Linux Kernel Loadable Wrappers[2]

Terrance Mitchem, Raymond Lu, Richard O'Brien, Kent Larson
Secure Computing Corporation
2675 Long Lake Rd
Roseville, MN 55113
mitchem@securecomputing.com, lu@securecomputing.com, obrien@securecomputing.com

Abstract

This paper describes the results of the Hypervisors for Security and Robustness (Kernel Hypervisors) program. Using the concept of a loadable module, kernel loadable wrappers (KLWs) were implemented in a Linux kernel. These kernel loadable wrappers provide unbypassable security wrappers for application specific security requirements and can also be used to provide replication services. KLWs have a number of potential applications, including protecting user systems from malicious active content downloaded via a Web browser and wrapping servers and firewall services for limiting possible compromises. This paper also includes a summary of the composability analysis that was done on the program.

Keywords: wrappers, security wrappers, kernel loadable modules, computer security, application security, browser security, Linux security

1. Overview

This paper describes the results of the Hypervisors for Security and Robustness program sponsored by DARPA's Information Technology Office. It describes the approach, called kernel hypervisors, developed on the program for selectively wrapping COTS components to provide robustness and security, and summarizes the major achievements of the program. Results of the program, including all documents produced and downloadable source code, is available at: http://www.securecomputing.com/khyper/ .

1.1 Introduction

A hypervisor is a layer of software, normally operating directly on a hardware platform, that implements the same instruction set as that hardware. They have traditionally been used to implement virtual machines[1]. Kernel hypervisors (hereafter called kernel loadable wrappers or KLWs) are a similar concept, but are implemented on top of an operating system kernel rather than on top of the hardware. These KLWs provide a set of "virtual" system calls for selected system components.

KLWs are loadable kernel modules that intercept system calls to perform pre-call and post-call processing. They can be used to provide an additional layer of fine-grained security control or to provide replication support. Their key features are that they can be set up to be unbypassable, since they are in the kernel, and they are easy to install, requiring no modification to the kernel or to the COTS applications that they are monitoring.

KLWs have a number of potential applications, including protecting user systems from malicious active content downloaded via a Web browser and wrapping servers and firewall services for limiting possible compromises.

1.2 Accomplishments

On this program, the following specific accomplishments were achieved.

[2] This work was supported by DARPA on contract number: F30602-96-C-0388.

- A framework was developed based on a master KLW, whose job is to coordinate installation and removal of individual client KLWs and to provide a means for management of these clients. The framework allows client KLWs to be stacked so that a variety of application specific policies can be implemented, each by means of its own KLW.
- A variety of specific client KLWs were developed to test and demonstrate the feasibility of the KLW concept Specific wrappers developed were:

    1. Netscape browser wrapper: to limit the damage that could be done when a user, browsing on the Internet, downloaded and executed malicious active content.
    2. Apache web server wrapper: to protect the server's data files from unauthorized modification and to limit the damage that could be done if the server was taken over by a malicious user.
    3. Replication wrapper: to automatically replicate files as they are being modified in a manner transparent to the applications performing the modifications.

    The Netscape browser and Apache web server client KLWs are actually generic wrappers that can be used to isolate an application and its data files in a separate domain so that the application is protected from other components of the system and vice versa.
- A management component was developed that provides the ability to dynamically reconfigure the policies that the various client KLWs enforce.
- Performance testing was done to measure the impact of the various client KLWs on the performance of their associated applications.[7]
- A composability analysis was performed to analyze the security implications of stacking client KLWs.
- The source code for the KLW components developed on this program has been made available on the web for other researchers. All code was developed for the Linux operating system.

1.3 Document Organization

The remainder of this document is organized as follows. Section 2 describes the KLW architecture, presents a high level view of the system components that were developed and discusses some of the possible uses of KLWs. Section 3 describes the composition analysis that was performed on the system. Section 4 documents some of the lessons learned on the project and open issues. The References section includes a list of cited and related documentation.

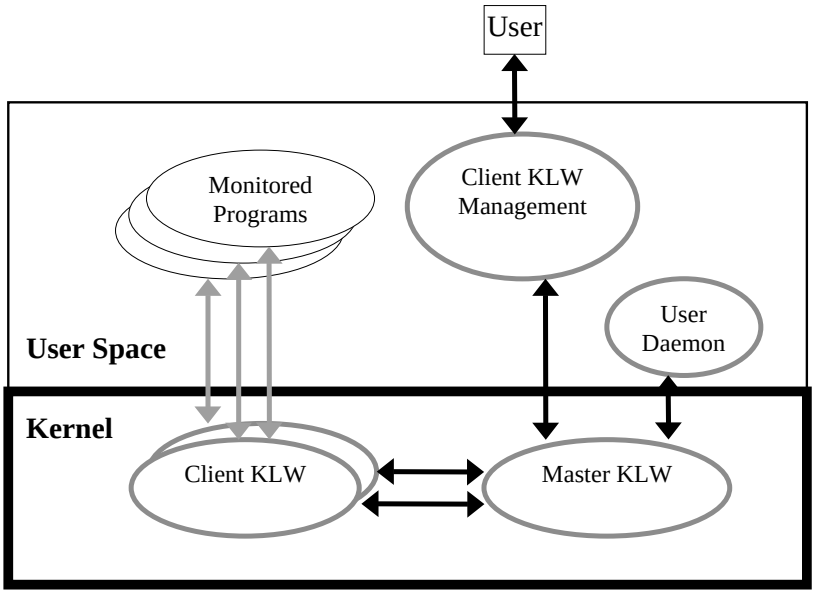
2. Approach

Section 2.1 presents an overview of the Linux KLW architecture that was developed on the program and includes a brief discussion of the benefits of the approach and how it relates to other research work. Section 2.2 then provides more details of the specific system components that were developed. More details of the implemented system can be found in the Design Report [3]. Section 2.3 documents some of the possible applications of the KLW technology.

2.1 Architecture

The KLW architecture is illustrated in Figure 1. There are three main components:

- The *master KLW* manages the individual client KLWs that are currently loaded and provides a control facility allowing users to monitor and configure these client KLWs.
- Specific *client KLWs* provide application specific policy decision making and enforcement. Each client KLW can wrap one or many applications. User daemons that run in user space can also be developed to allow the client KLWs to initiate actions in user space.
- The *client KLW management* module provides an interface for communicating with and configuring the client KLWs from user space.



- Master KLW provides communications and control
- Client KLWs provide application specific monitoring
- Client KLW Management provides the user interface for configuration of client KLWs

Figure 1. KLW architecture

KLWs are distinguished by the fact that they consist of *loadable kernel code* that is used to wrap specific applications. Other approaches have been used to provide wrappers for additional security or robustness including:

- Traditional hypervisors that replace the standard hardware interface with an interface that provides additional capabilities. This approach was used by Bressoud and Schneider[1] for building a replication KLW that intercepts, buffers, and distributes signals from outside the system.
- Special libraries that include security functionality and that are linked with an application before it is run. This is the approach taken by SOCKS[4] where the client links in the SOCKS client library. SOCKS is intended for use with client/server TCP/IP applications, but the concept of linking in special libraries can be used for any type of application.
- Wrappers that make use of an operating system's debug functionality. This is the approach taken by the Berkeley group[5].

Our approach is most like the last, but it differs in that we place our code directly in the kernel and do not use the system debug functionality. Loadable KLWs have a number of benefits.

- They are unbypassable. Since the wrapper is implemented within the operating system kernel, malicious code cannot avoid the wrapper by making direct calls to the operating system as could be done with code library wrappers. Any such calls will be intercepted and monitored.
- They do not require kernel modifications. All KLW code is implemented as modules that are loadable within the kernel while the system is running. There are no changes to kernel source code as is necessary with specialized secure operating systems.

- They are flexible. KLWs can be used both to implement a variety of different types of security policies and to provide replication functionality. Some of the possible applications are discussed below. A key feature is that KLWs can be "stacked", so that modular security policies can be developed and implemented as needed.
- They are not platform specific. KLWs can be used on any operating system that supports kernel loadable modules that have access to the system call data structure. This includes Linux, Solaris, and other modern Unix systems, as well as Windows NT.
- They can be used to wrap COTS software without any modification to the software (including, e.g., relinking). KLWs can monitor the actions of the COTS software and react in a manner that is transparent to the COTS application, other than it may be denied access to certain system resources according to the policy being enforced by the KLW.

The goal of KLWs is to protect against malicious code or to provide some type of additional functionality. Hence, the operating assumption is that the user can be trusted. This assumption implies that KLWs are similar to virus protection programs in that a user must not specifically disable them. In fact, KLWs can be set up so that they cannot be disabled, but they do rely on proper user administration and use.

2.2 Components of the System

Using the concept of a kernel loadable module, we implemented KLWs on a Linux kernel. Linux was chosen as the initial platform because it supports loadable modules, the source code is free and easily available so the results of our work are accessible to other researchers and developers, and it is widely used as a Web server platform and hence provides a good target for our approach.

Linux provides full support for kernel loadable modules[6]. Three system calls are supported that allow a privileged user to install and remove loadable modules and to list which modules are currently loaded. Once loaded, the KLWs in Linux run within kernel space and have full access to kernel data structures.

The remainder of this section discusses the three major components of our implemented KLW system in more detail.

- The Master KLW Framework
- The Client KLWs
- The KLW Management.

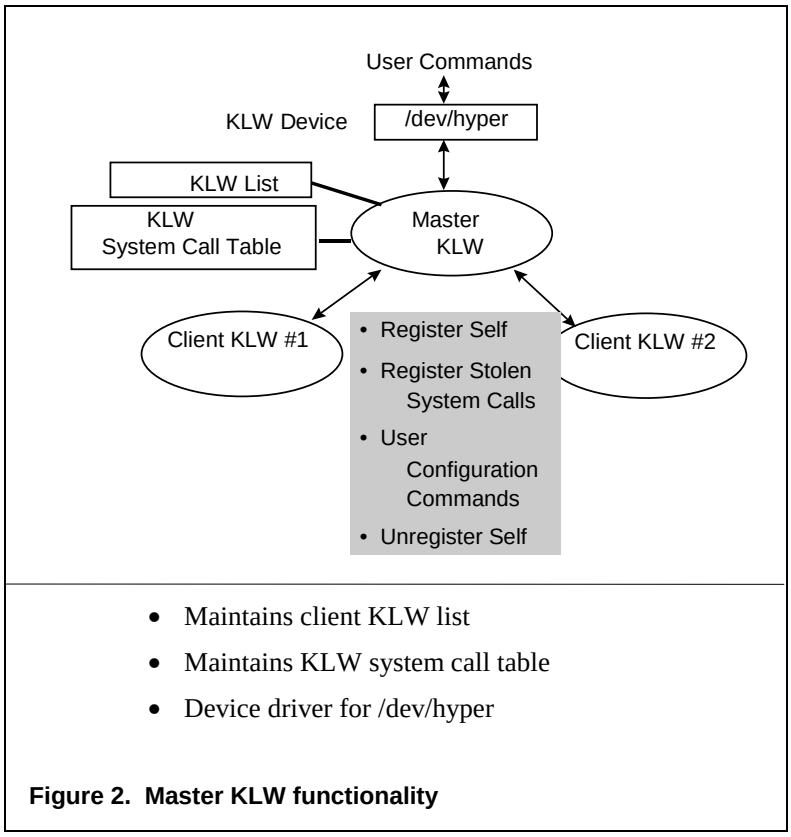


Figure 2. Master KLW functionality

2.2.1 The Master KLW Framework

A framework has been developed based on a master KLW, whose job is to coordinate installation and removal of individual client KLWs and to provide a means for management of these clients. The framework allows client KLWs to be stacked so that a variety of application specific policies can be implemented, each by means of its own KLW. The KLWs run in the kernel, but since they are loadable modules, they do not require that the kernel be modified.

Figure 2 illustrates the framework. The master KLW is loaded before any other client KLWs. A special application programming interface (API) has been defined that allows client KLWs to register and unregister themselves with the master KLW and to identify which system calls they need to monitor. The master KLW keeps track of all currently registered client KLWs and of the particular system calls that each client KLW is monitoring. When a client KLW module is removed via the rmmod call, it is the responsibility of the client KLW to de-register itself with the master KLW.

A special device, /dev/hyper, has been defined for communication between user space and KLWs. The master KLW acts as the device driver for this device. The device allows a privileged user to dynamically update the configuration information for a KLW, including updating the security policy that the KLW enforces. It also provides a mechanism that KLWs can use to communicate with user space daemons. Such daemons, for example, could be used to provide additional audit capabilities or replication services.

The master KLW provides special wrapper code for use with any system call that is being monitored. The actual monitoring of system calls is performed by redirecting the links in the kernel system call table to point to system call wrappers that the master KLW provides. This redirection of links is the only modification to the kernel that is performed and is done by the master KLW only on system calls being monitored. The wrapper code is invoked when the system call is made and performs the processing illustrated in Figure 3.

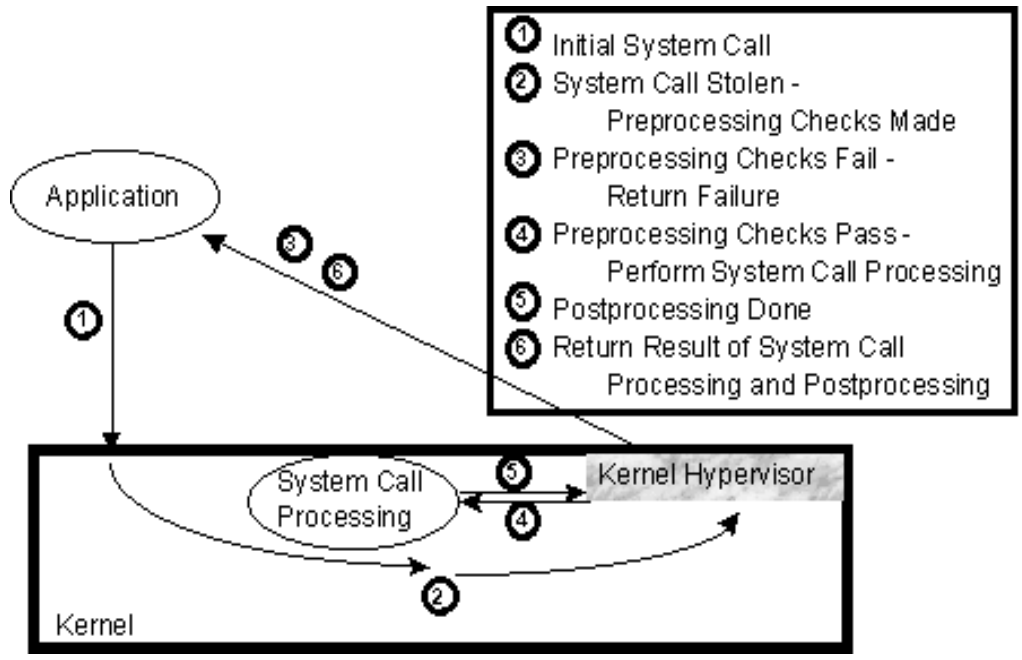


Figure 3. System call processing

Each system call has its own wrapper that follows the algorithm:

- For each client KLW monitoring this call, initiate that client's pre-call processing.
- If the call passes the pre-call processing, call the standard system call processing.
- For each client KLW monitoring this call, initiate that client's post-call processing.

Pre-call and post-call processing is used to enforce the client KLW's particular security policy or to initiate other actions by the client KLW. This processing could include additional auditing of system calls, including call parameters and results; performing access checks and making access decisions for controlled resources that the KLW is protecting; modifying system call parameters; and passing information to user daemons.

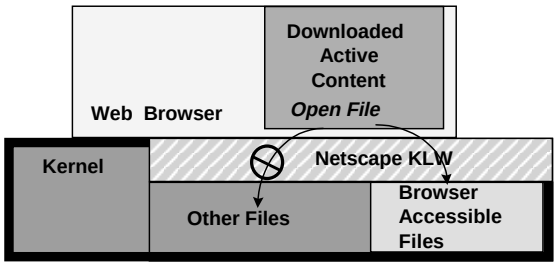
2.2.2 Client KLWs

Client KLWs are developed and loaded separately as needed. A single client KLW can be designed to monitor just one specific application or a number of different applications. Client KLWs could also be used to enforce other types of policies independent of any application.

To illustrate the practicality of the KLW concept, we prototyped three client KLWs: one for wrapping the Netscape browser, one for replicating files, and one for wrapping the Apache Web Server.

Netscape KLW

The goal of the Netscape KLW is to protect a user, browsing on the Internet, from downloading and executing malicious active content that might damage the user's system. The Netscape KLW accomplishes this by monitoring system calls made by the browser and enforcing a policy that only allows certain resources to be accessed. In particular, the set of files that the browser can open for read, and read/write access is controlled so that the browser effectively operates within its own limited execution context. While this does not prevent malicious code from accessing, and possibly damaging, resources within this context, it does limit the damage that could be done to only these resources.



In the case of the Netscape browser, the context includes the user's .netscape directory as well as limited access to other libraries needed by the browser to execute. Most files on the system, however, are not accessible to the browser and so cannot be damaged. To ensure that applications started from the browser as the result of a download, e.g. a postscript viewer, are also controlled, the KLW keeps track of all descendants of the browser and enforces the same policy on them as on the browser.

The security policy that the Netscape KLW enforces is stated as a set of rules identifying which resources the browser is allowed to access and what permissions the browser has to the resource. If there is no rule that allows access to a resource, then the KLW refuses any requests for access to that resource. The format of the rules is:

<*type*> <*identifier*> <*permissions*>

where <*type*> is either a file, socket, or process

<*identifier*> is either a file/dir pathname, an IP address, or a process ID

and <*permissions*> depends on the type.

For our Netscape prototype, only rules for the <*file*> type are used. For these rules permissions are:

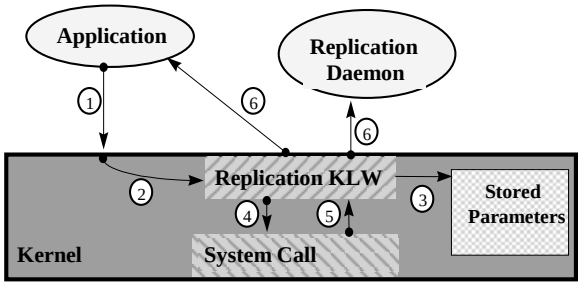
read, *write*, *read/write*, and *none*.

Certain conventions are used to simplify the statement of the rules. If an <*identifier*> is a file directory, then access to all files in that directory and all subdirectories is governed by the rule for the <*identifier*>, unless this rule is specifically overridden. Rules can be overridden by stating another, more specific, rule. For example, if a rule allows *read* access to all files and subdirectories of the directory /etc, then you can prevent users from accessing the file /etc/passwd

by including a rule for this file with a permission of *none*.

Replication KLW

The replication KLW is used to transparently replicate a file or set of files. The objective is to provide a replication facility that allows immediate backup of changes to a file without having to modify any applications that are making the actual changes.



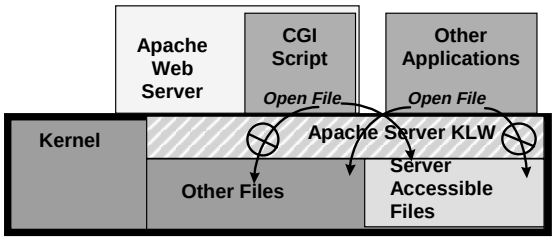
The replication KLW monitors all system calls that modify files. It looks for calls that access a file that is being replicated. When such a call is identified, the KLW caches the input parameters and allows the call to continue execution. If execution of the call completes successfully, then the KLW sends the cached input parameters to a replication daemon, operating in user space, that replays the call with the cached parameters on the copy of the file that it is maintaining.

Files can be replicated locally or across the network (using NFS) via this method. Results of the performance testing done are documented in the Test Report available on our website.

Apache Server KLW

Based on our experience with the Netscape KLW, a generic security KLW was developed that can be used to isolate any application in a compartment from which its access to files can be completely controlled. This generic security KLW, plus additional restrictions using techniques developed for the replication KLW, were used to develop a security KLW for the Apache Web Server.

The web server requires two types of protection. First, the server must be restricted to a subset of the file system so that any attack coming through the server will be contained.



This was accomplished by creating a new KLW, hyper_apache, which is essentially the Netscape KLW with a different policy.

The server's temporary scratch directory and logging directories are configured as read/write while the html files presented by the server are configured as read only by hyper_apache.

The second piece of server protection is to isolate the configuration and html files used by the server from modification by unauthorized individuals. This is accomplished by a new KLW, hyper_cop (for Circle of Protection). Once installed and configured, hyper_cop restricts access to the files within its circle from all but certain privileged users. This will prevent a rogue user from overriding the system and changing the content of the web site even if they get access to root. The attacker will be required to know who the privileged user is before changes are allowed.

This implementation of hyper_cop is sufficient to show proof of concept but some open issues remain. Hyper_cop is configured by reading in and storing a list of filenames to protect, which it checks each time an open() call is encountered. The filename being opened is checked against the list and an access decision is made. Unfortunately, relying on filename string comparisons is a very inaccurate way to identify files. To circumvent the protection, a user simply needs to access the file by a different name than the name being protected by hyper_cop. This can be accomplished by many methods including accessing the file through a symbolic link, through the /proc directory structure or simply by giving a relative path and name instead of the full name of the file. How to identify and control all such indirect accesses to a file is an open issue. This is not an issue with the Netscape KLW, since the policy that KLW enforces is that if a given filename is not recognized as one to which access is allowed, then the access is denied.

Stronger protection could also be achieved by developing a shell that works with hyper_cop to provide user authentication. Such a KLW/shell combination could be used to provide strong authentication before allowing a user to modify the protected files.

2.2.3 KLW Management

The KLW management component allows a user to obtain information on client KLWs that are currently loaded and to reconfigure them, if needed. (As noted earlier, KLWs are loaded and removed through standard Linux system calls.)

All KLW management is done via communication through the /dev/hyper device. The master KLW responds to requests for a list of all currently installed client KLWs and their current identifiers. These identifiers are used to direct requests to specific KLWs.

The management functions available for the Netscape KLW include:

- the ability to list the current rules that are being enforced
- the ability to clear the current rule set and load a new one
- the ability to change the log level that the KLW uses to determine what detail of logging it should do.

2.3 Possible Applications

KLWs can be used in a variety of ways to enhance the security and robustness of a system. This section discusses some of the types of uses followed by some specific applications.

2.3.1 Types of Uses

Auditing: In their simplest form, KLWs can provide an audit and monitoring functionality that merely records additional information about the system resources that an application is accessing. Such audit KLWs are also useful for determining what system resources an application accesses during its normal processing. For example, an audit KLW was used to identify what resources the Netscape browser and the Apache web server needed to run. Since the level of auditing can be dynamically adjusted and since KLWs also provide a control mechanism, they could be useful as a component of an intrusion/detection/response system.

Fine-grained access control: The most compelling use of KLWs is to provide dynamic, fine-grained access control to various system resources such as files, network sockets and processes. The type of control possible is what differentiates this method of wrapping applications from standard access control list methods that perform control based on a user attribute. KLW security policies can be based on users, but can also be based directly on the application. For example, the Netscape KLW that we implemented only monitored the Netscape application as well as any other applications started from within Netscape. Users had additional privileges to access their own .netscape directories, but could not override the restrictions in the Netscape KLW security database. These restrictions apply even to the root user. (In fact, an additional restriction imposed by the Netscape KLW was that the root user could not even run the Netscape browser.)

Label-based access control: While the rule sets described in Section 2.2 can be used to implement most simple policies, it would also be possible to use KLWs to implement more sophisticated label-based policies, such as a multilevel secure (MLS) policy or a type enforcement policy. To be able to implement these policies, a KLW would need a way to label system resources. This might be done by creating and maintaining its own list of labeled names, or by piggybacking the labels on system data structures that have available space. Because of the flexibility of the KLW, these label-based policies could be applied to only those portions of the system that required labels. And policies could be quickly changed, if needed, to adapt to the current operating environment.

Replication: KLWs can also be used to provide replication features by duplicating system calls that modify system resources, such as files. We have investigated one approach for doing this by using a user daemon to replay file access requests that are intercepted by a KLW. The replayed requests effectively duplicate that portion of the file system that is being monitored.

This replication facility could be used to provide dynamic file redo logs to support recoverability.

2.3.2 Specific Applications

Specific applications that KLWs can be used for include:

- Wrapping web browsers to protect the user from downloaded malicious active content.

  This is described in more detail in the section on the Netscape KLW. The most interesting point is that any applications that are started automatically from the browser, as well as any plug-ins, are also subject to the same restrictions that are on the browser.

- Wrapping web servers to protect against malicious attacks.

  KLWs can be used to ensure that if the web server is overrun the damage is limited. In particular, it can ensure that files being served by the web server are not modified inappropriately. Also, because they are often easy to overrun, CGI scripts are sometimes attacked by malicious users on the web. In particular, if a CGI script has the ability to connect to an internal system, then an attacker might be able to compromise this CGI script to launch an attack. By limiting which ports on an internal system a CGI script can connect to, a KLW can limit the attacker to only services on those ports.

- Wrapping services and proxies on an application gateway.

  Services, like Sendmail, often have unknown bugs that are only discovered when someone uses them to attack the system on which the service is running. Once again, KLWs provide a way of limiting the damage that such a compromise can cause.

- Adding new security features to a system.

  KLWs can be used to add security features, such as security levels, roles, or domains and types, to a system without requiring any additional modifications to the system. In fact, a special kernel security KLW could be implemented that performs all of the required security checks for any other KLW.

This is only a partial list of the possible security uses of KLWs. Any number of applications could benefit from the additional security that they can provide.

3. Composition Analysis Summary

This section contains a short summary of the composition analysis that was done on the program. The full set of specifications and documentation of the composition analysis is in the Formal Specifications report [8].

The application of composition to the analysis of a system provides similar benefits as the modular approach to writing code. Even if the design (and specification) of one component changes, as long as the properties and interfaces between that component and the others remain stable, then only the arguments made about the properties of that one component need to be re-verified. In other words, the assurance arguments for unchanged components become reusable.

3.1 Specified Systems

The composition framework that was used is the one developed on the Composability for Secure Systems program which is the successor to the framework developed under the DTOS program. This framework is based on the work of Abadi and Lamport[9] and of Shankar[10]. The differences between the Composability and DTOS approaches and that of Abadi and Lamport are described in more detail in the Composability for Secure Systems Refined Design Report [11] and in the DTOS Composability Report[12] respectively. For both the Composability for Secure Systems program and the DTOS program, the framework was specified in the PVS language.

The composability framework is based on a state transition model. Each *transition* is a triple of the form *(initial state, final state, transition agent)*. Depending on the needs of the analysis, the state may represent either the component state or the system state. The transition agent can represent a variety of concepts, including particular components, particular operations that a component supports, or threads that implement a component. Transition agents also can represent agents in the component's environment that are not part of the component.

Four separate, but related, systems were

specified. The four specifications are related via refinement. The second and third specifications are refinements of the first; the fourth is a refinement of the second. The systems are:

- no KLW loaded into the kernel; that is, just one component: the kernel. Since this system has just a single component, composability has nothing to add to the analysis of it. However, the kernel without any KLWs loaded into it forms the baseline system from which all of the other systems are built.
- a two component system consisting of the kernel and a single security KLW that monitors write requests loaded into the kernel
- an enhanced two component system consisting of the kernel and a single security KLW that monitors write requests and return parameters
- a multiple component system consisting of the kernel, a controller KLW and a set of security KLWs that monitor write requests. In this specification, the single KLW has been refined into a controller KLW and a set of KLWs each of which have a security policy defined. It is possible to implement a single KLW that would behave identically to a system composed of the components specified in this case, but the added complexity of specifying the security policy would make the task very complex.

Each specification describes the individual components of the system and the common state for the composite system. A list of properties satisfied by each component was derived, and composability theory was used to show that these properties are satisfied by the composite system.

3.2 Conclusions of the analysis

Since composability theory says that the properties of a system are just the conjunction of the component properties, the analysis seems to be trivial. In fact, the hypotheses placed on a set of components for composability are so weak, that the composition theorem seems to say almost nothing. However, the final results are not as trivial as they appear at first glance, and this is often the case with any mathematical construct which is defined well. The strength of composition comes not from the composition theorem *per se*, but from the way in which composition of two components is defined.

The interesting features of a specification come in the decision of what should be specified in each component's transitions, and what other transitions each component can tolerate. In fact, much of the strength of the properties relies on what things one can reasonably enforce on the environment of a component. The more that one can enforce on the environment of a component, the more strongly one can state (and perhaps more easily one can prove) the properties of the component. However, the counter argument is that specifying too much in the environments of the system components can specify away all functionality of the composite system (since the composite consists of only those individual component transactions that satisfy each of the other component's environment transaction specifications). The goal therefore is to specify in each component's environment only those things that are needed to prove the component's properties hold.

The first specification represents an operating system that is not protected by a KLW. The final three specifications represent refinements of a sort on that initial specification. They are not true refinements in the sense that the KLWs are enforcing a security policy in addition to the one already enforced by the kernel. However, this appears to be the most powerful application of composition. While the code for the kernel itself may be unavailable to the developers of the KLWs, the KLWs are kernel-loadable modules, and thus represent a refinement of the kernel. Since the code for the kernel may be off limits, the interfaces and properties of the kernel are both well-defined and stable.

This is an ideal situation for the re-use of the specification of the kernel properties. We know what they are to begin with. After loading the KLWs into the kernel, the properties of the composite system are just the conjunction of the properties of the KLWs and the kernel. That means that we can layer the desired behavior of the KLWs over the properties of the kernel with the assurance that these properties will also hold.

4. Lessons Learned and Open Issues

This section contains some lessons learned and open issues from the work done on the program.

4.1 Lessons Learned

An original goal of the program was to investigate whether kernel loadable modules could be used to add additional security and robustness to systems with minimal impact on the system and system applications. The results of the program clearly demonstrate that this is not only possible, but, in fact, has many advantages over other approaches for wrapping system components.

One of the more interesting results of our work is the validation of the concept of a master KLW to control the specific client KLWs and to provide communication between user space and specific KLWs for management. An alternative approach would be to construct one large loadable module that contained all of the functionality of both the master KLW and the specific client KLWs. This large module approach would be much less flexible, manageable and maintainable. Hence, proving that the more modular approach used on the program actually worked is a significant accomplishment.

Another interesting result is the development of a generic KLW that can be used to provide an isolated environment within which an application can execute. As demonstrated with both the Netscape KLW and the Apache Web server KLW, the generic KLW is easy to configure for applications with minimal work. By running the KLW in audit mode, the files that the application normally accesses can be identified and the appropriate configuration file constructed that allows only the accesses needed by the application.

The hyper_cop KLW provides the other most common type of additional access control needed: controlling the accesses to particular files for all applications and users on the system. The combination of the generic KLW with the hyper_cop KLW provides most of the additional access control that might be desired. As discussed earlier, however, there are still some open issues involving the hyper_cop KLW.

4.2 Open Issues

While the feasibility and usability of KLWs was shown on the program, there are some areas that were not fully addressed.

- Network and process control.
  The KLW framework is appropriate for controlling access to files, network resources and processes, however, only file access control was investigated on this program.

- Platform independence.
  While the KLW approach should work for any system that supports kernel loadable modules, it has only been tested on the Linux operating system. For other Unix systems, like Solaris, that support kernel loadable modules, it should be a relatively straight forward task to port the current framework. For a system like Windows NT, that is proprietary with little kernel documentation, it remains an open issue as to how well the approach will work. There are unofficial ways to insert code to intercept system calls on NT that have been documented [13]. However, whether these methods are sufficient to support the KLW framework is unknown. SCC is currently under contract with AFRL to develop KLWs for NT.

5. Summary

KLWs are an approach to wrapping applications to provide additional security that has a number of advantages over other approaches. Because they reside in the kernel, they cannot be bypassed. Because they are implemented as loadable modules, they do not require any modification to the kernel. Because they do not require modification to an application, they can be used to dynamically wrap processes that are started from other processes that have already been wrapped. Because they can be easily configured, the policy that they enforce can be dynamically modified as needed. By limiting the amount of damage malicious software can do, KLWs provide an approach to protecting one's system against current and future threats that may still be unknown.

The Kernel Hypervisors program has successfully demonstrated the feasibility and usefulness of using kernel loadable modules to add additional security and robustness to an application without needing to modify either the application or the operating system on which the application runs. Future research should now

focus on extending the framework developed on this program to other platforms and on developing new and enhanced client KLWs to provide security, reliability and recovery features not currently available.

6. References

[1] Thomas Bressoud and Fred Schneider. Hypervisor-based Fault-Tolerance. *Proceedings of the 15th ACM Symposium on Operating System Principles.* December, 1995. ACM Press.

[2] Terrence Mitchem, Raymond Lu, Richard O'Brien. Using Kernel Hypervisors to Secure Applications. *Proceedings of the 1997 Applied Computer Science Applications Conference*. December 1997.

[3] Secure Computing Corporation. Design Report, Hypervisors for Security and Robustness Program, CDRL A005. February 1998.

[4] M. Leech, et al. RFC 1928: SOCKS Protocol Version 5. March 1996.

[5] Ian Goldberg, David Wagner, Randi Thomas and Eric Brewer. A Secure Environment for Untrusted Helper Applications. *Proceedings of the 6th USENIX Security Symposium*. July, 1996.

[6] Bjorn Ekwall and Jacques Gelinas. Linux Modules 2.1.3 Documentation. June, 1996. Distributed with Linux source.

[7] Secure Computing Corporation. Test Results, Hypervisors for Security and Robustness Program, CDRL A006. February 1998.

[8] Secure Computing Corporation. Formal Specifications, Hypervisors for Security and Robustness Program, CDRL A007. February 1998.

[9] Martin Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems* 17, 3. May 1995.

[10] N Shankar. *A Lazy Approach to Compositional Verification*. Technical Report TSL-93-08, SRI International, December 1993.

[11] Secure Computing Corporation. Composability for Secure Systems Refined Design Report. February 1998.

[12] Secure Computing Corporation. DTOS Composability Study. February 1997.

[13] Mark Russinovich and Bryce Cogswell. Windows NT System-Call Hooking. *Dr. Dobb's Journal*, January, 1997.