Protection Domains

Eric Tamura
Computer Science Department
Brown University
etamura@cs.brown.edu

Contributions to the Project

When I joined the project in the beginning of the semester last fall, Joel and Aaron had already developed most of the ideas for the guardian model, as well as building a prototype guardian following that initial design. I then researched various alternative systems as well as familiarized myself with the code base for the guardian and kernel changes that had already been made.

The first technology that I researched was SELinux access control, and my job was to validate our belief that their goals were not in line with our own – namely, that it could not support a subject-based discretionary security model. This turned out, thankfully, to be the case. SELinux is built upon a mandatory access control policy, which means that users and owners do not have full and complete access to files that they create. Instead, access is governed by the access control levels specified in the security policy for the entire system. SELinux has several attractive features, particularly a semblance of role-based access control, as well as the idea of a user-configured security policy. It also uses the access control list support built into the Linux kernel. However, the inherent difficulty in producing a dynamic security policy validated the initial fears that the system would not be sufficient for our needs. Specifically, new policies must be recompiled and reloaded after every change, and although it could be modified enough to support our discretionary target, it would not be easy. The policy language is inscrutable enough to warrant a plethora of tools designed to automate the building of policies. Plus, the time it would take to build our own custom policy made it less attractive.

After deciding to reject SELinux, I researched the kernel mechanisms behind SELinux, Linux security modules. These presented an intriguing idea: that we could specify a kernel security module to be loaded in after boot-time that would govern whether or not files, sockets, or nearly any other I/O based system could be used. Within the kernel, there are several security checkpoints, where access is obtained from the security module subsystem before continuing. If there is no security module function to handle this request, then it simply proceeds. Internal discussions on Linux forums indicated that Linus was considering removing them from upcoming kernel distributions, based on the lack of support. To be sure, the web server containing the authoritative documentation on these modules was frequently inaccessible.

Despite ongoing debates, the system itself is usable and might have been suitable for our purposes. The security modules are stackable, meaning that an entire set of security checks for files, directories, and virtual memory can be placed on top of another. The system designates a primary security module, and stacked modules are secondary. If the primary module decides it does not want to handle a specific operation, it can relay

the request to the stacked module, which will handle it. Were we to use this approach, our security module would act as primary, and we would stack the existing capabilities module as the secondary – it would be important to do so, as the commoncap.c file containing these checks is used in the virtual memory system. We ultimately decided against using this approach because the appropriate security checkpoints occurred far too late in the call trace of the `open` system call. In order to make it usable, it would have to be modified significantly enough where using a modified `open` system call would require just as much work. As a result, we chose the latter.

I also investigated the use of extended attributes on the EXT3 filesystem for Linux. In our early forays into the guardian security model, extended attributes were the eventual target for the set of pre-authorized files for a restricted binary. However we came across several roadblocks. Specifically, there were several ambigious statements in the documentation for extended attribute support regarding the size of extended attributes. It was unclear whether only 4 kilobytes of disk space were allocated for the entire filesystem, or for each extended attribute (it was the latter). The system required several kernel patches and a recompile to install properly. The system was slightly difficult to interpret, and had several quirks, such as requiring all extended attributes to begin with "user." or else rejecting the access request. After getting our guardian/kernel modifications to work properly, we decided against using extended attributes because of the size limitations.

For most of the second semester, I tracked down the race conditions involved with the socket-duplication bug that plagued our guardian/kernel communications for the entire year. To recap, the bug basically refers to the problem where, when we connected to the guardian for the first time as a restricted program, we would store the file descriptor in the task struct, and it remained the same across `do_fork` calls. As a result, we would have several processes communicating over the same socket on the kernel side. Aaron initially began debugging the system, whose only symptom initially was that the restricted programs worked "some of the time." The errors began popping up during our attempts to get the firefox browser to work properly in a restricted context. It failed to appear during our testing with VIM because even though the editor appears to call `do_fork`, it does not perform much file I/O in its duplicated process.

However, Firefox runs a script which performs much of its configuration and internationalization settings, before launching the actual Firefox binary. As a result, there are several independent processes reading and writing from the same socket, which caused much variability in how far firefox goes in its startup sequence. Once we determined the cause of the bug, we remedied it by ensuring all new restricted tasks made their own connections to the guardian as well as removing all subsequent `sys_close` calls from the restricted process codepath. It is important not to close all of the file descriptors, because some of the tasks could have been spawned from calls to `sys_close` as opposed to `sys_fork`.

On a similar line, the preceding fixes allowed Firefox to start properly, but only intermittently. I then tracked down the remaining bugs in the `do_restricted_open` codepath for our restricted exec. Specifically, we were not dealing with restarted signals

properly. If the kernel notices a pending signal, it should return the internal errno value `-ERESTARTSYS`, which notifies the trap handler to restart the system call so that the caller does not notice an error. This would cause problems for the `do_restricted_open` codepath, as making duplicate send calls would cause the guardian to respond twice to the same request. To resolve this, I added an internal flag to the task struct that would allow us to determine whether or not the system call was interrupted. Another internal flag was added to determine whether or not a given process is a guardian, so that during `proc_exit` it can be treated specially and send signals to the remaining restricted processes so that they will know to exit as soon as possible.

Finally, I added in the remaining code to pre-authorize certain files for a restricted process before it makes the outbound request to the guardian. In `rexec`, the kernel must read the set of files from the filesystem before adding them to the in-memory data structure for the task struct. This code was relatively straightforward after discovering most of the pitfalls in doing I/O from within the kernel previously. During `sys_open` the kernel will query the set of pre-authorized files to determine whether or not the value is in the set. If it is, then it bypasses the guardian checks.

I spent much of the first semester researching alternative solutions to modifying the Linux kernel, and how feasible those solutions were. Most of the alternative ideas had good points, but were not great at allowing us to build the subject-based discretionary model that this project sought. During most of the second semester, I spent a significant amount of time simply debugging what turned out to be a fairly difficult race condition involving the file descriptors, mostly because it was unclear what part of the system was failing. At times, it appeared as though the Unix domain socket code was suspect, while at others, we were not sure that the kernel was passing the right requests to the guardian. In the end, it was plain as could be once Aaron and I saw the PID information being passed across the socket. I still find it slightly disconcerting that such a simple bug tripped us up for so long, but it definitely proved educational.