Sandboxing Applications

Vassilis Prevelakis
vp@prevelakis.net
Department of Computer and Information Science
University of Pennsylvania

Diomidis Spinellis
dds@aueb.gr
Department of Technology and Management
Athens University of Economics and Business

Abstract

Users frequently have to choose between functionality and security. When running popular Web browsers or email clients, they frequently find themselves turning off features such as JavaScript, only to switch them back on in order to view a certain site or read a particular message. Users of Unix (or similar) systems can construct a sandbox where such programs execute in a restricted environment. Creating such a sandbox is not trivial; one has to determine what files or services to place within the sandbox to facilitate the execution of the application. In this paper we describe a portable system that tracks the file requests made by applications creating an access log. The same system can then use the access log as a template to regulate file access requests made by sandboxed applications. We present an example of how this system was used to place Netscape Navigator in a sandbox.

1. Introduction

The sad truth is that whichever operating system one may be using, running large monolithic programs is a security risk. The original Unix philosophy of having simple dedicated tools that could be combined to carry out complex tasks is being abandoned. Instead, huge programs such as Netscape Communicator and the Star Office suite have been constructed by simply piling up one marginally useless feature after another. Nevertheless, users like fancy features and will use such programs despite our many philosophical and stylistic objections. Therefore, our rear guard action must be to alleviate the detrimental effects the use of such programs may have on the security posture of our system.

The concept of creating restricted environments for programs that are considered unsafe is by no means new. Unix itself offers many restrictions to what processes can and cannot do. Where additional security is required (e.g. by the `ftp` daemon) the `chroot(2)` system call is used to restrict access to a specific area of the filesystem. In this paper we examine how these access restrictions can be used to create a safe execution environment, and describe a tool that supports the construction and operation of such sandbox-type environments. We illustrate our approach by sandboxing Netscape Communicator and discuss the wider implications of the use of such mechanisms.

2. Access Restrictions

Let us consider the classic dilemma. A typical user receives a Microsoft Word file and would like to see what is inside but is afraid of what will happen to his workstation if the file contains a virus. One solution would be to place the program (Word) along with the suspect file in a controlled environment (which from now on we will refer to as sandbox) and open it there. If the file is infected, the effects of the virus will be localized, but not entirely eliminated as we shall see later on.

The sandbox must contain all the files needed for executing the application. Gathering a list of these files is a non trivial task. Applications depend in utilities such as `lpr` or `mail`, and on shared libraries, loadable modules, configuration files, and the operating system files required by various C library functions (IP service names, localized messages, time zone specifications, etc.). A key consideration is to make sure that the program does not escape from the sandbox, and that the sandbox system never assigns the program greater privileges than it would have if it ran outside the sandbox. Thus, the sandbox must disable access to setuid programs, or not allow them to be executed with permissions other then the ones

This work was supported by DARPA under contract F39502-99-1-0512-MOD P0001.

given to the user running the application. Moreover, to prevent the application from gathering information about the system by accessing files such as `/etc/password` and `/etc/hosts` we need to substitute them with “sanitized” versions that contain just the information that the application requires to perform its tasks.

More sophisticated systems like Janus [1] support centralized policies with fine grained control over the resources that the process may use (e.g. file descriptors, memory, file system space etc.). Our approach dynamically evaluates the requirements of a given program, creating a sandbox specification that will not affect its operation. In doing so we strive to balance what we would like to control against the effort in specifying these restrictions.

3. Approach Overview

Our target audience is the typical Linux or *BSD user, i.e. people with their own PC where they have root access, but are not Unix gurus or security specialists. This orientation has strongly influenced our approach. Our sandbox is based on system-provided services such as `chroot` and `mount.` Access to these system calls has important implications regarding system security. However, since our user already has root access we need not worry about the user abusing these calls and can concentrate on automating the generation of secure sandboxes.

Based on this premise, our approach for sandboxing applications is based on the following steps:

1. Run the application with known benign input to create access logs specifying its file access behavior.
2. Use the logs to create an access list that can be morphed into a typical `chroot` environment (like the one used by `ftpd`). The user can augment this list based on the particular requirements of the application.
3. Create the sandbox as a `chroot` environment based on the access list.
4. Run the application with untrusted input in the sandboxed environment thus denying it access to unauthorized files.

Nowadays, operating system releases every few months are commonplace. This places an enormous burden on the developers of software that requires “special” access to the operating system (e.g. kernel modules). While external kernel interfaces (system calls, `ioctls` etc.) evolve slowly, internal kernel data structures and interfaces are more volatile. Furthermore, subtle but annoying differences between the various systems make the support of kernel based programs a full time job. We, therefore, decided to stay in user land and invest our resources (time) on making the system portable and flexible.

4. The FMAC Tool

To support the approach we outlined, we designed the File Monitoring and Access Control (FMAC) tool. The tool implements a filesystem that mirrors the system’s existing file structure. With the FMAC filesystem mounted on the workstation, applications are run with a `chroot` operation that limits their access to the FMAC managed filesystem. The FMAC tool supports two modes of operation:

passive whereby FMAC logs file requests while allowing them to go through, and

active when FMAC only honors file access requests that are authorized by a user-specified access list.

Initially applications are run with FMAC in passive mode to create the access log (step 1 of our approach). After the sandbox specification has been created (step 2) FMAC is run in active mode (step 3) and applications can be executed with untrusted input in the FMAC `chroot` environment (step 4).

We implemented two versions of the FMAC tool: one based on a user-level NFS server and one on a Perl filesystem [2]. The user-level NFS server is efficient and highly portable. It is available on most *BSD and Linux distributions. The Perl filesystem trades runtime efficiency for flexibility. It runs without any modifications on any platform supporting Perl filesystems (currently Linux on Alpha and Intel CPUs). In the following paragraphs we describe the two FMAC implementations.

4.1 FMAC as an NFS Server

The FMAC NFS server runs as a user-level process without the need for NFS server support to be present in the kernel. The FMAC filesystem is mounted by the standard NFS client, so the system must be able to mount filesystems using the NFS protocols.

The FMAC tool uses a different port from the well-known NFS port, so that it can coexist with a standard NFS server. All requests from the FMAC filesystem are processed by the tool which performs a lookup in the access list in order to determine its response to the request. If the filename is found in the access list, then the permissions reported by the FMAC server are constructed by performing a logical AND operation between the permissions in the access list and the those in the underlying filesystem. If the filename is not present in the access list, the FMAC server reports that the file does not exist.

Normally NFS does not allow requests to cross filesystems (i.e. if you export two filesystems / and `/usr`, a client mounting only `/` will not able to access files on `/usr` unless this filesystem is mounted as well). The reason for this limitation is that inodes/vnodes are guaranteed to be unique only within a single filesystem.

In our system we want to be able to view the entire local file hierarchy as a single filesystem so that there is no need for multiple mount points within the `chroote`d environment.

We, therefore, modified the NFS server to allocate file handles dynamically and maintain a lookup table. The implication is that the NFS server is no longer stateless. This is contrary to the NFS philosophy of delegating state to the client, but in our case we felt that our decision was acceptable because:

- This is not a general-purpose NFS server but an application that is intended to run on the same machine as the sandboxed application. If the machine crashes so will the client.
- This only affects the passive mode when we have to construct the file access list in memory. In the active mode the access list is retrieved from a file. In this case the FMAC server may be restarted without disturbing the client.

4.2 FMAC as a Perl Filesystem

The Perl filesystem (PerlFS) is a combination of a Linux kernel module and a Perl extension that make it possible to write filesystem implementations in Perl instead of C. Perl filesystems are object classes conforming to the PerlFS interface. Compared to typical filesystem implementations written in C, Perl filesystems are less dependent on the underlying operating system implementation. The PerlFS interface abstraction isolates the filesystem implementation from operating system changes; the same code will run on all systems supporting PerlFS.

Like the NFS-server implementation, the Perl filesystem allows multiple partitions to be mounted under the same directory hierarchy and dynamically allocates unique inode numbers for existing files. Two Perl associative arrays are conveniently used to map inode numbers to file names and vice versa. To avoid name aliasing problems created by hard links across files, and the multiple ways a directory hierarchy can be traversed to reach a file, the native filesystem `stat(2)` system call is used to obtain the original unique device/inode number pair as a basis for creating the dynamic inode number.

The high-level interface of the Perl filesystem—trading runtime efficiency for flexibility—allowed us to implement the FMAC tool in less than 2000 lines of Perl code. We utilized Perl's excellent support for regular expressions to experiment with different ways to specify the file access request list. The author of the Perl filesystem module advertises it as an alpha version. However, after some recent improvements that added support for the `mmap` functionality needed to load executable files and shared libraries, we were able to run a number of programs in the `chroote`d environment without a problem.

5. Example

Netscape Communicator is a unified web browser and email client (among other things). Its use creates enormous security and privacy concerns since it has full access to the files of the user. Moreover, Netscape Communicator maintains files such as `bookmarks.html` and `cookies` that may provide hints about the browsing habits of the user. Our objective is to be able to run this program in a stripped down environment where:

- It will have no access to the user files.
- It will have access only to special sanitized versions of the system files.
- The user will be able to create temporary "new" installations of the Netscape user directories for accessing suspicious sites.
- The program will be able to function as a browser and a `pop`/`imap` client.

5.1 Methodology

We ran Netscape Communicator version 4.75 with the FMAC system in passive mode and we extracted, using FMAC in passive mode, the file hierarchy shown in Figure 1.

We separated the files into two categories, *system* files that are common to all installations (e.g. the `/netscape` hierarchy) and *user* files that are personal files accessible by the user. The former category is typically read-only and consists of files that are not owned by the user. These are the files that will be handled by the access control part of FMAC as we will see later on.

```
/bin/
      /sh
  /dev/
      ...
  /etc/
      /group
      /kerberosIV
              /krb.conf
      /localtime
      /login.conf
      /networks
      /pwd.db
      /resolv.conf
      /spwd.db
  /usr/
      /X11R6
          /bin
          /lib
              /X11
                  /XKeysymDB
                  /app-defaults
                  /locale
                      ...
      /bin
          /su
      /lib
           ...
      /libexec
          /ld.so
      /local
          /netscape
                  /netscape
      /share
          /zoneinfo
                ...
  /users
      /bob
          ...
  /var/
      /X11
          /app-defaults
      /db
          /kvm_bsd.db
      /mail
              /bob
      /run
          /ld.so.hints
```

Figure 1: Files required by the Netscape Communicator sandbox

For most files in the system category we simply need to ensure that access is read only. However a few of them deserve special attention. For example the `/etc/passwd`, `/etc/master.passwd` (or the hashed versions of them, `pwd.db` and `spwd.db`) and `/etc/group` should not be accessed directly since they probably contain information that we would like to keep out of the sandbox. Such information may include user names, group assignments and most importantly passwords. Another consideration is that there should be no setuid program in the sandbox.

The consequence of the above is that the list of files produced by the analysis phase must be examined to determine which files need to be replaced by sanitized versions.

Once the sandbox is built, we can execute the program using a command like:

```
chroot /users/bob/sandbox \
   /bin/su bob -c \
   /usr/local/netscape/netscape
```

Notice that we use the `su` program in the sandbox to reduce the process privileges to those of user `bob`. Since the suid bits are ignored within the sandbox, processes within the sandbox cannot use `su` to become root.

5.2 Sandboxing using only `chroot`

If we do not wish to use the FMAC tool for the active phase, we will need to create an actual file hierarchy by copying all the files included in Figure 1 to another part of the filesystem. We would then run the `chroot` command as in the earlier example.

This approach relies only on the standard system tools for the active phase. It also allows the sandbox environment to be moved to other similar platforms with little or no customization. Thus, in a company environment we can create the Netscape sandbox in one machine and then copy it to the machines of all the other employees.

However, copying all these files is wasteful and more importantly, we will need to keep track of all the duplicates and update them every time we upgrade the application, or the operating system. Employing links from within the sandbox to the actual files is quite difficult. Symbolic links cannot be used, while hard links require that the linked files are in the same filesystem and may only be used to link files, not directories. Also the extensive use of hard links, may lead to confusion.

5.3 Sandboxing using FMAC

The FMAC system creates a virtual filesystem by transparently providing access to the real files while at the time enforcing access controls on the basis of a user supplied access list.

To use FMAC we have to start the modified NFS server or the Perl filesystem module and mount the FMAC filesystem, inside the sandbox before closing the sandbox.

To facilitate the configuration of the FMAC system, the list of accessed file produced during the passive phase is the same has the same format as the ACL file used in the active phase.

The ACL contains one line for each file or directory. Its format is as follows:

<permissions> < path> [<actual path>]

permissions is a string that denotes the access permissions allowed for the file.

path is the path to the file requested by the sandboxed application,

actual path is an optional argument that allows us to provide a substitute for the requested file. Only files may be substituted.

For example Figure 2 contains an extract from the Netscape ACL:

```
__X_    /usr/lib
R___    /var/run/ld.so.hints
__X_    /var/run
__X_    /var
R_X_    /usr/libexec/ld.so
__X_    /usr/libexec
R_X_    /usr/bin/su
__X_    /usr/bin
__X_    /usr
R___    /etc/pwd.db  /var/tmp/pwd.db
R___    /etc/spwd.db /var/tmp/spwd.db
        . . .
```

Figure 2: Structure of Communicator ACL file.

The permissions apply only to the user (i.e. the traditional Unix "group" and "others" permissions are gone) because only the user will be accessing the files from within the sandbox. The permissions that may be specified are, `read`, `write`, `execute` (access for directories) and `create` (that allows the named file to be created if it does not exist).

Moreover, the sandbox does not grant additional privileges to the user – the sandbox permissions are *in addition* to the existing Unix permissions that apply to the files.

In the last two lines in Figure 2, we provide substitutes for the password database files (`pwd.db` and `spwd.db`).

5.4 Constructing the ACL

When running the application in passive mode we need to exercise as much of its functionality as we can. For example, not accessing the help files will mean that they will be left out of the ACL.

Clearly, we can manually edit the ACL afterwards and correct some obvious omissions, but its is usually easier to have the FMAC system prepare it for us.

A more serious question is how to treat directories that are going to have extensive file creation, such as the Web page cache directories of the Communicator. We did not want to clutter the ACL with definitions for these files, so we decided to provide some directories inside the `chroote`d environment that are not controlled by the FMAC system. We created the hierarchy shown in Figure 3

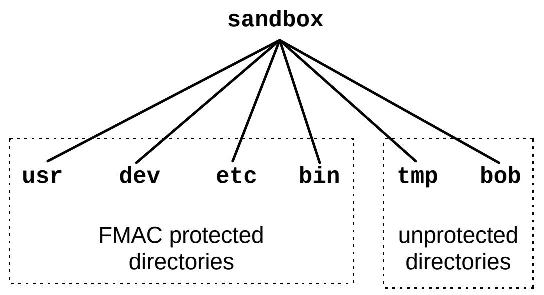

Figure 3: Layout of sandbox allows for both protected and unprotected hierarchies.

The directory containing the user files (`/bob`) is not the real home directory of the user; it is an empty directory belonging to the user, where Netscape will create its state files (`.netscape` and `nsfiles`).

We could easily have retained the real path to user's home directory (e.g. `/users/bob`) so that the entry for the user in the `/etc/pwd.db` did not have to be changed. In that case, directory `/bob` would have to be changed to `/users/bob`. However, since we are replacing the `/etc/pwd.db` file anyway, we can change the user information in the copy and make the sandbox layout a bit simpler.

For more complex file access scenarios, it may be easier to be able to specify unprotected file hierarchies within the ACL file. This is still an open question which we plan to investigate as we gather more experience through the actual use of the FMAC system.

When using file substitutions it is often desirable to be able to use the replacement files in the passive phase as well. In our earlier example, where we changed Bob's home directory, we wouldn't really be able to get sensible data out of the FMAC tool, unless the changed `/etc/pwd.db` was used.

We solve this problem by allowing the user to specify a template ACL file when running the FMAC tool in passive mode. This template file would contain file substitutions like the ones mentioned above. The contents of the template file are included in the ACL produced by FMAC, so it is not necessary to include the template file when running FMAC in active mode.

6. Discussion

Access control, like other security related problems has no clear-cut solution; rather, it requires a compromise that balances costs against perceived risk.

The costs include execution overheads, memory requirements, application configuration, and, most importantly, the definition of the capabilities that the restricted process is allowed to have. For example, setting up a Windows 2000 system under VMware in order to read files generated by Microsoft Word requires an investment in time and expertise that may not be justified by the end result.

The FMAC tools attempt to strike an acceptable balance by providing:

Ease of configuration. Allowing the FMAC tool to learn from the trial runs of the application, reduces the initial work required for constructing the sandbox. Anybody who has manually configured an anonymous ftp server that uses the `chroot` facility can testify how difficult it is to determine the files and the permissions required for the its correct operation.

Portability. By being a user level program, FMAC has minimal installation overhead and is largely independent of the release or type of the operating system.

Security. The access restrictions imposed by the FMAC tool are in addition to the restrictions that are placed by the underlying file system. Thus, using the FMAC tool will not degrade the security posture of the system.

In the next paragraphs we will discuss some of the problems and caveats that are associated with the use of the FMAC tools.

The `chroot` and `mount` calls require superuser access. It is, therefore, imperative that we lower the capabilities of the process immediately after the two calls complete and before `exec`ing the application. We must also make sure that no hooks exist in the sandbox that may allow the boxed application to escape. For example, although a shell must be present within the sandbox, we can use a restricted shell that provides only the bare essentials for the execution of the program. Users may not even have access to this program as they will be talking to the application. Moreover, files served via the FMAC system always have their `suid` and `sgid` bits cleared.

Particular care must be paid when running the application with the FMAC system in passive mode. During this phase, the application runs with all the privileges enjoyed by the user. Hostile activity on the part of the application will not be detected and may affect the security posture of the application when it runs with the FMAC in active mode.

Creating an access list file which is as complete as possible is also important because it reduces the need to go back and update the access list file later on. The user should try to use all the features of the application that are likely to be required in the future. For example, one feature that is seldom used during the active phase is the on-line help system. It is often required during the operational lifetime of the system, so it must be exercised during the passive phase so that the access list allows access to the help files.

The primary objective of the FMAC system is to prevent a user-level application such as Netscape Communicator from performing tasks that adversely affect the security of the user running the program, or the security of the machine hosting the application. Confining the program to a sandbox significantly reduces the possibility of undesired side effects. However, it is not a guarantee. A large and complicated program such as Netscape Communicator interacts with the system in many ways and it depends on a large number of system resources for its correct execution. For example when we follow a URL leading to the PDF file, the browser will automatically launch the Acrobat reader to display the page. Postscript files, streaming audio, video, etc. all require their own special helper applications. Placing inside the sandbox all the programs and devices that the helper programs need for their correct operation, will essentially negate the use of the sandbox.

On the other hand implementing workarounds for performing all these special tasks in a secure way, involves disproportionate amounts of work. For example, the simple act of sending a Web page to the printer involves the execution of `lpr` which is setuid root. The FMAC system will not allow `lpr` to run as root and the operation will fail. This may be overcome by depositing files in a “spool” directory and having a daemon running outside the sandbox send them to the printer. Clearly this will appeal to few people and even if deemed adequate, implementing it will add to the overhead of configuring the sandbox.

Even if everything is configured properly, private information may still leak. In the Communicator example, email messages will need to be stored inside the sandbox so that the email application can access them. Downloaded files, cookies and bookmarks may also contain private information. All these can be accessed by malicious code that manages to subvert the application.

One workaround is to have automated scripts regularly move files out of the sandbox and clean up files that may contain private information.

Another category of malicious behavior that cannot be trapped is that which exploits the application capabilities in a manner that is roughly consistent with the intended use of the application. One example is the Melissa virus which sent copies of itself to email addresses contained in the user's address book.

Given the above, it is evident that FMAC is not a panacea. Rather it is yet another mechanism that can protect the user under certain circumstances. In particular, the ability of FMAC to rapidly create a disposable environment in which to run a potentially nasty applet, or contact a suspicious site, makes it an extremely useful tool.

7. Related work

Over the years there have been numerous proposals for systems that impose discretionary access controls on programs. These systems can be roughly placed in three categories, in increasing order of complexity for the execution environment.

Systems that trust programs. Programs that have been vetted are considered trusted, while the rest are given only limited access. Examples include Microsoft Active X controls and the system presented by Lai et al [3]. These systems assume that all bugs or vulnerabilities can be detected before deployment. This assumption has been demonstrated time and time again as utopian. In [3] only thirty-two programs from the entire BSD 4.3 were considered trusted. Ironically, one of them was `/etc/fingerd`, a daemon later used by the Morris Internet Worm to break into systems.

Regulate file access. File access is considered a key capability in most systems since it involves the long term memory of the system. Unauthorized modifications to files can threaten the integrity of the system itself or the data that is stored in it. Even read-only access can be used to leak information to hostile parties. Controlling file accesses is, therefore, appealing both because it has significant impact and because file systems are often self contained OS subsystems that can be controlled with minimal modifications to the core operating system. Numerous systems are included in this category. The system described in [4] bases decisions on the file types (via the filename extension), while in the Exokernel [5], a credential-based system is used to specify file access. Credentials are used to determine which parts of the file hierarchy are accessible by an application. The system, however, is rather limited by the fact that permissions are hardwired into the system, the hierarchical capability tree may be up to eight levels deep, and the access-list based control mechanism is inflexible. Wichers et at [6] looked at the problem the other way round by attaching to files lists of programs that could access them.

FMAC relies on a custom filesystem providing the learning and `chrooted` access functionality. A number of projects have provided ways to create such filesystems, see [7] and the references therein.

Full access control. All requests made by the application are passed through a discretionary access control mechanism that enforces policy. The checks may be at the operating system call level as in Janus [1] and SubOS [8], or at the library call level [9]. Virtual machines such as the Java VM and VMware also provide a restricted environment in which programs may operate. Errant applications should only be able to cause damage to the virtual machine leaving the real system intact.

Recent versions of FreeBSD include the `jail` system call which is a more powerful version of the `chroot` facility that has been mentioned earlier. Like `chroot`, `jail` restricts the controlled process to a subset of the filesystem, but it also prevents the process and its children from issuing privileged requests such as creating device special nodes. The `jail` facility imposes a fixed access policy that cannot be altered without changing the implementation. Like a leash with a fixed collar its effective use is limited by the lack of flexibility.

Mobile code systems have to face many similar problems because they have to accommodate applications that are imported from the outside and hence are potentially hostile. The SANE architecture [10] includes a credential-based capability mechanism, while others [11, 12, 13] propose languages that define acceptable policies for mobile code.

Advisories from CERT and postings on security related forums provide ample evidence that many of the above systems fail to provide foolproof security. The guardians themselves often have flaws that may allow applications to escape from the sandbox and compromise system security.

8. Current Status and Future Directions

Both versions of the FMAC tools are currently fully operational. In addition to Netscape Communicator, the FMAC tools have been used to sandbox the Adobe Acrobat PDF reader and the `ghostview` application. We are currently investigating a mechanism whereby applications may negotiate an acceptable set of permissions with the FMAC system before executing, thus dispensing with the need to run the application in passive mode to construct the access list. Moreover, in the current system, files may not be added to the access list after the sandbox is running. We plan to investigate the possibility of asking the user for permission when trying to access a file that is not in the access list.

We intend to continue work on the system aiming at creating a fully fledged discretionary access control system for files. Moreover, we are currently investigating ways of controlling access to the network by dynamically creating special rules for the packet filtering facility in the kernel.

Acknowledgments

We would like to acknowledge the invaluable help of Sotiris Ioannidis and the other members of the DSL Lab at the University of Pennsylvania. Special thanks are also due to Ted Faber for his careful reading of the draft and his many right-to-the-point comments.

References

[1] Goldberg, Ian, David Wagner, Randi Thomas and Eric A. Brewer, "A Secure Environment for Untrusted Helper Applications," 1996 USENIX Security Symposium.

[2] Calvelli, Claudio, "The Perl Filesystem", http://dd-sh.assurdo.com/perlfs, April 2001.

[3] Lai, N. and T.E. Gray, "Strengthening discretionary access controls to inhibit Trojan horses and computer viruses," Proceedings of the 1988 USENIX Summer Symposium, pages 275-286, June 1988.

[4] Karger, P.A., "Limiting the damage potential of discretionary Trojan Horses," Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy," pages 32-337, April 1987.

[5] David Mazieres and M. Frans Kasshoek, "Secure Applications Need Flexible Operating Systems," *Proceedings of the 6th Workshop on Hot Topics in Operating Systems,* May 1997

[6] Wichers, D., D. Cook, R. Olsson, J. Cossley, P. Kerchen and R. Lo, "PACLSs: An Access Control List Approach to Anti-Viral Security," Proceedings of the USENIX SECURITY II Workshop, pages 71-82, 1990.

[7] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. *USENIX Annual Technical Conference,* Anaheim, California, January 1997.

[8] Ioannidis, Sotiris, Angelos D. Keromytis, Steve Bellovin and Jonathan M.Smith, "Implementing a Distributed Firewall," 7th ACM Conference on Computer Communications Security, November 2000.

[9] Ko, Calvin, George Fink and Karl Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," Proceedings of the 10th Annual Computer Security Applications Conference, Orlando, FL, 1994

[10] Alexander, D. Scott, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith, "*A Secure Active Network Architecture: Realization in SwitchWare*". IEEE Network Special Issue on Active and Controllable Networks, vol. 12 no. 3, pp. 37-45.

[11] Edjlali, Guy, Anurag Acharya, and Vipin Chaudley, "History- Based Access for Mobile Code," In the Proceedings of the 5th ACM conference on Computer and Communication Security (CCS'98). 1998

[12] Jaeger, T., A. Prakash, and A. Rubin, "*Building Systems that Flexibly Control Downloaded Executable Context,*" Proceedings of the 6th USENIX Security Symposium, 1996.

[13] Jajodia, S., P. Samarati, V. Subrahmanian, and E. Bertino, "A Unified Framework for Enforcing Multiple Access Control Policies," Proceedings of the ACM SIGMOD International Conference on the Management of Data, pages 474-485, 1997.