MAPbox: Using Parameterized Behavior Classes to Confine Applications

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

TRCS99-15

May 1, 1999

Abstract

Designing a suitable mechanism to confine commonly used applications is challenging as such a mechanism needs to satisfy conflicting requirements. The trade-off is between configurability and ease of use. In this paper, we present the design, implementation and evaluation of *MAPbox*, a general-purpose confinement mechanism that retains the ease of use of specialized sandboxes such as Janus and SBOX while providing significantly more configurability. The key idea is to group application behaviors into classes based on the expected functionality and the resources required to achieve that functionality. Classification of behaviors provides a set of behavior labels (class names) that can be used to concisely communicate the *expected functionality* of programs between the provider and the users. This is similar to the MIME-types used to concisely describe the *expected format* of data files. Classification of application behaviors also allows class-specific sandboxes to be built and instantiated for each behavior class. We present a study of the behavior and resource requirements of a set of commonly used applications and use the results of this study to define a set of application behavior classes. We also evaluate how effective this technique is in confining a variety of commonly used applications and how much overhead it introduces.

1 Introduction

The applications that form our computing environment are often unsecured. These include network applications, which are often vulnerable to buffer-overflow attacks;[1] plug-ins downloaded to play/display different data-formats, which are often provided in a binary form and difficult to check for security flaws; executable content for extensible applications such as `ghostview`, `gnu-emacs`, MIME-mail,[2] and `Word`[3]; and CGI scripts used for generating dynamic web documents, which can either be subverted or can interfere with other unrelated scripts. Unsecured applications can pose a threat to the security and privacy of a user as they run with her full privileges. Given the number of such applications in a typical computing environment and their size and complexity[4] it is impractical to even attempt to ensure that all the applications in one's environment are secure. A

[1] 30 of the last 70 CERT advisories [6] and 38 of the 83 Caldera advisories [5] are concerning buffer-overflow attacks.
[2] For example, `Happy99` [12] and `Back Orifice` [13] were both distributed as mail attachments.
[3] The recent Melissa worm [21] was written in Visual Basic for Applications.
[4] For example, `ghostscript` 5.5 is over 240K lines of C code; `pine` 4.0 is over 270K lines of code.

less ambitious, and perhaps more achievable, goal is to try to *limit* the amount of damage that can be caused by unsecured applications by limiting their access to resources.

Designing a suitable mechanism to confine commonly used applications is challenging as such a mechanism needs to satisfy conflicting requirements. The key trade-off is between configurability and ease of use. On one hand, such a mechanism should be configurable enough to allow different policies for different kinds of applications. On the other hand, it should be easy to use and non-intrusive. That is, the effort needed to administer and use it should be small. Furthermore, an end-user should be able to deploy the mechanism on her own, without special privileges. In addition to being configurable and easy to use, a confinement mechanism needs to be simple and reasonably efficient. Simplicity is important since simple programs are easier to secure; reasonable efficiency is needed as users are likely to bypass overly slow mechanisms.

1.1 Previous research

Creating confinement environments (also referred to as *sandboxes*) has long been an active area of research. Previous research into creating confinement environments has taken one of four approaches which differ mainly in the trade-off between configurability and ease of use. Several researchers have proposed some form of per-program access control [7, 9, 15, 16, 18, 26]. This approach is highly configurable but requires users (or administrators) to specify access-control information for every program. It can work well if the number of applications is small and changes infrequently. Current computing environments are dynamic (new programs are downloaded) and contain a large number of applications. The second approach uses finite-state machine descriptions of program behavior [17, 20, 22]. This provides even more configurability 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 complexity and the number of applications available on a single machine, it would be hard to develop suitable behavior descriptions.

The third approach considers each application provider (author/company/web site) as a principal and uses per-provider access-control lists (ACLs) [11, 14, 25]. 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, this improvement is achieved at the cost of configurability. Disparate applications from the same provider may be grouped into the same sandbox. To allow *all* these applications to run, a user may have to provide an overly coarse sandbox. 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 on the part of individual end-users.

The fourth approach consists of special-purpose sandboxes for specific classes of applications, e.g, document viewers [10], applets [8], global computing [2], CGI scripts [23] and programs that run with root privileges [24]. By limiting the scope of the confinement mechanism, these techniques significantly reduce the administration effort required. A class-specific sandbox needs to be configured only once for each platform. End-users can use these sandboxes with minimal additional effort. While each of these sandboxes are easy to use when they are applicable, they are limited in their applicability. For each application she would like to confine, an end-user needs to 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.

1.2 Key idea

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 configurability. 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 etc. Classification of behaviors provides a set of behavior labels (names of classes) which can be used to concisely communicate the *expected functionality* of a program between its provider and its users. This is similar to the MIME-types which are widely used to concisely describe the *expected format* of files. Accordingly, we refer to the behavior labels as *Multi-purpose Application Profile*-types (or MAP-types). For mobile code that is downloaded on-demand, the MAP-type can be specified in the HTTP header (similar to the MIME-type header used currently). For CGI scripts provided by users of a web-hosting service, the MAP-type of the script can be specified when it is submitted for installation. For programs that are installed as a part of an OS distribution (e.g., programs in `/usr/bin`), the MAP-types can be provided by the vendor in a well-known file (e.g., `/etc/maps`). For applications downloaded and built locally, the MAP-type can be specified either by by the provider (e.g., as a part of a README file) or by the end-user.

Classification of application behaviors also allows class-specific sandboxes to be developed. The MAP-type specified for a program can be used to locate and instantiate the appropriate sandbox before running the program. Currently, end-users can specify which document formats (or MIME-types) they are willing to view, which application is to be used to view a particular format and how should this application be invoked (usually specified in the `.mailcap` file). Similarly, MAPbox allows users to specify which behaviors (or MAP-types) they would like to allow, which sandbox should be used for a particular MAP-type and how it should be instantiated (specified in a `.mapcap` file). This mapping can be used to automatically locate and instantiate the appropriate sandbox for a program without requiring user intervention.

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. Since the binding of a MAP-type to its sandbox as well as the configuration of all sandboxes can be customized by end-users, end-users retain complete control over resource usage. Application providers gain *no* advantage by misrepresenting the behavior of their program. Note that, MAPbox limits an application to only one behavior.

This proposal raises several questions. First, can application behaviors be suitably classified? That is, do application behaviors and the corresponding resource requirements fall into clean categories? Second, how does MAPbox deal with a group of applications that exhibit similar behavior but need different resources? For example, `finger` and `whois` both connect to a remote server, request some information, receive a reply and print it to screen; however, they differ in the port they connect to and in the local profile files they may access. 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 MAPbox work in practice? That is, is it able to confine commonly used applications at an acceptable cost?

In section 2, we describe a study of the behavior and resource requirements of 50 commonly used applications. 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. In section 4, we describe how MAPbox can be configured and used. In section 5, we present an evaluation of

```
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.

MAPbox: its performance as well as how well it is able to confine commonly used applications. We conclude with a discussion of the limitations and open questions for this approach.

2 Identifying Behavior Classes

To determine if the behavior and resource requirements of commonly used applications can be grouped into distinct categories, we studied 50 commonly used applications.[5] Our goal in selecting these applications was to be able to study a large and diverse set of commonly used applications. For each application, we obtained a trace of the OS system-calls it made while executing a workload. To design the workloads for our study, we started with an intuitive notion of behavior classes such as editors, viewers, compilers, mailers, etc. For each class, we defined a series of tests with increasing complexity. 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, modify and exit; (4) for text editors, edit a file, spell-check it and exit; (5) for graphical editors, generate a postscript file and exit.[6] Structuring the workload in this manner facilitated trace analysis; once we had analyzed the less complex behaviors, the more complex behaviors were easier to analyze.

To collect the system-call traces, we used the `truss` utility available on Solaris. It traces system-calls, signals and machine-faults. 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 [4]. 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.

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, ability to create new processes, environment variables, specific devices, and signals. We started the task of identifying behavior classes with an intuitive notion of program behaviors and refined this notion using the resource requirements of individual applications. For each behavior class, we identified resources commonly required to implement core functionality. Some applications have additional resource

[5]The applications were `vi`, `pico`, `gnu-emacs`, `idraw`, `xfig`, `ghostview`, `pageview`, `xv`, `imagetool`, `xdvi`, `cc`, `gcc`, `c++`, `f77`, `javac`, `pine`, `elm`, `mailtool`, `trn`, `lynx`, `Netscape`, `hotjava`, `sh`, `ksh`, `tcsh`, `ftp`, `finger`, `rwho`, `whois`, `telnet`, `latex`, `dvips`, `bibtex`, `ical`, `xcalc`, `xbiff`, `xclock`, `make`, `xterm`, `sed`, `grep`, `comm`, `nroff`, `detex`, `deroff`, `compress`, `gzip`, `tar`, `ld`, and `httpd`.

[6]Most of the workloads used in this study are described in [4].

requirements that are not warranted by their core functionality. For example the Solaris C compiler opens a socket to a license server to check licensing information. Other compilers that provide the same functionality (e.g., `gcc`) do not require such resources. 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 is not simply the union of the resource requirements for the corresponding 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 class with the *actual resource requirements* of commonly used applications that provide the same functionality.

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, `finger` and `whois` both connect to a remote server, request some information, receive a reply and print it to screen. For this, they need to link in networking libraries and make network connections. However, they differ in the port they connect to and in the local profile files they may access. In this case, the server port and the profile file are the parameters of the behavior class containing `finger` and `whois`.

Table 1 presents the behavior classes we identified and their parameters. In addition to the parameters listed in the table, each class has two additional parameters – a `read-home` directory and a `write-home` directory. These directories are expected to hold all the files specific to the application. The application has read and execute access to the `read-home` directory and its descendants and has `write` access to the `write-home` directory and its descendants. For example, the `read-home` directory for `httpd` contains the configuration files for the web server whereas the `write-home` directory contains the logs.

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 clean 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.

3 Design and implementation of MAPbox

Our implementation of MAPbox runs on Solaris 5.6 and confines native binaries. We first describe how the sandboxes for individual behavior classes are specified. Next, we describe how MAPbox implements individual sandboxes.

3.1 Sandbox description language

We base our sandbox description language on the configuration language used by Janus [10], a class-specific sandbox for document viewers. Our language consists of eight commands: `path`, `connect`, `putenv`, `rename`, `accept`, `childbox`, `define` and `params`. Figure 2 provides a brief description for these commands (Appendix A contains a BNF description). Of these, the first three commands were used in Janus. For a detailed description of these commands, please see [10]. The last five commands are new to MAPbox and are described below. A sample sandbox specification is presented in Appendix C.

rename: many applications try to access sensitive files such as `/etc/passwd` and `/etc/utmp`. The `rename` command can be used to redirect such accesses to safe replacements. Among the

Behavior class	Parameters	Description
`filter`		Unix-like filters; cannot open files, access network/display or exec processes (e.g., `sed`, `grep`, `comm`, `detex`, `deroff`)
`transformer`	`infile`, `outfile`	can read `infile`, write `outfile`; cannot access network/display or exec processes (e.g., `compress`, `gzip`, image format converters)
`compiler`	`dir/filelist`, `libpath`, `outfile`	can read files in `dir/filelist` and on dirs on `libpath`; can write outfile; cannot access network/display; can exec `compilers` (e.g., `gcc`, `make`, `tar`, `dvips`, `latex`, `nroff`, `bibtex`, `ld`)
`editor`	`dir/filelist`	can read/write files in `dir/filelist`; cannot access network; can access display; can exec `filters/transformers` (e.g., `gnu-emacs`, `vi`, `pico`, `xfig`, `idraw`)
`viewer`	`dir/filelist`	can read files in `dir/filelist`; cannot access network; can access display; can exec `viewers` (e.g., `ghostview`, `pageview`, `imagetool`, `xdvi`)
`download`	`host`, `port`, `dir`	can connect to `host` at `port`; can write (but not read) files in `dir`; cannot exec processes; cannot access display (e.g., `finger`, `whois`, `rwho`, `ftp` get commands)
`upload`	`host`, `port`, `dir`	can connect to `host` at `port`; can read (but not write) files in `dir`; cannot exec processes; cannot access display (e.g., `ftp` put commands, HTTP file upload)
`mailer`	`mailbox`, `gateway`	can read/write `mailbox`; can connect to `gateway` on port 25; can access display; can exec `viewers` (e.g., `pine`, `elm`, `mailtool`)
`browser`	`hostlist`, `port`	can connect to hosts in `hostlist` at `port`, can access display; can exec `viewers` (e.g., `lynx`, `hotjava`, `trn`)
`info-provider`	`hostlist`, `port`, `dir`	can accept connections at `port` from hosts in `hostlist`; can read (but not write) files in `dir`; cannot access display; cannot exec processes (e.g., `fingerd`, `rwhod`, `ftpd` allowing gets)
`server`	`hostlist`, `port`, `dir`	can accept connections at `port` from hosts in `hostlist`; can read (but not write) files in `dir`; cannot access display; can `transformers` (e.g., `httpd`)
`shell`	`mapfile`, `classlist`	can read `mapfile`, can exec binaries given in `mapfile` (the corresponding sandbox is also specified); `classlist` can be used to limit the behavior classes permitted; cannot access network; cannot access display (e.g., `ksh`, `csh`, `tcsh`)
`game`		can access display; cannot access network; cannot exec processes
`applet`	`host`, `port`	can access display; can connect to `host` at `port`; cannot read/write files; cannot exec processes.

Table 1: Parameters and descriptions of the behavior classes identified by this study. In addition to the parameters listed above, each class has two additional parameters – the `read-home` and `write-home` directories. An application has read and execute access to files in its `read-home` and write access to files in its `write-home`.

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)
accept	used to control connections from remote hosts (e.g., accept allow udp 128.111.*.*:513)
putenv	used to add a variable definition to the environment (e.g., putenv HOME=/tmp/boxedin)
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 2: Brief description of the sandbox description language.

applications that attempt to access these files, many don't actually require information from them and are able to operate in the presence of such redirection (e.g., `ghostview`).

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. 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.

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[7] and read a platform-dependent set of configuration files.[8] 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. Appendix B 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.

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)

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

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

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 (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 directory to the application's write-home directory, and closing all unnecessary file descriptors. It then exec's the application to be confined. If no write-home directory is specified, a temporary directory is created in /tmp and is deleted after the confined program terminates.

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 [2, 10] and for user-level extensions to operating systems [1]. This interface allows us to intercept system-calls both on their entry to and exit from the operating-system. It also provides a structure containing 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 [4].

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/momletter.txt can be a symlink to /etc/passwd. To plug this hole, MAPbox completely resolves each filename (using resolvepath()) 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 on the stack of the confined process,[9] changes the register holding the system-call argument to point to this string and allows the operation 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 sandbox specification language can specify the confinement require-

[9]This is implemented using pwrite() and ioctl()s on the /proc file corresponding to the confined application.

ments for most but not all system-calls. For the remaining system-calls, MAPbox implements an application-independent policy. It does not intercept system-calls related to signals, threads and virtual memory and 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 at 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.[10] 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 [3]. `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.

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

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

Figure 3: Syntax for `.mapcap` entries. The first two arguments in an entry are mandatory and specify the `read-home` and `write-home` directories respectively. They can, however, be left empty.

```
filter(,)                 /fs/play/~acha/mapbox/sandboxes/filter.box
transformer(,,%a,%a)      /fs/play/~acha/mapbox/sandboxes/transformer.box
browser(,%c,*,80)         /fs/play/~acha/mapbox/sandboxes/browser.box
game(,)                   /fs/play/~acha/mapbox/sandboxes/game.box
```

Figure 4: Sample `.mapcap` file. The arguments in the entry for browsers specify that: (1) a new directory is created as the `write-home` directory and and that browsers are allowed to connect to any host on port 80. The empty entries for filters and transformers specify that these applications are not allowed to have either of `read-home` and `write-home`.

4 Configuration and administration

There are two ways in which MAPbox can be configured. First, by specifying which behavior classes are 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 behaviors: the list of behaviors 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 a behavior class with values for all its parameters. The MAP-types specify the behavior classes that the user is willing to allow and the acceptable combinations of their parameters. A parameter can be specified using as a concrete value, a regular expression, a numeric range, or a list. Multiple combinations of acceptable parameter values for a behavior class 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-value `%a`. Some behavior classes (e.g., `browser`) need a writable directory (to generate logs etc) but do not care where this directory is located. For such classes, the `write-home` parameter can be specified as `%c` with the semantics that a new directory is created for this purpose. The syntax for `.mapcap` entries is presented in Figure 3. A sample `.mapcap` file is presented in Figure 4.

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. This enforces the "no-home-directory" specifications (e.g., `filter(,)`).

- `%a` and `%c` can only be matched 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(,%c,www.cs.ucsb.edu,80)` would match the `.mapcap` file in Figure 4 whereas `browser(,%c,*,*)` 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.

5 Evaluation of MAPbox

We evaluated MAPbox from two different perspectives. First, we evaluated the effectiveness of the MAPbox approach for confining commonly used applications. For this, we tried to confine a large set of commonly used applications using suitable class-specific sandboxes. The goal of these experiments was to determine the effectiveness of this approach as well as of the class-specific sandboxes we have developed. Second, we evaluated the efficiency of the MAPbox implementation. For this, we ran experiments with interactive as well as non-interactive applications. The goal of these experiments was to determine the overhead seen by real applications due to confinement.

5.1 Effectiveness of MAPbox

For these experiments, we selected 30 commonly used applications.[11] We used the workloads developed for the resource requirements study to drive these experiments. For confining each application, we used the sandbox corresponding to its intuitive behavior class (`compiler` for `gcc`, `editor` for `gnu-emacs`, `browser` for `Netscape` etc). Of the 30 applications used in these experiments, 18 applications[12] are able to complete their corresponding workloads while confined. The remaining 12 are unable to complete their workloads for one or more of the following reasons:

Lack of a `pwd` system-call: several applications try to determine the current working directory by walking up the directory hierarchy. Figure 5 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. Three applications from our test suite, `cc`, `gcc` and `gnu-emacs` failed to complete their workloads due to this limitation. Determining the working directory is central to the functioning of many applications and is usually a safe operation by itself. However, operations on relative paths are may not be safe. From this, we conclude that a `pwd` system-call would be a useful addition from a confinement perspective.

Access to sensitive files: several applications are not able to function if they are denied access to sensitive files. Of the applications in our test suite, two, `lynx` and `Netscape` did not complete

[11] The applications used these experiments were `finger`, `telnet`, `ftp`, `lynx`, `Netscape`, `cc`, `gcc`, `javac`, `latex`, `dvips`, `vi`, `pico`, `gnu-emacs`, `sh`, `ksh`, `tcsh`, `ghostview`, `pageview`, `pine`, `elm`, `xcalc`, `xclock`, `xterm`, `idraw`, `xfig`, `xv`, `ical` `grep`, `compress` and `gzip`.

[12] These were `finger`, `telnet`, `ftp`, `latex`, `dvips`, `compress`, gzip, `grep`, `vi`, `pico`, `sh`, `ksh`, `xcalc`, `xclock`, `xterm`, `ghostview`, `ical` and `idraw`

```
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 5: System-call trace excerpt illustrating the `pwd` pattern.

their workloads due to this reason – `lynx` tried to access the password database via a `door()` call whereas `Netscape` needed access to `/etc/passwd` and `/etc/mnttab`.

Attempt to run setuid root program: `xterm` failed trying to invoke `pt_chmod` which is a `setuid` root program. We cannot trace `setuid` root programs using the `/proc` interface and do not allow confined application to run such programs.

Attempt to scan $PATH: `tcsh` did not complete its workload as it tries to scan all the directories in its PATH environment variable. Based on the Principle of Least Privilege, the sandbox for the `shell` behavior class disallows this operation (since `sh` and `ksh` are able to work without this).

Attempt to perform prohibited X operation: `xv` failed when it tries to scan the entire window hierarchy of the 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.

Attempt to access an unanalyzed device: currently, `pine` and `elm` fail as access to `/dev/ticosord` is denied. We do not yet understand the operation of this device and, therefore, have denied access to it.

5.2 Efficiency of MAPbox

For these experiments, we selected two sets of applications. The first set contained six commonly used non-interactive applications. The second set contained three commonly used interactive applications. The applications and their 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 repeated each experiment five times and reported the average. We conducted these experiments on a lightly loaded SUN Ultra-1/170 with 64 MB and Solaris 2.6. All files involved in these experiments were in the OS file-cache. We used the Solaris high resolution timer `gethrtime()` for all measurements.

Table 3 presents the results for the non-interactive applications. We found that the overhead caused by MAPbox for non-interactive applications varies greatly – from about 1% for `gzip-1MB`

application	type	workload
`gcc`	non-interactive	compile 10 C files, about 5000 lines of code
`latex`	non-interactive	compile 5 tex files, each about 300 lines
`dvips`	non-interactive	convert a 50 page DVI file to postscript
`ftp`	non-interactive	ftp 10 files of 32KB each
`gzip-1MB`	non-interactive	compress 4 1MB files
`gzip-8KB`	non-interactive	compress 32 8KB files
`grep`	non-interactive	search gcc source for *int*, 182 files
`vi`	interactive	write a C program to print first n fibonacci numbers
`pico`	interactive	write a C program to print first n fibonacci numbers
`ical`	interactive	scan six months of events, add one event per month

Table 2: Workloads for the test applications. The two gzip workloads were selected to compare the confinement overheads for processing a few large files and for processing many small files.

application	total time	total time with MAPbox	time in MAPbox	other overhead
`gcc`	14.5s	20.49s (41%)	4.41s (30%)	1.58s (11%)
`latex`	2.80s	3.06s (9%)	0.17s (6%)	0.09s (3%)
`dvips`	2.88s	3.26s (13%)	0.11s (4%)	0.27s (9%)
`ftp`	1.99s	2.32s (17%)	0.17s (9%)	0.16s (8%)
`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%)
`grep`	2.72s	2.76s (1.2%)	0.02s (0.6%)	0.02s (0.6%)

Table 3: MAPbox overhead for the non-interactive applications. 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.

and `grep` to 41% for `gcc`. To determine the cause of this variation, we analyzed the system-call traces for the test suite. We found that the applications that had large overhead due to MAPbox made frequent calls to file-system-related system-calls (`open`/`stat` etc). To obtain a fine-grain breakdown of this overhead, we added additional time 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). From this we conclude that the overheads are mostly due to operations that are necessary for confinement. We would like to point out that the overheads for five out of the seven test cases were below twenty percent which is not an unreasonable tradeoff for safe confinement.

It is difficult to precisely measure the MAPbox overhead for interactive applications since most of the time is spent waiting or polling for human input. As an estimate, we measured the time spent in MAPbox (for `ical`, an X application, we also measured the time spent in `Xbox`). Table 4 presents the results. These results suggest that the time spent in MAPbox for interactive applications is a

application	total time	time spent in MAPbox and Xbox
`vi`	145s	102ms
`pico`	170s	100ms
`ical`	400s	600ms

Table 4: Time spent in MAPbox and Xbox for interactive applications.

negligible fraction of the total execution time. Furthermore, there was no perceptible degradation of interactive response. To estimate the cost of additional context-switches due to intercepted system-calls, we added a counter to MAPbox to count the number of intercepted calls and reran the experiments. We then multiplied the number of intercepted calls by two times the cost of a single context-switch (kernel to MAPbox, MAPbox to kernel). We used `lmbench` [19] to determine the cost of a single context switch. In all cases, the estimated overhead due to additional context-switches was less than 10ms.

6 Potential limitations

We believe that the MAPbox approach provides a better tradeoff between configurability and ease of use than previously proposed approaches. Nevertheless, it has several potential limitations. In this section, we briefly discuss the potential limitations of this approach.

Portability: MAPbox depends on the ability to intercept system-calls for implementing a secure reference monitor. Currently, only Solaris and Linux allow system-call interception in this manner. As far as we know, this technique cannot be used for Windows 95/98. We are currently trying to figure out if it is possible to implement similar functionality in Windows NT.

Applications limited to single behavior: the MAPbox approach limits each application to one of the behavior classes. Many applications, however, are able to exhibit multiple behaviors. As yet, we do not know how to safely allow a single application to simultaneously belong to multiple behavior classes. In specific cases, it may be possible to create customized classes that allow a specific pair of behaviors. In general, however, we believe this is an inherent trade-off and that not all applications can be securely confined in all their generality. In particular, applications such as `gnu-emacs` which contain a powerful extension language.

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 and configure behavior classes and the corresponding sandboxes, it is conceivable that everyone defines different classes 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 experience with MIME-types which have a similar potential for configuration chaos. Instead of a "tower of babble", the set of MIME-types used by most users has converged to a fairly stable set.

Need for additional provenance: while MAPbox can allow users to confine applications to individual behaviors, it has no way of ensuring that they perform this behavior in a manner that is acceptable to the user. For example, recent events have shown that documents generated by Microsoft applications can have a unique user-specific ID embedded in them – which may or may not be desirable behavior from the point of view of the user. We believe that digital signatures can provide additional provenance that might help make such distinctions in some cases. We are considering ways in which digital signatures could be incorporated into the MAPbox approach.

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 while providing significantly more configurability. Based on a study of 50 commonly used applications, we have been able to identify fourteen behavior classes – `filter`, `transformer`, `compiler`, `editor`, `viewer`, `download`, `upload`, `mailer`, `browser`, `info-provider`, `server`, `shell`, `game`, `applet`. These classes have intuitive meaning and their 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 clean categories. We expect this classification to be refined based on further experience.

To evaluate the effectiveness of MAPbox, we tried to confine a set of 30 commonly used applications using suitable class-specific sandboxes. We found that 18 of them ran successfully. of the 12 that did not, 10 failed because they make potentially unsafe accesses and only two failed because of configuration problems (access to a device that we have not yet analyzed). From this, we conclude that the approach of using behavior-class-specific sandboxes is fairly effective in confining commonly used applications.

To evaluate the efficiency of MAPbox, we ran experiments with interactive as well as non-interactive applications. We found that the overhead caused by MAPbox for non-interactive applications can vary greatly – from about 1% for `gzip` and `grep` to 41% for `gcc`. On the other hand, the time spent in MAPbox for interactive applications is negligible. Also, in our experiments, there was no perceptible degradation of interactive response. From this, we conclude that the overhead of confinement is likely to be acceptable.

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 their lucid description in [10]. 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. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the 1997 USENIX Annual Technical Conference*, 1997.

[2] 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.

[3] Anonymous. The Xbox distribution. Available at __.

[4] Anonymous. __. Technical report, Dept of Computer Science, University of __, 1999.

[5] Caldera security advisories. Available at *http://www.calderasystems.com/news/security/index.html*, April 1999.

[6] CERT/CC Advisories 1988-1999. Available at *http://www.cert.org/advisories*, April 1999.

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

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

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

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

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

[12] The Happy99 worm. Available at *http://www.symantec.com/avcenter/venc/data/-happy99.worm.html*, 1999.

[13] Information on Back Orifice and NetBus. Available at *http://www.symantec.com/avcenter/-warn/backorifice.html*, 1999.

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

[15] P. Karger. Limiting the damage potential of the discretionary trojan horse. In *Proceedings of the 1987 IEEE Syposium on Research in Security and Privacy*, 1987.

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

[17] 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.

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

[19] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *In Proc. of 1996 USENIX Technical Conference*, Jan 1996.

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

[21] The Melissa virus (CERT Advisory CA-99-04-Melissa-Macro-Virus). Available at *http://www.cert.org/advisories/CA-99-04-Melissa-Macro-Virus.html*, 1999.

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

[23] L. Stein. SBOX: put CGI scripts in a box. In *Proceedings of USENIX'99*, 1999. To appear.

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

[25] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for Java. In *SOSP 16*, 1997.

[26] 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.

A Grammar for the sandbox description language

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 | *

Note that the `define` and `params` commands are not included in the above description. These commands are implemented as macros in a preprocessing step.

B A common specification file for Solaris 5.6

```
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_FILES /etc/netconfig /etc/nsswitch.conf\\
/etc/.name_service_door

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

C Sandbox example

```
# sandbox  spec for the browser class
# the browser sandbox takes four arguments -- the read and write
# home directories, the list of hosts it is allowed to connect to
# and the port(s) it is allowed to connect to.
params _READ_HOME _WRITE_HOME _HOSTLIST _PORT

# set up the env variables
putenv PATH=$_READ_HOME
putenv TERM=$_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 $_READ_HOME

# _COMMON_WRITE can be written by all (in this case /dev/zero)
# downloaded files are to be in _WRITE_HOME
path allow write $_COMMON_WRITE $_WRITE_HOME
```

```
# browsers are allowed to read X data files and libs
path allow read $_X_FILES

# browsers are allowed to read network config files and libs
path allow read $_NETWORK_FILES $_NETWORK_LIBS

# browsers are allowed to connect to all hosts in the argument
connect allow tcp $_HOSTLIST:$_PORT

# 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
```