Proceedings of the 9th USENIX Security Symposium

Denver, Colorado, USA
August 14–17, 2000

MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications

Anurag Acharya, Mandar Raje
Dept. of Computer Science, University of California, Santa Barbara

Abstract

Designing a suitable confinement mechanism to confine untrusted applications is challenging as such a mechanism needs to satisfy conflicting requirements. The main trade-off is between ease of use and flexibility. In this paper, we present the design, implementation and evaluation of *MAPbox*, a confinement mechanism that retains the ease of use of application-class-specific sandboxes such as the *Java* applet sandbox and the *Janus* document viewer sandbox while providing significantly more flexibility. The key idea is to group application behaviors into classes based on their expected functionality and the resources required to achieve that functionality. Classification of application behavior provides a set of labels (e.g., `compiler`, `reader`, `netclient`) that can be used to concisely communicate the *expected functionality* of programs between the provider and the users. This is similar to *MIME-types* which are widely used to concisely describe the *expected format* of data files. An end-user lists the set of application behaviors she is willing to allow in a file. With each label, she associates a sandbox that limits access to the set of resources needed to achieve the corresponding behavior. When an untrusted application is to be run, this file is consulted. If the label (or the MAP-type) associated with the application is not found in this file, it is not allowed to run. Else, the MAP-type is used to automatically locate and instantiate the appropriate sandbox. We believe that this may be an acceptable level of user interaction since a similar technique (i.e., MIME-types) has been fairly successful for handling documents with different formats. In this paper, we present a set of application behavior classes that we have identified based on a study of a diverse suite of applications that includes CGI scripts, programs downloaded from well-known web repositories and applications from the Solaris 5.6 distribution. We describe the implementation and usage of MAPbox. We evaluate MAPbox from two different perspectives: its effectiveness (how well it is able to confine a suite of untrusted applications) and efficiency (what is the overhead introduced). Finally, we describe our experience with MAPbox and discuss potential limitations of this approach.

1 Introduction

Designing a suitable mechanism to confine untrusted applications is a challenging task as such a mechanism needs to satisfy conflicting requirements. The key trade-off is between ease of use and flexibility. To be easy to use, a confinement mechanism should require little or no user input. As a result, such a mechanism is likely to provide one-size-fits-all functionality – that is, all applications being confined are allowed to access exactly the same set of resources. This limits the class of applications that can be used effectively while being confined. To be more flexible, a confinement mechanism has to either allow access to all resources to all applications (which defeats the purpose of confinement) or it has to somehow select the set of resources each application is allowed to access. To be able to select the set of resources that each application is allowed to access, such a mechanism needs *some* knowledge of the application's resource requirements as well as the user's intent.

Previous research into creating confinement environments (also referred to as *sandboxes*) has taken one of four approaches which make different trade-offs between flexibility and ease of use. Several researchers have proposed some form of per-program access control [4, 5, 7, 11, 12, 14, 21]. This approach is highly flexible but requires users (or administrators) to specify access-control information for every program. It can work well if the number of untrusted applications is small and changes infrequently. Several computing environments, however, are dynamic and contain a large number of applications (e.g., a web-hosting service which allows its users to run CGI scripts). The second approach uses finite-state machine descriptions of program behavior [13, 15, 17]. This provides even more flexibility as different sequences of the same set of accesses can be distinguished. To be used effectively, however, this approach requires a careful understanding of the behavior of individual applications. Given the size, complexity and the number of applications in modern computing environments, it would be hard to develop such detailed descriptions.

The third approach considers each application provider (author/company/web site) as a principal and uses per-provider access-control lists (ACLs) [9, 10, 20]. This groups applications from the same provider into the same sandbox. This is a promising approach since a user needs to deal with potentially fewer principals than the first two approaches. This makes it easier for the users to create and maintain the corresponding ACLs. However, disparate applications from the same provider may be grouped into the same sandbox. To allow *all* of these applications to run, a user may have to provide an overly coarse sandbox – which may or may not be desirable. Another potential problem is that the number of potential providers is large and growing. Creating and maintaining ACLs for a large number of providers can require substantial administrative effort.

The fourth approach consists of special-purpose sandboxes for specific classes of applications, e.g, document viewers [8], applets [6], global computing [3], CGI scripts [18] and programs that run with root privileges [19]. By limiting the scope of the confinement mechanism, these techniques significantly reduce the administration effort required. While each of these sandboxes are easy to use when they are applicable, they are limited in their applicability. For each application, one needs to *manually* find, deploy and instantiate the appropriate sandbox. In addition to being an administrative burden, using a variety of programs for sandboxing makes it harder to check the sandboxes themselves for security flaws.

In this paper, we present the design, implementation and evaluation of *MAPbox*, a confinement mechanism that retains the ease of use of application-class-specific sandboxes while providing significantly more flexibility. The key idea is to group application behaviors into classes based on the expected functionality and the resources required to achieve that functionality. Examples of behavior classes include filters, compilers, editors, browsers, document viewers, network clients, servers etc. Classification of the behavior of an application provides a label (the name of its behavior class) which can be used by its provider to concisely describe its *expected functionality* to its users. This is similar to MIME-types which are widely used to concisely describe the *expected format* of files. We refer to the label assigned to an application as its *Multi-purpose Application Profile*-type (or *MAP-type*). An end-user specifies the set of application behaviors she is willing to allow as a set of MAP-types listed in a `.mapcap` file. With each MAP-type, she associates a suitable sandbox. When an untrusted application is to be run, this file is consulted. If the MAP-type associated with the application is not present in the `.mapcap` file, the application is not allowed to run. Else, the MAP-type is used to automatically locate and instantiate the appropriate sandbox without requiring user intervention. We believe that this may be an acceptable level of user interaction since a similar technique has been fairly successful for handling documents with different formats. For MIME-types, end-users specify, in a `.mailcap` file, which MIME-types they are willing to view, which application is to be used to view MIME-type and how should this application be invoked.

In effect, MAPbox allows the *provider* of a program to *promise* a particular behavior and allows the *user* of a program to confine it to the resources *she* believes are sufficient for that behavior. For CGI scripts provided by users of a web-hosting service, the MAP-type for the script can be specified by the user when it is submitted for installation. For plug-ins and other applications that are downloaded on demand, the MAP-type can be specified in the HTTP header (just as MIME-types are specified for downloaded documents). For applications downloaded and built locally, the MAP-type can be specified by the provider (e.g., in a README file). Note that the provider of a program only specifies the MAP-type for the program, she *does not* specify the sandbox to be used. The association between MAP-types and sandboxes is completely under the control of the user of the program (being specified in the `.mapcap` file in the user's home directory).

This proposal raises several questions. First, can application behaviors be suitably classified? That is, do application behaviors and the corresponding resource requirements fall into distinct categories? Second, how does MAPbox deal with a group of applications that exhibit similar behavior but need different resources? For example, hotjava and trn are both browsers that connect to remote servers. However, they differ in the hosts they connect to, the port they connect to and the directory they use to store the downloaded information. Third, how are the individual sandboxes used by MAPbox to be implemented? There are conflicting constraints – on one hand, all accesses must be checked; on the other hand, the overhead should be acceptable. Finally, how well does this approach work in practice?

In section 2, we describe a study of the behavior and resource requirements of fifty applications. These applications were drawn from different sources: CGI scripts downloaded from a well-known CGI repository; programs downloaded from well-known program repositories; and applications provided as part of the Solaris 5.6 environment. Based on this study, we have defined a set of behavior classes and the corresponding sandboxes. In section 3, we present the design and implementation of MAPbox. Our implementation of MAPbox runs on Solaris 5.6 and confines native binaries. It also provides a sandbox description language that can be used to construct new sandboxes with relative ease. In section 4, we describe how MAPbox can be configured and used. In section 5, we present an evaluation of MAPbox. We evaluated both its effectiveness (how well it is able to confine a suite of untrusted applications) and efficiency (what is the overhead introduced). Our results indicate that the overhead of confinement is small enough ($< 5\%$ for CGI scripts, 1-33% for other applications) to be acceptable for many applications and environments. We found that a MAP-type-based approach is quite effective for confining untrusted applications. Of the 100 applications in our evaluation suite, only nine failed to complete their test workloads; of these five failed because they made inherently unsafe requests. We also found that mislabeled applications (i.e., applications that were labeled with a different MAP-type than their own) were not able to gain access to resources that the user did not wish to grant. We conclude with a discussion of our experience with MAPbox and the potential limitations of this approach.

2 Identifying Behavior Classes

To identify application behavior classes, we studied a suite of fifty applications. Of these, twenty were Perl-based CGI scripts that we downloaded from a well-known repository; another fifteen were programs downloaded from various well-known repositories; and the final fifteen were applications provided as part of the Solaris 5.6 distribution.[1] We ran each application on a Solaris 5.6 platform with several workloads. For each execution, we obtained a trace of the system-calls made by the application. To collect the system-call traces, we used the truss utility. For each system-call, it prints the name, arguments and the return value. As far as possible, we summarized these traces by identifying groups of system-calls and relating them to higher-level operations such as: accessing files, linking libraries, making/accepting network connections, creating child processes, accessing the display, handling signals etc. Figure 1 presents one such group. For other examples, please see [16]. In some cases, to verify the mapping between a higher-level operation and the system-calls it generates, we wrote short programs performing the operation and compared their traces with that of the application being studied.

To design the workloads for our study, we considered two alternative techniques. The first technique starts with an intuitive notion of application behavior classes such as editors, document viewers, compilers, mailers, etc. For each class, it defines a synthetic workload that exercises the *primary* behavior of the class. The second technique develops trace-based workloads by having a set of users to use individual applications and keeping track of user operations for relatively long sessions. Trace-based workloads have the advantage of being more realistic. However, many applications can exhibit multiple behaviors (e.g., gnu-emacs can be used as an editor, a news-reader, a mailer etc). Since our goal in this study was to identify the set of resources needed for the individual behaviors, we chose to use synthetic workloads instead of trace-based workloads. For example, for editors, we used the following workload: (1) start up with no file and exit; (2) start up with an existing file and exit; (3) start up with an existing file, delete 100 characters, add 100 characters and exit; (4) for text editors, edit a file, spell-

[1] The CGI scripts were ads, AtDot-2.0.1, authentication, banner, bbs, bookofguests, bp, browsermatcher, bsmidi, calendar, chat, counter, CrosswordMaker, DB_Manager, DB_Search, dcguest2, formmail, form_processor, guestbook, and juke. All of these are linked off cgi.resource-index.com, a well-known CGI repository. The downloaded programs were idraw, xfig, ghostview, xv, gcc, pico, pine, elm, lynx, agrep, xcalc, ical, xdvi, gzip, httpd. The Solaris applications were vi, pageview, imagetool, dvips, mailtool, trn, Netscape, hotjava, sh, ftp, finger, rwho, whois, telnet and sed.

check it and exit; (5) for graphical editors, generate a postscript file and exit. Workloads used for most of the other classes are described in [16].

Based on the results of this study, we identified a set of behavior classes and their resource requirements. We first determined the resources needed by each application. By resources, we mean files, directories, network connections (hosts and ports), the X server, other devices, ability to create new processes, environment variables etc. For each behavior class, we identified resources commonly required to implement the *primary* functionality of the applications in the class. Some applications make use of resources that are not really needed for implementing their primary functionality. For example the Solaris C compiler opens a socket to a license server to check licensing information. Other C compilers (e.g., `gcc`) don't need to make network connections. Based on the Principle of Least Privilege, we do not consider such resources as *requirements* for the corresponding behavior classes. Note that the resource requirements for a class are not simply the union of the resource used by a set of applications that we studied. Instead, they are the set of resources that we believe are *required* to implement the expected functionality for the class. In Section 5, we compare these *expected resource requirements* associated with a behavior class with the *actual resource requirements* of a large suite of applications that implement that behavior.

In addition, we identified a set of *parameters* for each class. Parameters of a class capture common patterns in the idiosyncratic resource requirements of the applications belonging to the class. For example, `hotjava` and `trn` are both browsers that connect to remote servers, download files and present them to users. For this, they need to link in networking libraries, make network connections, open networking-related device files (e.g., `/dev/{tcp,udp,ticotsord}`) and write files in a local directory. However, they differ in the hosts they connect to, the port they connect to and the directory they use to store the downloaded information. In this case, the hosts to connect to, the port to connect to and the directory to store the information would be parameters of the behavior class containing `hotjava` and `trn`.

Table 1 presents the behavior classes we identified and their parameters. We do not claim that the classification presented in Table 1 is either unique or complete. Our goal in identifying these classes was to demonstrate that application behaviors and the corresponding resource requirements *can* be grouped into distinct categories. We expect this classification to be refined based on further experience. This would be similar to the evolution of MIME-types which have been repeatedly refined as users have better understood their potential.

The classes described in Table 1 form a lattice based on their resource requirements. A class `X` is higher in the lattice than a class `Y` if the resources required by `Y` are a proper subset of the resources required by `X`. For example, applications in the `filter` class can access only `stdin/stdout/stderr` whereas applications in the `transformer` class can access `stdin/stdout/stderr` as well as `infile` and `outfile`. We present this lattice in Figure 2.

3 Design and implementation of MAPbox

Our implementation of MAPbox runs on Solaris 5.6 and confines native binaries. We first describe the sandbox description language provided by MAPbox which can be used to construct new sandboxes with relative ease. Next, we describe how MAPbox implements individual sandboxes.

3.1 The sandbox description language

We base our sandbox description language on the configuration language used by Janus [8], a class-specific sandbox for document viewers. Our language consists of eight commands: `path`, `connect`, `putenv`, `rename`, `accept`, `childbox`, `define` and `params`. Figure 3 provides a brief description for these commands (Figure 7 contains a BNF description). Of these, the first four commands were provided by Janus. For a detailed description of these commands, please see [8]. The last four commands are new to MAPbox and are described below. A sample sandbox specification is presented in Figure 9.

accept: this command is the server-side analogue of the Janus `connect` command. It can be used to control the set of peer hosts as well as the set of ports that the confined application can listen on.

```
open("/usr/lib/libsocket.so.1", O_RDONLY)          = 3
fstat(3, 0xEFFFEA00)                               = 0
mmap(0x000000, 8192, PROT_READ, MAP_SHARED, 3, 0)  = 0xEF7B000
mmap(0x000000, 8192, PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xEF7900
close (3)                                          = 0
```

Figure 1: The system-call sequence for dynamically linking a library in Solaris 5.6.

Behavior class	Parameters	Description
`filter`	None	cannot open files, access network/display or exec processes
`reader`	`dir/filelist`	can read files listed in `filelist` or contained in `dir` and its descendants; cannot write files, access network/display or exec processes (e.g, `cat`, CGI scripts that authenticate a user or provide a random image)
`transformer`	`infile`, `outfile`	can read `infile`, write `outfile`; cannot access network/display or exec processes (e.g., `compress`, `gzip`, image format converters)
`maintainer`	`homedir`	can read and write files in `homedir` and descendants; cannot access network/display or exec processes (e.g., CGI scripts that implement counters, guestbooks, bulletin boards, chat servers, etc.)
`compiler`	`homedir`, `filelist`, `libpath`, `outfile`	can read/write files in `homedir` and descendants; can read files in `filelist`; can read files in all directories on `libpath`; can write `outfile`; cannot access network/display; can exec other applications in the same class (e.g., `gcc`, `tar`, `dvips`, `latex`, `nroff`, `bibtex`, `ld`)
`editor`	`homedir`, `filelist`	can read/write files in `homedir` and its descendants; can read/write files in `filelist`; cannot access network; can access display; can exec applications labeled `filters` or `transformers` (e.g., `gnu-emacs`, `vi`, `pico`, `xfig`, `idraw`)
`viewer`	`homedir`, `filelist`	can read/write files in `homedir` and its descendants; can read files in `filelist`; cannot access network; can access display; can exec applications in the same class (e.g., `ghostview`, `pageview`, `imagetool`, `xdvi`)
`netclient`	`host, port, dir`	can connect to `host` at `port`; can read and write files in `dir`; cannot exec processes; cannot access display (e.g., `ftp`, `finger`, `wget`)
`mailer`	`homedir`, `[mailbox]`, `[gateway]`, `[mailcommand]`	can read/write files in `homedir` and descendants, can read/write `mailbox` file (if specified), can connect to `gateway` (if specified) on port 25; can access display; can exec `viewers` and `filters`, can exec the `mailcommand` (if specified) (e.g., `pine`, `elm`, `mailtool`, many CGI scripts that implement guestbook, mailing lists and bulletin boards)
`browser`	`homedir`, `filelist`, `hostlist`, `port`	can read/write files in `homedir` and descendants; can read files in `filelist`; can connect to hosts in `hostlist` at `port`, can access display; can exec `viewers` (e.g., `lynx`, `hotjava`, `trn`)
`netserver`	`homedir`, `hostlist`, `port`	can read/write files in `homedir` and descendants; can accept connections at `port` from hosts in `hostlist`; cannot access display; can exec `filters`, `transformers` and `maintainers` (e.g., `httpd`, `ftpd`)
`shell`	`path`, `mapfile`, `[maptypelist]`	can exec binaries found in the directories listed in `path`; can read `mapfile`; `maptypelist` can be used to limit the MAP-types of applications that be exec'ed; cannot access network; cannot access display (e.g., `ksh`, `csh`, `tcsh`)
`game`	`homedir`	can read/write `homedir`; can access display; cannot access network; can exec applications in the same class
`applet`	`host`, `port`, `path`	can access display; can connect to `host` at `port`; can read files in directories listed in `path`; cannot write files; cannot exec processes.

Table 1: Brief descriptions of the behavior classes identified in this study.

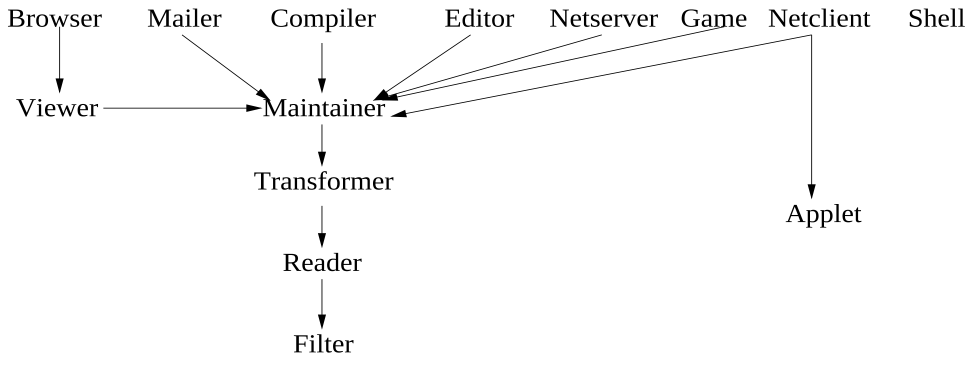

Figure 2: Lattice describing the relationship between application behavior classes.

Command	Description
`path`	used to `allow` or `deny` `read/write/exec` access to a list of files (e.g., `path deny read,write,exec /etc`). Wildcards are allowed; relative paths are not allowed; `deny` takes precedence over `allow`.
`rename`	used to redirect accesses to a particular file to a different file (e.g., `rename read /etc/passwd /tmp/dummy`). Wildcards and relative paths are not allowed.
`connect`	used to control connections to remote hosts and the X server. Must be specified as IP addresses; wildcards allowed (e.g., `connect allow tcp 128.111.*.*:80/128.32.*.*:8080`)
`putenv`	used to add a variable definition to the environment (e.g., `putenv HOME=/tmp/boxedin`)
`accept`	used to control connections from remote hosts (e.g., `accept allow udp 128.111.*.*:513`)
`childbox`	used to specify the sandbox to be used for processes forked by the confined application (e.g., `childbox viewer`). At most one `childbox` command allowed per sandbox.
`define`	used to define a symbolic value that can be used later (e.g., `define _NETWORK_FILES /etc/netconfig /etc/nsswitch.conf /etc/.name_service_door`)
`params`	used to define the parameters for a sandbox (e.g., `params infile outfile`). Parameters are refered to using a $ prefix (e.g., `$outfile`)

Figure 3: Brief description of the MAPbox sandbox description language.

The value `NON_SYSTEM_PORT` can be used to indicate any port not reserved for system services.

childbox: this command is used to specify a different sandbox for the processes forked by the confined application. If no childbox command occurs in a sandbox specification, the original sandbox is used to confine the children, if any. For example, children of `browsers` can be restricted to be `viewers`.

define: this command can be used to define symbolic constants which can then be used in other commands. Symbolic constants can be used to simplify the task of porting sandboxes across platforms. For example, to be able to access the network on many platforms, an application needs to link in a platform-dependent set of libraries[2] and read a platform-dependent set of configuration files.[3] Symbolic definitions can be used to isolate these dependencies. As long as the sandboxes are defined in terms of symbolic constants which are collected in a single file, porting the entire set of sandboxes is a matter of redefining the symbolic constants in this file. To support this, MAPbox reads a common specification file before it reads the specification file for a particular sandbox. Figure 8 presents an example of a common specification file for Solaris 5.6.

params: this command is used to define the parameters for a sandbox. This command can occur only once in a specification and must precede all other commands.

[2]On Solaris 5.6, `/usr/lib/libsocket.so.1`, `/usr/lib/libnsl.so.1.so.1` and `/usr/lib/nss_compat.so.1`.

[3]On Solaris 5.6, `/etc/netconfig`, `/etc/nsswitch.conf` and `/etc/.name_service_door`.

3.2 Implementation details

Initialization: MAPbox starts by reading the sandbox specification file (specified on the command line) and building the `Policy` structure. The `Policy` structure consists of eight components: (1) `read-list` (list of files that can be read), (2) `write-list` (list of files that can be written), (3) `exec-list` (list of binaries that can be `exec`'ed), (4) `rename-list` (list of files whose accesses are to be redirected to some other file), (5) `connect-list` (list of host/port combinations that the confined application can connect to), (6) `accept-list` (list of hosts that the confined application can accept connections from and the ports that it can bind to), (7) `env-list` (list of environment variables for the confined application), and (8) `childbox` (the sandbox to be used for child processes, if any). It first forks. The forked version sets up the environment for the application to be confined by: limiting the environment variables to those specified in sandbox, setting umask to 077, limiting the virtual memory use and datasize, disabling core dumps, changing the current working directory to the application's `homedir` directory,[4] and closing all unnecessary file descriptors. It then `exec`'s the application to be confined.

Interception mechanism: we use the `/proc` interface provided by Solaris 5.6 to intercept selected system-calls. The `/proc` interface has been previously used by researchers for building class-specific sandboxes [3, 8] and for user-level extensions to operating systems [2]. This interface guarantees that *all* system-calls are intercepted. It allows us to intercept system-calls both on their entry to and exit from the operating-system. The interception mechanism provides information about the identity of an intercepted system-call, its arguments, whether it is an entry or an exit, and the return value (if intercepted on exit). We intercept most system-calls on their entry to the kernel to allow or deny access to resources; we intercept a few system-calls on their return from the kernel to record a returned value (e.g., `fork`) or to control access to blocking communication calls (e.g., `accept` for which the identity of the peer is known only when it returns). MAPbox maintains a handler for every intercepted system-call (separate handlers are maintained for entry and exit). When a system-call is intercepted, the corresponding handler is invoked. To deny a system-call, the handler sets a field in the structure used to communicate between the kernel and MAPbox. A denied system-call returns to the application with an error code of `EINTR`. For a description of individual system-call handlers, please see [16].

Handling symbolic links: since Unix file-systems support symbolic links, simply checking the arguments for file-system-related system-calls is not sufficient to implement file-system-related checks. For example, `/tmp/letter-to-my-mom.txt` can be a symlink to `/etc/passwd`. To plug this hole, MAPbox completely resolves each filename (using the `resolvepath()` call available in Solaris) before checking it against the `Policy` structure.

Redirecting requests for sensitive files: to redirect requests for a sensitive file to a benign dummy file, MAPbox resolves all filenames completely and compares them with the completely resolved name of the sensitive file. If a match is found, it writes the name of the dummy file as a string on the stack of the confined process,[5] and changes the pointer to the filename argument to the intercepted system-call to point to this string. It then allows the system-call to proceed.

Confining child processes: MAPbox creates a separate copy of itself for every child of a confined process. To achieve this, it intercepts the `fork` system-call on exit and extracts the process-id of the newly created process. It then forks itself and attaches the child MAPbox process to the newly created application process. Unless specified otherwise, the child of a confined application is confined in the same sandbox as the parent. If, however, a different sandbox is specified (using the `childbox` command), the instance of MAPbox corresponding to the child process intercepts the subsequent `exec` system-call and reads the appropriate sandbox specification file.

Other system-calls: the MAPbox sandbox specification language can specify the confinement requirements for most but not all system-calls. For the remaining system-calls, MAPbox implements an

[4] If no `homedir` directory exists for the application, a temporary directory is created in `/tmp` for this execution and is used as the current working directory. This directory is deleted after the confined program terminates.

[5] On Solaris 5.6, this is implemented using `pwrite()` and `ioctl()`s on the `/proc` file corresponding to the confined application.

application-independent policy. It does not intercept system-calls related to signals, threads and virtual memory. For these resources, it relies on the security provided by the kernel. It also does not intercept system-calls that perform read/write or send/receive operations – depending on the checks performed for initializing operations such as `open()`, `creat()`, `socket()` etc. For others, it takes a conservative approach and denies all system-calls that it does not know to be safe.

- it denies calls that can be invoked only with super-user privileges (e.g., `mount`, `umount`, `plock`, `acct`, etc.).
- it currently denies calls to `acl()` which gets/sets the access-control list for a file. We have not yet seen these system-calls in traces.
- it denies all calls to `door()` except those used to query the host database.
- it allows `fcntl()` calls with F_DUPFD, F_DUP2FD (which return new file descriptors) and F_GETFD, F_SETFD (which read and write file descriptor flags) commands. It denies `fcntl()` calls with other commands.
- it allows a small number of `ioctl` calls on `stdin` and `stdout`. It currently denies all other `ioctl` calls. This call performs a variety of control functions on devices and streams. Properly handling `ioctl` requires a good understanding of the individual devices and their controls.

Confining X applications: The X protocol has been designed for use by cooperative clients. Any client application is able to manipulate or modify objects created by any other client application run by the same user.[6] This has been done for two reasons. First, it allows window managers to be written as ordinary clients and second, it allows clients to communicate to implement cooperative functionality such as cut-and-paste.

To confine X applications, we have developed `Xbox`, an X protocol filter [1]. `Xbox` has been designed to be used in conjunction of a system-call-level sandbox such as MAPbox and Janus and is to be interposed between an untrusted application and the X server. Before starting an untrusted X application, MAPbox sets the DISPLAY environment variable to a socket that `Xbox` listens on (`unix:4` by default). It then makes sure that the confined application does not bypass `Xbox` by denying direct connections to the X server.

`Xbox` snoops on all protocol messages and keeps track of the resources (windows, pixmaps, cursors, fonts, graphic contexts and colormaps) created by the confined application. `Xbox` can be easily extended to handle extensions to the X protocol. The current implementation handles the `SHAPE`, `MIT-SCREEN-SAVER`, `DOUBLE-BUFFER`, `Multi-Buffering`, and `XTEST` extensions. The confined application is allowed to access/manipulate only the resources that it has created and is allowed to read limited information from the root window (the operations it allows on the root window are both necessary and safe). All other requests regarding specific resources are denied (e.g., CreateWindow, ChangeWindowAttributes, GetWindowAttributes, InstallColorMap,ReparentWindow, ChangeGC, ClearArea, PolyPoint etc). In addition, the confined application is not allowed to query parts of the window hierarchy it did not create and is allowed limited versions of some operations that change the global state of the server (GrabKey, GrabButton etc). Other global operations (such GrabServer, SetScreenSaver, ChangeKeyboardMapping etc) are denied. Finally, the confined application is not allowed to communicate with other applications via the X server.

[6]The existing security mechanisms provided by the X server, i.e., the `xhost`-based mechanism and the `xauthority`-based mechanism cannot distinguish between multiple applications belonging to the same user.

4 Configuration and administration

There are two ways in which MAPbox can be configured. First, by listing the MAP-types allowed by the user in a `.mapcap` file; and second, by placing commands in a site-wide specification file which MAPbox reads when it starts up.

Specifying acceptable MAP-types: the list of MAP-types acceptable to the user can be specified in a `.mapcap` file. This file contains a sequence of entries consisting of (MAP-type, sandbox-file) pairs. A MAP-type consists of the name of a behavior class with values for all its parameters. The corresponding sandbox file contains a description of the sandbox that is to be used for this MAP-type. A parameter can be specified using as a symbolic value, a concrete value, a regular expression, a numeric range, or a list. Multiple combinations of parameter values can be specified using separate entries. Parameters for some behavior classes (e.g., `transformer`) include command-line arguments that will supplied only when an application runs (for `transformer`, the the input and output files). These parameters are specified by the meta-values `%a1`, `%a2`, `%a3` etc. These correspond to the arguments supplied to the program – in the same order as they are specified. Several behavior classes have a `homedir` parameter which specifies the home directory for the application. Typically, this is the directory in which all the files for the application reside and the application is allowed to read/write files in this directory and its descendants. To refer to the directory that the binary for an application lives in, MAPbox provides the meta-value `%h` (`h` for `homedir`).[7] The syntax for `.mapcap` entries is presented in Figure 4. A sample `.mapcap` file is presented in Figure 5.

To check if an application is to be allowed to run, the MAP-type specified by the provider is matched against entries in the `.mapcap` file. The rules for matching are:

- an empty argument can only be matched by an empty argument.
- meta-values, like `%a1`, `%a2` and `%h`, can be matched only by themselves.
- for all other arguments, the value provided by the application provider should not be more general than the value in the `.mapcap` file. For example, `browser(%h,www.aol.com,80)` would match the specification in the `.mapcap` file in Figure 5 whereas `browser(%h,*,*)` would not.

Implementing site-wide policies: as mentioned in Section 3, MAPbox reads a common specification file before it reads the specification file for a particular sandbox. In addition to making sandbox specification files more portable, this feature can also be used to implement site-wide policies. The purpose of this feature is not to deal with malicious users – it is easy to bypass this mechanism. Instead it is to rapidly respond to problems in a cooperative environment. Figure 8 contains a sample of a common specification file.

[7]Executables in a a software package are often placed in a "appDir/bin" directory whereas the resource files are usually placed in a separate subdirectory of "appDir" (e.g. "appDir/lib"). To handle this common case, MAPbox checks if the last element in an application's pathname is "bin". If so, it removes this element. For example, if the application lives in "/apphome/bin", this meta-value would expand to "/apphome".

5 Evaluation of MAPbox

We evaluated MAPbox from two different perspectives: its effectiveness (how well it is able to confine a suite of untrusted applications) and efficiency (what is the overhead introduced).

5.1 Effectiveness of MAPbox

For these experiments, we used a suite of 100 applications: the fifty applications used in the application characterization study mentioned in Section 2 and fifty additional applications. Of the additional applications, twenty were Perl-based CGI scripts from *cgi.resource-index.com*, fifteen were programs that we downloaded from different repositories and built locally and fifteen were applications from the Solaris 5.6 distribution.[8] We assigned each application a MAP-type based on the code (where available), the associated documentation (manual, man page, README file) and a trace of the system calls it makes.

We performed two sets of experiments. The first set of experiments were designed to check if the behavior classes identified in Section 2 were too restrictive. In other words, is MAPbox so restrictive that few or no applications can be successfully run while confined? The second set of experiments were designed to check if the behavior classes were too broad. That is, is MAPbox so lax that mislabeled applications (i.e., applications that were labeled with a different MAP-type than their own) are able to gain access to

[8]The CGI scripts were `jchat10c`, `kewlcheckers`, `kewlchess`, `mazechat`, `multimail`, `netcard201`, `picpost`, `postit`, `robpoll`, `SDPGuestbook`, `SDPMail`, `SDPUpload`, `search`, `showsell`, `UltraBoard_1.62`, `web_store`, `webadverts`, `webbbs`, `webodex`, `wwwchat30`. The downloaded and built programs were `gnu-emacs`, `lcc`, `javac`, `wget`, `ksh`, `latex`, `bibtex`, `xbiff`, `xclock`, `groff`, `gnuplot`, `mpeg_play`, `cjpeg`, `gzcat`, `md5sum`. The Solaris 5.6 applications were `tcsh`, `comm`, `detex`, `deroff`, `compress`, `tar`, `ld`, `talk`, `strings`, `sort`, `diff`, `s2p`, `find2perl`, `mpage` and `cc`.

```
entry          := behaviorClass (args) sandboxfile
behaviorClass  := filter | transformer | ...
args           := /* empty */ | arg | args arg , arg
arg            := value | list | %a | %c | /* empty */
list           := {values}
values         := values , value | value
value          := regexp | [ num - num ]
```

Figure 4: Syntax for `.mapcap` entries.

```
filter()                /fs/play/~user/mapbox/sandboxes/filter.box
transformer(%a1,%a2)    /fs/play/~user/mapbox/sandboxes/transformer.box
browser(%h,*,80)        /fs/play/~user/mapbox/sandboxes/browser.box
game(%h)                /fs/play/~user/mapbox/sandboxes/game.box
maintainer(%h)          /fs/play/~user/mapbox/sandboxes/maintainer.box
```

Figure 5: Sample `.mapcap` file.

resources that the user did not wish to grant? For the first set of experiments, we ran each application within the sandbox associated with its own MAP-type. For the second set of experiments, we ran each application within a sandbox that corresponds to a MAP-type other than its own. For both experiment sets, we ran these applications with workloads similar to those used in the classification study described in Section 2.

5.1.1 Is MAPbox too restrictive?

Of the 100 applications in our evaluation suite, only nine failed to complete their workload when run within the sandbox for their own MAP-type. Of these, six belonged to the original set of 50 applications that were used in the classification study described in Section 2, the remaining three belonged to the second set of 50 applications added for these experiments.[9] Of the 40 CGI scripts in the suite, one failed; of the 30 downloaded programs, five failed; of the 30 Solaris applications three failed. Of these nine programs, five (`xv`, `xfig`, `pageview`, `lynx` and `Netscape`) failed because they made unsafe accesses and the other four failed inspite of making accesses that we manually verified to be safe. Of the latter, two (`gcc` and `gnu-emacs`) failed because they made a sequence of requests that were individually unsafe but taken as a sequence, implement a safe operation. Since MAPbox makes decisions about each system-call independently, it is unable to detect such cases. The last two, (`cc` and `multimail`) failed because they do not fit into our current collection of MAP-types.

Applications that failed due to unsafe operations:: Three applications failed because they tried to perform unsafe X window operations: `xv` failed when it tries to scan the entire window hierarchy of the X server; `xfig` failed trying to allocate a colormap not owned by itself; and `pageview` failed trying to change an attribute of a window not owned by itself. Two other applications failed because they were denied access to sensitive files: `lynx` tried to access the password database via a `door()` call; `Netscape` needed access to non-empty `/etc/passwd` and `/etc/mnttab`.

Applications that failed due to local nature of checking: Several applications try to determine the current working directory, a safe operation by itself, by walking up the directory hierarchy using relative paths, which is an unsafe operation. Figure 6 illustrates this behavior using a system-call trace excerpt. MAPbox does not allow this operation since it denies all file-system calls with relative paths. Two applications, `gcc` and `gnu-emacs`, failed due to this limitation. Another application, the Solaris C compiler `cc`, also failed while performing this operation but had another reason for failure (see below). Note that this particular problem can be elim-

[9] As mentioned in Section 2, the resource requirements for a class are not simply the union of the resource used by a set of applications that we studied. Instead, they are the set of resources that we believe are *required* to implement the expected functionality for the class.

inated if the Solaris system-call interface is extended to provide a *getcwd()* operation directly. However, the general problem of not being able to distinguish safe sequences of potentially unsafe operations is inherent to the MAP-box approach. Based on our experience, however, we expect this problem to be rare.[10]

Applications that failed due to lack of a suitable MAP-type: Two applications failed as they could not fit into our current collection of MAP-types: `cc` (the Solaris C compiler), and `multimail` (a CGI mailing program). The Solaris C compiler fails because it connects to a license server and the sandbox for a `compiler` does not allow access to the network. If desired, this can be fixed by introducing a new MAP-type, say `licensed-compiler`, which includes the host and port number of the license server as parameters. The CGI mailer, `multimail`, fails as it invokes a program (`/bin/date`) that is not the mail command. If desired, this problem can be fixed by rewriting the program to directly determine the current time.

Note that only four applications from a diverse suite of 100 applications fail due to features of MAPbox. This indicates that a MAP-type-based approach is not too restrictive.

5.1.2 Is MAPbox too lax?

For each application used in these experiments, we selected a conflicting MAP-type, that is, a MAP-type that would allow the application to access resources that it would not be allowed to if correctly labeled. In effect, we picked a MAP-type that was not its own and was not an ancestor in the lattice shown in Figure 2. Of the 100 applications in our evaluation suite, not one completed its workload in these experiments. This provides evidence that MAPbox is not too lax.

5.2 Efficiency of MAPbox

To evaluate the efficiency of the MAPbox implementation, we ran two sets of experiments. In the first set, we used MAPbox to confine CGI scripts in a web-server environment and measured the additional latency experienced by web clients over a long-haul network. For these experiments, we used a suite of six CGI scripts. In the second set of experiments, we used MAPbox to confine non-interactive applications in a desktop environment and measured the increase in their execution time. For these experiments, we used a suite of six applications.

The applications used in these experiments and the corresponding workloads are listed in Table 2. We ran each application with and without MAPbox and measured the difference in end-to-end execution time. For each experiment, we also kept track of the time spent in MAPbox code. We conducted these experiments on a lightly loaded SUN Ultra-1/170 with 64 MB and Solaris 2.6 (i.e., the applications and the CGI scripts were run on this machine). All files involved in these experiments were in the OS file-cache. We used the Solaris high resolution timer `gethrtime()` for all measurements.

For the experiments involving CGI scripts, the server (Apache 1.0.2) was at the University of California, Santa Barbara on the US west coast and the client was at the University of Maryland, College Park on the US east coast. We ran these experiments between 1am and 3am Pacific Time when network congestion is usually light. Measurement of the end-to-end execution time was done at the client. The round-trip time between these sites (as determined by `ping`) was about 80 ms. To factor out the effects of transient congestion, we repeated each experiment 100 times and reported the minimum value as the result. For the experiments involving local applications, we repeated each experiment five times and reported the minimum value as the result.

Table 3 presents the results of all experiments. The overhead caused by MAPbox for CGI scripts was small ($< 5\%$) in all experiments. This is to be expected since only a small fraction of the end-to-end execution time in these cases was due to the execution of the scripts themselves; network latency, transfer time and other administrative costs (web server overhead, CGI invocation etc) contributed a large fraction of the execution time. The overhead caused by MAPbox for local applications varied greatly – from about 1% for `gzip-1MB` and `grep` to 33% for `gzip-8KB`. For five out of the six applications, the overhead was below 20%. From these results, we conclude that the overhead of confinement is likely to be acceptable for many applications and

[10] In case, we are mistaken in this expectation, it is quite easy to extend MAPbox to handle relative paths by using the `resolvepath()` system-call to completely resolve all relative paths.

```
stat64("./", 0xEFFFC620)                     = 0
stat64("/", 0xEFFFC588)                      = 0
open64("./../", O_RDONLY|O_NDELAY)           = 3
fcntl(3, F_SETFD, 0x00000001)                = 0
fstat64(3, 0xEFFFBC30)                       = 0
fstat64(3, 0xEFFFC620)                       = 0
getdents64(3, 0x0005A014, 1048)              = 608
close(3)                                     = 0
open64("./../../", O_RDONLY|O_NDELAY)        = 3
fcntl(3, F_SETFD, 0x00000001)                = 0
fstat64(3, 0xEFFFBC30)                       = 0
fstat64(3, 0xEFFFC620)                       = 0
getdents64(3, 0x0005A014, 1048)              = 280
close(3)                                     = 0
```

Figure 6: System-call trace excerpt illustrating the `getcwd()` pattern.

application	type	workload	application	type	workload
`ftp`	local	ftp 10 32KB files from `localhost`	`dvips`	local	convert a 50 page DVI file to postscript
`latex`	local	compile 5 tex files ($\approx$ 300 lines each)	`grep`	local	search gcc source for "int", 182 files
`gzip-1MB`	local	compress 4 1MB files	`gzip-8KB`	local	compress 32 8KB files
`guestbook`	CGI	post 100 1KB msgs	`wwwchat30`	CGI	post 100 128 byte msgs
`counter`	CGI	100 counter accesses	`kewlcheckers`	CGI	make 20 moves
`SDPUpload`	CGI	upload 10 64KB files	`webbbs`	CGI	post 32 8KB msgs

Table 2: Workloads used in the experiments. The two gzip workloads were selected to compare the overheads for processing a few large files with the overhead for processing many small files.

environments.

To determine the cause of the variation in the overhead for local applications, we analyzed their operation in greater detail. We found that the cost of using MAPbox depended on the frequency of file-system-related system-calls (`open`/`stat` etc). To obtain a fine-grain breakdown of this overhead, we added probes in the handlers for these calls and repeated the experiments. We found that most of this overhead (90% of the time spent in MAPbox) is due to two operations: (1) the *resolvepath* operation which is used to safely handle symbolic links by completely resolving a filename (65% of the time spent in MAPbox); and (2) reading the string containing the filename from the confined process's memory (25% of the time spent in MAPbox). These costs are inherent to the system-call interception technique and cannot be eliminated.

6 Discussion

We first present our experience with determining suitable MAP-types for applications. We then discuss potential limitations of the MAPbox approach.

6.1 Experience determining MAP-types

Of the 100 applications in our suite, 91 applications completed their test workloads. Of these, twenty applications were labeled `mailer`, nineteen were labeled `maintainer`, nine were labeled `compiler`, eight each were labeled `reader` and `transformer`, seven each were labeled `netclient` and `viewer`, six were labeled `editor`, three were labeled `shell`, two were labeled `browser` and one each were labeled `filter` and `netserver`.

application	total time	total time with MAPbox	time in MAPbox	other overhead
`ftp`	1.99s	2.32s (17%)	0.17s (9%)	0.16s (8%)
`dvips`	2.88s	3.26s (13%)	0.11s (4%)	0.27s (9%)
`latex`	2.80s	3.06s (9%)	0.17s (6%)	0.09s (3%)
`grep`	2.72s	2.76s (1.2%)	0.02s (0.6%)	0.02s (0.6%)
`gzip-1MB`	4.26s	4.30s (1%)	0.01s (0.2%)	0.03s (0.8%)
`gzip-8KB`	1.52s	2.02s (33%)	0.23s (15%)	0.27s (18%)
`guestbook`	49.1s	51.2s (2.2%)	0.36s (0.7%)	0.74s (1.5%)
`wwwchat30`	19.21s	19.62s (2%)	0.2s (1%)	0.22s (1%)
`counter`	25.4s	26.0s (2%)	0.32s (1%)	0.28s (1%)
`kewlcheckers`	16.94s	17.23s (1.7%)	0.1s (0.6%)	0.19s (1.1%)
`SDPUpload`	22.31s	22.9s (2.6%)	0.3s (1.3%)	0.29s (1.3%)
`webbbs`	26.12s	26.94s (3%)	0.31s (1%)	0.51s (2%)

Table 3: MAPbox overheads. All percentages are with respect to end-to-end execution time without MAPbox (second column). The time in the "other overhead" column includes kernel overhead for intercepting system-calls as well as the cost of the context-switches required to pass information between the kernel and MAPbox.

All the CGI scripts that we studied fell into only four MAP-types: `reader`, `maintainer`, `mailer` and `compiler`.[11] While we expected the first two MAP-types to be common among CGI scripts, the number of scripts that are able to send email was a surprise to us. Seventeen of the forty CGI scripts used in this study invoke `sendmail` and/or open a socket to a mail gateway. This included guestbooks, advertisement managers, homepage providers, web-based rolodex and bullletin board programs. These programs used email to notify users/administrators about events of interest. It appears that the authors of CGI scripts prefer to send mail for this purpose instead of writing to a log file (as is common in conventional applications).

6.2 Potential limitations

We believe that the MAPbox approach provides a good tradeoff between ease of use and flexibility. Nevertheless, it has several potential limitations.

Applications limited to single behavior: the MAPbox approach limits each application to a single behavior. Many applications, however, exhibit multiple behaviors (e.g., Netscape can be used as a browser, mailer and newsgroup reader). In some specific cases, it may be possible to create customized classes that allow a particular group of behaviors. In general, however, we believe this is an inherent trade-off between security and functionality: many applications cannot be securely confined in all their generality.

[11] Only one application was assigned the `compiler` MAP-type: `search` which compiles an index for all files on a website and looks it up on demand.

Confining `setuid root` programs: the system-call interception mechanism used by MAPbox does not work for `setuid root` programs. This is a necessary restriction as allowing a user-level process to intercept the system-calls of a `setuid root` program would provide a trivially easy way to become `root`.

Lack of standardized behavior classes: given that individual end-users are allowed (though not required) to create new MAP-types and the corresponding sandboxes, it is conceivable that everyone defines different MAP-types or uses different names for the same classes resulting in configuration chaos. While this is a possibility, we believe that it is unlikely to happen. As evidence, we point to the web community's experience with MIME-types which have a similar potential for configuration chaos. Instead, the set of MIME-types used by most users has converged to a more-or-less stable set.

Portability: MAPbox depends on the ability to intercept all system-calls for implementing a secure reference monitor. Currently, only Solaris and Linux provide this facility.

7 Conclusions

In this paper, we have presented the design, implementation and evaluation of MAPbox, a confinement mechanism that retains the ease of use of application-class-specific sandboxes, such as Janus, while providing significantly more flexibility. Based on a study of a diverse set of applications, we have identified a set of behavior classes which have intuitive meaning and whose resource requirements can be differentiated. We do not claim that this classification is either unique or complete. Our goal in identifying these classes was to demonstrate that application behaviors and the corresponding resource requirements *can* be grouped into distinct categories. We expect this classification to be refined based on further experience.

To evaluate the effectiveness of MAPbox, we tried to confine a large suite of applications (including Perl-based CGI scripts, downloaded programs and applications from the Solaris distribution) using suitable class-specific sandboxes. We found that a MAP-type-based approach is quite effective for confining untrusted applications. Of the 100 applications in our evaluation suite, only nine failed to complete their test workloads when run within the sandbox corresponding to their own MAP-type. Of the 40 CGI scripts in the suite, one failed; of the 30 downloaded programs, five failed; of the 30 Solaris applications, three failed. Of these, five failed because they made unsafe accesses and only four failed in spite of making accesses that we manually verified to be safe. We also found that mislabeled applications (i.e., applications that were labeled with a different MAP-type than their own) were not able to gain access to resources that the user did not wish to grant.

To evaluate the efficiency of the MAPbox implementation, we ran two sets of experiments – one set with CGI scripts and the other with local applications. We found that the overhead caused by MAPbox for CGI scripts was small ($< 5\%$) in our experiments. The overhead caused by MAPbox for local applications varied greatly – from about 1% to 33%. For five out of the six applications, the overhead was below 20%. We found that the cost of using MAPbox depended on the frequency of file-system-related system-calls. From these results, we conclude that the overhead of confinement is likely to be acceptable for many applications and environments.

Acknowledgments

We would like to thank the authors of Janus for making their implementation available. While MAPbox has been implemented anew and contains much functionality not provided by Janus, we benefited *greatly* from reading their code as well as its lucid description in [8]. We would also like to thank Paul Kmiec for suggesting the use of `pwrite()` to write a string into the stack of a confined process.

References

[1] A. Acharya. The Xbox distribution. Available at *http://www.cs.ucsb.edu/∼acha/software/xbox.tar.gz*, 1999. `Xbox` is a confining filter for X11 applications.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proc. of the 1997 USENIX Technical Conference.*

[3] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. Available at *http://www.cs.ucsb.edu/∼berto/papers/99-usenix-consh.ps*, Dec 1998.

[4] W. Boebert, R. Kain, W. Young, and S. Hansohn. Secure Ada Target: Issues, System Design, and Verification. In *Proc. of 1985 IEEE Symposium on Security and Privacy*, pages 176–83.

[5] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. of the Fifth ACM Conference on Computer and Communications Security*, 1998.

[6] J. Fritzinger and M. Mueller. Java security. Technical report, Sun Microsystems, Inc, 1996.

[7] T. Gamble. Implementing execution controls in Unix. In *Proc. of the 7th System Administration Conference*, pages 237–42, 1993.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proc. of the 1996 USENIX Security Symposium*, 1996.

[9] L. Gong. New security architectural directions for Java. In *Proc. of IEEE COMPCON'97*, 1997.

[10] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proc. of the Sixth USENIX Security Symposium*, 1996.

[11] P. Karger. Limiting the damage potential of the discretionary trojan horse. In *Proc. of the 1987 IEEE Syposium on Security and Privacy*, 1987.

[12] M. King. Identifying and controlling undesirable program behaviors. In *Proc. of the 14th National Computer Security Conference*, 1992.

[13] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings. 10th Annual Computer Security Applications Conference*, pages 134–44, 1994.

[14] N. Lai and T. Gray. Strengthening discretionary access controls to inhibit trojan horses and computer viruses. In *Proc. of the 1988 USENIX Summer Symposium*, 1988.

[15] N. Mehta and K. Sollins. Extending and expanding the security features of Java. In *Proc. of the 1998 USENIX Security Symposium.*

[16] M. Raje. Behavior-based confinement of untrusted applications. Technical Report TRCS99-12, Dept of Computer Science, University of California, Santa Barbara, Jan 1999.

[17] F. Schneider. Enforceable security policies. Technical report, Dept of Computer Science, Cornell University, 1998.

[18] L. Stein. SBOX: put CGI scripts in a box. In *Proc. of the 1999 USENIX Technical Conference.*

[19] K. Walker, D. Sterne, M. Badger, M. Petkac, D. Shermann, and K. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. of Sixth USENIX Security Symposium*, 1996.

[20] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for Java. In *Proc. of the Sixteenth ACM Symposium on Operating System Principles*, 1997.

[21] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL's: an access control list approach to anti-viral security. In *USENIX Workshop Proceedings. UNIX SECURITY II*, pages 71–82, 1990.

```
command        :=   path_c | rename_c | connect_c | accept_c | putenv_c | childbox_c
path_c         :=   path permission access_modes file_list
rename_c       :=   rename file1 file2
connect_c      :=   connect permission protocol ip_addr_list
               |    connect permission display
accept_c       :=   accept permission protocol ip_addr_list
putenv_c       :=   putenv name_val_list | putenv DISPLAY
childbox_c     :=   childbox class
permission     :=   allow | deny
access_modes   :=   access_modes , access_modes | access_mode
access_mode    :=   read | write | exec
file_list      :=   filename file_list | filename
protocol       :=   tcp | udp | *
ip_addr_list   :=   ip_addresses : port_addr
ip_addresses   :=   ip_addr , ip_addresses | ip_addr | *
port_addr      :=   port_num / port_mask | port_num | *
```

Figure 7: Grammar for the sandbox description language. Note that the `define` and `params` commands are not included in the above description. These commands are implemented as macros in a preprocessing step.

```
define _COMMON_LD_LIBRARY_PATH /usr/openwin/lib:/usr/ucblib

define _COMMON_READ /dev/zero /usr/lib/locale/*

# /dev/zero is a device file used for mmap's
define _COMMON_WRITE /dev/zero
# this is true in our environment
define _COMMON_TERM  xterm
# redirect X requests to the Xbox filter
define _COMMON_DISPLAY unix:4

define _COMMON_LIBS /usr/lib/libthread.so.1 /usr/lib/libICE.so.6\\
/usr/lib/libSM.so.6 /usr/lib/libw.so.1 /usr/ucblib/* \\
/usr/lib/libc.so.1 /usr/lib/libdl.so.1 /usr/lib/libintl.so.1\\
/usr/lib/libelf.so.1 /usr/lib/libm.so.1 /usr/lib/liballoc.so.1\\
/usr/lib/libmp.so.2 /usr/lib/libmp.so.1 /usr/lib/libsec.so.1

define _X_FILES /usr/openwin/lib/* /usr/openwin/share/*\\
/usr/openwin/bin/*

define _NETWORK_READ_FILES /etc/netconfig /etc/nsswitch.conf\\
/etc/.name_service_door

define _NETWORK_WRITE_FILES /dev/tcp /dev/udp /dev/ticotsord\\
/dev/ticlts, /dev/ticots

define _NETWORK_LIBS /usr/lib/libsocket.so.1 /usr/lib/libnsl.so\\
/usr/lib/nss_compat.so.1
```

Figure 8: A common specification file for Solaris 5.6.

```
# sandbox  spec for the browser class
# the browser sandbox takes three arguments -- the home directory
# the hosts it is allowed to connect to and the port(s)
# it is allowed to connect to.
params HOMEDIR HOSTSPEC PORTSPEC

# set up the env variables
putenv PATH=$HOMEDIR
putenv TERM=$_COMMON_TERM
putenv LD_LIBRARY_PATH=$_COMMON_LD_LIBRARY_PATH:$_NETWORK_LIBS
putenv DISPLAY=$_COMMON_DISPLAY

# _COMMON_READ and _COMMON_LIBS are accessible to all apps
path allow read $_COMMON_READ $_COMMON_LIBS $HOMEDIR
# browsers are allowed to read network config files and libs
path allow read $_NETWORK_READ_FILES $_NETWORK_LIBS
# browsers are allowed to read X data files and libs
path allow read $_X_FILES

# _COMMON_WRITE can be written by all (in this case /dev/zero)
# browsers are allowed to write HOMEDIR
path allow write $_COMMON_WRITE $HOMEDIR
# browsers are allowed to write networking device files
path allow write $_NETWORK_WRITE_FILES

# browsers are allowed to connect to all hosts in the argument
connect allow tcp $HOSTSPEC:$PORTSPEC
# broswers are allowed to connect to the X server
connect allow display

# all exec'ed children of browsers must be viewers
childbox viewer

# browsers are not allowed to access /etc/passwd
rename /etc/passwd /tmp/dummy
```

Figure 9: Sandbox example