Operating System Protection Domains, a New Approach

Aaron Myers
atm@cs.brown.edu
Computer Science Department
Brown University
May 12, 2008

1 Introduction

Secure computing has been an issue of high concern since the beginning of shared computers. As computers became more prevalent, so did resources shared between different users of computer systems, and naturally, users came to want to limit the access of other users to their own resources. To this end, many access control mechanisms have been devised whose general purpose are to allow access to system resources only to authorized entities. Though these methods have served us well for decades, their evolution has been overall slow.

Butler W. Lampson first formalized the general notion of access control within operating systems in 1972, with his introduction of the "matrix of access attributes"[2]. Though this foundational system is robust and general, it cannot scale to today's computer systems which potentially contain hundreds or thousands of subjects and objects. Furthermore, in the interest of simplicity, most actual access control mechanisms in the real world limit to some small set the type of actions which can be controlled.

Perhaps the most pervasive general security model found in modern operating systems are those implemented by Unix-like operating systems for protecting files. This scheme represents a pseudo role-based, discretionary access control mechanism, combined with very limited access control lists (ACLs). Under this system, files are the objects and users are the subjects. A file has a set of permission classes: one for the "owner" of a file, one for the "group" of the file, and one for all other users. The owner and group classes act as single-subject ACLs, and each can be set to a single user or a single group. The owner of a file is capable of changing the group class setting, as well as any of the permissions of an object. It is this aspect that makes this system "discretionary." Here, a group is defined as being a set of users of arbitrary size. The actual permissions can be set individually for each of these permission classes. The use of the group class is what makes this system pseudo role-based; an arbitrary collection of users can have a single group, all of whom have a set of rights based on the files whose group class is set to that group. This having been said, the use of groups does not in fact constitute full role-based access control. In a fully general role-based access control scheme, an object should be able to have multiple different policies for different roles assigned to them. Furthermore, full roll-based access control mechanisms usually allow for hierarchies of

roles, which Unix does not allow in general, though this is somewhat available through the use of the Unix super user, or "root."

Another common feature of Unix-like operating systems is what is referred to as the "effective user ID" system. Under this system, the user who is the owner of a file which is executable has the discretion to set the permissions of this file to be "set UID" or "set GID". If a file is set UID, then when it is executed, it will take on the user ID of the *owner* of the file, rather than the user executing the program represented by the file. Similarly, when a set GID file is executed, the program represented by this file will take on the group ID of the group class of the file, in addition to the groups of the user who is executing the program represented by this file. This facility is most often used to allow users to temporarily elevate their own privileges to perform some specific task or set of tasks. This is another role-based mechanism which is capable of providing great flexibility to the policies which can be enforced on a system.

To facilitate system administration and other tasks requiring privileges, Unix-like operating systems generally provide for a super user called "root." This user is capable of performing any and all actions on every object in the system. Furthermore, this user has complete discretionary control over all objects, and may obtain the role of any user in the system. When combined with the set UID/GID facility describe earlier, the capabilities of the super user can be arbitrarily passed on to other users by setting specific executables to be set UID/GID, and having the files owned by the root user.

One well-known mechanism common on Unix-like operating systems which leverages the privileges of the root user as well as the set UID/GID facilities is a program called "sudo." This program is owned by the user root and is set UID. When executed, this program interprets the settings contained in the file /etc/sudoers. This file allows one to specify that specific users, groups or all users be able to execute specific commands as another user, usually the super user.

Though Unix-style access control is rather inflexible on the whole, it has proven sufficiently robust as to have been successfully used for literally decades, and can still be seen in relatively recently-created operating systems such as Apple's OS X. However, there are many limitations to this system which prevent it from being robust enough to fully express certain security policies which could potentially reduce the ability of attackers to cause harm. As such, there has been some work done previously to extend and complement the Unix security capabilities.

Perhaps the most full-featured security extension available to a Unix-like operating system is that of SELinux[3]. This work, developed by and for the NSA, provides a series of hooks into many of the operating system facilities which have to do with resource control. SELinux provides a policy system which is capable of enforcing DoD-style mandatory access control, in which a central policy administrator is responsible for setting all ac-

cess control permissions for all objects in the system. Furthermore, SELinux allows for policies to apply to user-level programs in addition to a user as a whole. As we will see, it is this feature which is required to provide truly tight security in an environment where programs may access and be accessed by external, potentially malicious entities.

The work in this paper builds upon the work of Peng[4]. In this work, Peng introduced file system sandboxes as a way to interpose upon all file accesses performed by specific programs. This is done so as to require that all programs execute with the least level of privilege required for them to correctly perform their task.

This work is a continuation of the work done in [6]. In this previous work, we extended the work done by Peng to be implemented in the operating system kernel, and to interpose upon, specifically, the open(2) system call.

2 Motivation

As mentioned previously, Unix access control is very much user-centric. The vast majority of all access control decisions are based around what user is attempting to access a particular resource. Through the effective UID/GID system, per-program privilege elevation is possible. This is used, for example, to allow users to change their own password. Though this system has been effectively used for many years, it has serious drawbacks which must be addressed in order to move beyond the user-centric access control mechanisms.

The principle hurdle to providing better security for users in a Unix-like environment has to do with the privileges with which an ordinary program executes. Without considering programs which are set UID/GID, any program which is executed by a user has the full set of privileges granted to that user, regardless of what privileges that *program* actually requires to function. Thus, the program 'ls', whose purpose is to list the contents of directories, in fact executes with the permission to, for example, remove all of the files in the home directory of the user executing 'ls'. While this simple fact may not seem like much of a problem for small, self-contained programs like 'ls', it becomes a very important issue for any software which communicates with outside entities.

Consider, for example, a web server. Since a web server needs access to write to log files, read from many possibly disparate parts of the file system, and listen on privileged TCP ports, web servers often start and run as the root user. While this level of privilege is certainly sufficient to perform all the tasks a web server might need to perform, it is also far more privilege than the web server actually needs.

Consider further, the web browser. Web browsers in general need very few privileges to

do their basic functions. They need to be able to make network connections, read and write to some set of files to store settings and other internal data, and they occasionally need to write a file which the user requests be downloaded to some place in the file system that the user can write to.

Though in general Unix-style access control is very effective at allowing users to perform only the actions they are permitted to, there are little or no facilities to allow an individual user to restrict the privilege of a specific program or process they wish to run. Thus, a user must fully trust all of the programs he or she executes because there is no way of preventing a program from performing any and all operations which the user is capable of doing. Ideally, users should be able to execute arbitrary processes with the minimal set of privileges necessary for them to perform their legitimate purpose, and nothing more. In the case of the web browser and web server, these processes should only be able to access files and perform network communication which is explicitly allowed by the user executing these programs.

3 Sandbox Model

Ideally, a user should be able to execute programs in such a way as to minimize the privileges, and therefore minimize the damage, that an errant program could potentially do.

Perhaps the simplest policy which could be created would be for the program(s) executing within the sandbox to be denied access to all resources outside of the sandbox. While this would certainly be effective at preventing damage, it would almost certainly not be workable from a usability standpoint. Even if we assume that programs are able to execute at all without access to external resources, users would most likely be dissatisfied with the disablement of many of the features they are used to in their programs. Thus, some more complex policy is required.

SELinux[3], mentioned previously, allows for a rather flexible and extensible policy description. However, SELinux policies must be created statically by a central administrator, and cannot be modified on the fly. Furthermore, the use of Linux Security Modules, which must be written in C and loaded into the kernel, are a cumbersome way to express policy. In general, the use of hooks into OS kernel routines which deal with resource management is difficult to maintain and develop.

In the previous work of Tamura et al[6], we provided a new system call called "restricted exec." This system call, modeled after the regular exec(2) system call, execs a program in the current process which is then marked as being "restricted." Then, modifications were made to the kernel of the operating system in the paths of resource-allocating

routines which check the restricted attribute of the process. If the process is found to be restricted, then some extended attributes of the file which represents the program being executed are examined. These extended attributes contain a simple access control list which is meant to represent the minimal set of resources (limited to files) that this program requires to operate fully. If access is attempted on a file which does not appear in these attributes, then a user-level "guardian" process, which was specified at the time of the call to restricted exec, will be consulted to get the final determination of whether access to the given resource should be allowed.

Though this previous work is fully-featured, it is still cumbersome to express policy in this way. Furthermore, the policy description language is not sufficiently expressive as to allow completely arbitrary policies. For example, this scheme is very much file-centric, and it is not at all clear how this method would be adapted to perform access control on other types of resources. Finally, the implementation of this scheme was very challenging in that many kernel modifications must be made in many code paths dealing with resource allocation, all of which could have subtle, but potentially significant consequences to the behavior of the kernel.

In the current work, we leverage the implicit sandbox provided by an operating system running within VMware workstation to provide the protection of the host system against the actions of an individual process. We provide access to the necessary resources of the host system through different mechanisms of enforcement. For the purpose of this paper, we are concerned with the protection of files and network connections. Policy decisions are made by a single, configurable mechanism running on the host operating system, which communicates with the enforcement processes running on the host system. Through the use of the existing third-party system known as XACML, the extensible access control markup language, we support expressive policy description capable of describing truly arbitrary policy.

4 Design and Architecture Overview

In the previous works of Peng[4] and Tamura et al[6], both focused on implementing a tightly-knit system to implement subject-based access control in Linux. To that end, both works focused on making kernel modifications to the Linux kernel and communicating with user-level programs which were to be run on the modified kernel. In the current work, we attempt to provide a more modular system which together provides the fine-grained access control which we desire. Furthermore, rather than build the restricted execution environment by modifying all the relevant code paths in the kernel which have to do with resource allocation, we run the programs we want to sandbox within a virtual machine running in VMware workstation 6, and provide access to the required resources of the host machine through custom mechanisms.

To allow users to access files on the host system from within the VM, we use the built-in NFS client facilities of Linux. These allow a directory hosted on another machine to be mounted onto some directory within the local file system of the virtual machine. In the case of the web browser, we mount the directory contained within the user's home directory which stores the user settings of the browser, as well as a cache and whatever other files the web browser needs to run. In order for this to work, both the host system and the VM must be running the same version of the operating system, and in most cases, the same version of the relevant program.

On the host machine, we run a modified user-level NFS server. It is this program which serves the requests from the VM for access to files contained within the mounted directory structure. Thus, the protocol between the VM and the host for file accesses is simply that of NFS version 3. This server is modified to do additional enforcement upon file accesses. In XACML terminology, it is this program running on the host system which acts as the PEP, or policy enforcement point.

From here, the NFS file server program contacts another user-level program to perform the actual policy interpretation. In XACML terms, it is this program which acts as the PDP, or policy decision point. It is this program which compares the request being made to the user-level NFS server against the policy currently being enforced on the system. If the action being requested for the given resource is permitted by the policy, then the PDP signals to the PEP that this is the case. The reverse is also true in the case that the policy expressly denies the action on the given resource. In the event a conclusion cannot be drawn from the given policy, then the program acting as the PDP prompts the user of the system for what the verdict should be on this particular action.

A similar facility is used to observe and make decisions on network traffic into and out of the virtual machine. The concept is identical to the design of the file access protocol: the virtual machine talks to some user-level program running on the host system which acts as the PEP. This program communicates with the PDP in the same manner as the NFS server, and the verdict from this communication is used to determine the action of the PEP; either to allow or deny the requested action on the given resource.

This design, of having the guest OS resource managers communicate with various PEPs on the host OS, which in turn use a common protocol to communicate with the PDP running on the host can be seen in Figure 1. In the implementation described in this paper, the resource managers are, respectively, the NFS client implemented by the Linux kernel, and the IP stack implemented by the Linux kernel. The implementation of the PEPs, PDPs, and required kernel modifications is described in detail in the next section.

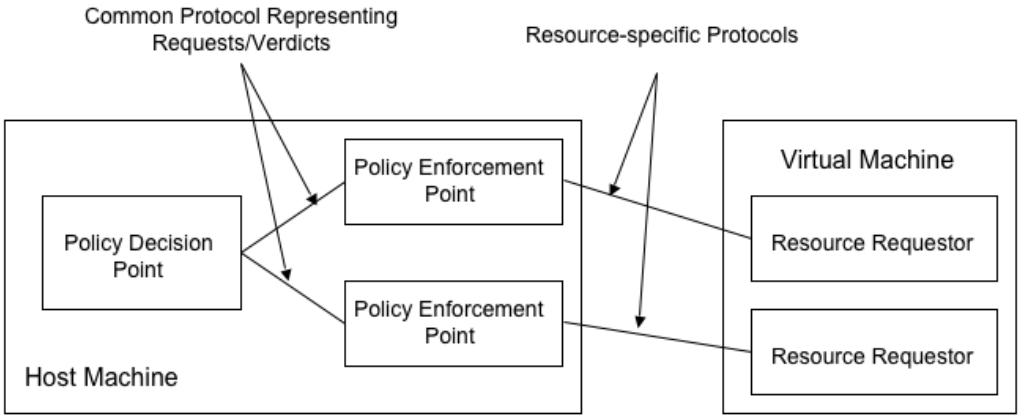

Figure 1: Architectural Overview

5 Implementation

This section details the implementation of the system so far described. For the rest of this paper, the version of Linux running on both the host and guest systems is Debian etch running a 2.6.18 kernel.

5.1 Modifying NFS Client Code

There are two problems with the current implementation of the NFS client in the Linux kernel which prevent the implementation from being usable as-is. While these problems do not necessarily cause incorrect behavior per se, they certainly limit the usefulness of the system if they are not addressed.

The first problem is that the Linux NFS client code uses an attribute cache to cache responses to NFS ACCESS calls. Most NFS clients, before opening a remote file, perform an ACCESS NFS request to obtain the permissions which are present on the file[1]. The NFS client may subsequently perform many operations based on these permissions. This causes difficulty for us because we'd like every file access to be run through our access checks, not just on the first time a particular request is made. The solution to this is to simply disable caching on the client side. This is done by modifying the function **nfs_do_access** in the file **fs/nfs/dir.c** of the Linux kernel. In this function, there is a check which short circuits the function call in the event an entry is found in the cache.

The second problem is that the NFS client code, before attempting to access a file, always

asks for the full set of permissions of a file, regardless of what permissions the subsequent access actually needs. e.g., even if the client just needs to read a file, it also asks if it has permission to write the file. This is done for performance reasons, as the response is cached, as previously noted. The solution to this is to make the client request only the permissions it actually requires for a particular request. This is again accomplished by modifying the function **nfs_do_access** in the file **fs/nfs/dir.c** of the Linux kernel to only request information about the permissions which are actually necessary.

5.2 Modifying NFS Server Code

In order to create the NFS server which runs on the host system, I chose to modify the user-level NFS server implementation unfsd 3-0.9.19. User-level NFS servers are rarely used on a large scale, mostly for performance reasons, but doing this in our case allows the startup and configuration of the whole system to be much simpler. Furthermore, in an environment such as the Brown CS department, where users by and large do not have local root access to their own machines, users may still run the system without special privileges.

The NFS version 2 protocol assumed that permissions on a file could be entirely determined by inspecting the Unix-style permission bits and UID/GID of the file. This allowed clients to attempt to deduce file access rights without necessarily explicitly talking to the server. This is a problem in our case since there is additional policy being enforced on file accesses beyond the simple Unix-style permissions. NFS version 3 corrects this issue by adding an ACCESS procedure, which is an explicit over-the-wire permissions check. It is here that we place the code to interpose on NFS traffic to and from the VM. In the code for unfsd, the relevant code is in the function **nfsproc3_access_3_svc**, in the file **nfs.c**. The permissions the NFS client is checking are accessible in this function via the argument **argp->access**. It is in this function that we place the code which communicates with the PDP to perform the additional access checks required for our system.

5.3 Running NFS Server on Host

Using the unfsd package described in the previous section, running the NFS server is basically trivial. unfsd implements both the MOUNT and NFS version 3 protocols, so you only need to run the one program to get both services. unfsd can be made to read from an arbitrary file to get its export information by using the **-e** option. Since NFS/MOUNT servers usually listen on privileged ports, we use the **-u** option to unfsd to have it listen on unprivileged ports. unfsd still registers with the port mapper, and thus mount and NFS requests will be directed to it. In order to prevent the program

from detaching, it should be run with the **-d** flag, which causes unfsd to not fork into the background at startup. This is useful so we can see debugging output as file accesses are made and verdicts are returned from the PDP.

5.4 PEP, PDP Communication

In order to implement the actual access control decisions, I opted to use the XACML policy description language. Unfortunately, since this technology is somewhat immature, I could not find an open source C or C++ implementation of a XACML policy interpreter. Thus, for simplicity, I decided to go with Sun's implementation which is written in Java. To make this work, I run the PDP as a stand-alone Java application which communicates with the NFS server over a plain old TCP socket. This decision and the relatively inefficient non-binary protocol is probably the main bottleneck in the performance of the overall system. The protocol I devised is a simple text-based protocol in which the PEP first sends the action being attempted to the PDP as a newline-terminated string, and then it sends the resource being accessed as another newline-terminated string. The PDP responds by sending a string containing the response (allow or deny) as a numeric code as a new line-terminated string.

5.5 Running the PDP

I wrote the PDP to start listening on port 4500 when it starts up and to accept any connection attempt made to that port. When a connection is made, the PDP spawns off a thread to handle all subsequent requests from the PEP which has connected. The NFS server lazily connects to this port at the localhost when it first needs to serve a request. Thus, one must start the PDP any time before the first request for a resource is handled by a PEP.

I designed the PDP such that the XACML policy interpreter is first consulted. If this returns a verdict of either PERMIT or DENY, then this is used as the final verdict from the PDP. Otherwise, if the XACML interpreter returns INDETERMINATE, the user is consulted via a prompt from the PDP running on the host operating system. This verdict is then used as the access decision. The default is DENY.

In Figure 2 we see a screenshot of Firefox running in VMware. All accesses to the **~/.mozilla** directory are being performed using the NFS protocol to a PEP NFS server running on the host. The output of the PEP can be seen in the top terminal to the left in Figure 2. The PEP is then communicating with the PDP which is running in the lower terminal on the left.

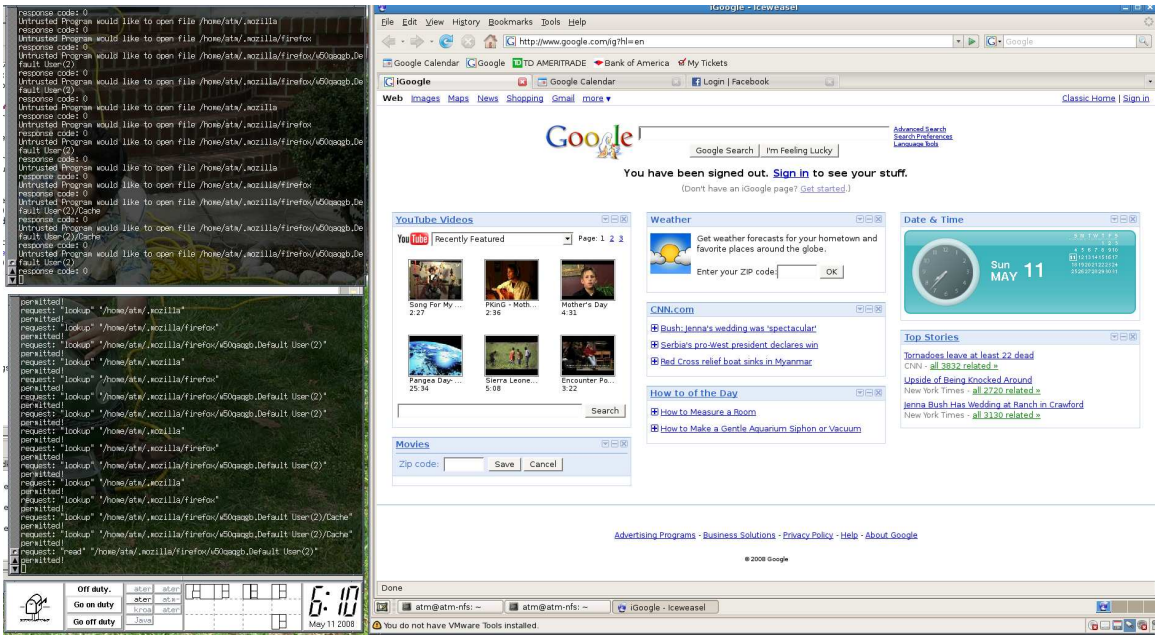

Figure 2: Screenshot of guest communicating with PEP communicating with PDP

5.6 Policy for Firefox, Thunderbird

As a first approximation, I created a XACML policy which is rather coarse-grained. My scheme is to mount three directories from within the guest OS: the standard **~/.mozilla** and **~/.mozilla-thunderbird** directories, and a special **~/insecure** directory. The **.mozilla** and **.mozilla-thunderbird** directories are the standard locations where Firefox and Thunderbird store all of their settings and user profile data. The policy I initially implemented allows full access by the guest OS to these directories. The "insecure" directory is insecure from the point of view of the host operating system. This is intended to be the sole location where files should be saved by the guest OS which need to be accessed by the host OS. File paths are matched to resources specified in the policy using regular expression matching. The operations which are allowed on a resource are identified in the policy using exact string matching. This policy can be seen in its entirety in Appendix A. Though this policy is somewhat coarse, it is effective at restricting access by the guest OS to only these files and directories, and at restricting the operations which can be performed to those specified in the policy.

Browser vulnerabilities and web pages which exploit these vulnerabilities are relatively commonplace today[5]. These vulnerabilities vary from a web page being able to read browser settings that it shouldn't, to severe vulnerabilities that could allow for arbitrary code execution. The latter type of vulnerability is increasingly being found in browser plugins, some of which are required if a user wishes to visit certain sites[5]. The enforce-

ment of this policy is capable of greatly limiting the damage that a vulnerable browser could potentially do. Though the vulnerable browser is still just as vulnerable, what actions it can perform on the host operating system is restricted to only being able to access the files in the shared directories.

A more restrictive policy would most likely identify some files within the relevant directory which could be marked as read only. Or, perhaps, to limit the outbound network communication to reduce or eliminate the possibility of sensitive user information from being obtained by malicious third parties.

5.7 Setting Up VMware Workstation to Allow for Filtering via iptables

There are several options one can use to get network access to a VM using VMware workstation. These include bridging, NATing, and host-only networking.

The simplest way to get network access is usually to use NATing, which VMware configures nearly automatically for you. Unfortunately, there is no way to get iptables to filter packets using this configuration. The problem is that this system uses the vmnet8 virtual network interface to communicate with a VMware-written NAT software daemon (vmnet-natd) running on the host. Despite being able to snoop this traffic using tcpdump, this communication is done completely at the MAC layer, and thus never gets touched by the IP stack where iptables lives.

Bridging is not really appropriate in this scenario because it breaks the desired encapsulation of having the VM only able to route traffic through the host.

This leaves only host-only networking as an option, and this is indeed what we want to do. Using this scheme, the guest will only be able to communicate with the host and with other VMs on the local virtual network. To enable access to the outside world for the guest, we can use iptables to setup a traditional NAT gateway for the guest, which can also allow us to do arbitrary filtering of this traffic using nfqueue and iptables. To configure this, the following steps were performed:

1. Configure support for host-only networking for VMware. Set the networking type to be host-only in the VMware workstation settings for the relevant virtual machine.
2. Edit **/etc/network/options** on the host and set **'ip_forward'** to **'yes'**.
3. Add the script provided in Appendix B to **'/etc/network/if-up.d/'**. Make sure it's executable. This will setup appropriate iptables rules to make the host machine a NAT gateway for the guest VM(s).

4. Make sure that your guest VM is configured to use host-only networking on vmnet1 (vmnet1 is the default.)

5. Edit **'/etc/dhcp3/dhclient.conf'** to contain the following two lines:

   supersede domain-name-servers [dns1], [dns2];
   supersede routers [private network gateway];

   Where **'[dns1]'** and **'[dns2]'** should be replaced by your default DNS servers, and where **'[private network gateway]'** is the IP address of the virtual network interface on the host (vmnet1 by default.) Adding the **'routers'** line will cause the guest OS to contact the host as it's gateway instead of VMware's software gateway. Adding the **'domain-name-servers'** line will cause the guest OS to contact those name servers instead of VMware's software gateway.

6. To actually interpose on the traffic going through the VM, run the following as root on the host:

   iptables -I FORWARD 1 -j NFQUEUE

   This will prepend a rule in the FORWARD iptables chain to make all packets go through the netfilter queue 0. With this rule in place, no packets will be forwarded into or out of the VM unless they are received by a user-level program reading packets from queue 0, and reinjecting them with an ACCEPT verdict. The netfilter source package includes a sample program which does exactly this. We extend this program, similarly to what was previously described for unfsd, to communicate with the PDP program, which will make policy decisions on all network packets. Though this is fully functional, the data received at the raw IP level does not provide sufficient semantic information to allow for meaningful policy to be expressed. The details of this are discussed in the next section.

6 Future Work

Though this system is fully functional, having been used by the author for day-to-day use for two months, there is certainly room for further extension of the work described here.

6.1 Extract Better Semantics from Network Communication

As described previously, a system was developed to interpose upon all network communication performed into and out of the guest OS. This was done by routing all traffic at

the IP level into and out of the VM through a NAT system configured using Linux iptables. Though this is very effective at snooping network traffic and potentially performing policy decisions on the traffic, doing so is challenging.

Ideally we would be able to make policy decisions based on the highest level protocol used by the sandboxed program. For example, if we wished to sandbox a browser, we'd like to be able to create policy based on HTTP requests, or at the very least TCP connections. If we want to sandbox a P2P application, we would ideally be able to make policy decisions based on the protocol being used by that application.

Because the data received is received as raw network packets at the IP layer, the semantics of the higher level protocols are not immediately available for policy enforcement to the PDP. This severely limits the usefulness of the network traffic interposition system.

6.2 More Fine-grained Identification of Sandboxed Subjects

Though the goal of providing more fine-grained access control beyond merely the user has been achieved by this work, the access control mechanism is still not as fine-grained as it could be. Because the system is only able to execute policy on the communication into and out of the virtual machine, the subject granularity is at the level of the entire guest OS. Ideally we would be able to express policy based upon the individual program or process running within the guest OS.

6.3 Automatically Create Good Policy Files

Though the policy description language of this system is relatively straightforward to use, creating policy files which express the truly minimal set of privileges required for a program to function is slow and tedious. Ideally we would be able to somehow generate policy files which describe the minimal or close to minimal set of resources and privileges required for a program to operate.

One idea for this might be to implement some sort of "record mode" in the PDP which, rather than attempt to restrict access to resources, just records the requests to access them. Then, the program whose minimal policy we are attempting to generate could be put through a sample run manually monitored by the user for malicious behavior. Ideally, commonality in the accessed resources, such as the base path of many file accesses, could be detected and refactored to produce condensed policy files. But, even in the absence of this feature, such a system should be capable of providing a good start to producing a minimal policy file.

7 Conclusion

Today's operating systems provide generally good access control policies which have proven to be sufficiently flexible as to have been used, basically unmodified, for literally decades. However, the security policies which we are able to express using these systems have not changed to meet the current needs of today's highly exposed systems.

We have presented here a system for effectively sandboxing individual applications such that the damage which they can do to the host system is minimized. This was done while still providing sufficient access to the resources of the host system as allow the sandboxed applications to be usable by ordinary users. The use of common and proven technologies in VMware and XACML allow this work to build on top of technology which is becoming increasingly common.

Though current OS access control systems are very good at controlling user access to shared system resources, we believe that providing access control that is more fine-grained than this has the potential to greatly increase the security of our systems. In general, the user of a computer system should be able to run arbitrary programs which cannot perform all of the actions of the user, but rather are capable of performing only the most limited set of actions the program actually requires. The work presented here makes this a reality.

8 Acknowledgments

First and foremost I would like to thank my advisor Tom W. Doeppner, without whom this work would not have been possible. I would also like to thank my parents, Dr. Alice Twining and Dr. Donald Myers, for their lifelong support of my academic career. Finally, I would like to thank my ever-supportive girlfriend Sarah Filman, who was understanding enough to let me work on this project at whatever hours I felt like.

References

[1] Brent Callaghan. *NFS Illustrated*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2000.

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

[3] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, 2001.

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

[5] Symantec. Symantec global internet security threat report. 2008.

[6] Eric Tamura, Joel Weinberger, and Aaron Myers. "Operating Systems Protection Domains". 2007.

APPENDIX

A Firefox/Thunderbird XACML Policy

```
<Policy PolicyId="MozillaPolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
 <Target>
  <Subjects>
   <AnySubject/>
  </Subjects>
  <Resources>
   <AnyResource/>
  </Resources>
  <Actions>
   <AnyAction/>
  </Actions>
 </Target>
 <Rule RuleId="DotMozillaRule" Effect="Permit">
  <Target>
   <Subjects>
    <AnySubject/>
   </Subjects>
   <Resources>
    <Resource>
     <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:regexp-string-match">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">/home/atm/\.mozilla.*</AttributeValue>
      <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
     </ResourceMatch>
    </Resource>
```

```
    <Resource>
     <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:regexp-string-match">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">/home/atm/\.thunderbird.*</AttributeValue>
      <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
     </ResourceMatch>
    </Resource>
    <Resource>
     <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:regexp-string-match">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">/home/atm/\.mozilla-thunderbird.*</AttributeValue>
      <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
     </ResourceMatch>
    </Resource>
   </Resources>
   <Actions>
    <Action>
     <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
      <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
     </ActionMatch>
    </Action>
    <Action>
     <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">write</AttributeValue>
      <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
     </ActionMatch>
    </Action>
    <Action>
     <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">lookup</AttributeValue>
      <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
     </ActionMatch>
    </Action>
    <Action>
     <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">delete</AttributeValue>
      <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
     </ActionMatch>
    </Action>
    <Action>
     <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">execute</AttributeValue>
      <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
     </ActionMatch>
    </Action>
   </Actions>
  </Target>
 </Rule>
</Policy>
```

B Script to configure Linux iptables Rules to Perform NAT

```
#!/bin/bash

PATH=/usr/sbin:/sbin:/bin:/usr/bin

#
# delete all existing rules.
#
iptables -F
iptables -t nat -F
iptables -t mangle -F
iptables -X

# Always accept loopback traffic
iptables -A INPUT -i lo -j ACCEPT

# Allow established connections, and those not coming from the outside
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -m state --state NEW -i ! eth0 -j ACCEPT
iptables -A FORWARD -i eth0 -o vmnet1 -m state --state ESTABLISHED,RELATED -j ACCEPT

# Allow outgoing connections from the LAN side.
iptables -A FORWARD -i vmnet1 -o eth0 -j ACCEPT

# Masquerade.
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

# Don't forward from the outside to the inside.
iptables -A FORWARD -i eth0 -o eth0 -j REJECT
```