Base Line Performance Measurements of Access Controls For Libraries and Modules

Jason W Kim and Vassilis Prevelakis
Department of Computer Science
Drexel University
Philadelphia, PA 19104
{jkim,vp}@cs.drexel.edu

Abstract

Having reliable security in systems is of the utmost importance. However, the existing framework of writing, distributing and linking against code in the form of libraries and/or modules does a very poor job of keeping track of who has access to what code and who can call what function.

The status-quo is insufficient for a variety of reasons. As the amount of code written that represents some kind of a rights-protected entity increases, we need a systematic, easily adopted framework for designating who has access to what code, and under which conditions.

While adding access controls to libraries and modules (as well as functions held securely within them), we also give regard to the performance characteristics and ease-of-use considerations. In this vein, we discuss the design and implementation of a framework (called SecModule) used for generating (and using) libraries under access controls, as well as performance measurements of invoking functions that are held inside the protected library.

1 Introduction

UNIX-like operating systems have a long and rich history deeply intertwined with the C programming language[10, 9]. The code development process today is remarkably similar to the early days (modulo the actual development tools, which have progressed since then). A programmer starts off with some source code, which then gets compiled into object files, perhaps grouped into a library, which is then linked together to form the final product.

As time goes by, the amount of written code increases, as well as its complexity and interdependency. Along with this is the fact that the amount of security vulnerabilities (both in number and in scope) are increasing as well.[2]. The apex of the current trend is not yet clear. However, we can identify three related cases where the current model of software organization and reuse is insufficient.

Suppose the existence of a piece of executable, reusable code that represents a significant investment of time, effort and capital. For the owner/creator of the code, the right to use, or invoke the functions in held in this library can be a valuable asset in terms of income, or perhaps merely recognition and acceptance. He would like some way to either get payed, or at least get recognition for his hard work. He may also wish to limit the possibility outright theft of the work.

Suppose the existence of a piece of executable code that represents a significant drain of computational resources. The owner of the host system may wish to control access to the rights to invoke this code, purely for the sake of preventing the host system from being flat-lined by over-use. The administrator may wish to limit access to the resource-hungry library according to certain criteria other than the carte-Blanche root access.

Suppose the existence of a piece of code that represent a critical component of a security infrastructure, such that misuse (intentional or otherwise) of the call can cause significant disruption of the host system. The maintainer would like to make sure that only those who have been certified to use those functions are allowed access.

We have identified three important issues related to software - financial (or fame) aspects, CPU usage and general system availability aspects, as well as security maintenance aspects, that our current model of sharing, and allowing access to chunks of important code held in a sharable library or module, is inadequate for the purposes at hand. The current UNIX methods for access control is purely binary, and coarse grain at that. All this point to a need for some kind of a trust-management[4, 3] framework for the creation and distribution of, and access to arbitrary pieces of code.

Providing a complete solution that solves the above three

1-4244-0054-6/06/$20.00

problems is likely to be beyond the scope of any single work. Nevertheless, we discuss a prototype software framework, called SecModule, which can be the basis of building such a system. Adding in authentication requirements to the calling of functions in a library is a very simple concept that is long overdue. Its very intuitive especially when thinking of computation as a protected resource.

This work attempts to answer the following questions:

1. What are the ways to generate a secure "handle" which will allow only processes that has successfully shown its valid credentials to allow to call or invoke functions within the library?

2. How are the risks of misusing the access handle minimized? That is, is it possible to limit the handle to be usable only by a validated process? In other words, only processes that have successfully authenticated should be given the ability to invoke the library method, and the handle must be valid only for a specific process.

3. What are the performance implications of having replacing normal libraries with the SecModule framework?

In the next section (2) We discuss additional background material. We then address questions 1 and 2 in section (3). Question 3 is addressed in the section (4). We conclude with lessons learned and possible future research.

2 Background

In this section, we discuss additional background material as well as related works.

One very important item that can be thought of as a spiritual ancestor to our work was not by software engineers, but by hardware designers. To be more precise, the developers of the Intel 80286/80386 CPU foresaw the need for having a hierarchy of access rights to (important) pieces of software[1].

In their widely disseminated works, the Intel engineers describe a set of "protection rings" which denote the amount of privilege that the software belonging to that ring possess. The innermost (or level 0) is the most privileged, whereas the outer ring (level 3) is the least privileged user-level code. Their foresight is underreported today. Matter of fact, newer versions of Intel CPUs do *not* possess the full four level hierarchy, and make do with two privilege levels. To be fair, what the Intel engineers saw was the privilege separation between the kernel, and periphery code such as device drivers, and finally the user level code. Unfortunately, software engineering as a science had not quite reached that level when the 80386 was released, and many system developers chose to use a simpler model, grouping device drivers and other system services in the same realm as the OS kernel.

Remote Procedure Call[12] can also be thought of as an early example of a "session managed" access to functions. It has very wide spread use in networked environments, and is used to implement vital services such as Network File System (NFS)[5].

Traditionally, UNIX and variants all feature a coarse-grain binary privilege escalation. Access rights were associated with a specific login ID. Because of this, it was difficult to formulate and enforce a fine-grain policy with respect to library access. And access to functions, once given access to the library containing it, was automatic and irrevocable.

To state by analogy, what we desire is to effect a more flexible "security region" where access levels and regions are not necessarily discrete. In other words, the question of access must now be delegated to some sort of a computational process that can enforce a complex policy that may not respond to a discrete privilege level.

Towards this end, there have been some positive signs. The OpenBSD operating system (`www.openbsd.org`) prides itself in being an operating system geared towards security in mind. Secondly, also related to OpenBSD is Systrace[11] which can be used to generate and enforce a fine grain policy based control over applications in the system calls they invoke.

Systrace does an admirable job - its only drawback is that the behavior of software captured by systrace is (counter-intuitively) too verbose. Because systrace guards the lowest level services provided by the operating system, certain higher level actions wind up being difficult to discern. For example, opening a window on an X11 based application is achieved through a fairly large set of system calls - and the final end result of the creation of a new window is hidden beneath the massive amounts of verbiage generated by systrace.

In other words, when one thinks of the sequence of system calls needed to implement a complex operations found in existing system or user supplied libraries, the fine grain control supplied by systrace, by itself, is immediately insufficient for the task at hand. Even worse, it may introduce subtle problems if the sequence of system calls used for implementing a higher level functionality is inadvertently interrupted in the middle by a misconfigured system call policy - resulting in the *library* code being in an inconsistent state. Because of this chaining of system calls for higher level actions, it may also be difficult to phrase a sufficiently precise systrace policies for applications that use a higher level abstractions many layers removed from the system calls.

To address this issue, through SecModule, we raise the

protective shield to the *library* level. In essence, we wish to provide fine grain policy enforcement over not just system calls, but calls to user level libraries as well. Our contribution is system which can be used to systematically formulate and formalize rights management for *software*. The access rights in question would be whether an entity p (which may be malicious) is allowed to execute some function f_i held secure in the library module m.

3 Generating a Secure Handle

A process p runs, and during its execution, p requests access to some function f_i contained in SecModule m. Initially, all images in m remain inaccessible to p. Once the request has been successfully processed, the SecModule system provides to p a handle h which allows access from p, and only p to f_i. The last criteria, to enforce that p and only p is allowed to access to f_i is ensured by the following:

The handle h, is a "co-process" that is started upon request for access to m. The actual dispatch to f_i in m is via an indirect call, managed by the OS Kernel.

Arguments and return values are marshaled and unmarshalled in the traditional stack passing mechanism, described next. The simplest policy is to allow access to m for the lifetime of p. Other policies may be implemented with this scheme.

A separate tool chain registers the SecModule m with the kernel, which must keeps track of the registered SecModules. At some point in time, the client process p, with credential c then makes a request to the kernel for access to the SecModule m. The kernel then verifies that c, is valid with respect to m's policy, and that m (consisting of name and version) actually is a registered SecModule. If so, then the kernel starts a new "co-process" h, linked with p, allowing a form of shared memory access between the two processes.

There are several problems in trying to share memory between processes using existing mechanisms (e.g. SystemV shared memory). First, any explicit shared memory model precludes sharing of large amounts of data. In fact, the required argument marshaling and unmarshalling develops the same flavor as that of the XDR (External Data Representation) Protocol used in RPC[12], and we were considering the generation of tools akin to `rpcgen` for SecModule.

It also became apparent that this design precluded its use for "retrofitting" much of the existing libraries into SecModules. The most fundamental call that was precluded was `malloc()`. Therefore, relying on an explicit shared memory model using existing OS primitives limits SecModule to "new" libraries.

The above problems go away if we presume that the processes p and h share the *entire data, heap and stack portions* of their virtual address space. It is important to point out that the text (or code) section *is not shared* between the client process and the handle. We achieved this by modifying several functions in the UVM[6] virtual memory subsystem of OpenBSD. With this approach, the handle has complete access to the entire data region of the client, such that even C library functions like `malloc()` can be placed inside a SecModule, working identically to its man-page specification within the SecModule framework. Some functions in `libc` *does* need to be handled specially, discussed later in 4.3.

3.1 Security Implications for the Operating System

In this section we answer a question that was implicitly stated in the prior section.

Why is the code body of f_i mapped to the handle h instead of the requester process p?

The answer is simple. With the limitation that C, assembly or some derivation thereof, is used to develop the application that spawns the in-memory process p, there can be *no trust* placed on any memory portion directly under the control of p. The code for f_i can not be available to p, because then p can jump past any guard code that protects f_i directly to the important parts, negating any protective aspects.

Assuming that the user who owns p received the credentials legitimately, the requirement for allowing access to m is still there. So the obvious solution is to control access to each call to f_i through a kernel level call, to get around the restriction that p can not directly access the code body of f_i. The arguments for f_i are passed on the shared stack like a normal (non-SecModule) function call. Then p invokes f_i indirectly by invoking a new kernel method `smod_call()` which will then verify that p did provide the proper credentials, and passes control over to h which will execute f_i on p's behalf.

This abstracted function call is not necessary if the OS and programming language itself did not allow arbitrary formulation of addresses and jumps, and code generation resources themselves are part of a trustworthy policy management. But such OS and language does not yet exist, and we are forced to accept this slowdown in order to increase the level of security.

In summation, the minimal set of changes needed in the OS are as follows:

1. Several new kernel level calls with the associated user level wrappers. See Figure 4 in Appendix A. The several `void *` arguments are pointers to structures that contain the needed arguments, i.e. additional information about the bodies themselves - generated through external tools.

2. Several new functions, as well as modifications of existing functions in the UVM virtual memory system. See figure 6 in Appendix A.

3. Processes no longer generate a core image when they crash. Certainly no Handle process should! Otherwise, f_i can be easily stolen by the user.

4. `ptrace()` and related kernel calls must not allow tracing of any processes associated with the handle.

4 Implementation Details

Our prototype of the SecModule system is implemented on top of OpenBSD v3.6 running on an PentiumIII PC. Figure 1 shows a high level overview of the steps by the client process p to access a function (in this case `malloc()`) held secure within the SecModule version of `libc`.

In step (1), the client's initialization code in `crt0` tries to open access to the module that holds the routine we want to access. Once the kernel has acknowledged that the requested module exists, the client executes the `smod_start_session()` call, which relays to the kernel the formal request by the client process for the module identified by `m_id`. The `smod_start_session()` needs a pointer to a structure that identifies all the modules

Figure 1. The SecModule Initialization Sequence

Assuming that the credentials check out, in step (2), the kernel forcibly *forks* the child process, creates a small, secret heap/stack segment for the handle, and executes the function `smod_std_handle()`, using the secret stack. This secret stack *is not available to the client*. Refer to figure 2.

On step (3), the handle starts the first phase of the handshake by executing the `smod_session_info()` system call, which informs the kernel that the handle is ready to go. This system call also forcibly unmaps the entire data, heap, and stack segment of the handle process and forces it to share the memory pages from the same address range from the client process (Refer to figure 2). This system call may load in additional code segments as needed to fulfill the requirements of the module.

Figure 2. Address Space Layout

On step (4), the client process concludes the handshake by calling `smod_handle_info()` which completes the internal synchronization data structures that the client and handle must use to communicate with each other. Then the client process's `crt0` completes by executing the main routine for the client, called `smod_client_main()`.

Inside `smod_client_main()`, in step(5), the client makes a call to `malloc()` which is in reality a relay to `SMOD_client_malloc()`. In step(6) The client stub routine invokes the kernel's `smod_call()` to start the actual call.

Some time later, in step (7), the handle receives the call, and relays the message to the real `malloc()` routine held inside it. Step (8) concludes by returning to the client.

Figure 2 gives a diagram of the address space of both the handle and the client processes. After the handshake completes as described above, the client process and the handle process is sharing the same pages for the address ranges that start just below the traditional OpenBSD data segment, to just above the end of the traditional OpenBSD stack segment bottom. All other portions *are not shared*. Specifi-

cally, the region marked "Secret Stack/Heap" is *only* available to the handle process's `smod_std_handle()`. The top half of that secret space is used as the stack space by `smod_std_handle()`, to avoid colliding with the shared stack between the client and handle.

Now we describe in detail the calling sequence, in so far as the shared stack space is concerned.

Figure 3. Stack Manipulations

In Figure 3, step (1) shows the state of the stack inside the client's assembly stub routine (e.g. `SMOD_client_malloc()`, before the kernel level call to `sys_smod_call()`. Step (2) shows the state of the client's stack inside `sys_smod_call()`. Notice that the assembly stub routine pushed in the unique identifier pair `moduleID, funcID` used to point to the function (and the module which contains it) that is being invoked. The top 2 elements from step (1) needed to be duplicated so that the kernel has the correct view of the relevant arguments. Technically, the kernel only requires `client_FP_1`. However, using only that exposes the kernel to unnecessary architectural dependencies. Step (3) shows the same stack from the view point of the handle, inside the `smod_stub_receive()`, which is executed by the handle to accept the invocation. Note that the handle has popped off all of the unnecessary elements on the stack above `arg1`. At step (3), the handle then relays to the actual library routine named by `moduleID, funcID`. The called function has access to the entire stack and data of the client process, as per normal (non SecModule) function call semantics. After the called function returns, in step (4), `smod_stub_receive()` then replaces the exact same arguments that the client stub routine had seen, so that it can properly return to the original calling location.

It is important to note that the handle process actually executes `smod_stub_receive()` using the secret *alternate stack* that was set up when the handle process initialized, shown in figure 2. Therefore, the execution of the handle-side stub routine can not disrupt the shared stack and data between the handle and client. In other words, `smod_stub_receive()` sets the stack to the *shared stack* before relaying the call to the actual library routine.

4.1 Modifications Required in the Kernel

The implementation of SecModule can be broken up into three parts. First involves the sharing of the data and heap. The second involves the proper synchronization between the client and handle. The third portion deals with making sure that the executable code held in the module is not made available to the client inadvertently.

To achieve the first goal, we can follow two equally good approaches. First is to rely on the existing UVM interface to mark the address space between the data and the stack as shared, then `fork()`. The second approach is to forcibly unmap the pages in one process and to then forcibly map the pages from the other process onto the first. We chose the latter approach, and added several new function for the UVM[6] virtual memory system in OpenBSD to achieve this. The first function we added was `uvmspace_force_share()` which uses existing UVM internal interface to first unmap all `vm_map_entry`s in the share region of the handle process, then to duplicate the actions of `uvmspace_fork()` by duplicating (and sharing) the entries from the client's process for the address range.

We must also ensure that the relevant pages remain "shared" even as the client process's heap/stack grows and shrinks. For this, we needed to modify the low level `uvm_fault()` routine, such that on a "unavailable mapping" error, `uvm_fault()` examines the faulting address with respect to the other process, to see whether it has a valid mapping for that address. If so, then `uvm_fault()` maps that entry onto the faulting address as a share. We also modified `sys_obreak()` to request additional heap space as *shared*, if the request came for one of the process in a SecModule pair. We also modified `uvm_map()` to create a shared mapping for the cases of the modified call from `sys_obreak()`.

The second goal of keeping the client and handle synchronized is much easier to achieve, as OpenBSD already comes with the proper kernel resources in the form of SYSV MSG interface. The `msgsnd()` and `msgrcv()` functions already contain efficient blocking and awakening that we desire for synchronization. So for the second goal, no changes were needed for testing purposes.

The third objective, of ensuring that the client process does not have direct access to the actual text of the functions held in a SecModule can be done in either one of two orthogonal approaches. The first approach is simply to encrypt the library using a secret key not revealed to the client process, using a a sufficiently powerful system like the the Advanced Encryption Standard[7]. We only encrypt regions in the library's text that do *not* correspond to relocation or linking data. That is, we do not touch any locations in the library that will need to be modified by the linking process. That way, the encrypted version of the library is still linkable using existing tools, but the unencrypted form will be available *only* to the handle process, after the kernel decrypts the relevant memory locations in the handle's text portion.

The second approach, which works well for dynamic libraries, is to simply have the kernel unmap the images of the shared library from the client's address space, as well as deny the ability of the client to load in plain text versions of the SecModule later on. As long as we can trust the fact that shared objects can only be directly accessed by the operating system, it is a perfectly valid way to maintain control over access to the libraries and remain carefree with regards to encryption.

There is nothing preventing both approaches being used, or using encryption to protect dynamically loaded libraries in a similar fashion.

4.2 Generating Stubs and Using Them

The SecModule system requires that a set of stubs be used to access the original symbols in the library. Our approach was to start off with the output of `objdump -t /usr/lib/libc.a | grep ' F '` and to slowly add in the ones we missed as we ran across them. The output of `objdump` was useful because they were guaranteed to be functions in the library. For the rest, we used the macro definitions already in the headers, as needed.

Most functions in the libraries targeted for SecModule conversion resolved to calling autogenerated assembly stub function. Because of the stack manipulation operations required by the stub, it can not be written in C, but must be duplicated for each library function that needs a a client side stub. This is done as part of the SecModule toolchain for processing libraries.

Using the SecModule `libC` is nearly identical to the traditional case, save that we must specify a custom linking procedure to make sure that the special `crt0` is linked in, and that the objects that hold the name and version of the needed SecModules, as well as the credentials that allow access to it are linked in. From the source code perspective, it is also nearly identical to the traditional case. The only change is that each source file (C, or C++) must have a single additional `#include` statement after all system `#include` statements, but before the user code, so that the SecModule client-side access functions which override the system header files get properly mixed in. Executing the SecModule enabled client must be preceded by the OS kernel's recognition of the SecModule about to be requested.

4.3 Special Functions

Certain function calls in the C library required special handling when they were converted over to the SecModule framework.

For `execve()` and variants, the action taken at the kernel level is to first detach the requesting client process from the SecModule system, kill the associated handle process, and then to run `sys_execve()` system call as per normal. If the resulting executable is a SecModule registered executable, its `crt0` will correctly execute the required set of system calls to set up a new SecModule session.

For `fork()` and variants, the ideal action is to duplicate the child process twice, and force the first child to be the handle for the second. This task is made complex by the fact that it is tricky to achieve a chained set of system calls on behalf of the child process after a fork(). Thus some of the heavy lifting for fork is implemented as a handle-side code that sits outside of the kernel. Multiple clients should not share the handle, because a many-to-one mapping of clients to a single handle introduces a performance bottleneck.

Obviously, `getpid()` and related calls must return the PIDs related to the client, not the handle! Similarly, signals, and scheduling routines like `wait()` and their variants must be modified such that they effect the *client*, not the handle.

This list of functions held in `libC` needing special handling may not be exhaustive. There are nearly 1500 global text symbols in the OpenBSD `libC`. Auditing them for correct behavior within the SecModule framework will take some time, even for the most enthusiastic programmer. The rule of thumb seem to be that if they involve scheduling, signals or processes, then they will likely need additional work in order for correct operation. It needs pointing out that *without* the sharing of data/heap/stack, adding in access controls for access to existing libraries such as `libC` becomes much more difficult.

4.4 More Security Considerations

When considering encryption[8], it is important to note that the secret keys that wrap the individual functions in m are never revealed to p. Once the SecModules are registered, the secret keys for each encrypted segment in m exist *only* in kernel space. As always, extreme care must be taken

when choosing the pseudo-random keys for the symmetric cipher that actually protect the bulk of m.

In our test case, there are two principals, the SecModule implementor and the client. The creation and registration of the SecModule is handled by the same principal. However, in more realistic scenarios, The SecModule m exists in a truly multiuser environment, and there is a third principal, which is the system s that hosts m. In cases like this, s must be a trusted party and the secret keys that protect m are encrypted using s's public key, and is shipped as part of m. In both cases, the operating system which hosts m has to be a trusted party. If this is not the case, then a security prerequisite is not met, and SecModule's guarantees become invalid.

Handling multi-threaded client programs is a special challenge when auditing their behavior. As others point out[13, 14, 11], its is possible for multi-threaded clients to first give innocuous arguments, evade the permissions check, and then *modify* the arguments on the stack.

Preventing this attack in a user-land process is more difficult, but it can be done in several different ways. First, we can simply *unmap* the entire data and stack region of the client (including all threads) during the kernel level execution of `sys_smod_call()`. A second approach is to also forcibly remove the client (and all threads related to the client) from the ready queue. The second approach has the benefit of having lesser overhead for the kernel. However, neither approach is very desirable in terms of client efficiency. Other approaches like partially mapping a stack segment read only are fragile and not necessarily more secure than the two brute force approaches. For the time being, we do not consider this issue.

4.5 Performance Characteristics

For our test case, we measured the cost of the indirect dispatch from the user process p, to the handle process h and back again. We compare against an identical no-op function implemented as a locally running RPC[12] service, as well as the native (non-SecModule) `getpid()` kernel call as a point of reference. Our test machine is described in figure 7 in appendix A. The measurements are detailed in figure 8. Our initial measurements demonstrate that invoking an unpacked SecModule function is slower than a simple kernel call (compare the difference between native `getpid()` and `getpid()` over SecModule), but is no more expensive to invoke than a locally served RPC call. Matter of fact, due to the tight coupling between the handle and the client, invoking a SecModule function *is roughly 10 times faster* than the identical function being executed via RPC. The function tested for both RPC and SecModule returns the argument value incremented by one.

5 Conclusions

We have shown an easy-to-use software framework which allows retrofitting of existing libraries, as well as develop new ones into a secured, session-managed environment. Our framework can be used to address the three issues raised earlier in section 1. We achieved our goals through selective sharing of memory pages between the handle and the client, such that the functions being executed by the handle on behalf of the client gets to access the entire data, heap, and stack space of the client process. We have a prototype implementation consisting of the kernel mods, a SecModule conversion of `libC`, and related userland registration tools. We will provide source code upon email request.

As discussed above in section 4.5, our performance numbers are quite good, with a factor of 10 increase over RPC. We believe that its possible to gain even greater performance gains by reducing redundant error checks and cross-address copies in kernel-to-kernel calls used for our protocol.

Thus, the performance numbers above (modulo any future improvements) serves as a good estimate of the lower bound for traditional "always allowed" access policy within SecModule. If we need to evaluate more complex policy statements, we can expect a corresponding slowdown in proportion to the complexity of the required access control check.

Our initial design included the use of KeyNote[3] policies as our definition language, but we have concluded that the integration of KeyNote policy engine with Systrace's own policy definition system requires some additional thought. Therefore, for this work, we avoid discussing the definition of nontrivial policies. Instead we concentrate on highlighting a framework in which such rules can eventually be specified and effected, given suffcient advances in policy definition and enforcement technologies.

References

[1] *The Intel IA-32 Software Architecture Manual*. Intel, 2001.

[2] S. M. Bellovin. An End State? In *Communications of the ACM*, volume 44. ACM, March 2001.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. RFC2704: The KeyNote Trust-Management System Version 2, 1999.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. *The Role of Trust Management in Distributed System Security*, chapter Secure Internet Programming: Security Issues for Mobile and Distributed Objects. Springer-Verlag, 1999.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. RFC1813: NFS Version 3 Protocol Specification, 1995.

[6] C. D. Cranor. *DESIGN AND IMPLEMENTATION OF THE UVM VIRTUAL MEMORY SYSTEM*. PhD thesis, Sever Institute, Washington University, August 1998.

[7] J. Daemen and V. Rijmen. *The design of Rijndael : AES–the Advanced Encryption Standard*. Springer, 2002.

[8] W. Diffie. The first ten years of public-key cryptography. In *Proceedings of the IEEE*, volume 76, pages 560–577, May 1988.

[9] B. W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, March 1984.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[11] N. Provos. Improving Host Security with System Call Policies. In *12th USENIX Security Symposium*, pages 257–272, 2003.

[12] R. Srinivasan. RFC1831: RPC: Remote Procedure Call Protocol Specification Version 2, 1995.

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

[14] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communication Security*, November 2002.

A Figures

```
301 STD { int sys_smod_find(const char *name, int version); }
;; sys_smod_session_info() is ONLY for the handle process,
;; that is, the handle process started by sys_smod_start_session()
303 STD { int sys_smod_session_info(void * sinfo); }
;; sys_smod_handle_info() is ONLY for the client process
;; that is, the client process started by sys_smod_start_session()
304 STD { int sys_smod_handle_info(void *hinfo); }
;; allows multiple versions
305  STD { int sys_smod_add(void *smodinfo) ; }
;;
306  STD { int sys_smod_remove(int m_id, void *credential, \
                        int credential_size); }
;; Requires assembly hooks to properly pass in the frame pointer,
;; and the return address to the kernel.
307     STD             { int sys_smod_call(void *framep, \
                          void *rtnaddr, unsigned m_id, int funcID) ; }
320 STD { int sys_smod_start_session(struct \
                         smod_session_descriptor *descp); }
```

Figure 4. Necessary Additions to the OpenBSD Kernel for Implementing Sec-Module

```
// Declare first arg to be an int, just to get
// it to compile with varargs.
typedef int (*SMOD_funcp)(int, ...);

// Called by the client to request access
// to the function identified by funcID
// Relays to sys_smod_call() ...
extern int
smod_stub_call(int funcID, ...);

// Called by the handle to actually execute the function
// pointed to by funcp on the shared stack
extern int
smod_stub_receive(void*shmsegp, SMOD_funcp funcp);
```

Figure 5. The C Declarations of Assembly Stubs

```
// New functions ..
// in uvm_map.c:
/* Where the original uvm_map() went to ... */
int
uvm_map_internal(vm_map_t map,vaddr_t *startp,
                 vsize_t size, struct uvm_object *uobj,
                 voff_t uoffset, vsize_t align,
                 uvm_flag_t flags);
/* Try to map the same anon in the same place in both processes */
int
uvm_map_shared_internal(vm_map_t map1, vm_map_t map2,vaddr_t *startp, vsize_t size,
                        struct uvm_object *uobj, voff_t uoffset,
                        vsize_t align, uvm_flag_t flags);
// The job of this function is to force the sharing of a portion of the VM
// between two processes.
// It achieves this by unmapping all pages between just a sliver above the
// traditional code space, reaching to the bottom of the stack space, and the
// relying on uvm_fault() to allow the sharing of the pages in between them.
int
uvmspace_force_share(struct proc *p1,
                     struct proc *p2, vaddr_t start, vaddr_t end);
// called by above.
int
uvm_force_share(vm_map_t map1, vm_map_t map2, vaddr_t start, vaddr_t end);

// Modified functions
// uvm_fault.c
int
uvm_fault(vm_map_t orig_map, vaddr_t vaddr, vm_fault_t fault_type,
          vm_prot_t access_type);

// uvm_map.c
int
uvm_map(vm_map_t map, vaddr_t *startp, vsize_t size, struct uvm_object *uobj,
        voff_t uoffset, vsize_t align, uvm_flag_t flags);
// In uvm_unix.c
int
sys_obreak(struct proc *p,void *v, register_t *retval);
```

Figure 6. Changes to the UVM Virtual Memory System

```
OpenBSD 3.6 (sys) #69: Tue Jan 25 03:52:35 EST 2005
cpu0: Intel Pentium III ("GenuineIntel" 686-class, 512KB L2 cache) 599 MHz
cpu0: FPU,V86,DE,PSE,TSC,MSR,PAE,MCE,CX8,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE36,MMX,FXSR,SSE
real mem  = 536440832 (523868K)
pcib0 at pci0 dev 7 function 0 "Intel 82371AB PIIX4 ISA" rev 0x02
pciide0 at pci0 dev 7 function 1 "Intel 82371AB IDE" rev 0x01: DMA, channel 0
    wired to compatibility, channel 1 wired to compatibility
wd0 at pciide0 channel 0 drive 0: <IBM-DPTA-372730>
wd0: 16-sector PIO, LBA, 26105MB, 53464320 sectors
wd0(pciide0:0:0): using PIO mode 4, Ultra-DMA mode 2
cd0 at scsibus0 targ 0 lun 0: <SAMSUNG, CD-ROM SC-140B, d005>

CLOCK_TICK_PER_SECOND is 100
```

Figure 7. Abbreviated Test System Information

	Number of Calls/Trial	Total Number of Trials
GETPID	1,000,000	10
SMOD(SMOD-getpid)	1,000,000	10
SMOD(testincr)	1,000,000	10
RPC(testincr)	100,000	10

Test Function	microsec/CALL	stdev(microsec)
getpid()	0.658000	0.00918937
SMOD(SMOD-getpid)	6.532000	0.29850740
SMOD(test-incr)	6.407000	0.07513691
RPC(test-incr)	63.230000	0.13482911

Figure 8. Performance Comparisons