Hardening COTS Software with Generic Software Wrappers

Published in the Proceedings of the 1999 IEEE Symposium on Security and Privacy

Timothy Fraser *tfraser@tis.com* TIS Labs at Network Associates, Inc.
Lee Badger *badger@tis.com* 3060 Washington Road
Mark Feldman *feldman@tis.com* Glenwood, MD 21738

Abstract

Numerous techniques exist to augment the security functionality of Commercial Off-The-Shelf (COTS) applications and operating systems, making them more suitable for use in mission-critical systems. Although individually useful, as a group these techniques present difficulties to system developers because they are not based on a common framework which might simplify integration and promote portability and reuse. This paper presents techniques for developing Generic Software Wrappers – protected, non-bypassable kernel-resident software extensions for augmenting security without modification of COTS source. We describe the key elements of our work: our high-level Wrapper Definition Language (WDL), and our framework for configuring, activating, and managing wrappers. We also discuss code reuse, automatic management of extensions, a framework for system-building through composition, platform-independence, and our experiences with our Solaris and FreeBSD prototypes.

1 Introduction

Commercial Off-The-Shelf (COTS) applications and operating systems are attractive to developers of mission critical systems because of their low cost. However, these COTS products typically provide only commercial-grade assurance, and may lack security features sufficient to meet mission requirements. A number of efforts have augmented the security functionality and assurance of COTS products by injecting layers of software into the interfaces between the COTS operating systems and their applications, or between the applications themselves. Once injected, these layers of software observe and modify the data passing through the interfaces, injecting and processing data for additional security protocols (encryption, authentication) or identifying data that is known to cause harm (access control, intrusion detection). In some cases, the application source must be modified to make the injected software effective, decreasing the savings associated with the use of Off-The-Shelf software.

The different kinds of security functionality added by these efforts are nearly as numerous as the efforts themselves. For example, a number of efforts add application-level access control and/or auditing to servers in the context of application-gateway firewall proxies [1], Wietse Venema's TCP Wrappers [25], and CORBA [30, 31]. DTE [2] and Janus[15] add mandatory access control (MAC) functionality at the system-call interface of COTS operating systems. Kerberos [29] and the Secure Socket Layer [34] provide the means to augment applications which communicate over the network with cryptographic functionality. Still other efforts focus on adding instrumentation to COTS operating systems and server applications to support intrusion detection [17, 23, 20, 21, 11, 12, 22], maintain synthetic jail environments to contain intruders [9], or both [32].

Each of the efforts listed above provides a useful solution in its own problem domain, but is generally limited in scope to a single kind of security augmentation, be it access controls, authentication protocols, or intrusion detection. In order to provide security, developers seeking to construct mission critical systems from COTS components may therefore find themselves faced with the task of integrating, assuring, and managing several of the above techniques, each implemented with its own individual mechanism. Additionally, development of new security enhancements is extremely expensive due to the almost-total lack of abstraction: security enhancements typically operate on low-level

[1]This research was supported by DARPA contract F30602-96-C-0333. This is TIS Labs technical report #0751. FreeBSD is a Registered Trade Mark of FreeBSD, Inc. and Walnut Creek CDROM. Solaris is a Registered Trade Mark of Sun Microsystems, Inc. Windows NT and XENIX are Registered Trade Marks of Microsoft Corp. 486 and Pentium are Registered Trade Marks of the Intel Corporation. UNIX is a Registered Trade Mark of the X/Open Company, Ltd. Trusted Mach is a Registered Trade Mark of Network Associates, Inc.

data structures such as system call parameters, IP messages, and network connections, and in many cases are system-specific.

This paper presents a set of techniques, and a prototype system, for reducing the development burden of security enhancements for COTS software components, for building assurance that separate enhancements function properly when composed, and for supporting reuse of already-developed enhancements. The emphasis of our work is on practical results; we have therefore formulated five key goals for maximizing the impact of our work:

abstraction Security enhancements should be insulated, to the extent possible, from low-level system details. Additionally, it should be possible for an enhancement to work with a system's API without the author having to enumerate all relevant (or worse, all irrelevant) system APIs. It should also be possible to design some enhancements to work without change on multiple operating system platforms.

ease of configuration Security enhancements should be easy to install, configure, use, and uninstall.

nonbypassability Software running under a security enhancement should not be able to circumvent the security policy enforced by the enhancement (e.g., mediation, auditing). This implies that security enhancements are protected from attacks by software running under their control.

compatibility Security enhancements must work with widely-used operating systems (e.g., Sun Solaris, Windows NT, free UNIXs) and applications (e.g., web browsers, compilers, office suites).

performance In most cases, the user should not perceive a performance loss due to enhancements. In particular, lightweight security enhancements (such as access control and some intrusion detection techniques) should exact almost no performance penalty.

We have approached these goals by combining three key techniques: a language for specifying security enhancements, a framework for controlling enhancements and their interactions, and a Loadable Kernel Module (LKM) architecture similar to that found in SLIC [14] for providing efficient and protected kernel-space enhancements on COTS systems.

Our main abstraction is the Generic Software Wrapper, or "wrapper." Wrappers are small state machine specifications. During run-time, state machines based on these specifications, called "wrapper instances," are associated with processes executing COTS applications. We express wrappers in Wrapper Definition Language (WDL), which is a superset of the C programming language [19]. Due to the expressiveness of C, WDL can express wrappers with a broad range of functionality. In addition to providing new language constructs tailored to the task of wrapping, WDL also provides a number of constructs to help make wrappers independent of operating system platform-specific details. This expressiveness and hiding of details make it possible to use wrappers as a common mechanism for implementing the security extensions described above in a coordinated and somewhat platform-independent way.

We have discovered that it is often useful for a system to deploy a number of wrappers simultaneously (e.g., each wrapper may implement a "mini-policy" such as access control for specific resources or intrusion detection). Additionally, it is often important to target specific wrappers to specific programs or users, thus focusing control on points of weakness instead of applying all controls globally (e.g., a wrapper might restrict a web server that is remotely accessible, but not affect any other programs or processes). To address these issues, we have developed a Wrapper Life Cycle framework for wrapper management and a model for wrapper composition. The Wrapper Life Cycle uses a small configurable rule-base to automatically manage the run-time relationships between wrapper instances and processes executing COTS applications. By handling the underlying details, the Wrapper Life Cycle allows wrappers to be managed through a simple high-level install/activate/duplicate/deactivate/uninstall interface. Working with the Wrapper Life Cycle, our wrapper composition model allows multiple wrapper instances to concurrently wrap a single process, each reacting in turn to the process's system calls. In addition to such "passive" composition, we are also developing support for "active" composition, in which one wrapper produces events that another wrapper can listen for. We discuss active composition in more detail in section 3.

Our strategy for providing protected and efficient enhancements for COTS systems is to run wrappers in kernel mode, and to provide their execution environment in an LKM. Figure 1 shows the general architecture of our Solaris 2.6 and FreeBSD 2.2 prototypes. (We are also developing a Windows NT prototype with a somewhat different architecture.) Our Wrapper Support Subsystem (WSS) is implemented as an LKM to permit dynamic installation. The WSS tracks running processes and evaluates "activation criteria" at appropriate times to activate new wrapper instances for pro-

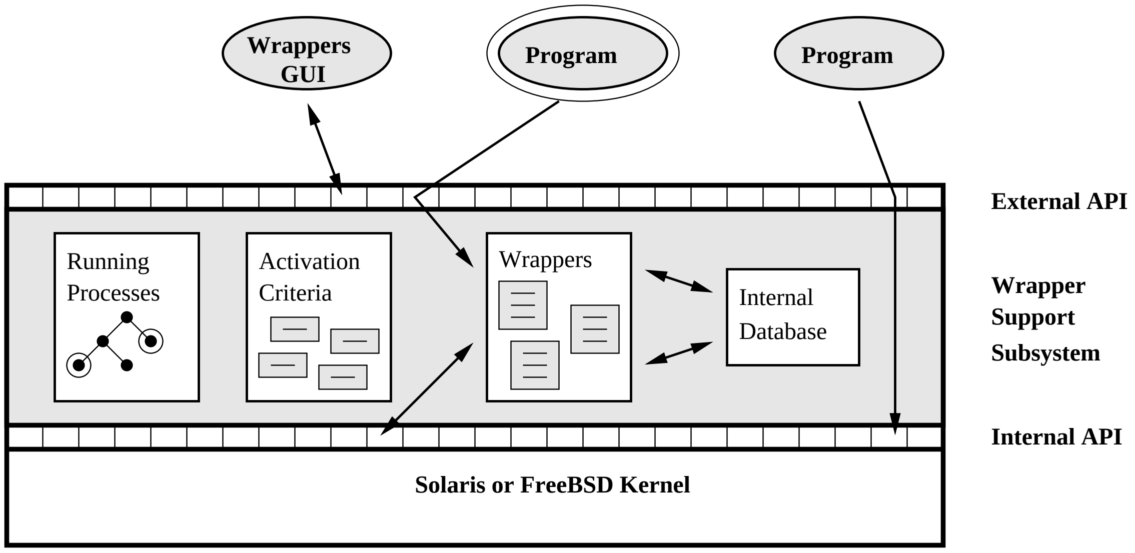

Figure 1. Architecture of the Generic Software Wrappers prototype on Solaris and FreeBSD.

cesses (e.g., the middle process in Figure 1 is wrapped). These wrapper instances "wrap" their processes by intercepting some or all of the system calls they make.[2] This interception effectively puts the wrappers in complete control of their processes' interactions with the operating system and with other processes; the wrappers impose no context-switch overheads and therefore are efficient, and wrappers execute in kernel space and are thus protected. Wrapper instances generally operate synchronously with the processes they wrap, executing just before and/or after the system calls indicated by their specification. (The Solaris prototype has not yet been made aware of threads, and assumes all system calls made by a given process originate from a single thread). For each system call, wrappers may observe and/or modify both the parameters specified by the caller and the values returned by the operating system.

Wrappers appear to provide an extremely flexible and powerful ability to tailor system characteristics to fit the needs of a particular environment, and to provide security using commercial operating systems and commercial applications. Section 4 outlines several strategies for adding security functionality to COTS operating systems and applications with Wrappers, and section 5 discusses several wrappers that we have implemented with our prototype. Based on our experience with building a FreeBSD and a Solaris wrapper prototype system, we believe that wrappers can be implemented using any UNIX system that supports dynamic loadable kernel modules, such as Solaris, FreeBSD, and Linux. Given sufficient documentation on the kernel interfaces exported to loadable kernel modules, access to operating system source is helpful, but not required. Additionally, we believe that wrappers can be implemented on Windows NT. While a wrapper approach must assume that underlying operating system mechanisms function as advertised, it has the ability to guard them from possibly damaging input (e.g., attacks on weak or overly privileged portions of a system's API), thus increasing their overall strength. The wrapper approach, therefore, contrasts but is complementary with, that of Trusted Systems[26] such as Trusted XENIX[27] and Trusted Mach[7], which are built from the ground up with support for enhanced security.

In the following sections, we present our central wrappers concepts, several applications for wrappers, design and implementation issues, capabilities and limitations, and performance. Sections 2 and 3 present our wrapper language and lifecycle model. Section 4 discusses some applications of wrappers to security, and limitations of the techniques. Section 5 covers design and performance issues. Finally, sections 7 and 8 discuss related work and our conclusions.

2 Wrapper Definition Language

A wrapper is a state machine that listens for specified events and, if it "hears" an event, takes actions

[2]Although it is imprecise, we often use the term "wrapper" instead of "wrapper instance" whenever the context is clear.

(A)

```
int     open(const char *path, int flags, mode_t mode)
ssize_t read(int d, void *buf, size_t nbytes)
int     reboot(int howto)
```

(B)

```
int{fdret,VAL_RET}   open{fileop, fdop}(char *path{path,nterm}, int flags, int mode{mode});
int{sizeret,VAL_RET} read{fdop}(int fd{fd},
                               char *buf{copy(buf,nbyte), iobuf, out},
                               u_int nbyte{nbytes});
int{STD_RET}         reboot{rootop}(int opt);
```

Figure 2. (A) C function prototypes of the C library functions corresponding to FreeBSD's open, read, and reboot system calls and (B) the system calls' characterizations.

such as:

augmenting the event by adding additional functionality to the event, such as encrypting a buffer or performing intrusion detection analysis on the event;

transforming the event by converting the event into one or more substitute events, for example, by rewriting a system call event's parameters, or by using alternative system calls instead of the requested system call; or

denying the event by preventing the execution of the event, and returning an error code.

We have designed our language, WDL, to make these tasks as simple as possible by allowing wrappers to easily refer to collections of system calls and by insulating the wrapper writer from low-level details such as parameter copying. A key element of our approach is to augment a system call API with semantic information that allows a wrapper to concisely reference portions of a system's interface without being mired in the details of a large API. An augmented, or *characterized*, interface binds attributes to interface elements; wrappers then can use these attributes to express concisely the functions of the interface to be intercepted and the conditions under which they should be intercepted. Additionally, these attributes provide the WSS the information it needs to copy and replace parameters or the return value, and also provide wrappers with convenient ways to refer to values being passed through an interface.

The majority of the abstraction provided by the Generic Software Wrappers prototype is derived from its use of characterized system call interfaces. In addition to substantially reducing the complexity facing a wrapper writer, a characterized system interface allows some wrappers to be portable between platforms when the APIs of the platforms can be described using the same attributes.

The characterized system call interface consists of the C language prototypes of the functions provided by the actual system call interface augmented with a standard set of tags. These tags map the platform-specific aspects of the function prototype to higher level WDL abstractions which are identical across platforms. Figure 2A shows the prototypes for the `open`, `read`, and `reboot` system calls from the FreeBSD system call interface[13]. Figure 2B shows the corresponding section of the system call interface characterization used by the FreeBSD prototype for comparison. The characterization demonstrates the abstraction provided by three groups of tags:

return value tags: The first group of tags consists of those following the function return values. These tags provide the wrapper writer consistent, named access to return values. For example, since all system calls returning a file descriptor are labeled with `fdret`, the return value for all of them can be accessed in a wrapper as `$fdret`. These tags also indicate semantics associated with the return value with respect to error conditions. This allows the wrapper writer to indicate that a system call should fail using abstract, WDL-based syntax which the WDL compiler converts to the appropriate, system-specific error return. For example, in the `bsd_noadmin` and `noadmin` wrappers in Figure 3, failure (`WR_DENY`) and bad permission (`WR_BADPERM`) will be translated to the appropri-

(A)

```
#include "../../wr.include/platform.ch"

wrapper hello_open {
   bsd::op{open}{
      wr_log("hello open %s\n", $path);
   }
}
```

(B)

```
#include "../../wr.include/platform.ch"

wrapper hello_path {
   *::pattr{path}{
      wr_log("hello path %s\n", $path);
   }
}
```

(C)

```
#include "../../wr.include/platform.ch"

wrapper bsd_noadmin {
   bsd::op{mount    || unmount || ptrace ||
           quotactl || acct    || swapon ||
           mknod    || adjtime || ktrace ||
           reboot   || settimeofday} pre {
      return WR_DENY | WR_BADPERM;
   };
}
```

(D)

```
#include "../../wr.include/platform.ch"

wrapper noadmin {
   *::opattr{rootop} pre {
      return WR_DENY | WR_BADPERM;
   };
}
```

Figure 3. Example wrappers hello_open (A), the wrapper equivalent of "Hello World", hello_path (B), a slightly more general "Hello World" example, bsd_noadmin (C), a wrapper that denies certain administrative system calls on FreeBSD, and noadmin (D), a portable version of bsd_noadmin.

ate error return (including the return value and the setting of the global `errno` variable, if appropriate) based on the tags.

function name tags: The second group of tags consists of those following the function names. These tags divide the functions provided by the system call interface into sets based on the functions behavior. For example, in the figure, both `open` and `read` are marked with the `fdop` tag, identifying them as functions that operate on file descriptors. When wrappers refer to functions in the system call interface by these abstract tags rather than by name, they achieve portability between platforms whose function names may be different.

parameter tags: The third group of tags consists of the tags following the function's formal parameters. Some of these tags divide the parameters into sets based on their purpose, in a manner similar to the function name tags described above. For example, the `path` parameter of the `open` function is marked with the `path` tag, identifying it as a parameter representing a path in the filesystem. Wrappers interested in filesystem paths can achieve portability by selecting system calls containing the parameter attribute `path` without naming the system calls directly. They can access the path parameter in system calls without concern for the location of the path in the parameter list, simply by using the `$path` variable. Other parameter tags indicate to the WDL compiler how parameters containing complex data types should be manipulated. For example, the tag `copy(buf,nbyte)` indicates that the length of the buffer parameter `buf` is dependent on the value of the `nbytes` parameter; the WDL compiler will automatically copy the parameter based both on the type of the parameter (in this case a character) and the length in units of that type. WDL provides features for copying complex data structures (e.g., arrays of records containing pointers) and also data structures whose types are actually determined at runtime (and are `void*` in the interface). Wrappers can achieve increased reliability by taking advantage of the Wrapper Support Subsystem's ability to transparently manipulate complex data types.

Figure 3 contains the WDL source for four simple wrappers that illustrate how wrappers are structured and how they can achieve abstraction. The first is the hello_open wrapper (figure 3A), which is the wrapper equivalent of the archetypical minimalist "hello world" program. This simple wrapper illustrates three characteristics of every wrapper:

It references a characterized interface by including a file containing a characterized version of the system's API.

It listens for events by specifying an *event subsystem*, in this case bsd for the BSD system calls, and specifies events within the subsystem to intercept. In the case of hello_open, the wrapper listens only for the event named "open." Using the attributes established by the characterization of the interface, wrappers may also listen for events having specified attributes and unions of such sets of events.

It takes an action by augmenting, transforming, or denying intercepted events. Hello_open augments the events it intercepts by simply generating a log message (and referencing the parameter value using the attribute from the characterized interface). By default, a wrapper gains control *before* an event (e.g., system call) occurs, however WDL also provides a keyword, post, that can be used to gain control after an event occurs to allow wrappers to do post-processing of events (e.g., translating data returned to the caller).

Figure 3B shows a slightly more general wrapper, hello_path: this wrapper uses WDL's wildcard capabilities to work for any event subsystem (e.g., any operating system), and uses the generic path attribute (via the pattr keyword, which stands for "parameter attribute") from the characterized interface to intercept any system call that uses a pathname parameter. This simple wrapper, which just logs pathnames that are passed through a system interface, will run on any system that has been characterized with a path parameter attribute.

Figure 3C and 3D show two versions of a simple wrapper that provides useful access control functionality. This wrapper addresses the fact that many root processes need the root privilege in order to access user files or to allocate system resources, but do not perform administrative functions, such as changing swap partitions or making new device special files. An attack on these processes can be devastating because the attacker then has administrative access to the system. The bsd_noadmin simply prevents the use of key system APIs that daemons such as web servers, FTP servers, and firewall proxies do not need and *should never use.* The bsd_noadmin wrapper simply listens for any of a specified set of events and, if they occur, denies them. While this is a simple wrapper, it contains a significant amount of system-specific detail and is not portable. The noadmin wrapper, in contrast, takes advantage of the characterization of the system API to intercept the same operations (or their equivalents) on any wrapper-supporting system.

The noadmin wrapper demonstrates simple and useful static access control. Many wrappers, however will be more complex, and, in particular, will require the ability to conveniently access stored configuration information, generate and efficiently store intermediate results, and produce output that can be easily accessed by administrators but is otherwise protected. To accommodate these requirements, WDL includes a lightweight, persistent, built-in database system that allows wrappers to store wrapper instance-specific, wrapper-specific, and global information very efficiently. WDL's database features use an SQL-like syntax, however operations on the WDL database are as efficient as local function calls because the database resides in kernel memory with the wrappers and because the database is not transactional: it does not implement traditional (and slow) database rollback and recovery features.

Figure 4 shows a wrapper, dbcallcount, that illustrates how wrappers can conveniently access and generate data. The dbcallcount wrapper uses the WDL database to create a table of system calls that a wrapped process calls, and to count how many times each system call occurs. Using the DBTABLE keyword, dbcallcount declares a table where each row has two columns, a name and a count. When the wrapper is activated for a process (we cover wrapper activation and management in detail in section 3), the WSS calls the wr_activate() wrapper lifecycle function and dbcallcount creates a new table for the process. Similarly, when the wrapper is duplicated (via fork() in UNIX), the WSS calls the wr_duplicate() lifecycle function and dbcallcount creates a new table for the new process; and, when a process terminates, the WSS calls the wr_deactivate() lifecycle function, and the dbcallcount wrapper deletes the table. The core of the dbcallcount wrapper is dedicated to catching and recording system calls. Using wildcards, the wrapper catches every system call. For each intercepted system call, the wrapper uses an SQL command to update the database (the special variable $$ is the name of the system call that the wrapper intercepted), and adds a new database record if this is the first time the system call has occurred.

Dbcallcount uses the WDL database in a fairly simple manner, however this is still sufficient to make the

```
#include "../../wr.include/platform.ch"
#include "../../wr.include/libwr.h"

wrapper dbcallcount {

    DBTABLE callcountTable {
        char(20) key name;
        int count;
    };
    callcountTable callcount;

    wr_activate() { /* create the table. */
        wql { create table callcount; };
    }

    wr_duplicate() { /* create the table. */
        wql { create table callcount; };
    }

    wr_deactivate() { /* Drop the table. */
        wql { drop table callcount; };
    }

    *::op{*} pre { /* Catch syscalls */
        int retVal;
        /* If syscall in db, ++ count. */
        /* If not, add call to the db. */
        retVal = wql {
                update callcount
                    set .count = .count + 1
                    where
                        .name = $$;
        };
        if (retVal <= 0) {
            wql {
                insert into callcount values
                    ($$, 1);
            };
        }
   };
}
```

Figure 4. The portable dbcallcount wrapper keeps a count of the number of invocations of each system call in the built-in persistent database.

wrapper's results available to administrators and to give the wrapper a reliable and high performance alternative to custom data structures for state management. In addition, the database provides a standardized and persistent data interchange format between wrappers: one wrapper can consume data generated by another simply by accessing the database.

Writing wrappers can be a complex, error-prone task. The key goal for WDL is to reduce complexity as much as possible while encouraging code reuse. Our wrappers have gained significant leverage on this problem through the use of characterized interfaces that allow semantic information of an interface to be reused in many different wrappers, allow wrappers to be much more concise, and also much more abstract. Additionally, the inclusion of a lightweight database into WDL has significantly improved our ability to write wrappers that generate persistent results, and also to monitor the behaviors of wrappers by viewing the database externally.

3 Wrapper Life Cycle

In order to intercept COTS component interactions and impose wrapper-supplied security policies, wrappers must relate at runtime to execution context structures such as UNIX processes. Additionally, wrappers require a control framework for managing their configuration, activation, and termination.

Typical systems create and terminate processes frequently. Depending on the applications running, large trees of processes may be spawned over a short period of time. If these processes are to run wrapped, the rules for associating wrappers with new processes must be simple, and the mechanisms used to relate wrapper-directed processing to the affected processes must also be simple. On one extreme, a single active wrapper might relate to an entire set of running processes; this approach, however, imposes a heavy burden on the wrapper to maintain context information on behalf of the different processes. On the other extreme, a wrapper might relate to only a single process: this would require a separate wrapper to be specified for each potential process, which is probably not feasible. We have adopted a third "middle" approach, which is to specify a single wrapper so that it may be used for a group of processes, and, at runtime, to activate an "instance" of the wrapper for each active process. Specified in this manner, wrappers are templates for creating wrapper instances which execute one-to-one with the processes they wrap. Each wrapper instance thus maintains context information specific to the pro-

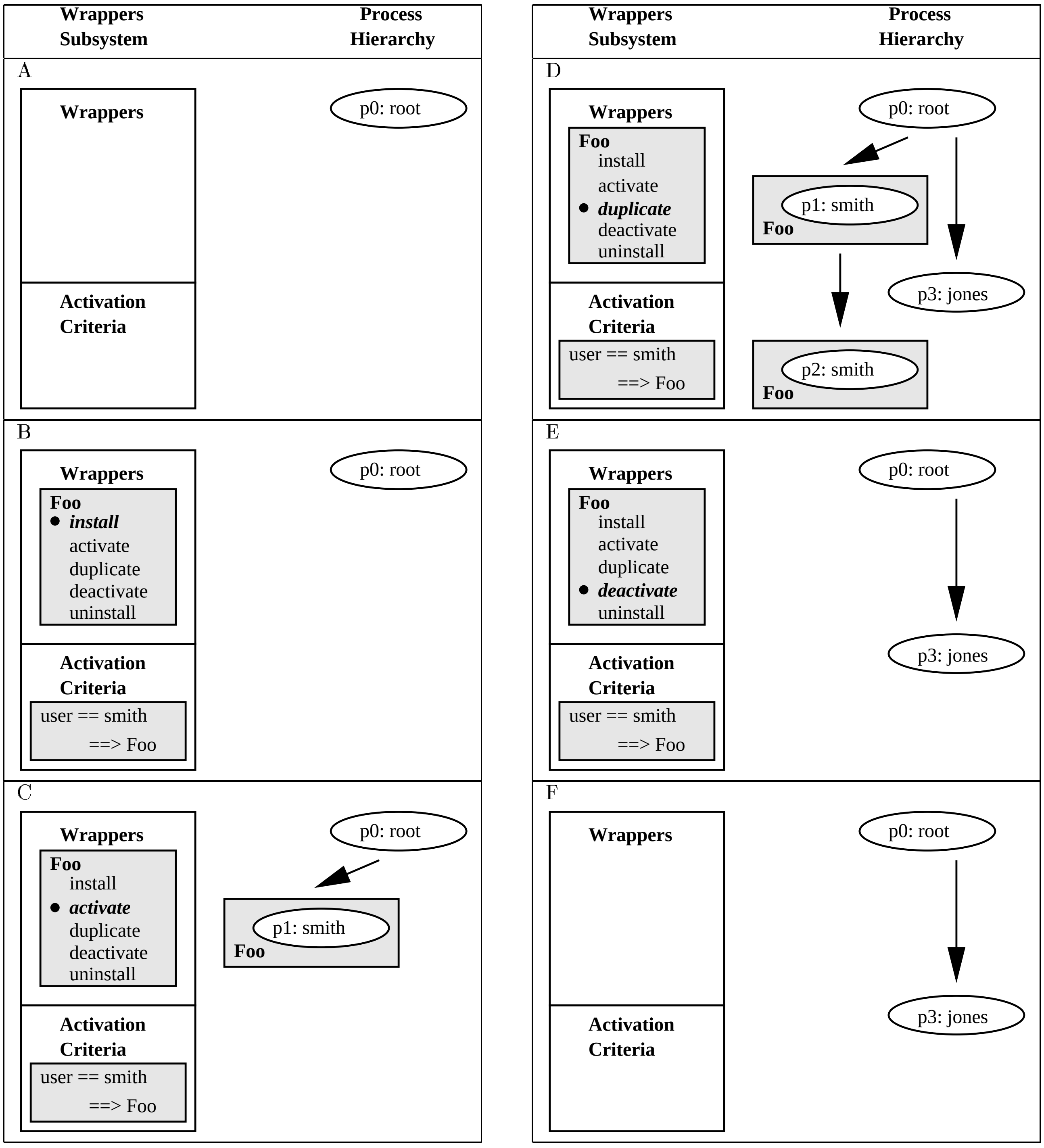

Figure 5. Steps of the Wrapper Life Cycle. (A) Initially, the WSS contains no wrappers or Activation Criteria, and the process hierarchy contains a single "root" process. (B) An Activation Criteria and the Foo wrapper are installed. (C) The creation of a qualifying process leads the Activation Criteria to activate an instance of the Foo wrapper. (D) The Foo wrapper is duplicated as p1 creates a child. (E) Foo wrapper instances are deactivated with the termination of p1 and p2. (F) The Foo wrapper is uninstalled.

cess it wraps and the wrapper template expresses logic common to all instances and any information required to coordinate the wrapper instances.

Our Wrapper Life Cycle framework identifies five key events in the life of a wrapper, marking the points in time where a wrapper may (or sometimes must) act to implement it's behavior relative to the dynamic process hierarchy. Figure 5 shows the state of a system's WSS and process hierarchy as a wrapper traverses the Life Cycle.

Initial state Initially, shown in 5A, no wrappers are loaded and, for simplicity, there is a single process owned by root.

Wrapper Installation Figure 5B shows the state of the system after an administrator (or a privileged program) has installed the wrapper `Foo`, and has specified an "activation criteria" for activating `Foo`. Activation criteria are boolean expressions that a wrapper system evaluates when programs are loaded: activation criteria determine if each program is wrapped, and which wrapper applies to each wrapped program. When the WSS loads a wrapper, it runs the wrapper's `wr_install` life cycle function (if one is specified in the wrapper), thus giving the wrapper an opportunity to initialize the database for wrapper instance activation.

Wrapper Activation Figure 5C shows the system state after a program load (`execve()` in UNIX) has matched the activation criteria: the running program (process `p1`) runs under the control of the `Foo` wrapper. Before the program begins to execute, the WSS calls the wrapper's `wr_activate` life cycle function to perform any process-specific setup.

Wrapper Duplication Figure 5D shows the system state after the wrapped process has duplicated itself (`fork()` in UNIX). Wrappers are inherited across duplications; before the child process starts to run, the WSS calls the wrappers `wr_duplicate` life cycle function, if any, to allow the wrapper to initialize process-specific state (e.g. process-local WDL tables).

Wrapper Deactivation Figure 5E shows the system state after the wrapped process has exited, thus calling the wrapper's `wr_deactivate` life cycle function, which allows the wrapper to perform any required cleanup of its local state and to commit any local results to the database.

Wrapper Uninstallation Finally, figure 5F shows the system after the `Foo` wrapper has been uninstalled. As with the other life cycle functions, uninstallation calls the wrapper's `wr_uninstall` life cycle function, thus giving the wrapper an opportunity to package final results in the database, or to delete them if they are temporary.

We have found that this life cycle approach provides substantial flexibility for applying wrappers selectively within a system. Selective application holds many benefits since performance impacts of process-intensive wrappers can be incurred only when necessary, and wrappers can be tailored to specific programs or environments without necessarily having to configure them for all programs on a system. We believe that this flexibility has the potential to significantly lower the cost of security extensions. Additionally, our wrapper life cycle makes it very easy to specify activation criteria that activate multiple wrappers for a single process. Support for multiple concurrent wrappers allows security functionality to be conveniently partitioned. For example, one wrapper may provide system-global, simple access control (like the `noadmin` wrapper in figure 3D) while other wrappers provide intrusion detection or other functionality that apply to targeted portions of the system.

To allow the use of multiple concurrent wrappers, we have developed a model for wrapper composition. According to our model, systems of cooperating wrappers may compose (combine) the functionality of several individual wrappers using two methods. The first method is "passive" composition; which occurs when multiple wrapper instances wrap the same process and intercept the same system call. Using passive composition, when a process calls the system call, each wrapper instance that wishes to intercept the event "pre," or before it occurs, intercepts the system call in turn, according to the order in which they were *activated* to wrap the process, until the flow of execution reaches the kernel. When the system call returns from the kernel, similarly, each wrapper instance that wishes to intercept the event "post," or after it occurs, is given a turn to intercept the flow of execution. In the return from the system call, however, the wrappers are given control in reverse order. This ordering of pre and post interceptions gives the passively composed wrappers a nested relationship that grants relative control to the outermost layers since they gain control first (on the way in) and have the last word about how the return parameters and codes are treated (on the way out).

Our real-world experience to date has been with passive composition. Passive composition has allowed us to write a number of simple-yet-useful wrappers and compose them. Passively composed wrappers have no knowledge of one another: this simplifies them but also makes it possible to compose incompatible wrappers since a wrapper may condition its behavior on parameter values it believes are from the running process, but that are actually from a wrapper that gained control first. In practice, this has not been difficult to avoid with reasonable configuration.

In our second method of composition, active composition, a wrapper may generate events that can be caught by other wrappers that are listening for them, just like system calls are caught. The primary advantage of active composition is that it will allow a wrapper to, at runtime, translate an operating system's actual events into more abstract events that can be consumed by a more abstract wrapper that can be much more portable. An additional advantage of this approach is that an abstract wrapper can avoid dealing with details from a characterized interface and can therefore, we believe, be much simpler. We plan to use this capability to formulate some key security policy models (such as the Biba integrity models [5], the Clark Wilson integrity model [10], or the Bell and La Padula MLS model [3]) as abstract wrappers. The goal of this investigation will be to make logic regarding such models more reusable and portable between systems.

4 Security wrappers: capabilities and limitations

Using appropriate activation criteria, wrappers can be applied selectively to points of vulnerability, or can be globally applied to all processes on a system. In either case, with respect to a wrapped program, the mediation and additional functionality provided by a wrapper is both nonbypassable and protected from tampering. These characteristics, in combination with the fine-grained control that wrappers may provide by potentially processing every system call, give wrappers a great deal of power to add and enforce security policies. Generic Software Wrappers appear particularly suitable for providing access control, auditing, intrusion detection, and application-independent security features:

access control Wrappers are capable of enforcing traditional access control schemes that make decisions based on a rule set and subject/object labels, such as Bell and La Padula [3], Biba [5], Clark-Wilson [10], and type enforcement [6, 2]. Wrappers are also suitable for enforcing state-based access control schemes, such as the Chinese Wall [8], in which access to a resource is conditioned on a subject's prior resource access history. Wrappers are able to base access control decisions on any state information that is available through a system call interface, allowing them to also enforce policies that are time-based, or sequence-based, such as two-man control. Wrappers are also suitable for implementing synthetic "jail" environments [9] in which process requests for sensitive resources are transparently remapped to alternative, less sensitive, resources.

In addition to controlling operating system abstractions, wrappers are suitable for adding protocol-based access control to communication streams. By wrapping a COTS server, a wrapper can provide the same access control and auditing functionality as a TCP Wrapper [25] or a firewall proxy without the need for a separate security-providing application. Such wrappers would intercept messages bound for a COTS server before they are processed, and perform the necessary access control mediation or auditing based on a set of configurable rules, which might be stored in the wrapper database. Such wrappers might be used to augment COTS servers with the ability to perform access control on their own application-specific operations, as described in OMG's CORBA specification, for example [30].

auditing and intrusion detection Access to system calls and their parameter values provides extremely fine-grained auditing capabilities. Unlike traditional auditing systems, wrappers can efficiently access behavioral data without first writing it to secondary storage. Additionally, wrappers have a potential for performing real-time policy-based audit data reduction, therefore providing more detailed auditing without overwhelming system resources. Wrappers are also suitable for implementing a variety of intrusion detection techniques such as specification-based detection [21], state-based detection [18], and sequence-based detection [12]. As discussed in section 5 below, we have implemented several of these techniques.

security feature enhancement Wrappers can transparently add some security extensions in an application-independent manner. Significant extensions include transparently encrypting files that an application creates, transparently encrypting communication streams between applications, and performing integrity validation checks on resources that an application uses. In addition, wrappers can maintain attributes with resources accessed by an application in order to restrict other programs from intentionally or accidentally damaging a key application.

Wrappers appear to be a very powerful and flexible strategy for deploying security functionality to popular end systems. There are limits, however, to what can be achieved using wrappers:

covert channels Wrappers are not well-suited for controlling information flow. In particular, covert channels [24] usually emerge from resource contention at low levels in a system's architecture, and wrappers are an extension at a system's interface.

assurance While wrappers can constrain the behavior of wrapped programs to protect the underlying system from malicious input, wrappers must assume that the underlying system is functional. If any unwrapped root program becomes malicious, that program can turn wrappers off. Furthermore, even if a malicious program is wrapped, the wrapper must implement a policy that effectively restrains its behavior to protect the system. While "root confinement" policies [36] can be specified with wrappers, policies that properly balance protection with the need for root programs to access resources in order to accomplish their missions may require detailed knowledge of the program's algorithms.

application-specific security enhancements Enhancements which require the wrapper to be aware of the wrapped application's algorithm or parameter formats can be implemented via wrappers, but with greater cost and reduced assurance because the algorithm and parameter data formats used by an application must be reverse-engineered by observing the application in operation. Access to application source or protocol documentation is required to provide a higher degree of assurance that the wrapper indeed understands the algorithm and formats completely.

To explore wrappers capabilities and limitations, we have developed two wrappers prototype systems.

5 Wrappers prototype systems

Our prototype currently runs on both Sun Solaris 2.6 and FreeBSD 2.2 and comprises roughly 42K lines of commented C, C++, Yacc, Lex, Perl, and Java. While some source code is platform-specific, a large portion is platform-independent and source code for all platforms resides in a single source tree. We are also porting our prototype to Windows NT, and now have some elements running in that environment. Our prototype consists of four key components:

a kernel-resident WSS The WSS is the core of the prototype. A key attribute of the WSS is that it is structured as a dynamically Loadable Kernel Module, which allows the WSS to be installed (by root) in unmodified COTS UNIX systems. We believe we can port the WSS for use with most UNIX systems that support dynamic kernel modules (e.g., Sun Solaris, Linux, FreeBSD). Figure 1 shows a block diagram of the WSS installed in a typical operating system kernel. As shown in figure 1, the WSS manages essentially four kinds of elements: 1) the set of installed wrappers and currently running wrapper instances, 2) a set of Activation Criteria, 3) an efficient, lightweight kernel-resident database system that wrappers use to retrieve their configuration information and to store wrapper-generated data streams, and 4) information about running processes such as whether they are currently wrapped, and if so, which wrappers currently control them. In addition to this management, the WSS also transitions system calls appropriately to handler code exported by individual wrappers, and manages the compositions of wrappers.

the WDL compiler The Wrapper Compiler `wrapc` translates wrappers written in WDL to C code, which is then compiled using the native C compiler into a kernel-relocatable object module; the generated object modules are loaded dynamically into the WSS, which uses the wrappers to generate wrapper instances according to the currently loaded set of activation criteria. Once installed, wrappers execute directly without a software interpreter. This improves the performance of the WSS, however it also implies that wrappers are trusted system extensions and that errors in a wrapper may have serious consequences. A number extensible systems techniques [35, 4, 33, 28] could be employed to remove this limitation; our research has focused on providing the most flexible platform possible for expressing varying security policies.

the activation criteria compiler The wrapper activation criteria compiler translates boolean expressions into C code, which is then compiled to produce small executable modules. These modules can be dynamically loaded into the WSS for run-time execution to determine the conditions under which individual wrappers should be activated.

The wrapper criteria language is flexible and allows wrappers to be activated based on predicates such as the user name, the program that is being run, the current working directory, etc.

the wrappers GUI In order to conveniently manage the wrappers prototype, we have implemented a Java-based GUI that communicates with the WSS through a set of new system calls that the WSS adds when it is installed. The GUI allows an administrator to install and uninstall wrappers, view and modify the database tables created by wrappers, set activation criteria, and track processes that are under the control of wrappers. As an alternative to the GUI, the same management functionality is also implemented by a set of command-line utility programs.

To gain experience with wrappers and test our prototype, we have implemented and tested a number of wrappers. The most complex example of enhanced security functionality we have constructed to date is `seq_id`, a wrapper which implements Forrest's sequence-based intrusion detection technique [11, 12]. According to the sequence-based intrusion detection technique, `seq_id` operates in one of two modes. The first mode is used to observe a wrapped process and build a database representing its normal system call behavior. The second mode is used to observe the wrapped process and detect deviations from its previously observed normal system call behavior as described by the database. These diversions correspond to intrusions, and are logged in a special report database table by the `seq_id` wrapper. While operating in both modes, `seq_id` incurs the overhead of intercepting all system calls and updating the behavior database for the wrapped process. The first mode of the `seq_id` wrapper is meant only for initialization; it should operate in the second, intrusion detection mode for the vast majority of its run-time. While operating in this second mode, the `seq_id` wrapper performs additional analysis whenever it detects an intrusion. In these cases, `seq_id` calculates a numerical measure of the magnitude of the deviation, and logs its result in the report database table. We report on the performance of `seq_id` in section 6. `Seq_id` requires 230 lines of WDL, excluding comments.

We have also implemented a number of simple wrappers to test various aspects of our prototype. The source for five of these simple wrappers are presented in section 2. Other, less simple, wrappers implement artificial "jail" environments by converting resource requests to alternative resources, perform file-oriented pseudo-cryptography, and perform specification-based intrusion detection using Ko's technique [21].

Based on our experience with `seq_id` and these other wrappers, we believe that our wrappers prototype can be used as a common implementation and management mechanism for a considerable range of existing security functionality extensions. Good candidates for implementation with wrappers are those techniques that apply some form of interposition to an interface; in some cases even those that, when implemented without wrappers, rely on modifications to COTS source.

6 Performance

Table 1 summarizes the results of a series of performance tests designed to estimate the overhead associated with the WSS and specific kinds of wrapper functionality. The first column of the table contains results from measuring the time taken to compile a Generic version of the FreeBSD kernel, a task which generates roughly 400 system calls per second on our testbed. The column compares the kernel build performance achieved by our testbed under the following five conditions:

no WSS: the baseline performance of the testbed without any modifications due to the wrappers prototype.

WSS only: the performance with the WSS loaded into the kernel, without any *installed* wrappers or Activation Criteria. In order to support the Wrapper Life Cycle, the WSS must intercept the fork, exec, and exit system calls made by all processes, wrapped or not. This mandatory interception imposes a 3-4% penalty on the kernel build's performance using our current wrappers prototype. While we have designed the WSS with performance in mind, we have not optimized the prototype yet, so improvements may be possible.

callcount: the performance when wrapped by the `callcount` wrapper. The `callcount` wrapper is a version of the `dbcallcount` wrapper shown in figure 4 which maintains its state using its own linked-list data structure rather than the kernel-resident database. Instances of the `callcount` wrapper intercept all system calls but perform only minimal processing after interception. `Callcount` adds roughly 1.4% to the base overhead of the WSS for a total of a 4.3-5.3% performance penalty.

dbcallcount: the performance when wrapped by figure 4's `dbcallcount` wrapper. This wrapper

	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%	1.1736	0.0579	0%	4.21	0.18	0%
WSS only	604.38	0.46	3.47%	1.1740	0.0541	0.03%	4.22	0.06	-0.24%
callcount	613.10	0.56	4.84%	1.1908	0.0307	1.44%	4.13	0.05	1.90%
dbcallcount	614.09	1.18	4.99%	1.1885	0.0634	1.25%	4.18	0.12	0.71%
seq_id	624.62	1.23	6.59%	1.1711	0.0446	-0.21%	4.07	0.06	3.33%

Table 1. FreeBSD Prototype Performance for Kernel Build and Web Server Benchmarks

shows only a small relative penalty associated with using the kernel-resident database rather than custom-made data structures, but as shown below by the Average HTTP Latency benchmark, this result is not conclusive.

seq_id: the performance when wrapped by the sequence-based intrusion detection wrapper described in section 5 running in detection mode. The 5-8% performance penalty observed for this wrapper on the kernel build test suggests the wrappers prototype can provide useful security functionality with reasonable overhead even for sophisticated system call analysis techniques.

The second and third columns of the table contain results observed for a custom-made web server benchmark that generates roughly 900 system calls per second on our testbed. 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 corresponding reply from the web server. The Average HTTP Throughput describes the rate at which the web server returns data to the web client, as measured by the web client. Although the results shown in these two columns generally tend to follow the same pattern of increasing performance penalties as the Average Kernel Build Time benchmark, the high variances and low overheads induced by the web benchmark workload make it difficult to meaningfully compare the relative performance penalties of each wrapper.

One exception to the trend of increasing overhead is the unexpected relationship between the callcount and dbcallcount throughput results. The small working set of the web server benchmark generally precludes any need to swap to secondary storage. However, the `callcount` wrapper, which periodically sends bursts of messages to the system log daemon, causes brief, periodic bursts of swapping on our testbed in time with its logging activity. This swapping artificially increases the penalty associated with the `callcount` wrapper, making it appear to impose greater overhead than the `dbcallcount` wrapper, which does not cause swapping. Two other exceptions are the -0.21% `seq-id` latency performance penalty and the -0.24% “WSS only” throughput performance penalty. We believe these numbers are anomalies induced by the high variance in the web server benchmark results.

The Average HTTP Latency and Average HTTP Throughput 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 64MB RAM and a 16-bit ISA Ethernet card running a Generic FreeBSD 2.2.7 kernel. The cron service was disabled during the tests. This machine was connected to a 66MHz Intel 486-based microcomputer with 32MB RAM and an 8-bit ISA Ethernet card via a simple Category 5 crossover cable. This second machine was used to run 32 WebStone 2.0.1 web clients through a series of four 10-minute trials using the standard WebStone 2.0.1 file set for each row in table 1. The Average Kernel Build Time results were produced by building a Generic FreeBSD 2.2.7 kernel on the Pentium-based microcomputer described above while completely disconnected from the network, using the standard build tools provided with the FreeBSD 2.2.7 distribution. Each row in table 1 represents the average result after four builds.

All tests other than the `seq-id` Average HTTP Latency and Throughput tests were conducted with the version of the FreeBSD prototype current on February 19th, 1999. The `seq-id` Average HTTP Latency and Throughput tests were conducted with the version of the FreeBSD prototype current on March 1st, 1999, which incorporated a small number of bug-fixes.

7 Related work

The Generic Software Wrappers project owes much of its direction as a common framework for security extension to the efforts cited in section 1. In particular, the basic concept of adding security functionality to a COTS application by introducing a new intercepting software entity rather than modifying the application's source is central to application-gateway firewall proxies [1], Wietse Venema's TCP Wrappers [25], and the Janus project [15]. Several projects use (or propose to use) mechanisms which are similar to those employed in Generic Software Wrappers. Specifically, the use of system call interception proposed in Sekar's intrusion detection approach [32] is similar to our use of the same mechanism. Also, other intrusion detection efforts [16, 23] have included Petri net-based mechanisms as state machines to keep track of events in series. We have employed the WDL sequence mechanism to accomplish the same task. While WDL sequences are not structured as Petri nets, they are state machines nonetheless.

The SLIC project [14] has demonstrated a system-call interception mechanism for kernel-resident software extensions on Solaris that is similar to the one used by Generic Software Wrappers. Using the SLIC mechanism, they have implemented a variety of security functionality extensions, including an encrypted filesystem and a Janus-like restricted execution environment. While it lacks an abstraction mechanism like WDL, an extension management mechanism like the Wrapper Life Cycle, and support for active composition, SLIC's interception mechanism is generally more comprehensive than the one found in the existing wrappers prototypes. Unlike the current wrappers interception mechanism, the SLIC mechanism is capable of intercepting operations at the virtual memory, virtual filesystem, and signal dispatching interfaces. SLIC therefore has the potential to wrap unusual COTS applications, like FreeBSD's `nfsd` and `nsfiod`. These applications spend most of their run-time hidden below the system call interface in inverted system calls, rendering their activities invisible to the wrappers interception mechanism. Future wrappers prototype development may incorporate elements of SLIC's interception technology.

8 Conclusion

We have presented techniques for augmenting the security functionality and assurance of COTS applications and operating systems using Generic Software Wrappers. We have described how a Wrapper Definition Language can be used to insulate wrappers from system-specific details, encouraging simplicity and portability, and allowing wrapper-writers to implement meaningful security functionality without the need for comprehensive knowledge of low-level kernel minutiae. We have also described how a Wrapper Life Cycle framework can form the basis of an configurable rule-based mechanism to automatically manage the relationships between processes and wrappers during run-time — an essential service where multiple wrappers must be directed to a critical subset of processes on a system. We have also demonstrated the effectiveness of these techniques on our Solaris and FreeBSD prototypes by implementing a number of wrappers, including one which implements sequence-based intrusion detection. Like all techniques, Generic Software Wrappers has its limitations. For example, at the system call level, events occurring at the application level of abstraction are difficult to perceive in the stream of low-level system calls. Also, wrappers cannot enforce information flow-based policies to eliminate covert channels. However, the Generic Software Wrappers technique can provide a common implementation and management mechanism for a wide variety of security functionality extensions including access control, auditing, intrusion detection, and cryptographic protocol enhancements. This common mechanism has the potential to ease the task of system developers who seek to coordinate and integrate security functionality extensions on COTS systems.

9 Thanks

The authors would like to thank the past and present members of the TIS Labs at Network Associates, Inc. Secure Execution Environments Group for their efforts in making Generic Software Wrappers a reality.

References

[1] F. M. Avolio and M. J. Ranum. A Network Perimeter with Secure External Access. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, Glenwood, Maryland, February 1994. TIS Technichal Report #0491.

[2] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A Domain and Type Enforcement UNIX Prototype. In *Usenix Computing Systems Volume 9*, Cambridge, Massachusetts, 1996.

[3] D. E. Bell and L. L. Padula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford, Massachusetts, 1976.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Operating Systems Review*, December 1995.

[5] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Massachusetts, 1977.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[7] M. Branstad, H. Tajalli, F. Mayer, and D. Dalva. Access Mediation in a Message Passing Kernel. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 66–72, May 1989.

[8] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[9] W. R. Cheswick. An evening with berferd, in which a cracker is lured, endured and studied. In *Proceedings of the Winter USENIX Conference*, 1992.

[10] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, California, 1987.

[11] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, Oakland, California, 1994.

[12] S. Forrest and T. A. L. Steven A. Hofmeyr, Anil Somayaji. A Sense of Self for UNIX Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, California, 1996.

[13] FreeBSD Online Manual. *OPEN, READ, REBOOT* . Section 2.

[14] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, 1998.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[16] Y. Ho, D. Frincke, and D. T. Jr. Planning, Petri Nets, and Intrusion Detection. In *Proceedings of the 21st National Information Systems Security Conference*, 1998.

[17] K. Ilgun. A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, 1993.

[18] K. Ilgun, R. Kemmerer, and P. Porras. State Transition Analysis: A Rule-Based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[19] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall Press, second edition, May 1998.

[20] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Computer Security Applications Conference*, 1994.

[21] 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.

[22] A. Kosoresow and S. Hofmeyr. Intrusion Detection via System Call Traces. *IEEE Software*, 14(5), September/October 1997.

[23] S. Kumar and E. Spafford. A Pattern-Matching Model for Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, 1994.

[24] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10), 1973.

[25] M. A. Monroe. Security Tool Review: TCP Wrappers. *;login:*, 18(6):15–16, November-December 1993. USENIX.

[26] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

[27] National Computer Security Center. *Trusted XENIX Version 3.0 Final Evaluation Report*, April 1992.

[28] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, October 1996.

[29] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. In *IEEE Communications*, September 1994.

[30] The Object Management Group, Inc., Framingham, Massachusetts. *The Common Object Request Broker Architecture and Specification*, revision 2.0 edition, July 1995.

[31] D. Sames and G. Tally. JTF-ATD NGII Composable Services Security Analysis White Paper. Technical report, TIS Labs at Network Associates, October 1998.

[32] R. Sekar, Y. Cai, and M. Segal. A Specification-Based Approach for Building Survivable Systems. In *Proceedings of the 21st National Computer Security Conference*, October 1998.

[33] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation.* Seattle, Washington, October 1996.

[34] D. Wagner and B. Schneier. Analysis of the SSL 3.0 Protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, November 1996.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[36] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining Root Programs with Domain and Type Enforcement. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, California, 1996.