

USENIX Association

Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25-30, 2001

USENIX

© 2001 by The USENIX Association. All Rights Reserved. For more information about the USENIX Association:
Phone: 1-510-520-5049 FAX: 1-510-520-5776 Email: office@usenix.org WWW: <http://www.usenix.org>
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Sandboxing Applications

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

Dionidis Spinellis
ds@ath.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 near 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 `rlp` 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

This work was supported by DARPA under contract F3962-98-1-0512-MOD P001.

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 use 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 on utilities such as `ls` 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 system programs, or not allow them to be executed with permissions other than the ones

given to the user running the application. Moreover, to prevent the application from gathering information about the system by accessing files such as `/etc/passwd` and `/etc/shells` 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 Jams [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.

More sophisticated systems like Jams [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.

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 `rlp`). 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 unsanitized 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, `ioctl` 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 mimics 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 or 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 protocol.

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 whether to allow 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 be able to access files on `/usr` unless this filesystem is mounted as well). The reason for this limitation is that inodes/volumes 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 `chrooted` 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 where 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 primitive arrays are conventionally 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 advises it as an alpha version. However, after some recent improvements that added support for the `swap` functionality needed to load executable files and shared libraries, we were able to run a number of programs in the `chrooted` 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 `bookmark.html` and `cookies.txt` 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 `popup` snap client.

5.1 Methodology

We ran Netscape Communicator version 4.75 with the methodology we used to create the sandbox by 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 `/usr/include` hierarchy) and user files that are personal files accessible by the user. The former

Google Exhibit 1139

Google v. VirtaMove

Oracle Ex. 1139

Oracle v. VirtaMove, IPR2025-00964