The Sandbox:
Improving File Access Security in the Internet Age

Luke Peng
Department of Computer Science
Brown University
Providence, RI 02912

May 14, 2006

1 Introduction

The longstanding working model for system security on Unix machines has consisted mainly in restricting access to files based on user-specific rights and permissions. The widespread success and use of the Internet, however, has afforded malicious parties the opportunity to expose a number of holes in this traditional security model. One such defect allows potentially dangerous applications downloaded or run from the Internet to operate with the full rights and permissions of the user executing the application, even if that application should only need access to a small subset of the many files accessible to the user.

A better security model would limit the files a process (or more generally, a subject) could access to only those that it truly needs. We call this subject-specific list of file access rights a Subject ACL (Access Control List). In effect the process is run within a virtual *sandbox*, with its file access permissions strictly defined by the Subject ACL. Note that this modification to the traditional security model is only an improvement, not a substitute: a process should only be granted access to a file if it has *both* the appropriate user-specific privileges as well as the appropriate subject-specific privileges.

This document provides a technical specification of how these Subject-ACLs are implemented and enforced with a Linux 2.6 kernel.

---

2 System Overview

Ultimately, this software system grants users the ability to safely execute a program without having to worry about whether the program will access any files beyond those specified in the associated Subject-ACL. This Subject-ACL will form a *sandbox* around processes, setting explicit boundaries on which files they are allowed to access.

The user is able to inspect the Subject-ACL associated with the executable s/he desires to run and add or remove items from the Subject-ACL as necessary. Moreover, since it can often be difficult to predict exactly which files a program will need before it is executed, the software system also provides a means for dynamically adding access rights to the Subject-ACL while a program is running.

The method by which a user executes a target process within a sandbox is the `rexec()` system call (short for "restricted exec"). This system call starts up a user-specified *guardian* process along with the target process, and sets up a communication channel between the guardian and target.

Subsequently, any time the target process attempts to access a file via a system call such as `open()`, the system call will, in addition to its regular access checks, also check the Subject-ACL associated with that process. If the requisite file permissions are not found in the Subject-ACL, the system call contacts the guardian, which in turn decides whether or not to augment the protection domain (usually by prompting the user to make a decision). If the guardian decides not to allow access, the guardian notifies the waiting system call, which just returns an error value indicating that the access check failed. If the guardian does decide to allow access, it will open the file and obtain a file descriptor on behalf of the restricted process and pass this opened file descriptor back to the system call, which can just return it.

There are two possible ways to specify the Subject-ACL that is to be associated with a program. The first option requires putting a copy of the Subject-ACL in the inode of the executable file to be run. When `rexec()` is called, the Subject-ACL will just be copied into the resulting target's task (Linux term for a process or thread) struct. The second option simply involves passing the Subject-ACL directly to `rexec()` as an argument. Obviously, some combination of the two methods could be used as well.

Importantly, any tasks (processes or threads) subsequently spawned by the target will inherit the same Subject-ACL and the same flag indicating that the task is running within a sandbox. Thus, there is no way for a task already running within a sandbox to augment its own protection domain by spawning other tasks. Only the appropriate guardian process has the ability to add privileges to a Subject-ACL.

The guardian process itself should be implemented at the user level by the application developer. Thus, there is a good deal of flexibility in how this process communicates with the user, whether it is via the command line or some sort of advanced GUI. The system, however, does ensure that the guardian can only add to the target's protection

domain those file permissions that it already has. Further, only the guardian process specified by the `rexec()` call has the ability to inspect the Subject-ACL of the target task and dynamically make changes to it on the fly.

---

3 Design Considerations

3A. Assumptions and Dependencies

The software system is designed specifically for Linux, and as such, involves making modifications specifically to the Linux (2.6) kernel. Although the basic premise of this software system is easily adapted to other flavors of Unix and other Operating Systems, this document focuses on the specific development of the system on Linux.

The first option (described above) for specifying the Subject-ACL of a program requires storing a copy of the Subject-ACL in the executable file's inode. The system makes use of Linux facilities for associating meta-data with files (known as Extended Attributes) to accomplish this. Extended Attributes are a relatively new feature of Linux (2.6 kernels) and depend on the underlying filesystem-specific implementation. Under the ext3 filesystem (with which this software is being developed), all Extended Attributes must fit on a single disk block (1, 2, or 4 Kilobytes). This space constraint could be a potential problem for very long Subject-ACLs, depending on how Extended Attributes are implemented in the future.

The Debian `attr` package for extended attributes is currently being used to set and modify the Subject-ACL associated with an executable's inode, since at this point the Subject-ACL is merely specified as an Extended Attribute of the executable.

In order to ease kernel development, the system has been tested and debugged using a Linux 2.6 kernel running on VMWare. Vim and Cscope are the major tools employed for software development.

3B. Open Issues

It is currently an open issue of whether to allow the Subject-ACL to specify directories under which all files and directories are granted the same access permissions.

The exact format of the Subject-ACL is still to be determined. Currently the system uses a string of ASCII text of the form "**`:6:/usr/fileOne:5:/etc/local/fileTwo:6:/tmp/fileThree:`**" Note that the access permissions for a file are indicated by the integer number (used in the same way as **`chmod`**) that precedes it in the Subject-ACL, and that items in the Subject-ACL are delimited by colons. Pathnames are used rather than inode numbers for the sake of usability and persistence. One major problem with this scheme is that colons are allowed to be part of filenames in Linux. A better standard format for the Subject-ACL

would be highly desirable.

---

4 Implementation Details

4A. Associating a Subject-ACL with a Task

I. *Specifying Subject-ACL as an Extended Attribute of an executable.*

The **getfattr** and **setfattr** commands of the Debian **attr** package are currently being used to store a Subject-ACL as an Extended Attribute of an executable file. An Extended Attribute (EA) consists of a name for the attribute and a corresponding value. The current name for the Subject-ACL EA is "user.perms". The value is a string of text in the format described above (in 3B) representing the actual Subject-ACL.

EAs utilize a namespace system, whereby attributes named under certain namespaces can be modified and accessed by any user (such as the "user" namespace), while other namespaces may only be accessible to the kernel (such as "system"). It is desirable to move the Subject-ACL EA into a more restrictive namespace once the mechanism for securely modifying and accessing the Subject-ACL EA is more fully developed.

Once the Subject-ACL is associated with an executable as an Extended Attribute, it is then relatively straightforward to copy the Subject-ACL into the corresponding task struct when the executable is run with `rexec()`. A new field has been added to the task struct object: "`char* subject_acl`". The code for `rexec()`first allocates enough space for a buffer to hold the Subject-ACL at the `subject_acl` pointer, and then copies over the Subject-ACL stored as an Extended Attribute into the allocated buffer. One consideration is whether it may be more efficient to allocate some extra space to allow for the buffer stored in the task struct at `subject_acl` to grow rather than having to allocate a new larger buffer any time the Subject-ACL needs to be longer.

The `do_exit()` code has also been modified so that the memory for the Subject-ACL buffer pointed to in the task struct is freed when the task exits.

II. *Specifying Subject-ACL by passing it as an argument to the* `rexec()` *system call.*

The other method for associating a Subject_ACL with a running task is to simply pass the Subject-ACL directly into the call to `rexec()` as an argument.

Again, `rexec()`copies the Subject-ACL into a buffer referenced by the `subject_acl` pointer in the task struct, just as in the case when the Subject-ACL is an Extended Attribute of the executable file. Obviously, if the Subject-ACL is specified both by an argument passed to `rexec()` as well as an Extended Attribute, the two Subject-ACL's

should be combined, and enough space should be allocated for the `subject_acl` buffer to hold the combination of the two lists.

One important security consideration in this implementation is that a task T is not allowed to call `rexec()`and pass in a Subject-ACL containing file permissions that T does not itself have in its own Subject-ACL. This prevents malicious programs from using `rexec()` to augment their own privileges or the privileges of other tasks.

4B. More on the `rexec()` system call

The only means by which a target task can be run in a restricted sandbox environment is via the `rexec()` system call. Just as many standard system calls like `sys_open()` are actually invoked by users primarily via wrapper functions provided by standard libraries, users should not call `rexec()` directly and should instead use the wrapper function `user_rexec()`. Callers of `user_rexec()` must, in addition to the arguments normally passed into regular `exec()`and the argument specifying the Subject-ACL, also pass in the arguments needed to execute a guardian process.

The first action that `user_rexec()` takes is to `fork()`, with the parent task executing the *target* task via a call to the actual `rexec()` system call, and the child task executing the *guardian* process.

In the *target* task where `rexec()` is invoked, before the task begins its normal execution, a number of steps are performed by the kernel. As a first step, all open file descriptors are closed in case any previously opened files are not specified by this target task's Subject-ACL.

Next, a Unix Domain Socket connection is established with the *guardian* task. This requires waiting until the guardian has set up its end of the socket connection. The current method used for waiting is to wait in a loop until the connection is successfully established, yielding the processor each time an attempt is unsuccessful. One potential issue with this approach is that waiting in the kernel in this manner makes the task unresponsive to signals, so there is no easy way to end the task if the guardian never establishes its end of the connection. Improvement on this approach is left to future work.

The *target's* task struct is then modified as follows: the Subject-ACL is copied into a buffer pointed to by `subject_acl`, the `is_restricted` flag is set to indicate that the *target* is being run within a sandbox, and the file descriptor for the socket connection with the *guardian* is copied into the `guardian_fd` field.

Finally, the appropriate *target* executable is `exec()`'d and begins normal (but restricted) execution.

The *guardian* process, implemented in user land, does the work of setting up its own end of the Unix Domain Socket connection and listens for the *target*'s attempts to connect to

it. After the connection has been established and as the *target* begins execution, the *guardian* starts listening for messages from the *target* regarding protection domain violations.

4C. Enforcement of Subject-ACL access policies

Since any user-level task that desires to use a given file must first make one of a limited number of system calls (such as `open()` or `socket()`) before it can obtain a file descriptor to work with that file, all such system calls are modified to check for Subject-ACL privileges in addition to the usual file-based privileges. For explanatory purposes, we will focus in this section on the `open()` system call.

Whenever a task with the `is_restricted` flag set calls `open()`, the kernel code first checks whether the target file is listed in the current task's Subject-ACL. If it finds a match, and the corresponding permissions listed in the Subject-ACL for that file are sufficient, then `open()` proceeds as normal (including the regular user-specific access checks). Otherwise, the kernel sends a message over the Unix Domain Socket (whose file descriptor is specified in the current task's task_struct) to the *guardian* task indicating that a file access violation has occurred. The kernel code then listens on the same Unix Domain Socket for the guardian's response.

Although the *guardian* process is specified and written by the application developer for user land to allow for more flexibility, it will generally follow a few essential steps when it receives a file access violation message:

Upon receiving the message over the Unix Domain Socket, the *guardian* prompts the user, asking whether the restricted task should be allowed to access the given file. The three possible responses are to (a) simply deny access, (b) grant one-time access to the file, or (c) persistently augment the task's Subject-ACL to allow future access to the file as well. If the user grants access, the guardian just opens the file on behalf of the restricted task. Note that the guardian is in this way limited to opening only files for which it already has the appropriate permissions. The guardian then just sends back a response message over the same Unix Domain Socket containing the opened file descriptor and indicating whether the file should be added to the *target's* Subject-ACL as an additional privilege.

The kernel code executing on behalf of the restricted task receives the guardian's response, and accordingly either adds the file to the Subject-ACL (if the message indicates that it should do so) and returns the already-opened file descriptor, or returns an access error code indicating insufficient permissions.

Note that some special cases may need to be handled differently. The file /dev/tty, for example is a special file that refers to a process-specific controlling terminal, if one exists. A problem arises if the *guardian* is executing in a different terminal, in which

case it will open up /dev/tty for its own terminal rather than the *target*'s terminal. The workaround for this case is to translate the /dev/tty path into a path to the more specific filename for the target process' terminal (such as "/dev/pts/4") and to pass this more specific path to the *guardian* for opening. Any other process-specific special files must also be handled in a unique manner.

4D. Handling the children of a restricted task

When a new task is spawned by a restricted task via a `clone()` or `fork()` system call, the child task can at best inherit all the file permissions specified by the Subject-ACL of the parent. It is important for the child task to be prohibited from "inheriting" any additional file access rights that its parent did not already have.

An important open question in regards to `fork()` and `clone()` is whether or not the new child task should be monitored by the same guardian as the original parent task.

If the guardian manages child (and grandchild) tasks as well, things might start to get complicated, as the guardian will have to keep track of all of the different tasks it is guarding. Since the guardian provides the user a means to dynamically modify a target task's Subject-ACL, these modifications would either have to be sent to all protected tasks, or the user would somehow have to identify the specific task(s) that should receive those privileges. There may also be a difficulty in the case of protection domain faults as it will be hard for the guardian to accurately and informatively indicate to the user which task is requesting additional privileges.

One solution to all this complexity could be to have a single authoritative Subject-ACL stored at the guardian for all the tasks that it guards. This simplifies the adding/removing of privileges and makes things clearer for the user. In this case, the Subject-ACL stored in the task_struct gets passed to the guardian one time at the beginning of the initial `rexec()` call, after which the Subject-ACL in the task_struct becomes obsolete as the guardian holds the only "official" copy. Checking of the Subject-ACL is then done in the guardian rather than in the system call kernel code of the target process. The simplicity of this method must be weighed against a loss of flexibility inherent in having only one Subject-ACL for many different tasks.

In either case, if the `fork()` or `clone()` call creates a brand new file descriptor table for the child task, the `guardian_fd` stored in the task_struct will no longer be valid, and so either this fd must be somehow migrated to the new table or a new connection must be established (further adding to complexity).

On the other hand, if we don't require the guardian to also manage the protection domain of the target's child tasks, we would instead have to create a new guardian to manage each new child task. This would unfortunately add a great deal of undesirable overhead to spawning a new task.

4E. Dynamic Runtime access to and modification of Task Subject-ACLs

The capability for dynamically changing and viewing the Subject-ACL of a running task, even when there has been no access permission error, is implemented in the form of a set of system calls. These system calls are specified to work only when invoked by the actual *guardian* process of the target task.

As has been mentioned earlier in this document, particular care and attention must be paid to ensure that only the actual *guardian* for the target task can successfully make these system calls. This restriction is fairly straightforward to implement, as the listing of all tasks guarded by the current *guardian* process is easily accessible from within the kernel.

5 Past, Present, and Future Work

Our initial implementation of the system was done completely in user land (no work was done in the Linux kernel). A C function named `open()`was written to communicate with the *guardian* process over a Unix Domain Socket. This function was compiled into a shared library such that the LD_PRELOAD environment variable could be set to point at this library. Thus any program run with LD_PRELOAD set appropriately would have its `open()` system calls intercepted by our library code.

This fairly rudimentary implementation did not deal specifically with checking and modifying the Subject-ACL, but rather assumed that the Subject-ACL was empty. Thus, all file accesses would result in protection-domain-fault messages being sent to the *guardian*.

The *guardian* process we created provided a simple command-line interface for the user in a separate terminal window. Whenever the *guardian* received a message over its Unix Domain Socket from our `open()` library code indicating that a program was attempting to open a file, it would prompt the user for input. If the user permitted the access, the *guardian* would open the file on behalf of the protected process and send the opened file descriptor back over the socket to the library code. Using the built-in features of Unix Domain Sockets, the protected process could receive the file descriptor and use it just as if it had opened the file independently. On the other hand, if the user denied the access, the guardian would send an error message over the Unix Domain Socket to the protected process, and the `open()` library code would return an access error value.

After completing this user-level implementation to demonstrate that our concept would actually work, we were able to move forward and translate what we had done into a more faithful implementation of our system.

At present, we have completed a basic working version of our system with different parts implemented either in the Linux kernel or in user land:

The function `user_rexec()`, a user-level wrapper for our `rexec()` system call, performs the work of both calling `rexec()` and starting a separate *guardian* process. The `rexec()` system call traps into the kernel, where it performs some initial setup steps (such as connecting to the *guardian* via a Unix Domain Socket and setting the `is_restricted` flag) before it executes the protected process.

Just as in the all-user-level implementation, an empty Subject-ACL is assumed by default. Thus, the kernel code handling the `open()` system call ensures that all attempts by a restricted process to access files require the permission of the *guardian* process.

The current system uses the same user-level *guardian* process implementation as in the initial system. Future work could potentially improve on this basic *guardian* with a more sophisticated GUI-based *guardian* sporting more advanced (and prettier) features.

The system has been tested extensively and shown to work quite well for various common Linux programs such as *echo*, *cat*, *more*, *less*, and *vim*. All of these programs have been sandboxed successfully without complication. Testing has consisted mainly in running these programs with `user_rexec()` and verifying that the *guardian* supervises all attempted file accesses on the part of the protected processes. A walkthrough of a sandbox test of *vim* (with screenshots) is contained in the *Appendix*.

Ours is a fairly basic implementation of the system described in this paper, but now that the fundamental framework is in place and has been shown to work correctly, it should be fairly straightforward to add more advanced features in the near future. The many possibilities for future work include improving on the handling of error conditions, adding support for the managing and checking of Subject-ACLs, and making design decisions on dealing with the child tasks of protected processes.

Appendix: Sandboxing vim:

The following screenshots demonstrate the use of our system to sandbox the vim program.

Fig. 1: The program "**test**" being executed is a simple C program that just does a `user_rexec()` of **/usr/bin/vim**. In this case, vim is trying to open the *Makefile* that is in the current directory. As you can see, the *guardian* is automatically started up in another window.

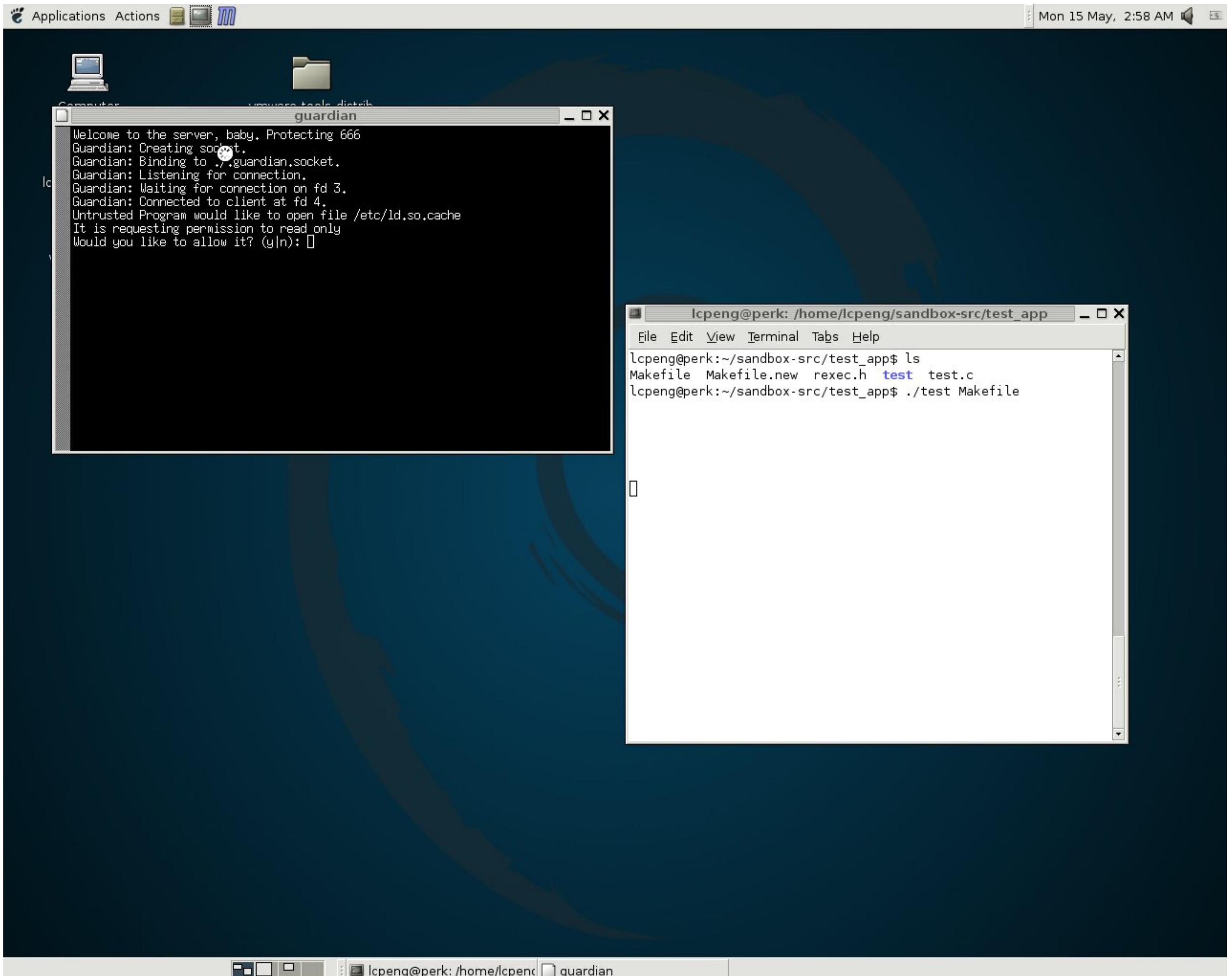

Fig. 2: As vim starts up, it needs to open up a number of standard libraries, configuration files, and of course the *Makefile* itself. Each time vim attempts to open a file for reading, writing, or executing, the *guardian* prompts the user for permission.

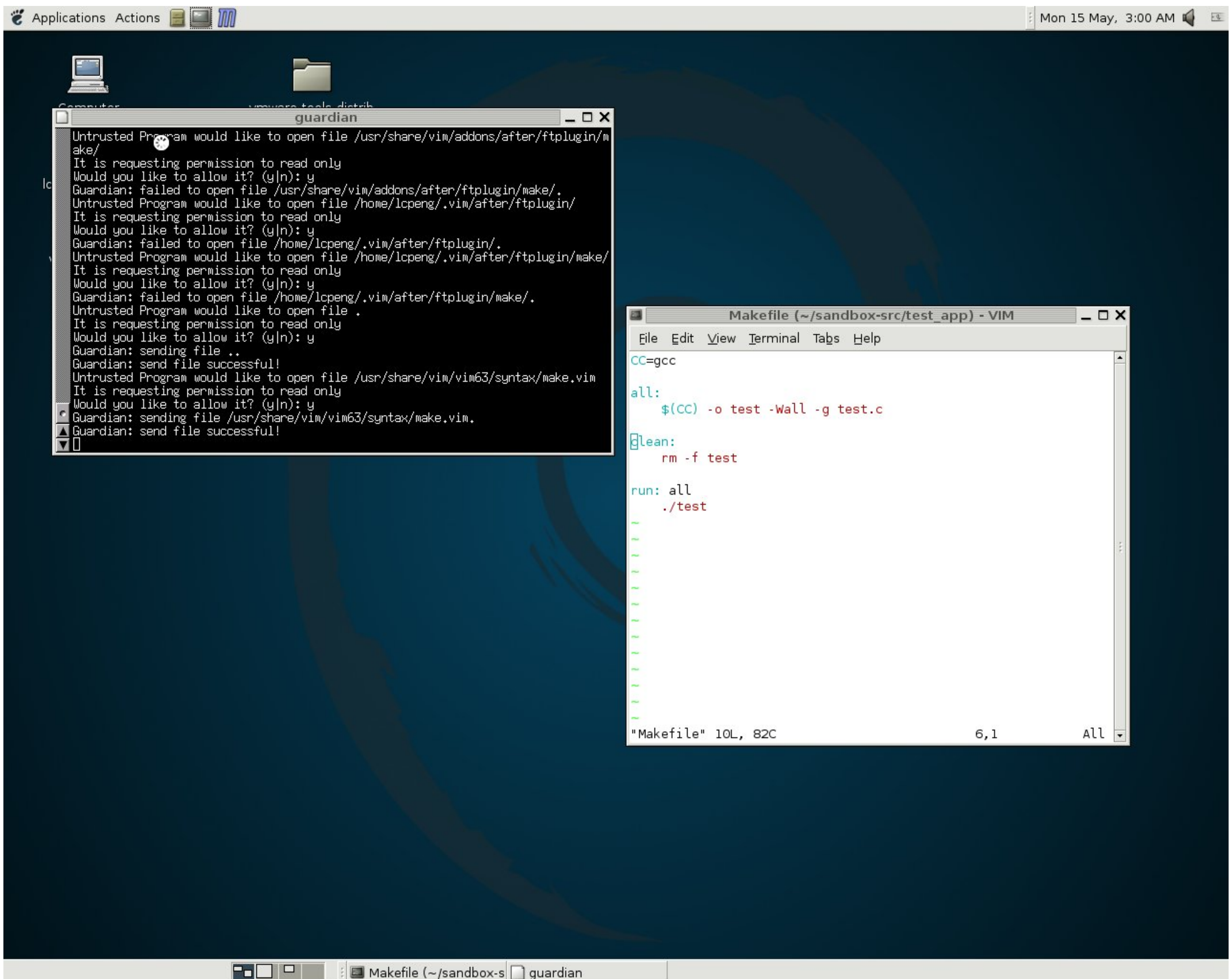

Fig. 3: Now the user makes some changes to the *Makefile* with vim and is ready to save his changes to the file *Makefile*.commented.

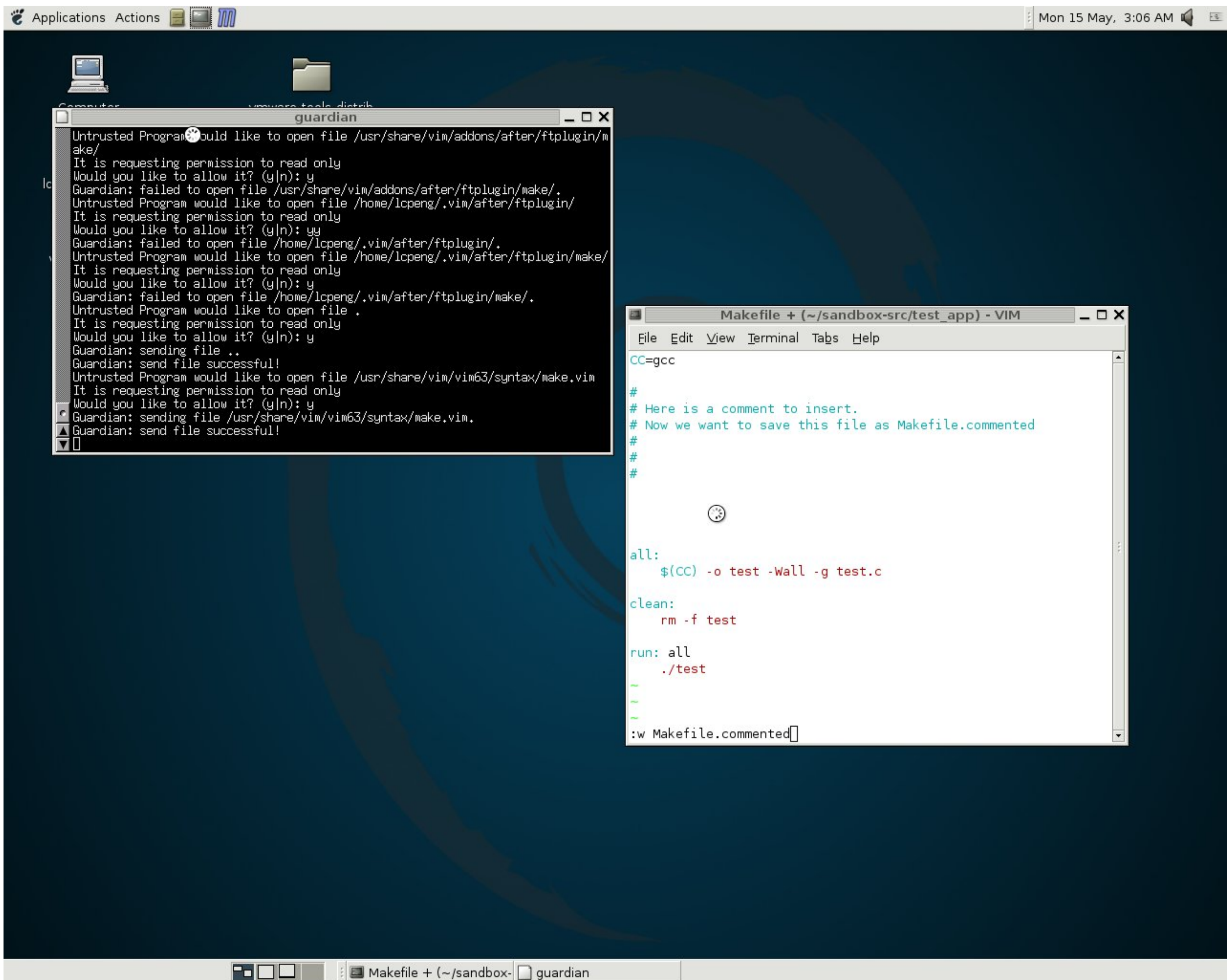

Fig. 4: The *guardian* prompts the user, who allows the write operation.

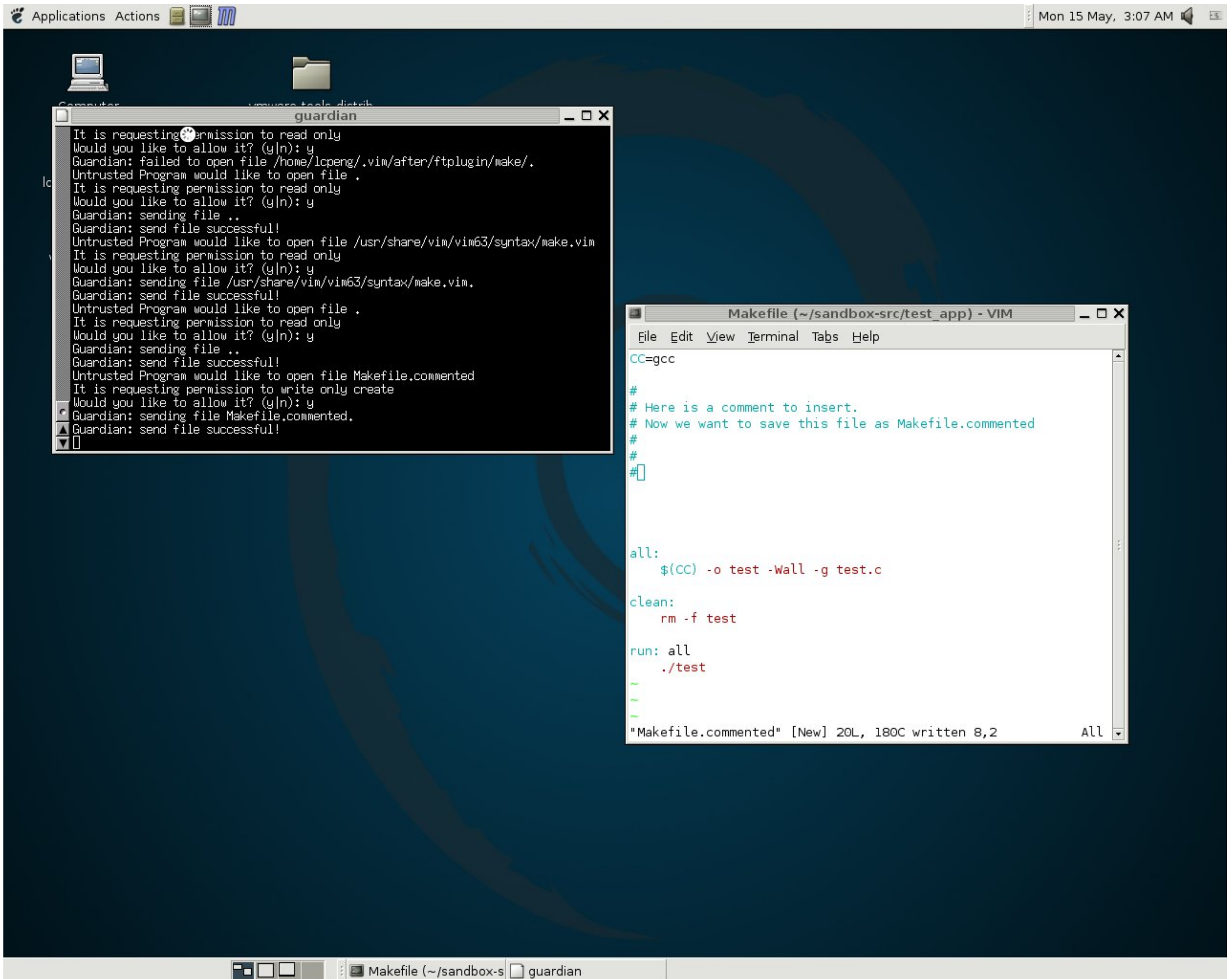

Fig. 5: Now the user makes a second change to the file in vim and tries to save again. This time, however, the user does not allow the write in the *guardian* window, so vim is not be able to save the second change. Now if we just quit out of vim…

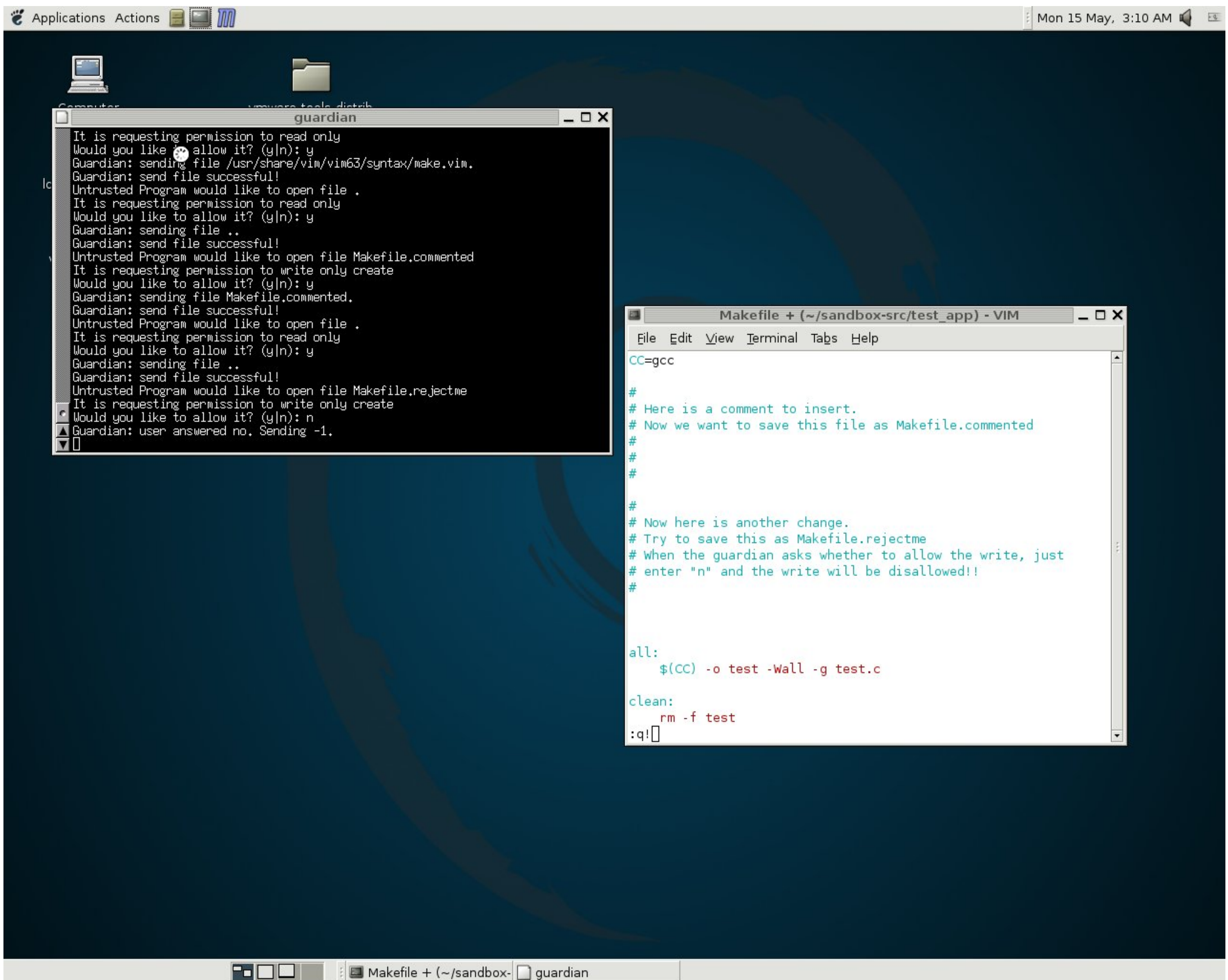

Fig. 6: …we see that the guardian window automatically closed. If we enter the *ls* command, we can see that the first change was saved as *Makefile.commented*, but that the *Makefile.rejectme* version did not get saved at all.