Operating System Protection Domains

Eric Tamura, Joel Weinberger and Aaron Myers

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-08-02
April 2008

Operating System Protection Domains

Eric Tamura, Joel Weinberger, Aaron Myers
Computer Science Department
Brown University
{etamura, joel, atm}@cs.brown.edu

1 Introduction

Access control plays a vital role in how almost any system is run, but it also varies a great deal across systems. Butler Lampson's original Access Control Matrix[1] first formalized access control. However, his system, even with a sparse matrix, proves too large and complex to maintain in reality. In systems with thousands and thousands of resources, across many different dimensions, it is simply unreasonable to maintain such specific information.

In response to this, many different generalized access control systems have developed. Some, such as discretionary access control policies, are very flexible but do not provide the guaranteed security that a system might require. Others, such as mandatory access control policies, provide the inviolable security that a system might need but lack adjustability and flexibility. Still others tried to provide a middle ground somewhere between the strict but secure mandatory policies and the flexible discretionary. One such example is a role-based security model as described by Sandhu, et al[3].

Many systems choose to maintain a semi-role-based security model in order to have both a secure and flexible system. One of the most common examples of this is the Unix security model. In the Unix environment, the system creates a hierarchy based on ownership and grouping principles. When a resource is created (namely, a file on the file system), the creator of the file is specified as the owner of that file. Creators are granted the ability to assign access privileges to that file based on their own rights, the rights of those in related roles (i.e. groups), and all others.

In many ways, groups act as a role would. Groups contain a set of users, and system resources can be assigned to a group. Thus security in Unix can be specified to a set of users through the use of a group. However, groups are not a complete role-based access control system as there is no hierarchy among groups and resources cannot be assigned to multiple groups, among other details. However, groups are a very practical and effective role-based system for most needs in Unix.

In addition to these standard access control mechanisms, there is a discretionary side to the Unix security model as well which includes the superuser and the effective user id systems. The superuser (also called the "root" user) is given the privilege of have complete discretionary control over the system, being able to access any system resource, change any system resource, and take on the role of any individual user. The root user is also allowed to grant a subset of his privileges to users through the "sudo" ability. This allows users, to execute specific, possibly limited, commands as the superuser. Additionally, the effective user id mechanism allows any user, including the superuser, to allow a program to run under his id. Thus, certain rights can be granted on an

individual basis to run as another user, allowing users to distribute their rights to some degree on a discretionary basis.

The discretionary and role-based systems in Unix have provided what turns out to be a relatively flexible and secure system, despite the presence of flaws and limitations. Some of these flaws, most notably the buffer-overflow security hole, are significant and can compromise the security of the system. The buffer-overflow hole refers to a program whose data buffer, stored in stack memory, is overwritten beyond its boundaries, affecting its return addresses and the code it returns to. If such a program has an effective id set to that of the superuser, it could grant superuser privileges to a malicious user. While this is certainly a limitation, it is preventable through properly written code and is not the focus of our work.

The basic problems are based on the user-centric security model of modern operating systems. These systems fail to present a semantic separation between the user and executing programs. Thus, a user must trust binaries he executes, since doing so inherently puts his system at risk. However, it is becoming increasingly popular to obtain and execute software from untrusted sources, such as the Internet. Thus, the semantic separation of users from executing programs has become increasingly important as users become less and less trusting of the software they are using.

There have been some attempts to extend Unix security. One of the most notable examples of this is SELinux (Security Enhanced Linux). SELinux adds a very flexible and extensible access control system. Through the addition and use of Linux Security Modules to the Linux kernel, SELinux allows for the creation of arbitrary security models. However, these security models are extremely complex and difficult to use, as well as operating under Mandatory Access Control policies, in contrast to discretionary models.

In response to the limited model of user-centric security presented in many modern operating systems, we present a new model of "Protection Domains" to guarantee the security of an execution of a program. The model presents a semantic separation between the user and the program so that a user does not necessarily need to trust a binary in order to execute it securely. Furthermore, we present a "Guardian System" that allows the user to implement arbitrary security policies easily and dynamically.

We allow users to perform a "restricted execution" of a program using a new system call, `rexec`. This specifically denotes the execution of a program in the operating system kernel as being a process to watch and to intercept its access to system resources. This provides a new model of specifically denoting restricted processes that are untrusted, or trusted only minimally. Combined with the notion of a "Guardian System," users can be assured of the integrity of their system.

We expand on the work of Peng[2] to create a model of access control that provides a secure method for the execution of untrusted binaries. We provide the framework and an implementation in the Linux environment for a guardian based system of watching these programs so their accesses to system resources can be monitored, and the access is left to the discretion of the user.

2 Motivation

Traditional operating system access control models only provide a view from one axis of Lampson's original access control matrix. While the security policy provides a model of user access control, it provides almost no notion of resource access control. Put another way, while a user can be prevented from accessing a resource on the system, a resource (i.e. an executing program) that is acting as a user cannot be prevented from accessing another resource.

This provides a good model for controlling users. Effectively, it allows a system to allow untrusted users to access it by limiting and controlling what they can do in the system. However, users have no way of guaranteeing the safety of arbitrary code that they may run on a given system. For example, a user might run an arbitrary program that would delete his entire home directory because the user has full read/write/execute access in this space. Thus, while an administrator can allow untrusted users onto the system, a user cannot (or, at least, should not) allow untrusted code to execute on the system.

We are motivated by a desire to provide a way for users to execute untrusted code safely and securely, without concern for how it might affect their system resources in a traditional operating system. Despite several attempts to resolve this, most systems fail to provide a truly useable, flexible, and secure model. We believe the Protection Domain model addresses these issues.

3 Guardian Model

In order to build a protection domain for the execution of a program, a security policy must be defined. More specifically, there must be some sort of method for allowing and denying access to certain system resources. How exactly such a policy will be expressed is not obvious.

One policy is to automatically deny from within the kernel all access to system resources by a program that is within a Protection Domain. Obviously, this is not a desirable policy as it is completely inflexible, and will cause almost any reasonably sized program to fail.

This idea can be expanded further. Another type of policy could be to statically assign some group of resources that a program can access. This group of programs could simply be a list, or a complex regular expression that defines files. However, the problem is that even with a regular expression defined in the kernel, it is virtually impossible to pre-define every possible valid access a program could make in system resources, especially for programs that have not been written yet.

Thus, a different method for defining policies is necessary. Based on the previous example, it is apparent that the policy system needs to be dynamic to accomodate different types of programs with different types of access restrictions. It also seems necessary that a user be able to define his own policy since the trust of the program lies with the user.

Our answer to these issues is in the guardian model we develop. The guardian model relies on a user defined program that "watches" a restricted process, logging the restricted program's resource access, and performing security checks on it. In our model, the guardian program communicates with the kernel of the operating system such that the operating system alerts the guardian of access points in the restricted program for which a security decision is made.

The key semantic notion for the guardian model is that the guardian program is user defined. The kernel simply alerts the guardian of the access points, to which the Guardian responds with either "accept" or "deny," either allowing or rejecting the restricted program's resource access. How the guardian decides on an "accept" or "deny" response is completely up to the guardian process. Thus, the guardian process allows for simple, arbitrary security policies to be defined.

In actuality, it is much better to think of a guardian as a security policy than as a "watch dog" process, as this is the intended behavior. This abstraction allows us to separate the policy from the internals of the operating system, as well. The idea of a guardian being separate from the operating system and enforcing a policy in a user context is a very important part of the guardian's flexibility.

Another possible Guardian model would be to define a separate policy language that the operating system could interpret dynamically. We feel, however, that by providing a system call-level API, a user can more effectively intergrate their policies into their session. One useful example of this would be a Firefox monitoring program, implemented as a plug-in, that it executes.

One of the most basic security policies imaginable would consist of a guardian that asks the user for his decision about whether the restricted program should be able to access a specific resource. We have included such a guardian with our software. Such a simple policy can be quite powerful, and the guardian model allows for many ideas to be extended from this one. For example, a program enforcing such a policy could be easily modified to allow access to any file that is in an arbitrary directory, while then asking the user about any other accesses. Alternatively, a guardian could allow a user to input a regular expression for all files that a restricted process can access. It is easy to see that a dynamic, user-defined guardian policy can quickly become quite powerful.

One of the most important aspects of the Guardian is its security. Since it is meant to be a user level process, what prevents a malicious program from bypassing the Guardian? The first aspect of this is its communication with the kernel. Upon execution of a restricted program using the `rexec` system call, a specific guardian process is also executed. The kernel is alerted of its presence, and the guardian then connects to the kernel using a predefined protocol.

This naturally leads to the question of what prevents the restricted program from connecting to the kernel itself as its own guardian? In actuality, nothing prevents this. The key insight is how the guardian responds to access requests relayed by the kernel. Before, we presented a notion of replying with "accept" or "deny." In reality, the Guardian does not simply reply with an "accept" if the access is accepted, but replies with the access to the resource itself.

For example, in our Unix implementation, on an "accept," the Guardian will open the file resource that is trying to be accessed and pass the file descriptor of the resource to the kernel using Unix Domain Sockets. This ensures that the Guardian itself can access the resource before conferring an "accept." If the Guardian tries to reply with an accept, but without a valid file descriptor for the requested file, the kernel will be unable to give the file descriptor to the restricted process, and thus, the restricted process will effectively be denied access to the system resource.

There may exist reasons, such as a shell running in a restricted context, that may cause a guardian to run in a restricted context. Since the restricted property is passed across all calls to `fork()` and `exec()`, it is simply a matter of calling `rexec()` on a binary that then executes a guardian and the restricted process. This forces the child guardian's file accesses to pass through the original guardian before being processed. This further restricts the child guardian from granting access to files that it itself cannot grant.

This system also presents several other interesting opportunities for sandboxing. For example, a "shadow file system" could easily be developed as a guardian. Imagine that every time the restricted process attempts to access a file the guardian copies over said file to some other directory and returns "accept" along with the file descriptor of that "shadow file." Neither the kernel nor the restricted process would have any indication of this occurrence, and the restricted process could modify this file freely without damaging the original file at all.

The guardian model provides a secure, simple interface for developing security policies. By associating every restricted process with a guardian, we can write arbitrary secure policies and even more valuable abstractions. Furthermore, the guardian is a logical abstraction, and it provides a mechanism to communicate with the user or policy about what the restricted process is doing.

4 Kernel Modifications and Implementation

In order to support this model of guardian and kernel communication, we made several changes to a base Linux kernel version 2.6.10. Specifically, we added a new system call as well as augmented the process duplication codepath and file opening codepath. We added features to mark restricted processes, since these processes need a means to communicate with userspace from within the kernel. Also, the filesystem is now used from within the kernel to pre-authorize files for restricted executables. As described previously, we also implemented a rudimentary guardian that queries the user for access, as opposed to following any "set" policy.

4.1 `rexec` system call

We implemented a new system call, called `rexec()`, which is short for restricted exec. Its purpose is to mark a new process as restricted, which will force it to communicate with the guardian for all file access. As explained in the guardian model, restricted processes

may perform arbitrary file access once the file descriptor is opened, but until this point, they must defer to the guardian to obtain permission. Alternatively, file access may be explicitly pre-authorized via special access control files, described later. The `rexec` system call involves duplicated structures both in user-space as well as the kernel, to ensure that message passing involves the same messages and structures.

We added several fields to the Linux `task_struct`, which also operates as the scheduling unit, as tasks are scheduled and descheduled by the Linux scheduler. Processes can contain several tasks, and these tasks have their own PIDs, even though they may all be part of the same thread group (and thus, the same process). These PIDs merely serve to identify tasks within the kernel framework. Our additional fields include flags to detect whether or not the task is restricted, whether or not the given process is a guardian, and whether or not the process was interrupted by a signal. Another flag tracks whether or not the task was spawned by a `sys_clone` call (as opposed to a `fork`). Other important additions include the file descriptor data for the parent's connection to the guardian, as well as the PID of the guardian communicating with the current task. Both of these latter items are necessary in order to ensure proper behavior is maintained when new tasks are created to support threads as opposed to actual processes.

During the call to `rexec()`, the kernel determines the presence of an access control file that explicitly outlines files that should be pre-authorized for the restricted process, with a specified set of permissions. For example, pre-authorized files and permissions might include read permissions on all files in `/usr/lib`, a standard directory for libraries. Presently, we expect that this mechanism be used for standard system libraries, such as the C library, caches for the loader, and other general operating system files whose accesses are hidden from most applications. It should be used for files that are frequently accessed and generally do not allow many modifications. Otherwise, larger, more complex security policies can be associated with the guardian, which can support the idea of allowable directories (and subdirectories) for blanket authorizations.

On calls to `do_fork`, this linked list is duplicated to ensure that the new task has the same access privileges as the parent. However, on subsequent calls to `rexec` (from a forked process, for example), the list is immediately destroyed, so that restricted processes cannot gain additional rights beyond those associated with the binary that they represent. The newly `rexec`ed process will then regain privileges corresponding to the access control file for its binary, if it exists. We are well aware of the algorithmic implications of this linked list; better data structures could be used, such as a hash table, to speed up, algorithmically, lookups of authorized files during `open` calls.

The rexec code could be modified to use the extended attributes of a restricted binary to store its pre-authorized files, instead of actual files on the filesystem. However, this presents several issues, the largest of which is the maximum size. Extended Attributes on Linux can only occupy at most 4 kB of disk space (1 disk block), which could be problematic with a large set of authorized files. However, this does ensure that there are no namespace conflicts, since we must currently define a naming convention to ensure that the kernel reads the file containing the list of pre-authorized files correctly.

4.2 `open()` and Guardian Behavior

The `open` system call, traditionally used to open files and return file descriptors, has also been significantly modified to handle the guardian communication pathways described earlier. `sys_open` is the typical Linux gateway for checking permissions and user validation prior to opening the file handle in the filesystem and returning the file descriptor to the user. As such, it was natural to modify this function to handle the socket communication with the guardian.

Presently, on a call to `sys_open`, there is a check to determine whether or not the process is restricted. If it is not, then the system call proceeds normally. Otherwise, it will check whether the pathname for the requested file is pre-authorized in the `task_struct`. If it is, then the system call proceeds as if it were not restricted. If the file is not pre-authorized, then the kernel defers to the guardian for the decision on whether or not to grant access to the file.

Communication to the guardian is performed via stream sockets using the Unix domain (as opposed to internet sockets). During the call to `rexec`, the kernel connects to the guardian and stores the file descriptor for this connection in the `task_struct`. During subsequent open calls, the kernel sends the guardian a request to open a file, and then blocks, waiting for a response. Both the guardian and kernel share data structures used for this communication in order to ensure uniformity in structure and size.

When the guardian accepts a new connection, it spawns a new thread to deal with the associated kernel task's requests, and continues to block while waiting to accept new tasks. These new threads are responsible for blocking on `recv`, waiting for data to be sent from the kernel representing a file request. Presently, the guardian performs a `stat()` on the file and then obtains a file descriptor for the file in question, if possible. The guardian then uses its security policy to determine its response to the kernel. Presently, the guardian prompts the user for a decision at run-time. Alternatively, the guardian could be configured to deny access to all files outside of a certain subdirectory, or to only authorize files with a specific owner. As explained earlier, the guardian's policy framework is very flexible.

If there were an error, or the file did not exist, the guardian will send appropriate error data, including `errno` values, back to the kernel. However, if the guardian intends to relay success to the kernel, it sends the file descriptors to the kernel using the `sendmsg` call over the socket. In this way, the guardian can transfer credentials to another process. Because the kernel is accepting the file descriptor (as opposed to a user-level application), we bypass any checks to determine whether or not the user is authorized to accept the file rights.

Once the low-level `recv` call completes in the kernel, we verify that a signal did not interrupt the socket operation. If it did, then we need to return the appropriate error value to allow the system call to restart, and bypass the original request to the guardian, so as not to put duplicate responses in the socket's buffers. If everything completed successfully, then the file descriptor has already been inserted into the file table for the specified process, and we can now simply return the file descriptor obtained over the socket.

Once the guardian's process `exit`s, it triggers signals being sent to all restricted/guarded processes to exit immediately. Because all guarded processes are owned by the same user who spawned the guardian, there are no issues with permissions when sending all of the restricted processes the KILL signal (-9). In order to implement this, we currently scan through all possible `task_struct`s to determine whether a process is restricted and if its guardian was the exiting process. We justify the large search space since the guardian is not expected to exit very frequently. Even if it does, it should tear down all of its restricted processes since they will have no means of contacting the guardian for file access once it has exited.

5 fork() System Call

As the restricted exec (rexec) plays a large role in our system, so should modifications to the `fork` system call, as `fork` and `exec` tend to operate hand in hand. In Linux, process duplication occurs in several routines, such as `vfork`, `fork`, as well as `sys_clone`. However, all of these routines act as wrappers around one major kernel function – `do_fork`, which actually performs the process duplication with the aid of other helper functions. As a result, we place all of our restricted process hooks and checks in the `do_fork` routine because it acts as the funnel for all higher level process duplication.

There are several security implications involved in duplicating a restricted process. First, restricted processes should not be able to circumvent the current security policy, which restricts file access. In other words, restricted processes should not be allowed to access files they would otherwise not be able to access by forking and spawning a new process. Secondly, the new process should be restricted, because its parent process was currently flagged as being restricted. The new process should be forced to delegate all decisions for file access to the guardian, who will decide whether or not to grant access for files not explicitly pre-authorized. The current value of the restricted flag in the Linux `task_struct`, as well as the list of pre-authorized files, is duplicated in the child process. Currently opened files in the file descriptor table for the parent process should also appear in the file table for the duplicated process. If a file were open in the restricted process, either the guardian allowed it or the file was already pre-authorized. In either case, the file should remain open for the child process.

In our first several iterations of the kernel modifications to support the guardian model, we encountered a serious bug that hindered development for a significant amount of time. In our changes to `do_fork` we naively duplicated all of the fields related to our system in the task struct, as well as allowing blind duplication of the file descriptor table. This reflected a superficial view of the file table as well as of functions calling `do_fork`. The first major flaw in this reasoning is that a duplicate process would need its own connection to the guardian if it ever intended to perform any meaningful communication with said guardian. Without this new connection the file table for the child task would appear as follows in Figure 1.

In this example, the duplicated process maintains a file descriptor pointer to the same socket connection to the guardian as did its parent process. Resolving this may seem

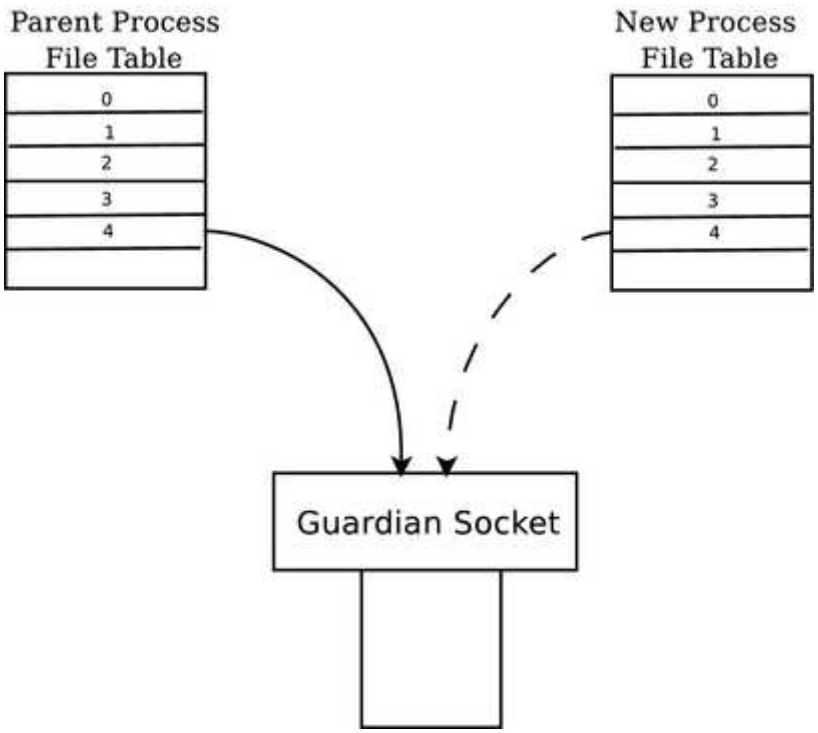


Figure 1: Process File Table and Forking

like a very intuitive fix, but debugging the system to discover this flaw took a significant amount of time because of the race conditions inherent in two distinct processes reading and writing to the same socket. In debugging the socket output, it remained unclear whether or not restricted processes actually sent and received data properly. Eventually, we attached PID data to the guardian messages and realized the issue at hand.

Compounding this issue was the notion that only process duplication routines call `do_fork`. However, `sys_clone` and friends also funnel down to `do_fork`. `sys_clone` is the Linux kernel mechanism used for implementing threads, as well as acting as a lightweight process duplication method. In order to combat the first issue raised above, we simply closed the parent task's connection to the guardian. However, were `sys_clone` the function to have invoked `do_fork`, it could possibly result in the two tasks sharing the same file table, as is the case when it is used to support threads. In this case, the newly created task should not close its parent's connection to the guardian, as doing so would cripple its parent task's ability to open files. In all other cases, the child task should close its copy of the parent's socket file descriptor at its first available opportunity (the first `open` call after `do_fork`). We decided to implement this file descriptor closing mechanism lazily (on the first `open()`) as doing so in `do_fork()` with a large number of spinlocks proved quite difficult.

6 Design Decisions

Although our project did not rely on the use of very sophisticated data structures, we made use of several interesting design decisions in its architecture. Specifically, the userspace to kernelspace communication presented several difficulties in building this project, but ultimately provides the backbone of our security model. Additionally, we rely heavily on Unix domain sockets and several features present in this local socket model.

Most data structures issues are avoided by adding fields to the Linux `task_struct`, in order to support our security model. The fields are added to the `task_struct` to facilitate accessing this data, as it is innately tied to each specific task. Most of the fields are simply used as flags and contain binary values, such as those that indicate whether or not a signal interrupted the system call, or whether or not the task is a `clone` of another. The other fields store file descriptor data for communication with the guardian via sockets and parent process values for these sockets:

```
struct task_struct{
   ...
   //rexec fields
   char is_guardian;
   char is_restricted;
   char is_cloned;
   char interrupted;
   int guardian_fd;
   int parent_guardian_fd;
   pid_t guardian_pid;
   acl_struct_t task_acl;
}
```

In our pre-authorization structures, which we represent as a `task_acl`, we provide a **set** abstract data type, which is backed by a linked list. The linked list is composed with nodes that contain paths and the permissions that a restricted process is allowed for each path.

We built a new system call, `rexec`, in order to provide a hook to connect the guardian to the process it restricts. To provide maximum flexibility, we added this new system call to maintain existing frameworks as opposed to significantly modifying `exec` and derivatives, which might have been impacted with the new codepaths for restricting a process. Additionally, it would have meant modifying a large number of the arguments to `exec` to have different semantics in the restricted case, which probably might have brought new inconsistencies with malformed programs. In this way, only processes that explicitly want to restrict binaries will do so. Further, the restricted binary need not be modified in any way, since the only modification required is a `rexec` instead of an `exec` to start it, and a simple wrapper would suffice.

On a related tangent is the question of why the guardian exists as a user program in the first place. Had the guardian been developed in the kernel, we might have avoided

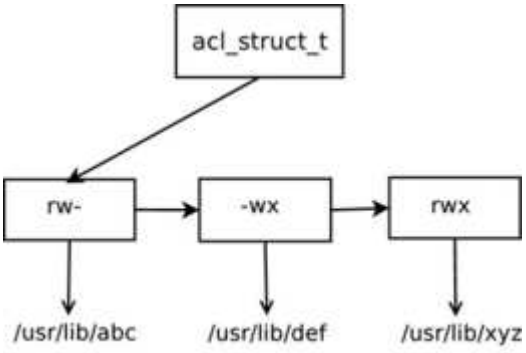


Figure 2: Pre Authorized Files Linked List

race conditions and other bugs altogether. However, the guardian must exist in a user context to provide interchangeability and flexibility to the user. If it existed as a kernel module, we would have to add a large number of checks to ensure that the module were loaded before proceeding, as well as reducing the number of people who could write their own guardians. With a simple API and library, it becomes a much more straightforward task to build an entirely new guardian that interfaces with our library to communicate with the kernel. Additionally, guardians can be interchanged quickly and easily so long as they all communicate with the kernel in the same fashion.

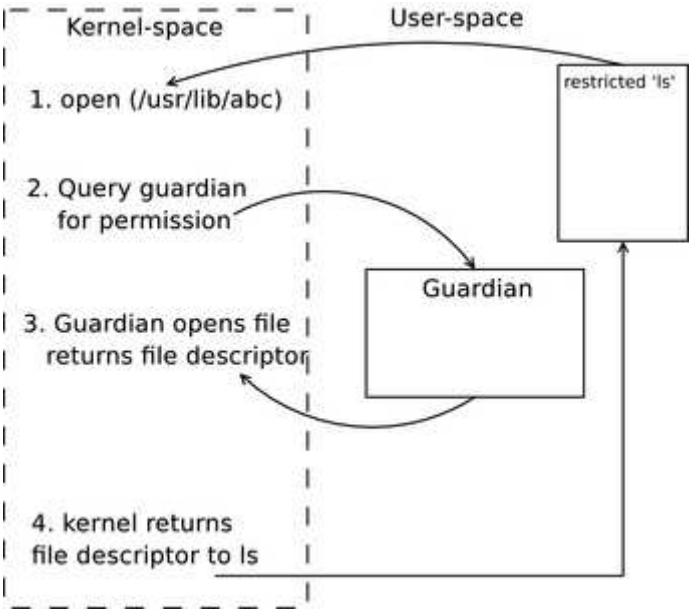


Figure 3: Kernel space vs. user space

Finally, we rely heavily on the Unix domain socket system call `sendmsg` both in

the kernel as well as in a guardian process. This system call enables one to pass file descriptors in a message structure, as well as ancillary data. It allows the guardian, acting on behalf of the user, to open the file and then pass its file descriptor to the guarded process that has no indication that this transfer took place. Essentially, this is exactly what the guardian should be able to do – given authorization from the user for a specific file, analyze the file and ensure the user has access before passing it back to the guardian. Without this functionality, it would have been significantly harder to pass the file descriptor over a socket and insert it into the restricted process' file table.

7 Debugging and Problems

As elucidated previously, the file descriptor issues inherent in `do_fork` presented a significant roadblock to developing a proper prototype of this system. We also encountered several issues stemming from signals that are noticed during a system call. In nearly every other case than our modified system calls, in general, system calls do not depend on other system calls. This is important because if the kernel operation realizes that it needs to suspend an operation in order to deal with pending signals, the system call will be restarted. Were there a dependency chain of system calls, it is difficult to determine whether or not one in the dependency chain had already completed.

This problem manifested itself in our modified `open()` call to support socket communication to the guardian. Essentially, the `open()` call now comprises three distinct operations: the message to the guardian, the reply, and the act of using the file descriptor passed to us from the guardian. However, because this dependency chain of three system calls now exists, it became difficult to discern whether or not the `recv` on the socket was interrupted call while waiting for the reply from the guardian. To combat this, we added an additional field to the `task_struct` (as described previously) which determined whether or not the operation had recently been interrupted to deal with a signal. If this were the case, as soon as the restricted open were restarted, it would not send the request to the guardian, and instead wait for its reply. Had the kernel re-sent the file request to the guardian, another reply would be waiting in the socket for this file descriptor, which would adversely affect the output of the next call to open, which would mistake this data in the socket as its reply.

Further issues with our modifications to the system calls involved the support for signaling guarded processes when the guardian exited. Our initial implementation attempted to emulate the behavior of `sys_kill` but was not immediately successful. Specifically, we attempted to duplicate the code but at a much lower level in the call trace. Initially we worried that we could not signal the processes properly as the current user without failing some kernel permissions checks. However, insightfully, we recognized that because all guarded processes are `exec`'d and run with the UID of the owner, all normal signal system calls should work properly. This is because the owner of a process can always send any desired signal to his processes.

8 Future Work

We have created a working Protection Domain system including a basic guardian for user communication. However, there are many areas the system can be improved. This includes known inefficiencies in our system, but also expanding of the system itself.

Our data structure in the `task_struct` should support much faster than a linear search. Given more time, a hashtable or a hash set would prove to be a better choice, as it would provide amortized O(1) search time to determine whether or not the target element was in the data structure. Given that in our current implementation, the entire list must be searched on every restricted open, it is imperative that a better data structure be used for faster searches.

Additionally, we need a much better way to kill guarded processes than searching through the entire PID space. A better solution would be to add another linked list to the `task_struct`. This list would be used by the guardian process in order to catalog all guarded processes that need to be killed upon exit. If a guarded process exits prior to the guardian exiting, then it should clean up after itself and remove itself from this list. In this case, the guardian must signal all of its guarded processes anyway, so a O(n) operation is entirely appropriate; in either way, it would be much more appropriate than searching the entire PID space for restricted processes.

Another flaw in our current system is that when we tag a specific process as being the guardian, we actually tag the terminal that then forks and exec's the guardian binary, so in essence, it is the parent of the guardian. However, since the aterm and the guardian go hand-in-hand, there is no qualitative difference between the window and the actual process. A better system would allow us to target the actual guardian binary rather than its parent.

There are many ways that the guardian system could be extended to provide interesting control of system resources. For example, as mentioned earlier, a "shadow file system" could be developed from within the guardian. Instead of rejecting questionable system resource access, the guardian could copy the requested file to a new file and return the new file's file descriptor. The restricted process could modify this file all they want, and it would not affect the original file. Thus, it gets its own "shadow" copy of the original file. Many other interesting guardians potentially exist as well, such as an interpreter for a generalized security policy.

Currently, the protection domain system only controls access to the file system. However, there are other resources in an operating system that one might want to restrict. One of the most interesting would be inter-process communication through sockets. Currently, this is left completely unrestricted. It is not unreasonable to imagine wanting to prevent a process from accessing the network, for example. This brings up many interesting questions, such as how to identify the socket connection to the guardian. For TCP connections, do you identify it using the IP address? Is this useful enough to a program/user to make education security decisions about it? Do you separate different types of inter-process communication, or treat them all as equals?

9 Conclusion

Tradiational operating systems provide a solid security model for protecting system resources. However, as systems become more and more complex, there is a need for more fine grained security policies. Access control is not just about controling user access; there is a very important need to be able to control the access of specific programs.

We have presented a model for addressing this issue. The protection domain solution presents a system for modern operating systems that adds a new semantic notion of individual, process-specific access control, allowing and denying programs access to system resources, not just users. We also show this model to be extensible in a number of valuable ways, such as the ability to create arbitrary security policies and to create a shadow file system. When properly used, protection domains should allow a user to comfortably and safely execute arbitrary binaries without having to worry about how they will affect the system.

Furthermore, we built a version of Linux that incorporates protection domains into the kernel so that we actually have a functioning, working guardian system. This system is shown to be useful, especially with the addition of file system support for access control lists.

Protection domains provide an important new semantic notion of process-specific access control that has not been previously seen in operating systems, or has previously been too complex as to be effectively used. We believe that protection domains will prove to be a useful and helpful addition to access control and security in modern operating systems.

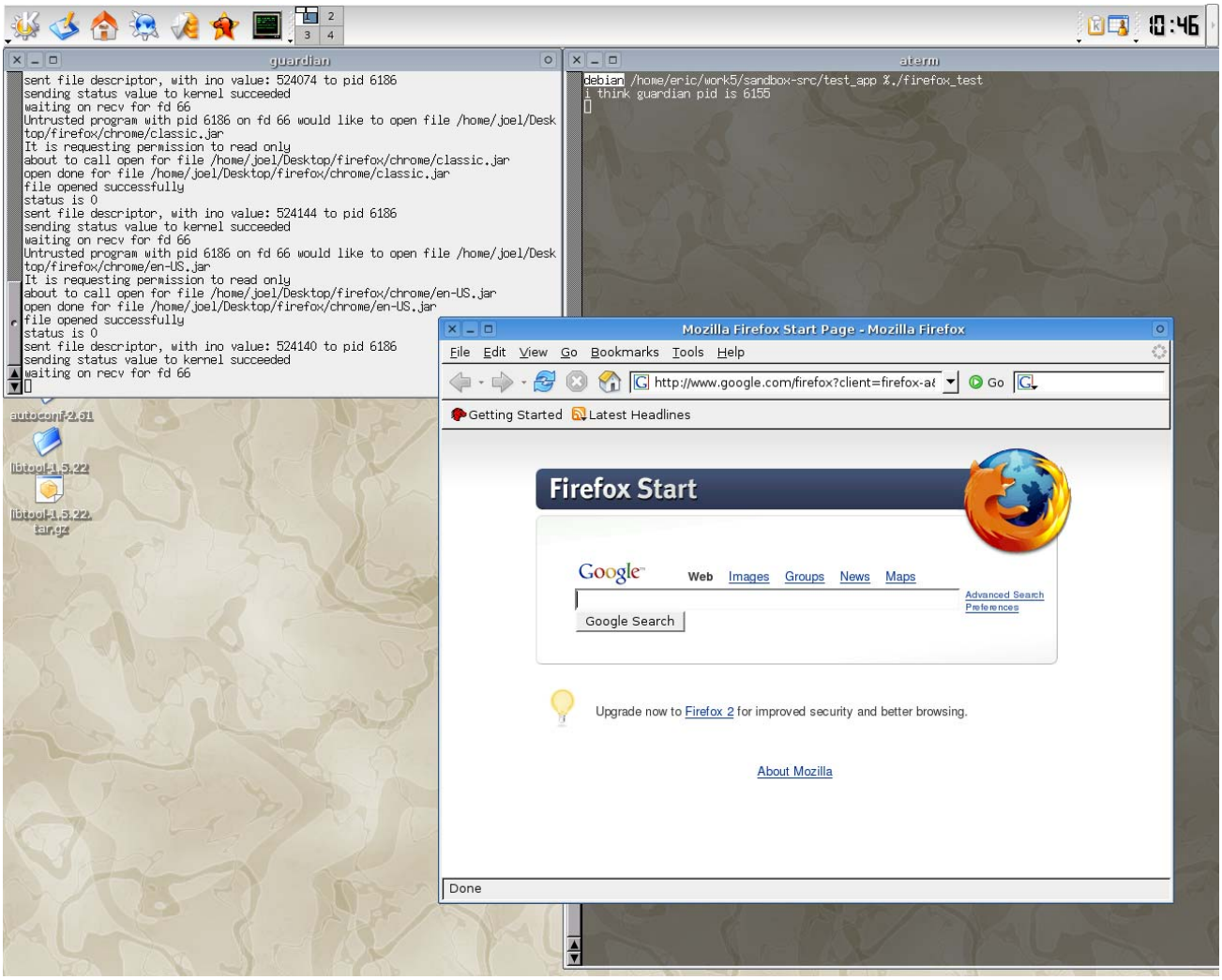
References

[1] B. Lampson. "Protection and Access Control in Operating Systems". In *Operating Systems, Infotech State of the Art Report 14*, pages 309–326. Infotech, 1972.

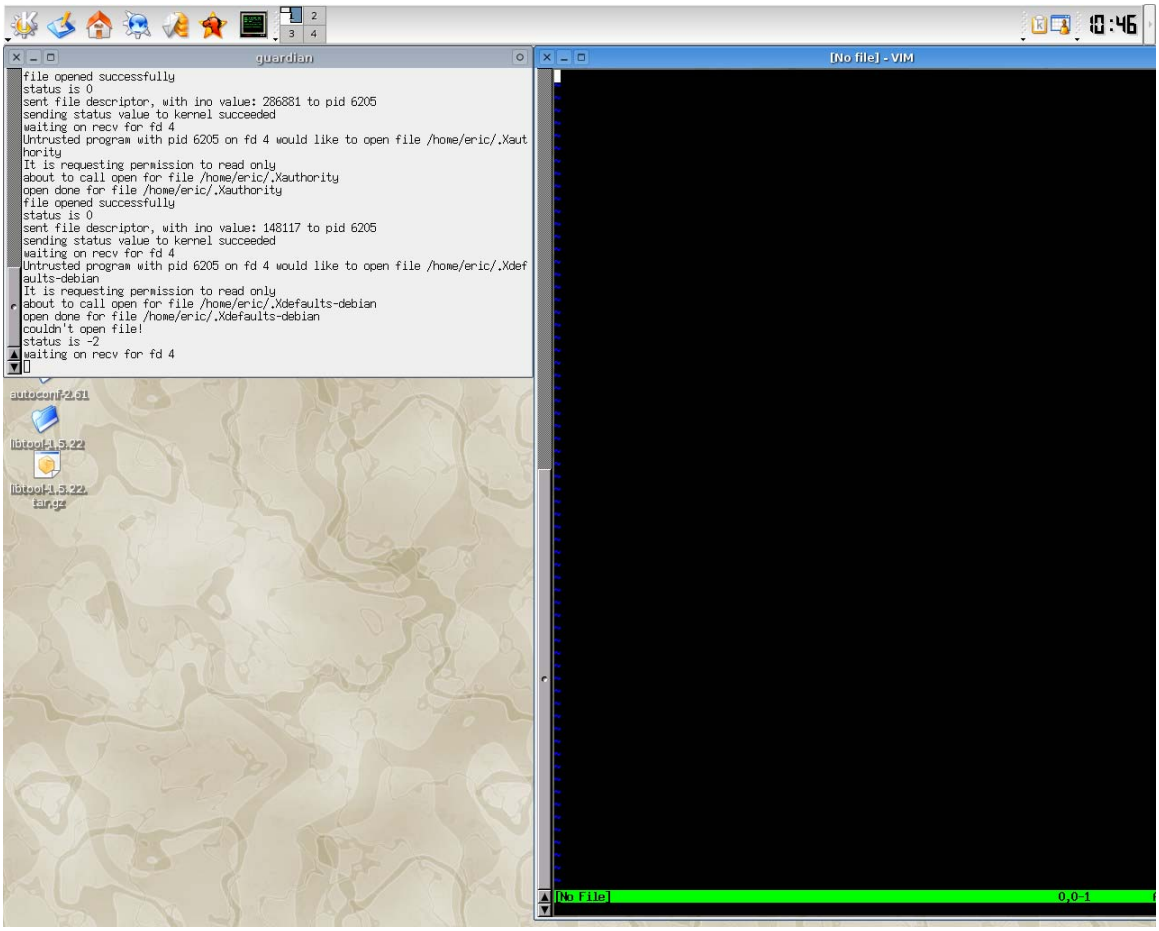
[2] Luke Peng. "The Sandbox: Improving File Access Security in the Internet Age". 2006.

[3] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

APPENDIX



Firefox run under a Guardian.



VIM editor run under a Guardian