Operating System Protection Domains

Xiaowei Wang
Computer Science Department
Brown University
xiaowei@cs.brown.edu

1 Introduction

One of an operating system's duties is to provide a trustful security architecture to protect user information from threats coming from all aspects the information world. Trojan horses, viruses, and other threats are all aimed at users' personal information. Therefore, the concept of access control has been introduced to system security area years ago. Typically, it guaranties that only processes with proper authorization are allowed to access certain system resources. Access control has become the foundation of system security architecture.

Butler Lampson's original Access Control Matrix[1] is an early model of access control. Lampson's paper claims that a computer system can be abstracted as a set of objects O, which is the set of resources to be protected, and a set of subjects S, which consists of all active entities. For a single active entity to access objects in a set O, it must be assigned a set of rights sufficient to access the target objects. Access Control Matrix could be implemented as access control list, a list of permissions attached to an object in computer security.

Discretionary Access Control (DAC) is an implementation of access control matrices, which stores them sparsely by columns. In DAC policy, the owners of objects have discretion as to who is allowed to access them. It is always implemented as a role based system. Where we have subjects such as user, group and others, and read/write/execute rights. For usual DAC implementation, the controls are discretionary in the sense that a subject with a certain access rights is capable of passing that permission on to any other subject. That is, the owner of a file, can pass its read or write access permission to other users existing in the system.

Mandatory Access Control (MAC) refers to the type of access control by which the operating system defines policies, constraining the ability of a subject or initiator to access or generally perform some sort of operation on an object or target. MAC also allows loading new access control policies into system, which can provide protections to a narrow subset of the system.

With mandatory access control, system security policy is centrally controlled by a security policy administrator or the operating system itself; user does not have the ability to override the policy, for example, granting access to files that would otherwise be restricted. By contrast, DAC, which also governs the ability of subjects to access objects, allow users the ability to make policy decisions and/or assign security attributes. MAC-enabled systems allow policy administrators to implement organization-wide security policies. Unlike in DAC-only systems, users cannot

override or modify this policy. MAC-enabled systems allow security administrators to define a central policy that is guaranteed (in principle) to be enforced for all users.

Occasionally, a system as a whole is said to have "discretionary" or "purely discretionary" access control as a way of saying that the system lacks mandatory access control. On the other hand, system can be said to implement both MAC and DAC simultaneously, where DAC refers to one category of access controls that subjects can transfer among each other, and MAC refers to a second category of access controls that imposes constraints upon the first. For instance, FreeBSD, a free Unix-like operating system, provides both role-based[2] DAC mechanism and MAC facilities.

Even though usual DAC implementation has provided a feasible and flexible method for access control concerns, there are obvious problems: in the usual implementation of DAC, a process could automatically get to access a file merely because the file's permission vector allows the process's UID access. That is, the usual DAC implementation cannot ensure the Principle of Least Privilege: a process must be able to access only the information and resources that are necessary for its legitimate purpose. The usual DAC implementation always fails to separate the user and their running programs. After a program has been launched by a user, its processes could access any resources that its user is allowed to access. A normal user has to trust the program after it starts. Overflow problems, virus, kinds of Trojans and spywares are all aimed to cheating users for their permission to run and then behave like the user, but steal information or break local system. It is a significant drawback.

This project, based on a Linux system, fixes this problem by adding a capability framework provided by a security module. This project can provide users a way to ensure their programs only have minimal privileges, a limited subset of files. Whenever a program attempts to go beyond these privileges, the program is suspended and there is an alert to the user—the user is able to stop such dangerous behavior. The owner of this framework can be either a normal user or a privileged user. To achieve this goal, we added new features directly into the system kernel, allowing it to monitor a running program's behavior, even if it has already gotten permission from its user and started to run. We allow user to perform a "restricted execution" for untrusted program using a new system call "rexec". This new system call will set up a communication channel between the untrusted program and a guardian process, which is used by the user to monitor and control its program.

Generally, my work, in this project, is to re-implement an earlier project by previous students[3], in a recent version of Linux kernel, making it more robust and useable; making it support web browsers, such as Firefox and Google Chrome, well; and design complex test cases to show how well our new features work in providing additional but important protection to a system.

2 Motivation

In operating systems with the usual DAC implementation, users must always trust what they run.

The running program always has the same privileges as its user, which might be far more than the privileges it needs. This provides attackers the opportunity to use the user's permissions for their own purposes. One case is the overflow problem. Attackers can use overflow, such as buffer overflow or stack overflow, to attack the user. This strategy effectively prevents many types of malware-based attacks. For example, the strategy provides protection against stack- and buffer-overflow attacks, in which an attacker sends carefully crafted arguments to a susceptible program, causing it to transfer control to code provided by the attacker. This code now runs with the privileges of the UID of the process running the program.

Another motivation for this project is that, since software applications have become more and more complex, especially web applications or applications downloaded from the Internet, users become gradually uncertain about what they are running. They want a way to monitor these applications' behavior, even after providing them with the ability to access the local file system. This project provides a means for users to efficiently monitor the behavior of any applications they do not trust. This is also the reason why my work concentrates on using web browsers. It is expected that we can provide a useable model or security plan for Internet users.

Modern browsers—Windows IE, Google Chrome, and Firefox—already provide sandbox mechanisms to ensure system security while accessing the Internet. Attackers invariably find ways to get around these mechanisms. The most famous case happened when IE started to support ActiveX Objects. Not only does ActiveX provide IE with a plug-in feature that enriches the applications users can use, but also provides it a series of interfaces that allow access to local resources. Firefox and Chrome browsers support powerful plug-in extensions as well. Recently, Google added a Native Client (a kind of sandbox) [4] feature to Chrome, which provides Chrome applications the ability to run C/C++ code within a Native Client. Even though these features are sandboxed, recent experience shows us there will always be ways to get around these sandboxes and access to the local system.

Therefore, we provide the Protection Domain Model to monitor browser behavior and the applications launched by the browser.

3 Protection Domain Model

This part introduces the framework of this research project, discussing the design of the protection domain model. And finally this part provides a clear system architecture for this protection domain system.

3.1 System Design

In Linux systems, everything has been abstracted into files, even devices such as disks. Typically there are normal files, directories and devices as well as symbolic link files. These files represent all the resources. No matter what kind of resource a user wants to access, a file is opened to access it.

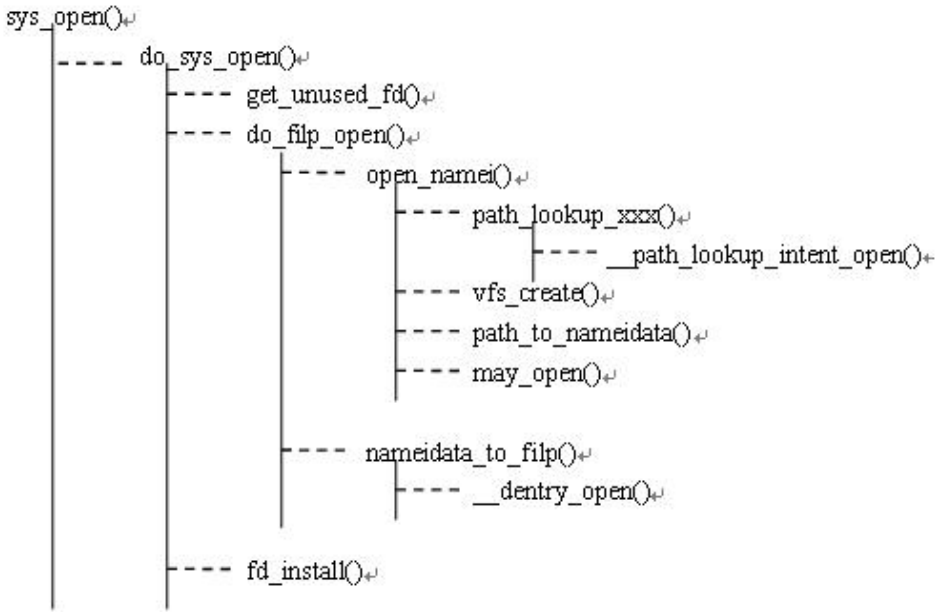

Figure 3.1 Normal File Open Procedure

The picture in Figure 3.1, above, shows how the normal file opening procedure works. At the top level, Linux system provides the open() interface for application program to open files, with various flags indicating permissions. This interface not only contains the name of file, but also the privilege flags showing which types of operations (e.g., read-only or write-only) will be performed. The open() interface calls a system method named do_sys_open(),which calls do_filp_open at the end. These two methods are both implemented in the kernel, and are the primary routines used for opening a file. These two methods do basic jobs, such as allocating a free file descriptor, initializing basic file structures and handling the access checks to see if the user is allowed to do what he or she wants.

As mentioned above, in order to monitor processes in running programs and protect the file system, users are eager to know what the untrusted running program might do to the local system. Because all objects in Linux are represented as files, all processes, whether malware or normal, have to use the open(). The task for our protection domain model is quite clear. It must interpose itself on open calls, making sure users can filter all attempts to open files. That is, whenever the monitored running program attempts to access the file system, it should be suspended, giving the user, if desired, a chance to determine if the access should be allowed.

The protection domain framework consists of a bidirectional communication channel between a guardian process representing the user and the monitored applications. Via this channel, users are able to apply their security policies for restricting running programs. If a monitored program “breaks” the system, say via a well designed stack overflow attack, the user will be alerted of attempts to access files that shouldn’t be accessed.

3.2 Communication Channel

For Linux/Unix system, virtual memory is partitioned into two parts. One is user space, the other is kernel space. Kernel space is the memory space strictly served for running kernel, kernel

extensions such as user's own system calls or modules, and most device drivers. In contrast, user space is the memory used by all user applications, such as your web browser and vi editor, which can always be swapped out when necessary. Linux system allocates processes enough virtual memory space to run. Each process has its virtual memory divided into both kernel space and user space. When a user application wants to access some resources by using open system call, it will trap from its user space into kernel space.

Based on the bidirectional communication mechanism, we could divide our whole system architecture into two parts. One part is kernel modification, including introducing a new system call--restricted execution, and modifying open() and fork() methods. All these modifications are aimed to make restricted processes able to communicate with a guardian process; the other part is a guardian process running as a user application. Apart from communicating with monitored processes, the guardian should also provide enough convenience for usability concern.

The bidirectional communication mechanism is an option for our protection domain framework, however, it introduces a key problem. As user applications are finally trapping into kernel space for accessing resources, a guardian process which is always running in user space is requested to provide a feasible way to send/receive information to/from kernel space. And the same thing should be also expected from the kernel part.

Here we summarize the work of previous students on this topic. Basically, two ways have been considered.

One is mentioned in "Operating System Protection Domains, A New Approach"[5]. It is a way that, generally, a virtual machine has been used as a sandbox providing isolated environment in which to run applications such as a browser. The authors use built-in NFS client facilities of Linux, allowing a directory hosted on another machine to be mounted onto other directory on the local file system of the virtual machine. On the host machine, a modified user-level NFS server is running to monitor the requests from the VM for access to files in the mounted directory structure. That is, the monitored application running in virtual machine communicates with the host machine via NFS service. This is certainly limited by virtual machine, and not what we want here.

One reason is the project mentioned in this paper is very cumbersome to use. For web browsers, it must ensure both the host system and the VM running the same version of operating system, and in most cases, the same version of the relevant program; the other reason is the server couldn't determine which tabs in the virtual machine from the monitored browser were generating requests and thus couldn't do per-tab filtering of requests.

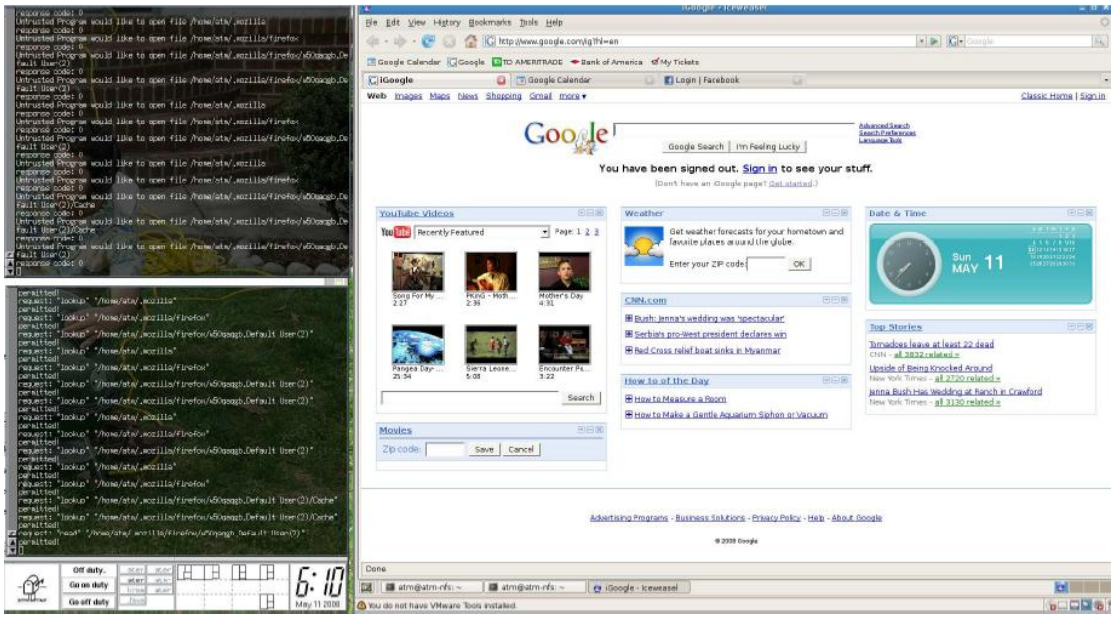

Figure 3. 2 Screenshot of the VMware Solution

The other method is to use Unix Domain Socket or IPC socket to build up a bidirectional communication channel in this project. Unix Domain Socket is a data communication endpoint for exchanging data between processes executing within the same host operating system. Unix Domain Socket provides an in-memory socket structure, which both can be used by processes running in kernel space and user space. The process from one side can create a socket and listen to it. While the process from other side creates a new socket, and make this socket connected to the previous one.

By comparing these two options, Unix Domain Socket is finally chosen as the base of this communication channel in the system.

Unix Domain Socket is quite similar with network socket. However, since all communications happen between processes in the same machine, there's no worry about message sending and receiving cost between different endpoints. What's important, Unix Domain Socket provides desired functionality which makes our framework easy to implement and extend. One can attach any structures we needed with a socket message.

In our project, a guardian process will always listen to a certain socket. While a monitored process, when it is going to access some files and trap into kernel space, will create a new socket and establish a connection to the guardian socket, at the beginning. Then, via this socket, messages could be used to encapsulate the request details, such as which process requests which object and the corresponding privileges, and sent back to the guardian process. After user makes a decision whether to allow the access or not, a socket message containing user's decision will be returned to the monitored process, which can impact the open procedure in reverse.

3.3Kernel Part

Another problem is how to distinguish restricted processes from normal processes, after application has been launched. After all, we only want processes from monitored application programs to communicate with our guardian process, not all processes launched by the user.

The solution is to add a new field, a flag, into the process structure to show whether this process is trusted or not. This field should be only changed when we launch the application in a restricted way. To set this field, a new system call is needed. For untrusted applications, instead of using normal execution procedure, do_execve system call, we use our own system call, rexec (restricted execution). This system call could be just a wrapper of do_execve, but with a step to set a flag in process structure changing a normal process to a monitored process.

As untrusted applications sometimes have multiple processes, which means if we treat the process which call do_execve(), as a parent process, we also need to monitor its child processes, which are created by forking or cloning the parent and always share the same privileges with the parent. So it is required to change normal fork procedure, making it can change the type of child processes to be monitored.

The most important kernel change is to modify normal file opening procedure, open(). After launching applications via restricted execution system call, processes can be distinguished according to the flag set in their process structures. Then, in file opening procedure, we can catalog processes by using this flag. If the open request is from monitored process, we should follow our restricted open procedure, otherwise normal file opening procedure should be followed. During the restricted open procedure, it is not necessary to overwrite the actual opening procedure. We just need to make sure the restricted open method communicate with the guardian process efficiently. In order to achieve this, the restricted open procedure should not only encapsulate enough information and send them back to the user, but receive user's decision correctly. Apparently, after getting user's choice, a normal open procedure can be used to deal with corresponding choices.

3.4 Guardian Part

The Guardian is running as a special user application. It listens to a fixed guardian socket waiting for restricted processes establishing new connections. Every time there is a restricted process willing to connect to the guardian socket. The Guardian process helps set up a communication channel with the monitored process and then receive requests and send back user choices. It always keeps listening to the fixed guardian socket waiting for new connections, until the user shuts it down. After restricted processes finish running, the guardian process is also responsible for closing the corresponding sockets.

What's more is that the guardian part should provide good usability as it directly used by users. As mentioned before, during the startup of Google Chrome, Chrome browser accesses more than 6,000 files. Most of these files are libraries files, or other trivial files. And the majority of requests are only asking for read only permission. If the user has to explicitly approve each of them, no one will want to use the approach. There will be no usability at all. Thus we need to come up with an

automatic or semi-automatic method for skipping trivial files such as libraries, and only bother users when there are attempts to access important files.

4 Implementation Details

This part introduces the implementation details about this project, from basic structures used in this project to kernel modifications, as well as how to implement guardian parts and improve its usability.

4.1 Structures Used In the System

4.1.1 task_struct

task_struct is a structure encapsulating all the information about each process. It is the "process structure" mentioned above. The Linux kernel provides a macro called "current", pointing to the current process's task_struct, which can be used to refer to the current process. For this project, I added two fields into this structure, "is_restricted" and "guardian_fd". "is_restricted" is the flag field mentioned above, which is used to indicate whether this process is monitored or not. The "guardian_fd" remembers the socket's file descriptor after a monitored process establishes a connection with the guardian process. Default value of this filed is -1, indicating there is no connection yet. These two fields will be widely used in the kernel parts.

4.1.2 open_args_t

This structure is used during the communication between kernel process and guardian process. It is attached to the socket message, encapsulating basic information such as message type, filename(must be full path), process id, opening flags. "type" field in the structure, stands for message's type. There are two kinds of messages in the system till now. The most important message type is "open". This type indicates there is an open request. "pid" in this structure provides the monitored process's identifier, to tell the guardian process which restricted process sends the message. This structure has to be both defined in the kernel and in guardian part in the user space.

4.1.3 Other structures

"agree_file_t" and "acl_node_t" are two structures only used in guardian part, aiming to help provide better usability. "fd_set_t" encapsulates the "fd_set" structure. "fd_set" structure represents socket's file descriptor set for selecting operation in typical usage of Unix Domain Socket. After the guardian process binds to a fixed socket, it will blocked on the select method, monitoring existed socket's set and waiting for new connections. Every time, there's a new connection coming(a new socket is created), a corresponding file descriptor should be added into fd_set set. "fd_set_t" is a wrapper for "fd_set", defining basic manipulations, such as add a new file descriptor into the corresponding "fd_set" set and so on.

4.2 Kernel Modifications

4.2.1 rexec system call

Because we can only set a process to be monitored when the application is launching restrictedly . It is a feasible way if we implement a system call, which is used to launch application restrictedly. To ensure all the processes from this application are monitored. A restricted parent process must be forked in this system call, and this parent process must call do_execve(). Then all processes of the application would be child processes of this restricted process, and be restricted (set flag) eventually.

4.2.2 fork

To ensure that all processes of untrusted application are also restricted. During the procedure of forking a child process, we also set the forked process's "is_restricted" flags to 1, indicating that the new process is monitored, just like its parent.

4.2.3 open

There should be two kinds of open procedure in Linux kernel. One is a normal open procedure which is used to handle open request from non-restricted processes. The other should be a restricted open procedure, which, of course, is used to handle request from restricted processes. Therefore, we need to add a new branch into the do_sys_open() method. If "is_restricted" flag is set, we need to do a restricted open, otherwise a normal open procedure is sufficient.

There are 4 steps in a restricted open. (1)Checking whether the restricted process has already connected with the guardian process. If not, which means it is a new coming process, and we need to establish a new socket connected to the guardian process first. After creating a new socket, we need to made the process remember the corresponding socket's file descriptor for future reuse. The file descriptor will be kept in "guardian_fd" field; (2)Encapsulating process request into "open_args_t" structure and attaching it to a socket and then sending back the socket message to guardian process. It is of significance that we do a double check here. Sometimes after guardian_fd referring to a new socket, this in-memory socket might be closed but not deleted by other kernel processes. So we cannot directly reuse such closed socket, a check is needed before we could use. If it was closed by other processes, a re-open operation is needed. (3)Waiting for user decision back from guardian process; (4)According to different responds, executing different open procedures. Here has 3 kinds of user decisions. One is fully agree a restricted process's request; One is deny a request; The rest is partially agree with a request. When user fully agree a request, we can directly execute a normal open procedure to open the requested files; When user just partially agrees with a request, open flags needed to be reset. For instance, if restricted process wishes to open a file with "read write" permission, but the user only agrees with "read" permission. Then the open flags should be changed from "read write" to "read" only. A normal open procedure should be executed then, using new open flags; When the user deny a request, a error value will be returned as do_sys_open()'s return value, this error value actually corresponds to no permission error pre-defined in Linux kernel.

4.3 Guardian Part

The guardian part in this project is generally implemented using a single thread model. I provide an extendable architecture which one can easily modified the whole architecture into a multi threads structure, but I suggest not. By using single thread architecture, there's no need to consider shared resources between different threads. For instance, if you make one thread listen to the guardian socket accepting untrusted processes requests and another thread deal with such request, we need to waste time on passing such requests between these two threads. It is not worthy, let alone the time spending on locking and unlocking shared resources among different threads. The most efficient way is to use one single process handle all these communicating work, from creating guardian socket to handling requests. However, assistant threads can be used to deal with other jobs. Such as a watch_dog thread to monitor when should end the guardian process. If there's no new connections any more, say all restricted application programs finish running and there is enough idle time, the guardian process could be ended, no need to wait on select function any more.

4.3.1 Communication

From the previous section, it is known that, for monitored processes, the only important feedback from the user, is how the user decide to change its open flags. Therefore, the guardian process only need to send back user's choice on open flags. For example, whether the user permits a read-write operation or read-only operation or just say no to the request. Therefore, for return message, user's choice will be attached as an integer, which always stands for the open flags.

4.3.2 Usability

Usability is of paramount importance, as no one wants to reply to thousands of requests, just for launching a simple application. To deal with this problem, in the guardian part, I classify the requests from monitored processes into two categories. One is the "read-only" request, which means the untrusted processes only request for read only access to a certain file and cause no modification to the local file system. The other is "other" request, such as write-only, append and so on, which will be most likely harmful to the local file system. It is quite clear that user should be much more careful when he or she deals with the second kind of requests. And it is also apparent that there will be a great improvement of usability if we simply skip most of the "read-only" request.

But can we simply skip these read-only requests? Of course not, a better way is to provide an "ignore" list, which contains resource entries (always be the full path of a file or directory) that an request of these entries can be allowed without user's permission. These files should be libraries or caches, temporary files which are trivial enough to be ignored. And I also provide an "important" list, which contains file entries that should never be ignored. That is, no matter which kind of request it is, the process has to gain user's permission to continue. I implement the "important" list as parameters sending to rexec() system call. After restricted execution is called, the rexec system call will encapsulate these important filenames into open_args_t structure but with a type "acls" which is different from the type "open", and send them back to the guardian process. Both "important" and "ignore" list, work for read-only and other request.

To make it even better, users can change "ignore" list on the fly; Users could also add a parent

directory of a specific file into the ignore list; For multiple processes, I also provide each process a list to check if an entry has been already allowed to access for this certain process. But "ignore" list is used as a global list shared by all processes; All these lists can remember user's choices. For instance, if a parent directory has been approved, user can tell guardian process to add it to "ignore" list. Then all the following files belongs to this folder will be treated as trivial files and ignored; Apart from changing "ignore" list on the fly, the list can be pre-defined. Just like "important" list, resource's entries can be added offline; I also provide key word policy in the program. If a filename starts with words such as "/lib", it is much more likely to be a library file which are trivial enough, and can be ignored. If accidently, an important file also have these key words, users can add this file into "important" list to make sure this entry be checked.

By using these methods, the guardian demo is much more easier to use.

5 Testing

There are three testing cases for the project. The first is a vim editor test case, which uses the guardian to monitor the vim editor's behavior. This is a basic test case, which could be used to test if the guardian process can communicate with restricted processes correctly; the second and third test cases are both web browsers—Firefox and Google Chrome. These two test cases are much more complicated than the vim editor testing since both of the browsers will access to thousands of files even just during startup. From these browsers, one can not only access the local system, but also run plug-ins or even executable codes directly.

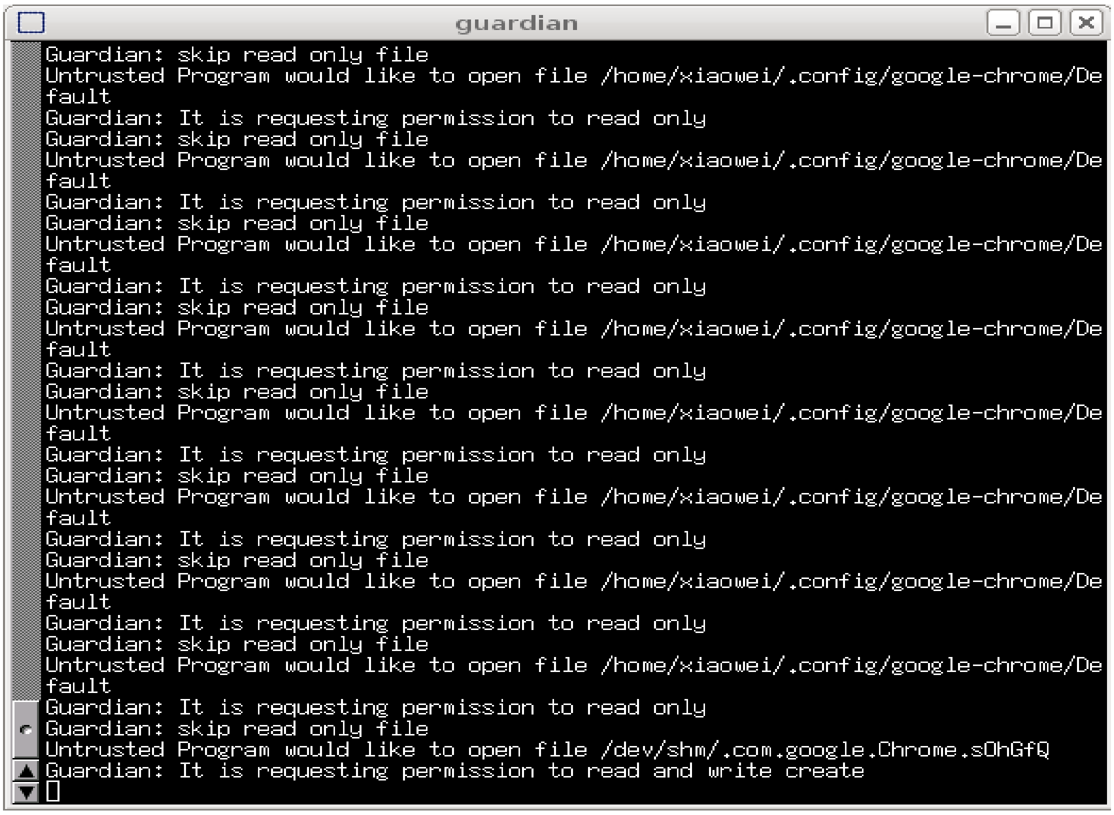

Figure 5.1 Guardian Monitor

I concentrate on the Chrome testing case to show that the implementation works well, and in particular provides enhanced security when users use browsers. In this test case, one executes a notepad within the Chrome browser, but both the browser and the notepad process are monitored by the guardian process. This is important since I actually get around Chrome's sandbox mechanism and run an executable program which directly accesses the local file system. That is, even if an attacker can break the browser's security policy, the browser and its plug-ins are still monitored by our guardian process, providing a last-ditch defense.

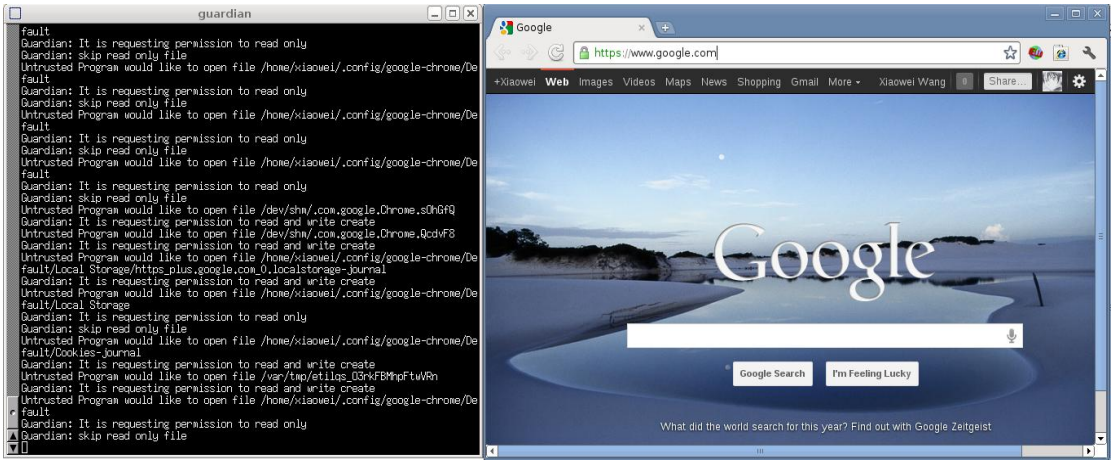

Figure 5.2 Restricted Google Chrome

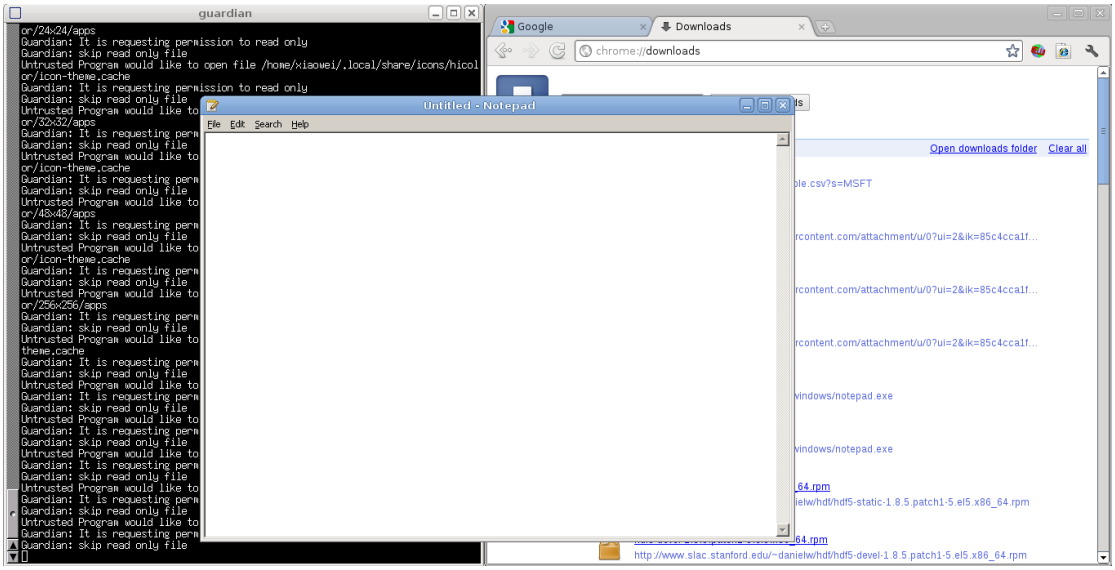

Figure 5.3 Restricted Notepad Launching by Google Chrome

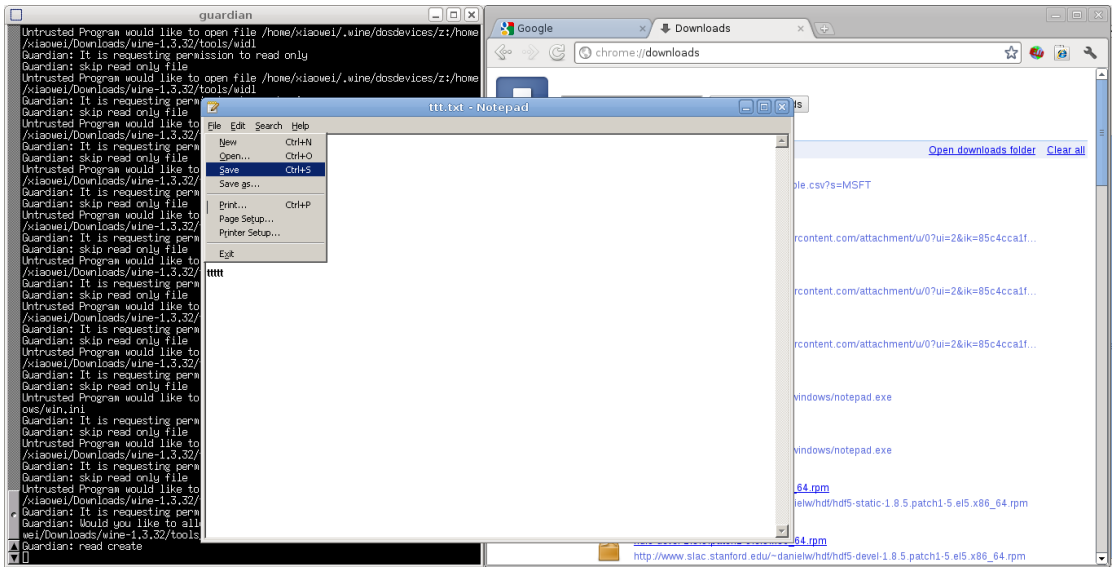

Figure 5.4 Restricted Notepad Trying to Modify a File

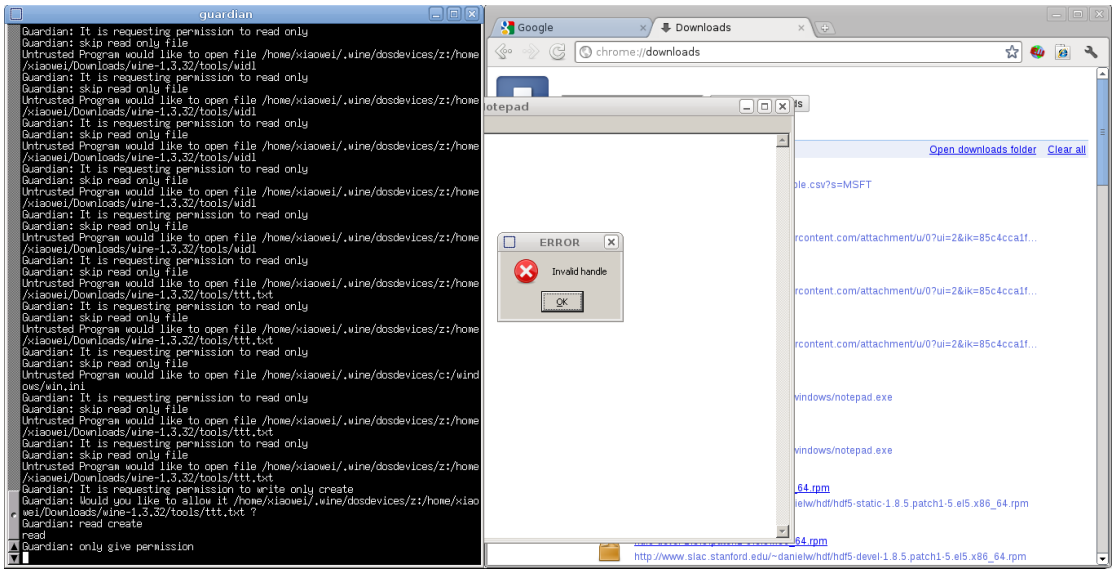

Figure 5.5 Permission Denied by User

In the above figures, Google Chrome is running under a Linux system, Debian 6. A wine layer has been installed in the Debian system for running Windows applications. In this case, the application is a simple notepad.

Wine is an open source application, which allows programs written for Microsoft Windows to run on Unix/Linux systems. Wine is not an emulator but can be seen as an abstract layer which duplicates functions and DLLs in Windows by alternative implementations in Unix/Linux systems. These functions and DLLs are usually essential for Windows programs. The downside to wine is you won't be able to run Windows applications that aren't reachable from a wine drive (like C:). In this case, the notepad application must be stored in a reachable wine drive.

It is not easy to run an executable program such as a shell script within Chrome on a Linux system—what we're doing is attempting to launch a successful attack on Chrome's sandbox.

That's why I introduced a wine layer to the Linux system, and run a Windows application. Note that, there's no need to use the wine command, one could use our guardian framework as usual. But all processes are launched restrictedly and monitored by our framework. As it is also hard to directly get around Chrome's sandbox, the notepad program should "cheat" user's permission to run. That is, for test purpose, user should allow the program to run, even if there's an alter from Chrome itself. In this case, after user clicks the notepad's tab in Chrome and allows it to run, wine will translate all the application's system calls(WIN32 APIs) to the corresponding Linux system calls and then make the notepad run.

Apparently, it is a dangerous behavior to the local system. Similar programs but malwares can also cheat user's permission to run from browser and use wine or other tools to access the local file system and make harmful impacts. However, our framework in this project monitored everything. From the figures above, one can see the "dangerous" notepad program wants to modify some file in the local system, but it was monitored and stopped by the guardian user!

6 Future Work

Though this system is fully functional and has been tested and optimized significantly, there is certainly room for future extension of the work described here.

First, the usability still has room for improvement. How to design a better method automatically distinguishing trivial files from important files, and how to provide the user a much more attractive and convenient user interface, instead of using a terminal-style interface, are both questions should be dealt with in the future. And we could also provide a suggestion and catalog mechanism. That is, files are classified into different catalogs. Then, according to the catalogs, our guardian framework could provide users suggestions whether this is an important file or just a library files or something else. Actually, it is impossible to distinguish trivial files and non-trivial files, only from their names. A new idea should be introduced into this policy to help classify;

Second, instead of directly modifying the system's kernel, there might be a better way to implement the whole project, to make the guardian much more easier to transplant. Now because of the need to modify and recompile system kernel, it is not such easy to install this guardian framework to other machines. If the framework could be implemented as an new module added to the kernel, it would be much easier for user to install. Restricted execution system call shall be easily implemented as a kernel module, which could be install or uninstall on demand. However, how could we distinguish between normal processes and restricted processes in a transplantable module, and how could we make child processes of a restricted process also restricted in it? What if a knowledgeable attacker could get around this by bypassing the module and making the normal system calls directly? These are key problems needed to be solved before we do such modify.

Third, applications' sockets might be closed by other processes sometimes. Then we cannot always reuse the old socket to accomplish communicating jobs. After the old socket is closed for

some reason, there must be a failover mechanism helping restricted processes reconnect to guardian process. Although, I provide such failover part and fix this problem, it is only a bug fix, not a solution. Future study is needed to figure out the real reason, why these sockets are closed while running and which processes close them.

7 Conclusion

Modern operating systems always provide a solid role-based security model for protection system resources. Some of the enhanced systems, such as FreeBSD, also try to introduce Mandatory Access Control frameworks to enhance their security. Access control becomes a really important topic which should be considered while designing a system. It is not just about controlling user access; but also should be able to control the access of user's running applications.

This protection domain project augments the usual DAC framework to provide the user effective monitoring and control of specific, which ensures the user's processes may access only a limited list of files. This framework provide a bidirectional communication channel between restricted applications and their user, which could arm user with abilities to deny or partially agree restricted processes' accessing request. When properly used, user should be able to comfortably and safely execute arbitrary binaries without having to worry about how they affect the system.

We believe that protection domain, the framework introduced in this report, will be proved to a useful and helpful addition to enhance system's access control ability and of course, system security.

References

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

[2] 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

[3] Eric Tamura, Joel Weinberger, Aaron Myers. "Operating System Protection Domains".

[4] Bennet Yee, David Sehr, Gregory Dardyk, etc. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". 2009.

[5] Aaron Myers. "Operating System Protection Domains, a New Approach". 2008

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

APPENDIX

Usage of this guardian framework

1. agree the request : press enter or type "y"
2. deny the request: type "n"
3. partially agree the request: input permissions that are granted
4. granting access of a directory: input full path of a directory, the directory must be, at least a parent directory. If followed by "&", the choice will be shared among different processes
5. parameters could be passed to the framework
    (1) "**--restricted-file**" followed with important files
    (2) "**--remember-yes**" indicating remember user's choice, it is the default value
    (3)"**--remember-no**" indicating not remember user's choice
    (4)"**--log-on**", dump basic test log to "guardian_dump" file, the default value is "**--log-on**"
    (5)"**--log-off**", turn off test log
    (6)"**--skip-read**", indicating whether to skip read only requests or not