Joel Weinberger

Protection Domain Masters Project

The Protection Domains project has had many different areas to work on throughout the project. At some point or another, every member of the group touched almost every part of the project. These areas include initial development of the Protection Domain model, development of the guardian model, initial proof-of-concept, file system development, and debugging of major issues. This will focus on areas of my greatest contribution.

The first area was the development of the guardian. Initially, the work was on designing a system that would prove flexible and powerful enough to be a useful addition to the access control model, yet simple enough for average users to understand and be able to use. The development of the model went through several major design shifts. Early on the idea was to add a built in guardian that simply provided the user with an interface with which to monitor the resource accesses of a restricted process.

We realized early on, however, that this would not provide a flexible enough model. We modified the design to reflect a broader, more generalized understanding of what the guardian semantics really should be. This initially started as a Guardian hierarchy, but we realized that this would be rather complicated from the user's perspective. Furthermore, through a simple proof-of-concept that we developed of the guardian, we showed that this was also a difficult model to implement.

We began working on a user land to kernel interface for the guardian. This utilized Unix Domain Sockets to provide communication between the kernel and the guardian, including the passing of file descriptors. We decided to pass file descriptors so that a guardian must be able to access a file to which it is granting access. Thus, a malicious restricted process cannot rely on a malicious guardian to grant it unfettered access to the system.

We have shown this to have further advantages through the idea of a “shadow file system” that allows the guardian to pass alternative files to a restricted process, instead of the files it requests. Conceptually, if a restricted process wants to modify a file, and the guardian is unsure if the restricted process is malicious, instead of returning the file descriptor of the requested file, it returns a file descriptor to a “shadow file,” which is a copy of the original file in a new, temporary location. The restricted process cannot tell the difference between that file and the requested file (unless he passes the path onto another program) and thus modifies it as if he were modifying the real file. The guardian can then at some later point check to see if these modifications should be merged back into the original file, or can feel free to disregard any changes made.

This is further enhanced by the development of our guardian API. While we have built a useable, useful guardian, it is easy to imagine someone wanting to develop their own guardian, such as in the “shadow file system,” or a guardian that accepts regular expressions as access control lists. Thus, we have built an API library to greatly simplfy the process of building a guardian. While the API is not *required* (any program can interface directly with the system call level interface with the kernel), it brings the actual communication down to only a few relatively simple functions that allow full guardian to kernel communication.

We also went through several stages of guardian development. This included both multi-threaded and event-based implementations, the latter of which became the basis for our final guardian model. The multi-process communication of the guardian with the various restricted processes it might be dealing with proved to be vary troublesome, and provided many hours of debugging. While I certainly did take part in this work, especially early on, it did not remain one of my major focal points.

We also spent a great deal of time working on file system support for access control lists. Conceptually, it is not desirable to constantly have to make access control decisions by referring to the guardian. It places undue stress on the guardian and the kernel/guardian interface, especially when you know that certain programs will access a set of files regularly and without negative impact.

One of the original ideas was to develop an access control system that utilized extended attributes in the ext3 file system. Extended attributes are meta-information that are associated with every inode in ext3 that allows you to store arbitrary information in a key-value pair system. This seemed to be almost exactly what we desired, since we wanted information in the form “path : access control decision.”

However, we quickly learned that extended attributes were not exactly what we wanted. Besides locking our system into one particular file system and needing to break the file system/virtual file system abstraction in the process, extended attributes have several important limitations. Extended attributes are limited to one block, which would limit us to 4 kilobytes of meta-information. That would leave us with two possibilities: too small access control lists or referencing other files. The first would greatly damage our usability and the second would violate almost the entire point of using extended attributes.

Thus we remodeled the file system aspect to simply use regular files owned by root and readable only by root in order to build access control lists. While this would potentially leave us with name space conflicts, we do not anticipate many programs to be conflicting with our name space choice. Furthermore, we do not have the file-size limitation of extended attributes.

This remains a viable system because a program would not be able to access an access control file if the guardian does not allow it. Even if the guardian does allow it, it would need root permissions to do anything to it. Thus it remains unaccesable by guarded processes.

We also spent a great deal of time developing the method by fork and related system calls would pass the restricted property. Because of the nature of Linux process forking and the relationship between processes and threads, this was an arduous process. For example, restricted processes carry a great deal of metadata concerning pre-approved files. Thus, when copying task structure information for a thread versus a process, you have to be careful how much you copy versus what you simply re-point to. There were many issues concerning forking that took a great deal of time.

Additionally, I worked on several other important aspects of the project aside from these main focal points. As mentioned earlier, I was a part of the solution to some of the major problems encountered in the guardian/kernel interface. There were also issues with actually setting up our own system calls. Our Protection Domain project provides a new way of viewing security in a traditional operating system environment. I was instrumental in developing several key aspects of the system, including the guardian model, the file system support, and the process forking mechanisms.

For more information about this work, please see the paper
"Operating System Protection Domains"
by Eric Tamura, Joel Weinberger, and Aaron Myers