for violations from the corresponding valid behavior specifications. One useful type of specifications is the set of valid operations of a program [7]. We have encoded the specifications for several programs (e.g., *imapd, fingerd, lpr, lprm, ftpd, httpd*, etc.) describing their valid operations as ID wrappers using WDL. Each specification-based ID wrapper is configured, using the activation criteria, to wrap the execution of the program with which the specification is concerned. We found that the WDL itself is very suitable for expressing the set of valid operations of a program.

Figure 3 shows part of a specification of the *imapd* program, encoded in WDL, that is concerned with the valid parameter values of *open-read*, *chmod*, *fchmod*, and *execve* operations.

The first clause specifies that after a successful open system call with the read-only flag on (lines 1-2), the action block (lines 3-7) will be executed. The action block obtains the inode information of the opened file and checks whether 1) it is world-readable, 2) owned by the invoker, or 3) created by the program execution (checked by the local function created()). It raises a violation if all conditions are false. In short, the first clause detects any bad open-read operation: on a file that is neither publicly readable, owned by the invoker, nor created by the program execution itself. Similarly the second and third clauses raise a violation if the program performs a *chmod/fchmod* operation on a file not created by the program execution. The last clause specifies that the wrapper intercepts the execve system call before it executes, issues a violation, and prohibits the call by returning an error code to the caller immediately (`return WR_D_BADPERM`).

The partial specification illustrates that criteria for event interception can be specified very easily in WDL. In addition, accesses to system call parameters can be accomplished easily through special \$ variables. For example, the \$path variable on line 10 denotes the path name of the file in the *chmod* system call. A reference or assignment to a variable effectively reads/modifies the corresponding argument of the intercepted system call. WDL

```
bsd::op{open}
($errno == 0 && $flags | O_RDONLY != 0) post {
    struct wr_stat s;
    wr_fstat($fdret, &s);
    if (!WorldReadable(s) && Owner(s) != _uid && !created(s.nodeid))
         violation();
};
bsd::op{chmod} pre {
   struct wr_stat s;
   wr_stat($path, &s);
   if (!created(s.nodeid))
        violation();
};
bsd::op{fchmod} pre {
   struct wr_stat s;
   wr_fstat($fd, &s);
   if (!created(s.nodeid))
        violation();
};
bsd::op{execve} pre {
   violation();
   return WR_D_BADPERM;
};
```

Figure 3: Partial imapd program behavior specification in WDL

also handles the copying of argument data between user space and kernel space automatically, allowing wrapper developers to focus on the key aspects of a wrapper instead of low-level programming details.

With ID wrappers, we can monitor programs for improper modifications to objects that otherwise cannot be accomplished using traditional audit trails. In particular, ID wrappers can examine data read/written to specific files without a huge overhead. Using this capability, we wrote an ID wrapper that examines the *passwd* program to ensure that when a user (say Joe) invokes the *passwd* program, the program modifies only the part of the password file associated with the password of Joe. If there is a vulnerability in the *passwd* program that allows the attack to control the program to arbitrarily modify the password file (e.g., changing the user ID of a user), this ID wrapper is able to detect such an attack.

3.2 Signature-based Techniques

Signature-based ID systems detect intrusions by observing events and identifying patterns which match the signatures of known attacks. An attack signature defines the essential events required to perform the attack, and the order in which they must be performed. Different ID systems represent signatures in different ways. The State Transition Analysis Tool (STAT) [5], for example, represents signatures with state transition diagrams. During runtime, these diagrams direct the operation of finite state machines that represent possible intrusions in progress. The STAT system advances these state machines from state to state as it observes events that match parts of attack signatures. If the STAT system observes a sequence of events that ultimately moves one of these finite state machines to its final state, the STAT system declares that it has detected an intrusion.

We have implemented the `Mailstat` wrapper, an example of STAT-like ID which attempts to detect a well-known attack on a commonly-used UNIX mail daemon. The signature of this mail daemon attack is effectively hard-coded in the structure of the Mailstat wrapper. When deployed, the `Mailstat` wrapper wraps all processes on the system, and intercepts and examines every system call that might correspond to an event in the mail daemon attack signature. It uses a database table to store the state of the finite state machines representing possible attacks in progress. Whenever `Mailstat` observes a system call that matches the first event in the mail daemon attack signature, it creates a new finite state machine by adding a new line to the table. As it intercepts system calls and observes events, it advances the state of the appropriate finite state machines according to the mail daemon attack signature's state transition diagram. When any finite state machine in the table reaches its final state, the `Mailstat` wrapper indicates an intrusion and reports the identities of the processes which caused the events leading to its detection.

3.3 Sequence-based Techniques

The sequence-based intrusion detection approach by Forrest [3] calculates an anomaly value for a program execution based on the number of sequences the program generates that are missed in a precomputed database of sequences. The technique has been found to be effective under offline evaluation using audit data collected from different environments. It requires properly-constructed norma sensitive to program versions and configuration, and can in some cases require significant processing resources to perform anomaly calculation in real time. We have structured `Seq_id`, our sequence-based ID wrapper, to address these issues.

`Seq_id` runs in two modes: record mode and detect mode. In record mode, `Seq_id` automatically generates a normative sequence database for each program executed. Using `Seq_id`, we have generated a per-program database for every program executed on our test machines. To increase efficiency and simplicity, we have slightly modified the algorithm described in [2] to merge some sequences, which would remain unique in the original technique. Initial comparison tests between the two algorithms indicate that the detection accuracy is similar. In detect mode, `Seq_id` decides if each observed system call completes a sequence stored in the program's database of normal behavior. `Seq_id` measures the magnitude of each deviation, and reports those of sufficient magnitude.

4 Experiments and Performance Measurement

To evaluate the intrusion detection wrappers with respect to their ability to detect attacks, we tested the ID wrappers with several existing attacks. These attacks exploit vulnerabilities in security-critical programs that possess privileges to obtain a shell running as root. We describe the programs and the attacks below.

imapd Some versions of the Internet Mail Access Protocol (IMAP) server contain a number of buffer-overflow bugs that allow a remote user to obtain a shell running as root (CERT Advisory CA-97.09). We obtained an exploitation script to one of the bugs from RootShell (www.rootshell.com). The exploit script carefully crafts the input to *imapd* that exceeds the size of a special stack buffer and presents the name to the IMAP server to overwrite the saved instruction pointer and execute the planted machine code. The code then executes a shell running with root. We wrapped *imapd* using a specification-based ID wrapper `Imapd_id` specific to *imapd* and a sequence-based ID wrapper separately. Both wrappers were able to detect the exploit script's attack.

lpr Due to insufficient bounds checking on arguments which are supplied by users, it is possible to overwrite the internal stack space of some versions of the *lpr* program while it is executing. This can allow an intruder to cause *lpr* to execute arbitrary commands by supplying a carefully designed argument to *lpr* (AUSCERT Advisory AA-96.12). These commands will be run with the privileges of the *lpr* program. When lpr is setuid root it may allow intruders to run arbitrary commands with root privileges. We simulated the attack using a script from RootShell. We wrapped lpr using a specification-based wrapper tailored for lpr and the wrapper was able to detect the attack.

lprm The program *lprm* is part of the printing subsystem. The program is used to remove a job in the printer queue. There is a buffer-overflow vulnerability in some versions of this program that allows a local user to execute arbitrary commands with root privileges. We obtained a script from Security Bugware (http://161.53.42.3/c̃rv/security/bugs/list.html) and tested a specification-based wrapper written for *lprm* with the script. The specification-based wrapper detected the attack when *lprm* was tricked to execute the Bourne shell.

binmail The *binmail* program is the back-end mailer that delivers mail messages to

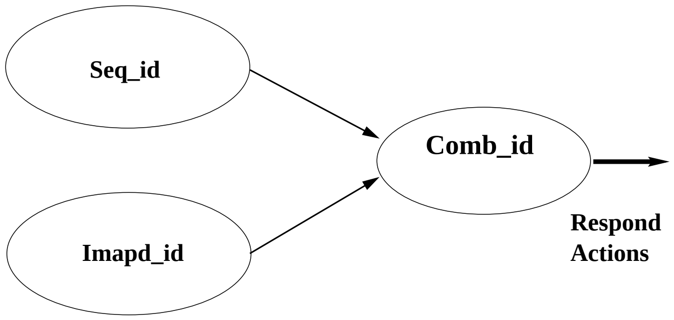

Figure 4: Composing Two ID techniques

users' mailboxes. It does so by appending the messages to the mailbox files directly. In some old versions, *binmail* changes the ownership of a user's mailbox (usually `/var/spool/mail/<username>`) back to the user after it appends a message if the mailbox file is not owned by the user initially. In particular, the *binmail* program (`/bin/mail`) in 4.2 BSD Unix fails to reset the setuid bit of the mailbox file after it appends a message and changes the owner of the file [5]. An attacker, who creates a mailbox file with the setuid bit on for the superuser, can trick *binmail* into making the file to be setuid root by invoking *binmail* to send a mail message to root. We deployed the `Mailstat` wrapper and tested the wrapper with an exploitation script we created. The wrapper detected the intrusion immediately.

4.1 Combining Multiple Techniques using Composition

Our wrapper frameworks allow multiple ID wrappers to cooperate to enhance their performance. The Common Intrusion Detection Framework [6] discusses several ways ID components cooperate with each other. We performed an experiment in which two ID wrappers cooperates to reinforce each others findings. Figure 4 depicts the configuration. A sequence-based wrapper and a specification-based wrapper are used to wrap the *imapd* programs. Every system call performed by *imapd* will be intercepted by both wrappers (The order will be determined by the loading sequence). Each wrapper will analyze the operations of *imapd* individually and generate an abstract warning event to the abstract wrapper (Com_id) when they find an attack. The abstract wrapper judges the output from both *Seq_id* and *Imapd_id* and accepts it when both ID wrappers think the program is under an attack. In this case it will kill the process.

We tested the composite ID wrappers using the *imapd* attack described in the last subsection. An interesting observation is that the two wrappers detect the attack at different system call. The `Imapd_id` detected the attack when the program executes a Bourne shell (at the *execve* system call). The `Seq_id` detected the attack several system calls after the *execve* system call. The abstract ID wrapper `Com_id` killed the process after it receives warning from both wrappers. Potentially, such configuration could reduce the false positive rate as the whole IDS will detect a false attack when both techniques produce a false positive. However, it could also cause some attacks to escape the detection if only one technique detects the attack. Thus, further research is needed to determine how to best combine different techniques.

4.2 Performance

We have studied the performance of the intrusion detection wrappers. We measured the overhead caused by the intrusion detection wrappers on the running time of programs using a Kernel Build test, in which the time taken to compile a Generic version of the FreeBSD kernel was measured. Also, we measured the overhead caused by ID wrappers from a user's perspective, in terms of latency and throughput, for a Web server and a FTP server.

	Average Kernel Build Time			Average HTTP Latency			Average HTTP Throughput		
	time (s)	σ (s)	penalty	time (s)	σ (s)	penalty	t-put (Mbits/s)	σ (Mbits/s)	penalty
no WSS	583.43	0.53	0%	0.657	0.025	0%	7.455	0.21	0%
WSS only	604.38	0.46	3.47%	0.652	0.0023	-0.081%	7.456	0.08	0.01%
Seq_id	624.62	1.23	6.59%	0.687	0.017	4.52%	7.038	0.164	5.61%
Http_id	-	-	-	0.705	0.0247	7.38%	6.928	0.157	7.08%
Http_id & Seq_id	-	-	-	0.744	0.018	13.26%	6.607	0.127	11.39%

Table 2: FreeBSD Prototype Performance for Kernel Build and Web Server Benchmarks

	Average FTP Latency			Average FTP Throughput		
	time (s)	σ (s)	penalty	t-put (Mbits/s)	σ (Mbits/s)	penalty
no WSS	28.2418	0.9019	0%	8.776	0.093	0%
WSS only	28.3332	1.0773	0.32%	8.768	0.069	0.09%
Seq_id	28.30125	1.0835	0.21%	8.743	0.076	0.38%
Ftpd_id	28.3592	0.7954	0.42%	8.756	0.085	0.23%
Ftpd_id & Seq_id	27.9224	1.2007	-1.13%	8.573	0.012	2.31%

Table 3: FreeBSD Prototype Performance for FTP Server Benchmarks

Table 2 summarizes the results of the performance tests for Kernel Build and for a Web server. The first column shows the average time taken to compile the FreeBSD kernel 1) under normal conditions, 2) with the WSS loaded into the kernel, and 3) with the sequence-based intrusion detection wrapper `Seq_id` wrapping the compilation process. The second and third columns of the table contain results for a custom-made Web server benchmark. The Average HTTP Latency column describes the delay a Web client experiences between the moment it makes a request and the moment it receives the Web server reply. The Average HTTP Throughput describes the rate at which the Web server returns data to the Web clients. We measured the latency and throughput of the Web server when the Web server is wrapped by Seq_id, Http_id, and both Seq_id and Http_id. The results were produced by a custom-made Web server benchmark executed with an Apache 1.3.0 Web server and the WebStone 2.0.1 benchmarking software. The Apache Web server ran on a 166MHz Intel Pentium-based microcomputer with 32MB RAM running a Generic FreeBSD 2.2.2 kernel. Two Pentium 400MHz machines were used to run 32 WebStone 2.0.1 Web clients through a series of 10 15-minute trials using the standard WebStone 2.0.1 file set for each row in table 2.

Table 3 shows the results of the performance tests using a custom-made FTP server benchmark. The Average FTP Latency column describes the delay a FTP client experiences between the moment it makes a anonymous request and the moment it receives all the data from the server. The Average FTP Throughput describes the rate at which the FTP server returns data to the FTP clients. The table denotes the latency and throughput of the FTP server under controls to Seq_id, Ftp_id, and both Seq_id and Ftp_id. Ftp_id is a specification-based wrapper that restricts the operations that can be performed by the FTP server. The FTP server (ftpd) in the FreeBSD 2.2.2 distribution was used in the tests. The Average FTP Latency and Average FTP Throughput results were obtained in a smiliar manner to the HTTP results using a modified WebStone software that performs anonymous FTP fetches instead of HTTP fetches.

The performance results show that WSS alone imposes 3-4% penalty on the compilation time of the FreeBSD kernel. `Seq_id` adds another 3-4% to the compilation time of the FreeBSD kernel. Impact caused by WSS on the latency and throughput of a Web/FTP server is minimal, possibly because WSS only intercept the *fork*, *execve*, and *exit* system calls, which are used infrequently in a Web/FTP server.

The sequence-based wrapper and the specification-based wrappers impose approximately 5-7 % overhead on the Web/FTP server, and their impacts add up when they are used together. While we have designed the wrapper toolkit and ID wrappers with consideration for performance, we have not optimized the prototype; therefore, performance can possibly be improved.

5 Related Work

Balasubramaniyan et. al. [1] have proposed the use of autonomous agents for intrusion detection. They have developed an architecture for the autonomous ID agents. Our idea is similar to their agent idea in that ID wrappers can be viewed as kernel-resident ID agents. Their conjecture is that the performance of the agents can be improved if they are implemented inside the kernel. Our results support their conjecture; in particular, kernel-resident agents can be very efficient and impose very little performance penalty on a system.

Sekar et. al. [11] have devised an efficient method of implementing a form of specification-based intrusion detection in the kernel. Some of the implementation strategies employed by Sekar's method are similar to those we have employed in ID wrappers. For example, both efforts associate individual kernel-resident state machines ("wrappers," in our terminology) with each application process under observation, using interposition techniques at the operating system's system call interface to enable these kernel-resident state machines to observe application process behavior at a fine-grained level of detail. Sekar's effort concentrates on the efficient implementation of a single form of specification-based intrusion detection, and has achieved a result which allows the intrusion detection system to handle multiple patterns with the same low overhead as a single pattern. Our effort, in contrast, has sought to produce a general framework for the implementation of multiple intrusion detection algorithms, as well as a convenient means for managing their simultaneous deployment and composition. Both our effort and Sekar's have observed favorably low overheads in terms of observed application performance degradation due to the use of kernel-resident intrusion detection. Sekar's technique resulted in overheads of no more than 1.5% in *ftpd*, *telnetd*, and *httpd* benchmarks documented in [2].

6 Discussion

The idea of moving intrusion detection functions into the kernel is not new and has been hinted at in the literature. Balasubramaniyan et. al. discussed the advantages and disadvantages of integrating intrusion detection agents inside the kernel [1].

In-kernel intrusion detection has several advantages. First, overhead due to extra context switching is avoided – a system call is analyzed by the ID logic at the same kernel context at which the system call executes. In addition, information is registered and processed at or near the place where it is produced, reducing the time and resources for transferring the information to the analysis engine. Also, this proximity allows prompt detection and reduces the possibility of the information being modified by an attacker before it gets to the ID analysis engine. Lastly, it is harder for an intruder to tamper with the ID system as the attacker would have to modify the kernel (e.g., by defeating the kernel's memory-protection mechanism).

However, the kernel-resident implementation strategy also has its disadvantages. First, kernel-resident ID systems are not portable across platforms. Second, a misbehaving ID system can do much more damage if it is running in the kernel rather than in user space because it has full access to the system. Third, entities in the kernel can have a large impact in the host behavior by slowing down fundamental operations (e.g., kernel data structures, accesses to memory, disk). Fourth, entities inside the kernel are very difficult to manage and configure. Finally, kernel programming is at a low level of abstraction, where the resources available provide very limited functionality when compared to the higher-level abstractions available in user space.

Our work essentially illustrates that in-kernel intrusion detection is feasible and practical provided that the kernel-resident ID system is designed and coded carefully. With minimal adjustment, many intrusion detection techniques can be implemented to run inside the kernel efficiently without impacting the host behavior. By using the Generic Software Wrapper Toolkit as the basis for implementing kernel-resident intrusion detectors, our approach inherits the advantages of in-kernel intrusion detection while avoiding the problems of portability, manageability, and the low-level nature of kernel programming. We strongly believe that further investigation of in-kernel intrusion detection is worthwhile and necessary.

7 Conclusion and Future Work

We have described our effort to enhances IDS capability by exploiting the execution environment offered by software wrappers. In order to take advantage of the potential for increased functionality and performance in kernel-resident intrusion detection systems, we have begun the development of a Generic Software Wrapper-based ID support framework, and have explored this framework's ability to ease the implementation, management and simultaneous composed deployment of three major intrusion-detection algorithms. We have described our ID-support extensions to the basic Generic Software Wrapper Toolkit, and how these extensions eased the implementation of our prototype ID wrappers. Based on our experience and the results of our performance benchmarks, we predict that many ID techniques can be efficiently implemented as kernel-resident wrappers. In all of our benchmarks, the overall observed application performance penalty associated with the use of our ID wrappers never exceeded 7.4%.

In addition to increased efficiency, ID wrappers derive several other benefits from their kernel-resident Generic Software Wrapper-based implementation. First, the interposition capability of the wrappers system provides ID wrappers with a greater range of fine-grained event data than is available to user-space techniques which must rely upon log-based audit data. All system calls and their parameters are visible to ID wrappers. Second, this interposition capability and the generality of the C-based wrapper implementation language allows wrappers to respond to intrusive events as they occur, with a broad range of response functionality. Finally, using the wrapper framework, kernel-resident ID components can be configured and managed easily to enforce a global ID policy and possibly to interoperate with large scale IDS running in user space.

Our most promising direction for future research concerns the composition of multiple intrusion detection wrappers at run-time. The ability to simultaneously apply multiple complimentary intrusion detection techniques to the same event stream appears to present a potential means of providing more accurate detection. Another promising direction involves utilization of wrapper's ability to examine data read/written to specific files or connection endpoints (e.g., sockets) to detect attacks that cannot be spotted by just looking at parameters of system calls. Other directions include cooperation with large-scale intrusion detection systems, the development of distributed ID wrappers, and efforts to improve the trust-worthiness and safety of the kernel-resident ID module.

References

[1] J. Balasubramaniyan et al. An Architecture for Intrusion Detection using Autonomous Agents. Technical Report TR-98-05, Department of Computer Science, Purdue University, June 1998.

[2] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX) 2000*, 2:84–99, 2000.

[3] S. Forrest et al. A sense of self for unix processes. In *Proceedings of the 1996 Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 6-8 1996.

[4] T. Fraser, L. Badger, and Feldman M. Hardening COTS software with generic software wappers. In *Proceedings of the 1999 Symposium on Security and Privacy*, 1999.

[5] K. Ilgun, R. Kermmerer, and P. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[6] C. Kahn et al. A common intrusion detection framework. *Journal of Computer Security*, 1998.

[7] C. Ko, G. Fink, and K. Levitt. Automated Detection of Vulnerabilities in Privileged Programs Using Execution Monitoring. In *Proceedings of the 10th Computer Security Application Conference*, Orlando, FL, December 5-9, 1994.

[8] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, California, 1997.

[9] S. Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Department of Computer Science, Purdue University, August 1995.

[10] Karen Oostendorp, Christopher Vance, Kelly Djahandari, Benjamin Uecker, and Lee Badger. Preliminary Wrapper Definition Language Specification. Technical Report #0684, Trusted Information Systems, Inc., 1997.

[11] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. *Proceedings of the 8th USENIX Security Symposium*, pages 63–78, 1999.