Behavior-based Confinement of Untrusted Applications

Mandar Raje
Department of Computer Science
University of California, Santa Barbara

TRCS 99-12

January 1999

Abstract

In my thesis, I propose a class-specific sandboxing mechanism to confine untrusted applications. The key idea is to identify different application classes like editor, browser, mail client, shell, filter, server etc and to confine applications belonging to each class in a sandbox that is tailored to the expected behavior/requirements of the class. For example, the sandbox for a MIME-mail client could be restricted to allow it to spawn only a set of helper applications explicitly listed in the mailcap file; the sandbox for an editor could be restricted to disallow network accesses and process creation. Such a mechanism retains the ease-of-use of sandboxes while significantly increasing their flexibility. End-users do not need to maintain complex access control lists or interact frequently with the security subsystem; nor do they need to depend solely on a digital signature. They can configure their systems by specifying the set of classes they would like to allow.

To evaluate the feasibility of my proposal, I have: (1) defined a set of application classes and have populated them based on a study of system-call traces of commonly used applications; (2) implemented a infrastructure that uses the /proc-interface to confine native binaries; (3) developed configuration files for the different application classes that I have encountered; (4) have integrated this infrastructure with an X proxy that confines untrusted X applications to windows and other X resources that it creates (and a small number of global attributes); (5) evaluated the overhead introduced by this mechanism.

Contents

Chapter 1

Introduction

In the fall of 1988, a graduate student at Cornell University introduced a computer *worm* in the Internet. It replicated uncontrollably for several days. Eventually, it infected over six thousand computers nationwide and overwhelmed their processing capabilities. The worm exploited three major weaknesses of Unix systems. The first of these was in *fingerd*. The bug involved overwriting the daemon's input buffer, which resulted in transferring control to a shell that was controlled by the worm. The second weakness was in *sendmail*. By using sendmail's debug option, the worm issued a set of commands instead of user addresses. The third major weakness was poorly chosen *passwords* and the fact that most of the encrypted password files were publicly readable.

In another incident, a remote administrative tool called *Back Orifice* was developed for Windows. This tool is distributed in the form of a *trojan horse*. Once it is installed, a remote operator anywhere in the Internet can gain access to the system and can do anything that the user can do – all without any outward indication of her presence. She can change, delete or rename files, change preferences, change the window registry or can trash it altogether. Every time the *infected* system is booted, this tool started itself and listened on a port for commands.

In both these incidents, an *untrusted* program which somehow managed to run on user systems as a user process, attacked the user systems. By *untrusted*, we mean a program originating from a different administrative domain than that of the user with whose identity it runs. Today, running such programs is central to many technologies and applications. Some of these applications are as follows:

- Web servers often execute *cgi-scripts* while serving http requests. Users like to install their own programs, scripts (games, for example) on web servers so that they can be accessed through browsers. The commercial drive to customize web content makes extensive use of CGI-scripts. Here cgi-scripts are the untrusted programs which run in the server's environment.

- Audio and video data on the Internet comes in a variety of formats. Each comes with its own installable *plug-in*, a program that understands it. These programs need to access the audio or video devices and display information like fonts, colormaps, etc.

- The text *documents* that we download do not always contain passive data. For example, Microsoft word documents contain macros, which must be interpreted by the system of the user downloading the document; Postscript, is a language in itself. Downloading and viewing the documents amounts to running untrusted programs.

- *MIME* (Multipurpose Internet Mail Extensions) is designed to represent a variety of data formats. Data with these multiple formats can be sent over mail and can be viewed by de-multiplexing the documents to helper applications. MIME-enabled mail clients can execute shells, Tcl and other external programs such as ftp.

- The goal of *global computing systems* [19] is to allow different users to participate in very large computations by downloading code from brokers and running it on their systems. This way, large applications can make use of thousands of idle machines in the Internet. The applications need memory and network access.

- The evolution of *active networking* research will permit installation of network services inside the network fiber. The idea is to process data as it passes through the network. This allows network services (format converters, filters, etc) to be installed at the intermediate network nodes. For this technology to evolve, services developed must utilize router resources and occasionally reroute the packets they process. Network services might use the network to dynamically load the code or they might use the filesystem to save state of the packet belonging to the same application.

In all these scenarios, systems need to run untrusted processes. Security mechanisms for such processes usually choose to be conservative and deny access to all but a few local resources (eg.Java1.0 [6]). Condor [27], for example, migrates system calls to the original host and limits access to local resources. This limits the set of applications that can be executed.

The goal of the work discussed in this dissertation is to develop a more flexible mechanism to control accesses to local resources. Instead of applying a uniform security policy to every untrusted application, we believe that applications can be partitioned into different classes based on their behavior such that a separate security policy is used for each class.

To understand the behavior of applications and to determine the resources they need, we analyzed system call traces of a variety of applications. We collected the information about the files accessed, libraries linked, binaries executed, display usage, environment variables. We used this information to define behavior classes such as filter, transformer, compiler, editor, viewer, mailer, browser, server, shell and determine the access control and resource requirements for each class.

We have designed and implemented a confinement mechanism which monitors untrusted applications by intercepting calls they make. Each application is confined in the confinement environment for its class. Individual requests are allowed or denied based on the class specification. Such a mechanism retains the ease of use of more restrictive mechanisms (such as Java1.0) since users only need to specify the behavior classes they wish to allow. Users do not need to maintain complex access control lists, nor do they need to depend solely on digital signatures. Note that class-specific confinement can be combined with digital signatures for accountability.

In chapter 2, we describe the study of applications. In chapter 3, we describe the language used to specify the confinement configurations for application classes. In chapter 4, we describe the application classes we have identified. In chapter 5, we describe a framework that uses confinement configurations to control access requests made by untrusted applications. In chapter 6, we talk about the performance overhead introduced by such a mechanism. In chapter 7, we discuss related work.

Chapter 2

Study of Application Behavior

To understand the behavior of applications we studied their system call traces. We summarized the low-level operations recorded in these traces to form higher-level operations such as (1) accessing files (2) linking libraries (3) making network connections (4) accepting network connections (5) forking processes (6) handling signals.

2.1 Workload

While conducting the study, we considered different functions that each application is capable of performing. Applications need different resources while performing different functions. A finger client, for example, reads the *passwd* file only if a domain name is not specified, which in a normal case it would not access. We first describe the workloads we used for different applications. We then describe how we summarized the resulting traces.

To design the workloads, we started with an intuitive notion of application behavior classes such as: editors, viewers, compilers, mailers, etc. For each class, we defined a series of workloads in increasing degree of complexity. This facilitates the analysis; once we have analyzed the less complex behaviors the more complex behaviors are easier to analyze. For example, we first analyze a trace of an editor invocation that does not edit a file. It provides the information about the *initialization segment* of the editor. Once we have this information, it becomes easier to observe other resources that the editor needs in order to edit a file. In the following section, we list all applications that we study and the workloads for each of them.

2.1.1 Editors

We studied three text editors `vi, pico, xemacs` and two graphical editors `idraw, xfig`. The workloads for editors were: (1) start up an editor and exit; (2) start up with an existing file and exit; (3) start up the application with an existing file, modify and exit; (4) for text editors, edit a file, spell-check it and exit; (6) for graphical editors, generate a postscript file and exit.

Many editors are capable of performing other functions, eg. `emacs` can run a debugger, interact with a shell, etc. We did not consider these functions relevant to the editor behavior and therefore did not include them in the workloads. On the other hand, spell-checking is useful functionality for text editor and we included it for our work-load.

2.1.2 Viewers

We studied two document viewers: `ghostview, pageview` and two image viewers `xview, imagetool`. The workloads for the viewers were: (1) start up a viewer application and quit; (2) start up an application with an existing file and quit; (3) print a file; (4) change the orientation of (document) file or the size of an image file; (5) save an image in a file; (6) for image viewers, grab a region, try one of the algorithms (oil painting, for example). There are many image processing algorithms which can be applied and we found that these algorithms have the same resource requirements.

2.1.3 Compilers

We studied five compilers: `cc, gcc, c++, f77` and `javac`. The workloads were: (1) invoke a compiler without any file; (2) compile a source file into an object file; (3) compile a source file into a binary file; (4) compile a file and link some object files; (5) compile a file that uses libraries; (6) compile a file and link it to libraries; (7) compile multiple files.

2.1.4 Mailers

We studied two mailers: `pine` and `elm`. The workloads were: (1) open and close an empty mailbox; (2) open and close a mailbox with one message; (3) delete one (the only) message in the mailbox; (4) list a mailbox with multiple messages; (5) refile a message, with mailbox empty and not empty after the refile; (5) send an existing file; (6) send a message with and without using an alias; (7) forward a message; (8) reply to a message.

2.1.5 News client

We studied one news client `trn`. The workloads were: (1) start up the newsreader using default profile and exit; (2) get the headers new news in one news group and exit; (3) get the headers of one news group, read all messages of that news group and quit; (4) subscribe to a news group; (5) unsubscribe to a news group and quit.

2.1.6 Browsers

We studied three browsers: `lynx, netscape` and `hotjava`. The workloads were: (1) start a browser with no arguments and quit; (2) start a browser with a known page and quit; (3) start a browser, follow one html link and quit; (4) follow one *.ps* link, quit `ghostview` and quit the browser; (5) follow one *.tar.gz* link, download a file and quit.

Browsers are capable of spawning helper applications (`ghostview, xview`). We believe that the pattern for spawning all these applications will be similar. We did not analyze the traces produced by the helper applications when we studied the browsers; these applications were studied as a part of their own classes.

2.1.7 Shells

We studied three shells `sh, csh` and `tcsh`. The workloads were: (1) start a shell without reading profile files and exit; (2) start a shell in the default mode and exit. (3) use a shell for filename completion; (4) use a shell history; (5) run a simple shell script. (7) use pipes, redirection and background operators;

2.1.8 Network Clients

ftp client

We studied the Solaris ftp client in the following situations: (1) login to some ftp server and log out; (2) list a set of files on remote server using `ls`; (3) download multiple files on local machine; (4) upload multiple files.

finger client

We studied the Solaris finger client in the following situations: (1) finger without arguments; (2) finger mraje@cs.ucsb.edu (local domain); (3) finger acha@cs.umd.edu (remote); (4) finger acha@cs.umd.edu mraje@cs.ucsb.edu; (5) finger -l mraje@cs.ucsb.edu.

telnet client

We studied the Solaris telnet client in the following situations: (1) telnet to a remote node and exit; (2) telnet to a remote node, read a file and exit; (3) telnet to a remote node to port 80 and exit.

2.1.9 Other applications

We also studied a suite of other applications: `ical`, `xcalc` `latex`, `make`, `xbiff`, `xclock` and `xterm`.

2.2 Collection and analysis of traces

The traces were collected using a system command called *truss*. The argument to truss is an executable or a process-id. It executes the command or attaches itself to the process with the specified id. It produces a trace of system calls the process performs, the signals it receives, and the machine faults it incurs. Each line of the trace output reports either a fault, a signal name or a system call name with the arguments and return values. It can also trace system calls made by all children of this process.

We summarized these traces by identifying groups or *patterns* of system calls and relating them to higher-level operations. In some cases, to verify the mapping between a higher-level operation and the system calls it generates, we wrote small programs and compared their traces with that of the application being studied.

Table 2.1 lists all the applications and also gives the length of the trace for each application. The lengths are given for the simplest behavior. For the most part, traces are composed of the patterns that we discuss in the next section. We discuss additional (relatively rare) patterns in the subsequent sections.

Application Class	Application	Lines in traces
Text Editor	vi	210
	pico	187
	emacs	567
Graphics Editor	xfig	1352
	idraw	1236
Browser	lynx	230
	hotjava	36988
	netscape	25044
Viewer	xview	2828
	ghostview	6420
Mailer	pine	747
	elm	803
Newsreader	trn	454
Shell	ksh	157
	sh	274
Compiler	cc	277
	gcc	265
	g++	609
	javac	6174
Network client	finger	130
	telnet	412
	ftp	253
Other apps.	dvips	1054
	latex	2256
	make	230

Table 2.1: Length of system call traces for some applications.

```
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 2.1: *Loading a library.*

2.2.1 Example: finger client

We present the process of summarizing system call traces using the trace for the Solaris finger client as an example. Most of the patterns in the traces are straightforward. We can determine what the application is trying to do just by observing the pattern. Patterns that fall in this category are loading of dynamic libraries, network communication using sockets, creating a process, reading-writing files, setting signal handlers, and polling the standard input.

Loading libraries

Figure 2.1 presents an example using *libsocket.so.1*. This pattern represents the process of linking the dynamic library. The `open` call opens the library file. The `fstat` call obtains the information about this file. The pair of `mmaps` map the library into the process's address space; `close` closes the file. This shows how dynamically linked libraries are mapped into a process's address space.

Accessing network configuration file

Figure 2.2 presents a trace fragment illustrating the pattern. The netconfig database contains information about networks that are connected to the system. The `open` call opens it for reading, the `fstat` gets its file system information. The sequence of `reads` and `llseeks` is where the file is actually read. Finally, the file is closed.

Network communication

This pattern is illustrated in Figure 2.3. The client connects to the finger daemon using the `so_socket` and `connect` system calls. This creates a socket descriptor (4)

```
open("/etc/netconfig", O_RDONLY)               = 3
fstat64(3, 0xEFFFE638)                         = 0
read(3, " #n #   T h e   " N e t".., 8192)     = 1064
read(3, 0x00029D5C, 8192)                      = 0
llseek(3, 0, SEEK_CUR)                         = 1064
llseek(3, 0, SEEK_SET)                         = 0
read(3, " #n #   T h e   " N e t".., 8192)     = 1064
read(3, 0x00029D5C, 8192)                      = 0
llseek(3, 0, SEEK_CUR)                         = 1064
close(3)                                       = 0
```

Figure 2.2: *accessing network configuration file.*

```
so_socket(2, 2, 0, "", 1)                       = 4
connect(4, 0xEFFFED40, 16)                      = 0
write(4, " m r a j e", 5)                       = 5
write(4, "rn", 2)                               = 2
read(4, " L o g i n   n a m e :  ".., 8192)     = 401
read(4, " N o   u n r e a d   m a".., 8192)     = 331
read(4, 0x0002CF2C, 8192)                       = 0
llseek(4, 0, SEEK_CUR)                          Err#29
close(4)                                        = 0
```

Figure 2.3: *communicating on the network.*

```
llseek(0, 0, SEEK_CUR)                          = 6928
write(1, " [ c s . u c s b . e d u".., 728)     = 728
```

Figure 2.4: *printing results.*

```
open64("/etc/.name_service_door", O_RDONLY)     = 3
fcntl(3, F_SETFD, 0x00000001)                   = 0
door_info(3, 0xEF6A9370)                        = 0
door_call(3, 0xEFFFCA98)                        = 0
```

Figure 2.5: communication using door.

which is used to communicate with the server. The next two `writes` send the user-id (mraje) and the domain name (`cs.ucsb.edu`) arguments to the server. The subsequent reads correspond to the result of the query. Once all the data is read, the socket is closed.

Print results

Once the application (finger client) has all the information, it prints it on `stdout`. This is illustrated by pattern in figure 2.4.

2.2.2 door calls

Door is a family of system calls that provides a new flavor of interprocess communication between processes on the same hosts. The file `/etc/.name_service_door` is a door to the naming service cache daemon (`nscd`). The `nscd` provides caching for passwd, group and hosts databases through standard `libc` interfaces like `gethostbyname` and `gethostbyaddr`. Calls to these interfaces get translated into door calls. In the context of the current application, `door_call` is trying to find the IP address of the host 'cs.ucsb.edu'. To understand more about door calls, we wrote a short program that calls `gethostbyname, gethostbyaddr, getgrnam,` and `getpwnam` and observed the traces for this program. These traces had a common door pattern (shown in figure 2.5). The type of the function invoked at the server end is passed within a compound argument, and is not visible in the

```
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 2.6: *pwd pattern.*

traces.

2.2.3 The pwd pattern

This pattern consists of directory reads, starting with the current working directory and going one step at a time to the root directory. See figure 2.6 for an abridged version. It gets the information about each directory in the path and all files in it using `getdents`. We compare this pattern with that of system command 'pwd' and found it to be similar.

2.2.4 Communication with the X server

One trace generated when an application connects to the X server is shown in figure 2.7. Applications use /tmp/.X11-pipe, a FIFO, to communicate with the X server. The connection with the X server is established with the `open` system call. X requests are sent with `write` and `writev` system calls and replies and events are received using `read` and `readv` system calls.

```
open("/tmp/.X11-pipe/X0", O_RDWR)          = 9
fstat(9, 0xEFFFC970)                        = 0
fcntl(9, F_SETFD, 0x00000001)              = 0
writev(9, 0xEFFFCEC0, 4)                    = 48
fstat64(9, 0xEFFFCD50)                      = 0
fcntl(9, F_SETFL, 0x00000080)              = 0
read(9, "0100/v/0/0/0 D", 8)               = 8
```

Figure 2.7: *X communication pattern.*

2.2.5 Other patterns

There are some patterns idiosyncratic to individual applications. For example the Solaris C compiler opens a socket to a licence server to check licensing information. Other compilers (eg. gcc) do not generate such a pattern. In the definition of application behavior classes we ignore these patterns, as we believe they are not a part of the core functionality of the applications.

Chapter 3

Confinement Configuration Language

In this chapter, we describe the language we use to specify the resource constraints for different application classes. Resource constraints can be specified at the individual system call level or at the resource level. In the first section we describe the choices for the abstraction level. Next, we describe the language we use, using BNF notation and the constructs in the language. We present a sample configuration in section three. In the last section, we discuss other languages and their relation to our proposed language.

3.1 Abstraction Level

Since all resources are accessed using system calls, one alternative for a configuration language would be to allow or deny individual system calls. For example, if a user wants to protect the file */etc/passwd*, she can specify that certain calls (`open, access, stat, creat, chmod, link, unlink, symlink`) are to be denied when the file argument is */etc/passwd*. The other alternative is to specify access-control at the level of individual files, hosts, ports, display, etc. Allowing the user to specify access controls for individual system calls provides flexibility at the cost of being less intuitive. However, our analysis of system calls traces indicated how this additional flexibility is not useful and the access controls at the resource-level for most of the part are adequate. As a result we have chosen to base our language around resources instead of system calls.

Commands	Resource
`path,rename`	File system
`connect,accept`	Network, Display
`putenv`	Environment Variables
`childbox`	Process's child

Table 3.1: Mapping between commands and resources

3.2 Constructs

The language consists of six constructs, each one controlling a resource. Table 3.1 gives the mapping between the commands and the resources.

The grammar for the language using BNF notation is given in figure 3.1.

3.2.1 *path*

The *path* command controls access to the file system. Each path command makes a part of the file system visible or invisible, by either allowing or denying access to the files or directories provided as the arguments. The usage is:

```
path allow | deny mode-list path1 path2 ...
```

The mode-list consists of read, write and exec. All combinations of these modes are allowed. Currently, all relative pathnames are disallowed. Many applications, however, need the working directory and therefore, make `stat` calls with relative pathnames, which are denied. This problem can be remedied by providing `pwd` as a system call. Use of wild cards is also allowed for specifying the path argument. For example:

```
path allow read,write /tmp/*
```

allows the process to read and write to all descendents of /tmp.

Figure 3.2 illustrates the use of path. It presents the path commands for the Solaris finger client.

command	:=	path_c | rename_c | connect_c | accept_c | putenv_c |
		childbox_c
path_c	:=	path permission access_modes dir_list
rename_c	:=	rename dir_pair_list
connect_c	:=	connect permission protocol ip_addr ”:” port_addr
		| connect allow display
accept_c	:=	accept permission protocol addr_list ”:” port_addr
putenv	:=	putenv name_val_list
		| putenv DISPLAY
childbox	:=	childbox class
permission	:=	allow | deny
access_modes	:=	access_modes ”,” access_modes | access_mode
access_mode	:=	read | write | exec
dir_list	:=	directory dir_list | directory
dir_pair_list	:=	dir1 dir2 dir_pair_list | dir1 dir2
protocol	:=	tcp | udp | *
addr_list	:=	ip_addr addr_list | ip_addr
ip_addr	:=	dot_addr ”/” dot_addr | dot_addr
port_addr	:=	port_num port_mask | port_num | /* empty */

Figure 3.1: Grammar for the language using BNF notation.

```
path allow read /dev/zero
path allow read /etc/netconfig /etc/.name_service_door
path allow read libsocket.so.1 libnsl.so.1 libsec.so.1 libc.so.1
path allow write /dev/zero
path allow read,exec /usr/bin/finger
```

Figure 3.2: *Configuration for finger client.*

3.2.2 *rename*

A lot of applications access files like /etc/passwd, /etc/utmp, /etc/services which contain sensitive information. The applications are designed to check the existence of such files using the `access` system call. It is potentially dangerous to let applications read these files. The solution we propose is to trick applications into reading some other files in place of the original files. This is done using the `rename` command. The syntax is:

```
rename file1 newfile1 [....  filen newfilen]
```

When an application tries to access file1, it is replaced by newfile1 and the process is allowed to continue.

```
rename /etc/passwd /tmp/dummy
```

In the example above, `rename` will result in all accesses to /etc/passwd being redirected to /tmp/dummy. Note that this approach does not always work. For example, applications such as `pine` and `lynx` access `/etc/passwd` to find out the login-id of the user running them and cannot work when the accesses to `passwd` are redirected.

3.2.3 *connect*

This command controls the network connections made by the application. The connections are granted or denied based on the IP address of the destination and its port number. The syntax is:

```
connect allow|deny tcp|udp|* ip-addr[/ip-addr-mask] [:port[/port-mask]]
```

For example,

```
connect allow udp 128.111.40.28
```

allows all udp connections to the host with 128.111.40.28. In another example,

```
connect allow tcp 128.111.52.15:8080
```

allows only tcp connections to 128.111.52.15 on port 8080 (web server). In addition, masks can be used to match a range of IP addresses. The matching algorithm takes the requested IP address, bitwise ANDs it with the mask, and tests whether the result is equal to the ip-addr field. i.e.

```
connect allow tcp 0.0.0.0/0.0.0.0:80
```

allows all `tcp` connections to port 80 at any IP address, and

```
connect deny * 128.32.0.0/255.255.0.0
```

disallows all connections (`tcp` or `udp`) to any port at any IP address of the form 128.111.x.x. Note that IP addresses must be used, domain names are not allowed.

3.2.4 *accept*

The *accept* command is similar to the connect command. It controls server applications. It specifies the hosts that are allowed to connect to the server application. The usage is:

```
accept allow/deny tcp/udp (ip1, ip2, ..., ipn)/* :port[/port-mask]
```

The example which allows two specific hosts to connect is:

```
accept allow tcp 128.122.52.15 128.111.40.28
```

In addition, we can specify the port the server application can listen on. A special value for the port field is NON_SYSTEM_PORT, which means the application must select a port which the system is currently not using. If port number is omitted, then server can listen on any port. The arguments consist of the protocol string, `tcp` or `udp`, and a list of IP addresses a server can accept connections from. Wildcards can be specified to indicate that the server is allowed to accept connection from any host and any port.

3.2.5 *putenv*

The untrusted application's execution environment can be controlled using the *putenv* command. The syntax is:

```
putenv name=value
```

This manipulates the untrusted application's environment. Here `name` is the name of the environment variable and the `value` is the value we want it to be set to. For example,

```
putenv PATH=/usr/bin:/usr/local/bin/
```

sets the PATH environment variable. In the above example, putenv command sets the PATH for the confined programs. Providing appropriate values for PATH is important, since the application is allowed to run binaries from the directories in the PATH. Other important environment variables include LIBDIR for libraries, HOME which is the home directory for untrusted application and TMP which points to the temporary directory. All environment variables must be explicitly included using `putenv`.

3.2.6 Display

We can enable access to the display by using two separate commands. First, we need to set the DISPLAY environment variable, because it determines where the application connects, to communicate with the X server. We achieve this using the *putenv* command in the following way:

```
putenv DISPLAY=unix:4
```

The next thing is to allow connection to the X server. We use *connect* command in the following way:

```
connect allow display
```

This would allow connections to the X filter `xbox`. The `xbox`, interposed between X clients and the X server, selectively filters the requests generated by the clients. We provide an application independent X access control policy of confining applications to access their own resources.

3.2.7 *childbox*

Several applications spawn child processes as a part of their operation. These processes may belong to a different application class than their parent. For example, helper applications spawned by browsers are usually viewers. We can use `childbox` command to specify the child configuration. The usage is:

```
childbox class
```

For example:

```
childbox viewer
```

indicates that all processes spawned by the process being confined are confined in a viewer confinement environment.

3.2.8 Non-configurable system calls

The constructs described in the previous section control access to the filesystem, network and environment. There are other resources provided by the operating system which cannot be specified using this language. Examples include signals, threads, doors and memory. For all these resources, a common application-independent access-control policy is used. For some of these resources like signals and memory, the security provided by the operating system is sufficient. Others like doors, ioctls and fcntls are selectively allowed based on the arguments. Only the calls that we were able to ensure to be safe are allowed.

- The `door` calls are primarily used to query information from the host, group and password databases. We allow interaction only with the host database.

- System call `fcntl` provides control over open files. The call looks like: `int fcntl(int fildes, int cmd, /* arg */ ...);`

  The available values for cmd include F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETOWN, F_SETOWN and other values.
  F_DUPFD, F_DUP2FD which return new file descriptors and F_GETFD, F_SETFD which read and write file descriptor flags are allowed. We have not seen a need for the remaining flags and have, as a result, disallowed them.

- The call `acl` can get or set a particular file's access control list. The filename is provided as one of the arguments. Clearly, changing the access control list by untrusted applications is not safe, so we disallow that. Two other options are reading access control list or reading the number of entries in the list, both of which are allowed.

- The call `ioctl` performs a variety of control functions on devices and streams. Handling ioctls requires a good understanding of the devices and their controls. Currently, we allow a small number of ioctls that we have seen a need for and that we have determined to be safe. We disallow all other ioctls.

- Calls like `read, write, lseek` are never monitored since the protection provided by monitoring `open` is sufficient. Similarly the calls related to memory mapping like `mmap, munmap, mcntl` and calls related to the signals, `signal`, and `sigaction` for example, are not monitored since the protection provided by the operating system is sufficient.

- System calls that can be invoked only by the processes with super-user privileges (eg. mount, umount, plock, acct, etc.) are denied.

3.2.9 Other considerations

In this section we discuss a few additions we have made to our language, in order to make it more configurable.

Home directory

We observe from the traces that a lot of applications use `.rc` files to store user preferences across invocations. For example *csh* uses .cshrc file which contains the initial configuration, the initial values for the environment variables, command aliases, etc. Many of the applications also need some file system space to create, read and write temporary or permanent files. To cater these needs, we allow applications to have their own HOME directory. They can read or write into this directory. The home directory can be set using the *set* command. For example,

```
set HOME /usr/home/mraje
```

sets the home directory to `/usr/home/mraje`. The question that arises is, what happens to the files written by the application after it completes the execution. We let the user decide whether to keep those files on the disk or to erase them after the application has executed.

Defining symbolic constants

To make the configurations portable across multiple platforms, we provide a set of symbolic constants that a user can define in a site specific manner. For example, if an application is allowed to access the network then it must read some files and load some libraries which provide the socket calls. For Solaris 2.6 these files would be /etc/netconfig and /etc/nsswitch.conf and libraries would be libsocket.so.1 and libnsl.so.1. We specify these files using the following primitives:

```
define _NETWORK_READ /etc/netconfig /etc/nsswitch.conf
define _NETWORK_LOAD /usr/lib/libsocket.so.1 /usr/lib/libnsl.so.1
```

Now when defining applications, a user can use the symbolic constants like _NETWORK_READ and _NETWORK_LOAD instead of specifying all the files. Other constants defined are COMMON_PATH, COMMON_READ, COMMON_WRITE, COMMON_EXEC, COMMON_NETWORK_READ and COMMON_X_READ. All such constants are defined in a single file which can be used in creating a configuration file for applications. All the site specific details can be put into this file, which makes the

```
# configuration file for application finger client
# syntax: finger username@hostname

# can set the home to save states after execution.
set _APP_HOME /tmp

putenv PATH=_COMMON_PATH
putenv HOME=_COMMON_HOME
putenv LD_LIBRARY_PATH=_DEFAULT_LD_LIBRARY_PATH

# includes the default directory for read, write and execute.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow exec _COMMON_EXEC

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

#should be allowed to connect to IP addresses
# specified in the definition.
connect allow tcp 128.111.0.0/255.255.0.0:79
```

Figure 3.3: *Configuration for finger client.*

configurations portable across platforms. Users are free to create new symbolic constants, if needed.

3.3 Example configuration

We now present a complete configuration of the *finger client* using the language we have proposed (figure 3.3).

The configuration starts with the *set* command which sets home to /tmp, thereby allowing a finger client to read or write into /tmp. Using the *putenv* commands the environment for the finger client is set. Most of the commands are self explanatory, except the *connect* command. It is configured so that the finger client can connect to any host inside *cs.ucsb.edu* domain and can only connect to port 79, where the finger daemon is supposed to be running.

We describe the configurations for different application classes in the next chapter.

3.4 Related work

In this section, we briefly describe languages proposed by other researchers and compare their approach with ours.

`Janus`, which is a system developed at Berkeley, is designed to confine the helper applications spawned by browsers. Our language is based on the configuration language of `Janus`. We use the *path, connect* and *putenv* constructs which are originally proposed by them. To this, we have added the *rename, accept* and *childbox* constructs.

In an alternative approach, Ko and Fink at UC Davis present a method for detecting exploitations of vulnerabilities in privileged programs by monitoring their execution using audit trials. For controlling access to the filesystem, they define a set of operations on files viz. read(file), write(file) and execute(file). To allow an application to read the file */etc/netconfig* we would say: *read("/etc/netconfig").* They also associate certain attributes with a file which are: its name, its uid, its gid and its access modes. Additional constraints can be specified using these attributes. For example, we can specify:

```
exec("/usr/bin/mail") :- U.uid = 0;
```

For the network part, Ko and Fink have a command *bind* which is similar to our *connect.* For example, bind(25) will ensure that a mailer connects only to a mail port, and nowhere else. Since they focus on local *privileged* applications, they do not verify the IP address. They also don't differentiate between the client and server behavior of the applications. They do not provide any support for monitoring accesses to the display. On the other hand, their language is more powerful than ours as it allows regular expressions and predicate logic.

In USTAT [4], which was developed at UCSB, penetrations are represented by state diagrams. The system predefines sets of files. The sets are are Restricted read files, Restricted write setup files, Files authorized to read Fileset, Non-writable file executables etc. Using these sets, they define various intrusion scenarios in form of a table or state transitions which at runtime can be matched against the audit trails to find out a possible attack. The language is limited to specifying conditions in terms of files and user-ids and group-ids. The system is being extended to detect network-based intrusion detection (Netstat). A language for state transition representation of scenarios in intrusion detection systems is also being developed at University of California [20]

Several other researchers have proposed languages to allow users to specify access-control policies and framework to compose these policies [14] [15] [16] [17]. Three of them [15] [16] [17], propose logic-based declarative languages and use inference mechanisms of various sorts to compose policies. Blaze et al. [14] propose a language that contains both assertions and procedural filters.

Chapter 4

Behavior Classes

This chapter describes the behavior classes we have identified. We started the task of identifying behavior classes with an intuitive notion of program behaviors. We refine the classes based on the information gleaned from system call traces. Most of the behavior classes are defined with parameters. The parameters limit the scope of the applications. For example, an `editor` is parameterized with a set of files which it can edit. Those are the only files the editor is allowed to edit. Similarly, the class `network client` is parameterized by the port it is allowed to connect to - a mailer can be restricted to port 25, whereas a finger client can be restricted to port 79. We first describe the common symbolic constants used by many configurations. We then describe the classes we have identified.

4.1 Symbolic Constants

As discussed in the previous chapter, we allow every untrusted application to have its own directory called *home* directory. The application can read or write any file in that directory. Also, there are some environment variables like PATH, LD_LIBRARY_PATH, some libraries and files that are common to the applications. They are defined separately as symbolic constants and those constants can be used to specify the configurations of individual classes. This makes the individual configurations portable. Please see Appendix B for the values of these constants for Solaris 2.6.

COMMON_PATH: This is the common value for the `PATH` environment variable. This is the path along which the executable will be searched, when `exec` is called.

By default COMMON_PATH is set to the *home* of the application.

COMMON_LD_LIBRARY_PATH: This is the common value for the `LD_LIBRARY_PATH` variable. This is the path along which the libraries are searched. For Solaris 2.6, it is set as:

```
define COMMON_LD_LIBRARY_PATH /usr/lib:/lib:/usr/ucblib
```

Note that the actual libraries that can be loaded are still decided using the *path* command.

COMMON_READ: This is the common set of files that can be read, and the common set of libraries that can be linked. By default, all applications are allowed to read files from their home directory. So home directory is included in this set. On Solaris 2.6, it also contains */dev/zero* which is used by all the applications for mapping an unbacked segment. The libraries that are included are `libc, libdl, libelf, libm, libmp, libsec`, etc. For a detailed listing, please see Appendix B.

COMMON_WRITE: This is the default set of files that can be written by any application. This set includes */dev/null* and all the files in the home directory of the application.

X_READ_SET: Applications that are allowed to connect to the X server need access to X libraries which contains X library calls and information such as fonts, shapes, etc. For Solaris 2.6, X_READ_SET is set to `/usr/openwin/lib`.

NETWORK_READ_SET: When an application communicates over the network, it needs to access files like `/etc/netconfig` and `/etc/nsswitch.conf` and libraries like `libsocket, libnsl` which are included in this set.

4.2 Behavior Classes

In this section we describe different behavior classes with examples of each class. This description is meant to give a broad idea of behaviors included in each class. The configurations for all behavior classes can be found in the appendix A.

4.2.1 *filter*

This is the simplest type of behavior class in terms of access control requirements. It takes its input from the standard input, processes it and prints the output to the standard output. Applications belonging to this class cannot access the network or the display. They also cannot *exec* any processes.

Example applications are unix filters `sed`, which is a stream editor, `grep` which is used to search a file for a pattern, `sort` which sorts text files and `comm` which finds lines common to two files.

4.2.2 *transformer (input, output)*

The transformer class is very similar to the filter class, the only difference being that the applications in this class take their input from a disk file, specified as the first argument and write the output to a file, specified as the second argument. As with `filter`, the applications cannot access the network or the display and cannot *exec* any processes.

Example applications are utilities such as `compress, gzip`, etc. Other important applications which belong to this class are format converters. For example, audio or video data can be converted from one form to the other. (jpeg to .bmp, for example).

4.2.3 *compiler (directory / list of files, libpath, output)*

This class is similar to the transformer class in a sense that it takes the input specified by the first argument and generates an output file. There are, however, two differences. The first difference is that the input consists of multiple files or a directory containing these files. Second, files that are not mentioned on the command line can be accessed and used to generate the output - for example, `libc.a` for the compiler. As with the `transformer` class, the applications cannot access the network and the display. Compiler applications may access libraries which must be present in the directories contained in the libpath.

Example applications are `cc, gcc, f77`. Other examples of this class are `latex, tar` and `dvips`. Latex is a formatting and typesetting tool. It takes single or multiple tex files as input and generates a .dvi file as its output.The `tar` command takes a directory as an argument and creates an archive of all the files in that directory.

Some compilers like the Solaris C compiler open a socket connection to exchange some licensing information. We do not consider this to be a part of *compiler* class and disallow all network connections.

4.2.4 *editor (dir/list of files)*

Applications belonging to this class can edit files from a list of files or a directory. Editors have read and write access to these files. They can't access the network. They have access to the display.

This class covers both text and graphic editors. Both types of editors require helper applications. For example, text editors require spell checkers ispell, typesetters and formatters such as troff and other utilities like `tee, expr` and `deroff, sort`. Graphic editors need access to graphics-specific libraries and fonts. For Solaris 2.6, these libraries are contained in the directory `/usr/openwin/lib`. The symbolic constant which defines the path for the libraries is X_COMMON_READ.

Examples of the text editors are `vi, pico` and `emacs` and graphic editors are `xfig, idraw`.

4.2.5 *viewer (directory / list of files)*

This class is similar to the *editor* class, except that it can only read files that it has access to but cannot modify them. The files that can be read by the applications are given by the argument, which can either be a directory or a list of files. Viewers are not allowed to access the network. Access to the display is allowed. The applications are not allowed to *exec* a process.

The examples include `xv, ghostview` and `pageview`. Though these viewers can actually write files into directory and have file browsers associated with them, we do not consider this as a part of safe viewer behavior.

4.2.6 *download (host, dir, port)*

Applications belonging to this class can get information by connecting to the host specified in the argument. The information downloaded is put in the directory specified by the second argument. The applications are given write permissions to this directory. The applications are not allowed to *exec* any processes, nor can they access the display. Example applications are `finger client` and `ftp client`.

Depending upon the port number, different *download* applications can be differentiated. For example, finger client connects to port 79 whereas the ftp client connects to port 21. Actually, ftp can also upload files, but we allow only ftp *get* behavior as a part of the download class.

4.2.7 *upload (host, dir, port)*

Applications belonging to this class can upload information by connecting to the host specified in the argument. The information uploaded is present in the directory provided by the second argument. Read-only access is given to the directory. The applications are not allowed to *exec* their own processes, nor can they access the display.

Again, depending upon the port number, applications of this type can be differentiated from one another. One example is an `ftp client` which uploads files using the *put* command. It connects to the port 21. An example is a `turnin` client, which uploads files by connecting to a specific port.

4.2.8 *mail client (mailbox)*

Examples of this applications class are `pine, elm` and `mailtool`. These applications fetch mail from the mailbox. They need to have read, write permissions for the mailbox (which is a file). Mail clients are also capable of sending emails using either *sendmail* or by connecting to the mailer daemon. This is the only connection allowed. Access to the display is allowed. Mail clients are allowed to spawn processes. The spawned processes can only be of the type *viewer* - this allows mailers to display most but not all MIME types (application/tcl and application/csh, for example, cannot be handled).

4.2.9 *browser (list of hosts, port)*

Applications belonging to the browser class can *connect* to any of the hosts specified in the argument. They can download information and store it into their home directory. They can connect to the X server. They are allowed to spawn any process which is of the class *Viewer*. The applications can connect to the port specified as a parameter.

Example applications are `netscape, lynx` and `hotjava`. Most of these applications spawn viewer applications such as `xv, ghostview`. Applications of this class are allowed to spawn children of class `viewer`. Netscape and lynx open files such as `/etc/utmp` which has user and accounting information, `/etc/passwd` which is a password database, and `/etc/mnttab` which has mount table information. We *rename* those files with dummy files.

4.2.10 *information provider(list of hosts, dir, port)*

Applications in this class are allowed to accept network connections from the hosts specified in the list. Information is contained in the directory specified by the second argument *dir*. The applications have read-only access to the information. They do not have access to the display. They cannot *exec* any processes. The application is allowed to bind to and listen on just the port specified as a parameter.

A typical example of this class is the `turnin server` which accepts a set of files from different users. Other examples include `ftpd, fingerd`, game servers and teleconferencing server which coordinate between different clients.

4.2.11 *server(list of hosts, dir)*

This class is similar to the previous class except that applications belonging to this class can *exec* processes. These processes must be of the type *transformer* which is discussed earlier. Applications have read-only access to the information contained in the directory. They do not have access to the display.

Example applications of this class includes `web-servers`. They can spawn cgi-scripts which access the local information and generate the output. Other examples include web proxies.

4.2.12 *shell (mapfile, list of behaviors)*

Applications of the type *shell* can execute binaries. The binaries which are allowed to be executed and their corresponding behavior(s) are given in the mapfile specified as the argument. Complete pathnames must be specified for the binaries. The behaviors that are allowed to be executed are listed in the list of behaviors. The examples of this type are `tcsh, csh` and `ksh`.

4.2.13 *game*

Game is a special type of application. It typically has access to the audio device and the display. We consider only single user games, so disallow games from accessing the network. Games are also not allowed to *exec* any processes. Examples are `pacman, doom-II`, etc.

4.2.14 *applet*

This is the class that has access controls similar to that provided for the applets in Java 1.0. The applications of this type, are allowed to access display, are allowed to connect only to the host where they come from. All other network accesses are denied. Access to local files are denied. The applications are not allowed to *exec* any process.

4.3 Relationships between classes

Now let us understand how these classes are related to each other. We start with the `filter` class, since it needs the least resources. Applications of this class take input from the `stdin` and send output to `stdout`. The applications cannot access local files or the network. The `transformer` class is a superset of the filter class, since it takes its input from a file and generates its output to a file. This means, if we allow applications of the transformer class to run, we also allow applications of filter class to run. Similarly, `compiler` class is a superset of the transformer class, since it can access multiple files and libraries. Notice that all the three classes discussed do not access the network, the display and except for the compiler applications, others can not `exec` any process.

Applications which belong to the `viewer` class can display a set of files. They cannot modify those files, nor can they `exec` any process. What differentiates them from the three classes discussed above is that they have access to the display. The class `editor` is a superset of the class viewer, since it can modify the files. Applications of this type can also `exec` editor-related executables.

All the classes discussed above do not access the network. The applications which access the network can be divided into two categories: network clients and network servers. `Browser, upload` and `download` are the classes which are network clients. Upload and download can connect only to a specific host and send or receive the information. Browsers on the other hand, can connect to multiple hosts and can also spawn helper applications. Network server classes are `information provider` and `server`. The difference is that the server can spawn processes and information provider cannot.

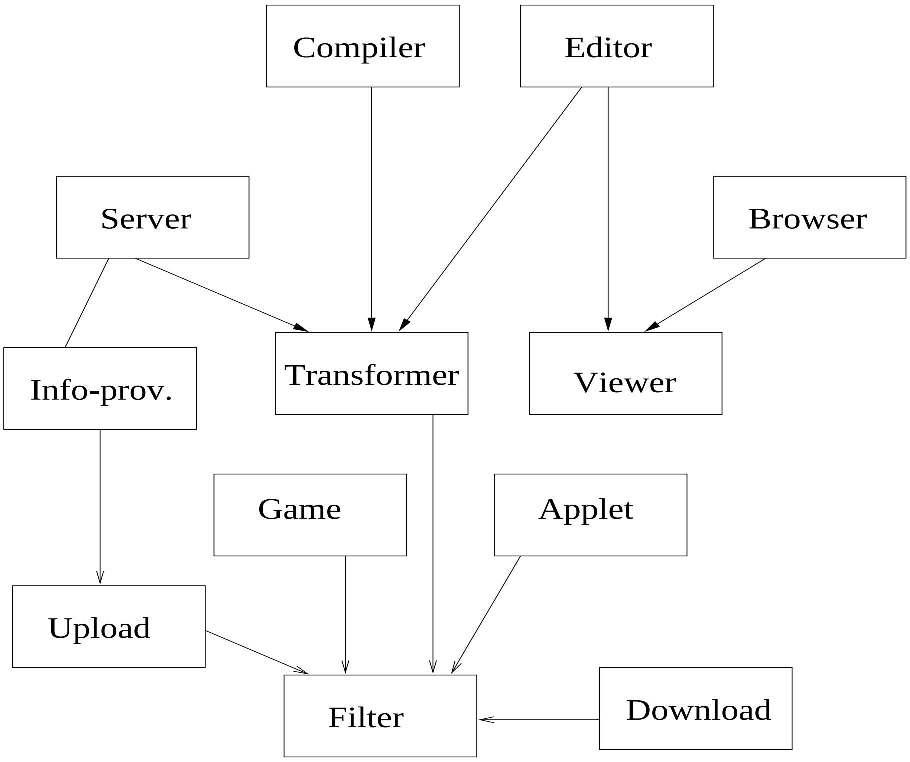

Figure 4.1: Relationship between Classes.

Chapter 5

Implementation

We have implemented a confinement system called `mapbox` which confines native binaries. In this chapter, we describe how our system works. We start by discussing the mechanisms used to intercept system calls. We then explain how we handle intercepted system calls. In the next section, we explain the operation of our system describing flow of control for a single system call. Next we describe how mapbox handles non-routine system calls such as `door, fcntl` and `ioctl`. Finally, we describe how mapbox can be used to confine untrusted applications.

5.1 Interception mechanism

The crux of the implementation lies in intercepting system calls and allowing or denying them based on their arguments and the security policy applicable to the class. To intercept system calls before they are executed, we can do static library interposing, binary editing or we can use the `/proc` interface provided by several operating systems.

Library interposing: We can intercept system calls by interposing a library that implements the generic system calls API. It can check the operations and then use the actual `syscall` interface to invoke the system calls. The advantage of this technique is performance. It has a low overhead for intercepting calls. Its disadvantages are: (1) applications cannot always be re-linked and (2) applications can bypass the library using the same `syscall` interface used by the library. For example, an application, instead of calling *open* can call *syscall* with SYS_OPEN as argument, which essentially has the same effect.

Binary editing: Binary editing can be used to insert checks before the relevant system calls. For example, for every `connect` system call in the binary, we can insert a call to a function which checks the IP address. The advantage of such a mechanism is its low overhead. The disadvantage is that we cannot have a dynamic library loaded at runtime. Everything must be linked statically.

`The /proc interface`**:** In our implementation, we use the interception mechanism provided by the `/proc` interface which has a higher overhead compared to the other two mechanisms but is more general as it allows dynamically linked libraries and cannot be bypassed. `/proc` is a file system that provides access to the state of each process in the system. Files in /proc are named after the process-ids (pids) of each process running on that system. Standard system calls like `open, read, write, close` can be used to access `/proc` files. `/proc` also contains the image of all running processes in the system and can be used to read or write data into address space of the individual process. Information and control operations can be performed through `ioctls`. To give an idea of how this interface is used in intercepting system calls, we will go through the relevant part of the implementation.

An untrusted application runs with a process-id, say PID. To get access to its image `mapbox` opens a file `/proc/PID` using `open` system call. Then it initializes two sets of flags. One is `sysentryset`, a set of system calls that it intercepts on their entry to the kernel. The other is `sysexitset`, a set of system calls that it intercepts on their exit from the kernel. Not all information needed by mapbox is available on the entry to the kernel. For example, a call like `accept` is blocking and the IP address of the client is available only when the call exits.

```
ioctl(tracedfd, PIOCSENTRY, &sysentryset);
ioctl(tracedfd, PIOCSEXIT,  &sysexitset);
```

As shown above, after the sets are initialized, control instructions (ioctls) are given to the process file (with id `tracedfd`), asking it to stop on encountering those system calls listed in the entry and the exit set.

Whenever the operating system stops the untrusted application, say because it encountered some call listed in one of the sets, it passes the information about the stopped process (untrusted application) in a structure called `prstatus`. This structure contains information about why the application was intercepted (entry or exit in our case), what system call was intercepted, its arguments, and the return value if the call was trapped on exit. It also contains other information like signals, process-ids, group-ids, stack size etc. which is not used by `mapbox`. Once `mapbox` has the information about the system call intercepted and its arguments it can take the appropriate action, ie. either allow or deny the system call. If the system call is to be denied, mapbox sets a flag in the structure `prstatus` asking the kernel to abort the call.

5.2 Handlers

Every call trapped has a handler associated with it. When the operating system passes the information related to an intercepted call to mapbox, mapbox invokes the appropriate handler. Handlers make use of the policy structure to decide if the call is safe.

Policy structure, which is a part of mapbox, is a collection of data structures that contain the information specified in the configuration. It consists of a set of lists:

1. read-list : files/directories readable by the application.
2. write-list : files/directories writable by the application.
3. execute-list: binaries executable by the application.
4. connect-list: IP addresses to which the application can connect.
5. accept-list : IP addresses of authorized clients.
6. rename-list : files/directories that must be renamed.

These lists are initialized by mapbox using the configuration file. For example, the following command will add */tmp/* /usr/home/docs* to the read-list.

```
path allow read /tmp/* /usr/home/docs
```

Now, if */tmp/foo* is being opened for reading, the open handler will scan the list to check for the file. If the entry for that file or appropriate directory is found in the list, then the corresponding action (allow or deny) is returned.

`mapbox` starts by reading the configuration file whose location can be specified on the command line. For each command specified in the configuration, appropriate initialization routines are invoked. After the initialization, mapbox is ready to run the application that is to be traced. A child process is created and child's state is cleaned up. This includes setting umask to 077, setting limits on virtual memory use, disabling core dumps, changing to the application HOME directory and closing unnecessary file descriptors. The `mapbox` then starts executing the application.

5.3 Flow of control

To illustrate the operation of `mapbox`, we trace the flow of control for a single system call.

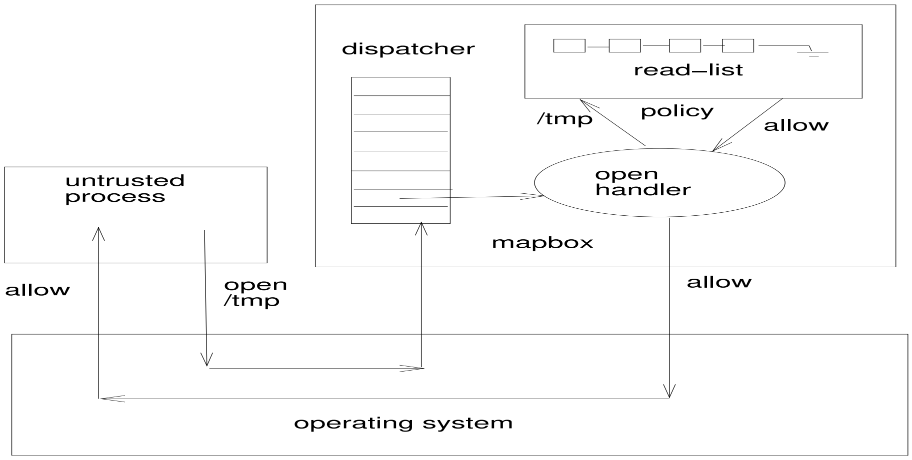

Figure 5.1: Flow control.

Suppose an application makes an `open` call. First, the call enters the kernel. Now since a handler for that call is registered, all the information related to the call is bundled into the structure `prstatus` and is passed to `mapbox` by the operating system. The `mapbox` then invokes the `open` handler. The open handler, depending upon the arguments (name of the file and access mode) and the entries in the appropriate list (read, write or execute), decides to allow or deny the call. If the call is to be denied, then `mapbox` sets the abort flag and resumes the interrupted application.

There are two extra context switches involved in this process. One switch happens when the operating system passes the control to `mapbox` passing all the information about the call. And the other switch happens when `mapbox` passes the control back to the operating system. If binary editing or library interposition is used as the interposition mechanism, these extra context switches can be eliminated.

5.4 Intercepted system calls

The set of system calls which are intercepted is decided by the commands used in a particular configuration. Each command (path, rename, connect, accept and childbox) is mapped into a set of handlers.

5.4.1 path

This command controls access to the file system. It maps into a set of handlers which handle the file related system calls. In this section we discuss such calls and also the criteria for allowing or denying them.

open: This system call opens a file for the access specified by the mode in the argument. The mode can be read-only, write-only, read-write, append or create. This call succeeds iff there exists a path command that permits access (with the appropriate mode) to the desired file.

creat: This system call creates a file in the specified directory. This call succeeds iff there exists a path command that allows reading and writing in the directory in question.

symlink: `symlink` call has two arguments, name1 and name2. It creates a symbolic link name2 to the file name1. If this call is to succeed, then there must be path module(s) giving write access to file name2 and read, write and execute access to file name1.

exec, execve: Each of these calls executes a binary file. Both read and execute permissions to the file must be given by a path command in order for this call to succeed.

pathconf, stat, lstat: The `pathconf` call returns the current value of a configurable limit or option associated with a file or directory. The `stat` and `lstat` calls get the status of a file provided as an argument. To get these values, the file must be made readable by a path command.

access: This call checks the file for accessibility according to the bit pattern contained in the second argument. The access permissions to be checked are R_OK, W_OK and X_OK (read, write and execute respectively) or the existence test F_OK. To check for read, write or execute permissions, the corresponding permissions must be given by a path command.

link, rename, unlink, rmdir, utime, utimes, chmod, chown, mknod: The `link` call creates a new link for an existing file and `unlink` deletes this link; `rename` renames the file by a name specified by the second argument; `utime` and `utimes` set the access modification times for the file; `chmod` changes the access modes and `chown` changes the owner of a file; `rmdir` deletes the directory entry for the file and `mknod` makes a special directory entry. All these system calls need write permissions to the file.

One important thing to note is that the path or the filename provided as an argument to these calls can be a symbolic link. This can be a security hole. For example, a file `/tmp/poem` can be a symlink to the file `/etc/passwd`. Thus all read requests on `/tmp/poem` will access the password file. To prevent such a thing, we use the `resolvepath` procedure to determine the actual file which will be accessed. The call is allowed to proceed only if the appropriate permissions are given for the actual file.

5.4.2 connect, accept

These commands control access to the network. They map into a set of handlers which handle the network-related system calls. In this section we discuss such calls.

socket, socketpair: The socket call creates a `socket` for communication. Basically, if `accept` and `connect` do not occur in configuration file, then the application is not allowed to communicate over the network. In such a case `socket` call will be denied. The `socketpair` call creates an unbound pair of connected sockets. It is handled in a similar manner.

connect: This call requests that a connection be made on a socket. The destination address is specified as an argument. This address must be specified by a `connect` command. Protocol listed in the command must be `tcp`. The port specified in the argument and that specified in the connect command must match. Otherwise the call will be denied.

bind: This call assigns a name to an unnamed socket. This is used by server applications. The address is checked for based on `accept` commands. If a match for an address, port and protocol is found, the call is allowed to continue.

sendto recvfrom: These calls occur in both client and server-side code. The `sendto` call sends a `udp` packet to the IP address specified in the argument. The `recvfrom` call receives a `udp` packet from the specified IP address. The connection must be explicitly allowed by either connect or accept for this call to succeed.

accept: This call is made only by server side applications. It accepts a new connection from a client and creates a socket to communicate with the client. The `accept` system call is intercepted on its exit from the kernel and the address of the client is checked for in a list of hosts specified in the `accept` command.

Only if the address specified in the argument is present in the list and if the port numbers match, `accept` is allowed.

5.4.3 putenv, childbox

A new copy of `mapbox` is created for every child of an untrusted process. This requires system call `fork` to be intercepted on its exit from the kernel. The return value of the fork is the process-id of the child. A new copy of mapbox starts monitoring the child. The behavior class for the child might be different from that for the parent. This class is specified by the `childbox` command. If the two classes are different, the `exec` handlers, must reinitialize the policy structures for the child.

The `putenv` command does not require any handlers. The environment variable specified in the command is set for the untrusted process before it is created. For example, the following command will set the PATH variable.

```
putenv PATH=/usr/bin:usr/local/bin
```

5.5 Special cases

There are some cases which are handled differently. They are (1) connection to the display (2) `ioctl` calls (3) `door` calls and (4) the `rename` command.

X clients use the DISPLAY environment variable to make connection to the X server. DISPLAY holds information about the IP address or unix domain socket where the X server listens. We have designed and implemented an X filter, which listens on a specific port. To make untrusted clients connect to the X filter and not to the actual X server itself, we set the DISPLAY environment variable for the untrusted process. The X filter monitors the messages between clients and the server. The design of X filter is discussed in the next section.

The other special case is the `ioctl` handler. These calls perform a variety of control functions on devices and streams. The arguments of these calls are file descriptors of the open devices and commands associated with a particular device. To handle `ioctls`, we would be required to keep track of all open devices and streams and to associate the commands with the right device. We would also need a priori information as to which commands for each device are safe and which are not. In our implementation, we choose to be conservative and allow `ioctls` related only to `stdin` and `stdout` which are known to be safe.

As discussed in chapter 2, the door family system calls get information from the host, group and password databases. This is the new IPC mechanism introduced by Sun Microsystems and is still in its experimental stages. Currently, we allow applications only to interact with the host database.

Implementing `rename` command requires `mapbox` to write a string into the process's memory. This string is the name of the file which substitutes the actual file. Using `ioctls` on the `/proc` file for the process, we write the string on the process's stack. The address of the stack is provided in the information passed by the operating system to `mapbox` as a field in the `prstatus` structure. Changing the filename also requires changing the pointer to point to the new string. This can be done with the help of ioctls which *get* and *set* the register which holds the pointer.

5.6 X protocol filter

The X server is designed to facilitate the cooperation between X clients. This is done for two reasons: (1) to make clients independent of the window configuration and (2) to allow different clients to communicate with each other (eg: for cut-and-paste). This design has security implications when untrusted X clients are allowed to run with other X clients.

5.6.1 Resources

There are six different types of resources that X supports. These are `window, pixmap, cursor, font, graphic context` and `colormap`. An X client, during its lifetime, creates resources, alters them or destroys them. Every resource is identified with an ID. To reduce the network traffic between X clients and X server, the resource related messages that travel from the clients to the server are indexed using the resource ID. Server has all the information about different resources created by the clients.

There are four different types of messages that travel between the clients and the server: (1) Protocol *requests* are generated by clients and sent to the server; (2) protocol *replies* are sent from the server to a client in response to certain requests; (3) *events* are sent from the server to clients and contain information about a device action or about a side effect of the previous request and (4) *errors* which tell the client that a request was invalid.

5.6.2 Potential problems

Each resource has a set of requests associated with it. For example, for windows, we have the following requests: CreateWindow, ChangeWindowAttributes, GetWindowAttributes, DestroyWindow, ReparentWindow and Mapwindow. If an X client knows the ID of a window created by some other client, it can issue the aforementioned requests on that window. Thus, it can manipulate that window even if some other client created it. Clearly, this is not desirable for untrusted applications. Another outcome of this mechanism is that untrusted X clients can listen to events for other X clients. They only need to know the ID of the target windows to listen to their events.

Another possible attack would use requests like GrabServer, ChangeKeyboardMapping, SetScreensaver, SetFontpath, etc. These requests are global in that they modify properties global to X server or make exclusive use of the server.

X clients communicate with each other using the protocol request, `SendEvent`. Selections are passed using this request. Selections can be faked, such that malicious X clients can make other clients pass certain information.

5.6.3 X filter

X filter is an interposing agent between an X client and the X server. It listens to all communication between an untrusted X client and the X server. It filters out all *unsafe* requests. Events and errors pass unfiltered. To secure the system, from the attacks discussed in the last section, X-filter applies the following policies:

- A client is allowed to access only resources that it has created. For example, if a client has created a window with id 344, then only that client can access the window.

- All requests that have global impact are disallowed.
- Selections are disallowed.

Appendix C at the end gives information about how each protocol message is handled.

5.7 Usage

After studying functions of the different components of `mapbox`, let us understand how we can actually use it to confine binaries. The simplest case is to invoke it directly from the command line. The usage is:

```
mapbox [-i] [-b symconstfile] [-f configfile] application
```

The `-i` option makes the application interactive. We can specify the path for the configuration file using the `-f` option. We can also specify the file with the site specific constants using the `-b` option. For example, we can run the editor `vi` using the following command:

```
mapbox -i -f $LIBDIR/editor.conf vi /tmp/foo
```

Here, LIBDIR is the environment variable which points to the directory where configuration files are stored. This application vi opens a file foo (if allowed); all the calls made by `vi` will be monitored by `mapbox` using the configuration `editor.conf`.

Since we already have a list of different classes and configurations associated with them, it would be nicer not to have to specify the configuration file everytime we invoke mapbox. In order to achieve this, we maintain an *behavior class to configuration file mapping* in the file *.mapcap*. Now the command mentioned above can be specified as:

```
box editor vi /tmp/foo
```

box is the executable which invokes the `mapbox`. Before it does that, it opens `.mapcap` and searches the entry for class *editor*, which is specified as the second argument. The entry looks like:

```
editor mapbox -i -f $LIBDIR/editor.conf HOME=/tmp DIR=/home %s
```

The first field is the name of the class (*editor* in this case). What follows is the command which is to be executed for an application belonging to that class. As discussed earlier, many of the configurations have parameters which are specified in .mapcap file. In this example, there are two parameters that are specified in the command. They are HOME (HOME directory of the application) and DIR (list of directories editor applications are allowed to edit). The *box* must translate these parameters into appropriate commands. For example, the parameter HOME=/tmp must be translated as:

```
set HOME=/tmp
```

The command the box executes then is:

```
mapbox -i -f $LIBDIR/editor.conf vi /tmp/foo
```

Chapter 6

Performance Evaluation

In this chapter, we present evaluation of the overhead of confinement using `mapbox` We first describe the experimental setup. Next, we explain various components in our system. We present two different sets of measurements: micro and macro benchmarks. Using micro-benchmarks, we compute the overhead for individual components of our system. Using macro-benchmarks, we measure the overhead for complete applications. We conclude with a discussion of the results.

6.1 Experimental Setup

We ran our experiments on a SUN, Ultra-1, 167MHZ machine running Solaris 2.6 with 64MB of memory. The measurements were taken on a lightly loaded machine. We tried to reduce load on the machine. For micro-benchmarks, we measured the overhead for individual system calls using a test program which makes the call a large number of times and computed the average time for the call. In addition we added timing code to `mapbox` to measure the time spent inside `mapbox`. For macro-benchmarks, we ran an application and measured its end-to-end latency. We wrote a generic wrapper which takes the application name on the command line, runs it and computed the latency. The time measurements were taken using the Solaris high resolution time function *gethrtime*.

6.2 Timing the components

Figure 6.1 shows the different events that occur in the system when a process makes a system call. Without `mapbox`, the call enters the kernel and the kernel executes

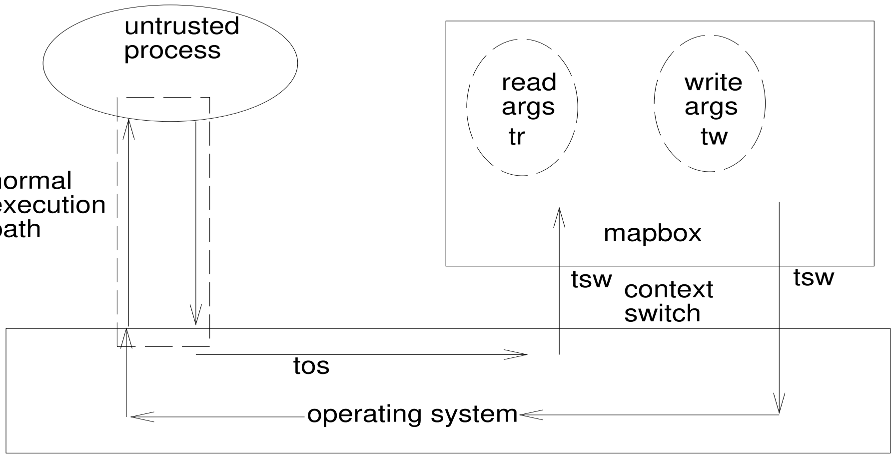

Figure 6.1: Timing Diagram.

the call before passing control back to the process. With `mapbox`, the control takes a different path. The kernel passes control to `mapbox`, if a callback for that call is registered. The `mapbox` passes the control back to the kernel asking it to either allow or deny the call. If the call is allowed, kernel executes it and passes the control back to the process. If the call is denied kernel passes control back to the process with appropriate error code. The time taken by the following events collectively constitutes the overhead.

- The operating system searches for a callback registered for the call and bundles the system call related information in the structure `prstatus`. This is the minimum cost that must be paid if we use the `/proc` interface for interception. This time is labeled `tos`.
- The operating system passes control to `mapbox` and vice versa. These are the extra context switches introduced by using `mapbox`. This time is `tsw`.
- Mapbox invokes a handler. The cost equals the cost of a function call within `mapbox`. For some system calls like `open`, we need handlers, whereas for some other calls like `mount` we do not need handlers and these calls do not incur this cost.
- A handler either reads or modifies arguments of the call. Handlers for calls like `open` read the arguments while for handling *rename*, we write into process's memory. These times are denoted by `tr, tw` respectively.

6.3 Micro-benchmarks

We selected `getpid` and `open` as the system calls to test our benchmarks. The call `getpid` is cheap as it does not have any arguments, whereas the call `open` has a string argument which needs to be read from the process's memory and is more expensive. The call *getpid* returns the ID of the process calling it. The call *open* opens a file with the path specified by the first argument and using the file access modes specified by the second. Each call was invoked a thousand times and the average time was calculated. This time without `mapbox` is denoted by `tn`. We then executed the program with `mapbox` and determined the values: (1) average time per call spent in `mapbox` and (2) average time per call as seen by the application. From these numbers we computed the time spent in the individual components mentioned in the previous section.

Workload: *getpid*

We used the following benchmarks for the *getpid* call. Table 1 gives the performance numbers.

- No callbacks were registered; null entry and exit sets were specified. Once these sets were registered with the operating system the callback search mechanism is turned on, even though the sets are empty. Since we repeatedly invoke *getpid* system calls, it tells us the overhead incurred by the operating system in checking for a callback. There are no extra context switches involved and zero time is spent inside `mapbox`. Let the time for this case be denoted by `tem`. Therefore, we have the following equation:

$$\boxed{\texttt{tos = tem - tn}} \tag{6.1}$$

  `tos` is the extra time spent by the operating system for the callbacks search.

- An intermediate variable `th` was calculated as follows. We intercepted *getpid* on its entry to kernel and the `mapbox` was set up to execute no handlers for getpid. We then used `th` to calculate the extra time spent by the operating system in packing all the information related to this call in the structure *prstatus*. We obtained the context switch time `tsw` using the following equation.

$$\boxed{\texttt{tsw = (th - tn - tos)/2}} \tag{6.2}$$

case	avg time w/o `mapbox`	avg time with `mapbox`	`mapbox` time(%)	o/s time
no callback	2.97	4.8	0(0)	4.8
entry, no handler	2.97	109.7	7.7(7)	102
exit, no handler	2.97	110.5	8.2(7)	102.3
exit, empty handler	2.97	114.6	8.8(8)	105.9
exit, read return value	2.97	128.7	9.3(7)	119.4
exit, modify return value	2.97	173.5	61.3(35)	112.1

Table 6.1: Results in μs for different cases of *getpid*

- We trapped `getpid` call on its exit from kernel. We used this benchmark to determine the difference of overhead between intercepting call at the entry and exit from kernel.
- We intercepted `getpid` on its exit and read the return value. We denoted the time taken to read the return value inside `mapbox` as `tr`.
- We intercepted `getpid` on its exit and modified the return value. This required us to modify a register from the set of registers for the process. We used two *ioctls* to do this, one which read the register set and the other which set the register set. We defined the time taken for this modification operation as `tw`.

From this we concluded that for a handler that reads the argument the end-to-end latency `tot` is given by:

$$\boxed{\texttt{tot = tn + tos + 2*tsw + tr}} \tag{6.3}$$

Based on the numbers in table 6.1, we can conclude that most of the cost of confinement for simple calls that do not read or write arguments can be attributed to the operating system.

Workload: *open*

For `open` system call, we used three different workloads. All the files were in an NFS filesystem.

- We invoked `open` with and without a handler. We also measured the overhead for `rename` command. For `rename`, a string must be written into the process

case	total time	total time with `mapbox`	time spent in `mapbox`(%)
entry, no handler	219.40	367.78(1.68)	9.919(1.05)
entry, empty handler	219.40	374.94(1.71)	12.013(1.05)
entry, read arguments	219.40	516.84(2.36)	151.47(1.69)
entry, remap file	219.40	680.08(3.1)	280.95(2.28)
gzip, 1Mb files(ms)	3823.72	3855.48(1.01)	64.68(1.02)
gzip, 32K files(ms)	1656.06	1665.55(1.01)	72.10(1.04)

Table 6.2: Results in μs for different cases of *open*. The numbers in parentheses are ratio of total time with and without `mapbox` (column 3) and overhead added by `mapbox` (column 4).

stack. This was done using a `pwrite` call. The pointer was changed using `ioctls`. The results are presented in table 6.2.

- The handler associated with `open` is more complex than the handler for `getpid`. It scans list(s) according to the access mode. (read-list, write-list and execute-list) To observe the effect of the length of this list on the cost of `open`, we varied the number of nodes in the list from 1 to 32 in multiples of 2. The results are presented in table 6.3.

- Finally, we measured the effect of intercepting `open` calls in presence of significant computations between them. For this experiment, we ran `gzip` for two datasets, 4 1MB files and 32 8KB files. Results are presented in table 6.2. Each reading is the average of a thousand consecutive runs.

From the results in table 6.2, we can say that the overhead introduced by open handler is quite high. To determine the breakdown of the overhead, we timed different components of the handler. We observe that the *resolvepath* system call, which resolves symbolic links, takes around 250 μs (65% of the time) and reading argument (filename string) from process's memory takes 150 μs (35% of the time). These two factors together account for 90% of the time spent inside `mapbox`. Therefore, in table 6.3, we see little variation in values as we vary the number of handlers.

nodes is read list	total time	total time with `mapbox`	time spent in `mapbox`(%)
1	219.40	766.66(3.49)	405.96(2.85)
2	219.40	769.26(3.51)	408.05(2.86)
4	219.40	779.50(3.55)	416.58(2.90)
8	219.40	806.20(3.67)	437.83(3.00)
16	219.40	832.42(3.79)	468.96(3.14)
32	219.40	879.88(4.01)	519.75(3.37)

Table 6.3: Results in μs for varying number of handlers. The numbers in parenthesis are ratio of total time with and without `mapbox` (column 3) and overhead addded by `mapbox` (column 4).

6.4 Macro-benchmarks

For macro-benchmarks, we ran complete applications using `mapbox` and measured the slowdown in their end-end execution time. We also gathered information about the total number of calls made by the applications and number of calls intercepted by `mapbox`.

Non-interactive applications

The applications and their workloads are listed in table 6.4. These applications were confined in the appropriate confinement environment. We used a wrapper program to accurately time the end-end execution time. We took 10 sets of measurements and eliminated the outliers. The intuition is that other programs running in the system can cause fluctuations in the values. We measured the end-to-end time and the time spent inside `mapbox`. We see for some applications (gcc), the overhead is high. This is due to a large percentage of the calls intercepted are `open, stat` and `access` calls, which require complex handling including `resolvepath` and reading the process' memory.

Interactive applications

It is difficult to measure the performance of interactive applications accurately. The main reason being the variability in human input. As a lower bound we measure

application	workload
gcc	Compile 10 C files, about 5000 lines of code.
latex	Compile 5 tex files, each about 300 lines.
dvips	Convert .dvi to .ps, about 50 pages.
ftp	ftp 10 files of 32KB each.
gzip	gzip 4 1MB files and 32 8KB files.
fgrep	fgrep gcc source directory for *int.*(182 files).

Table 6.4: Workload for non-interactive applications

application	total time	total time with `mapbox(%)`	time spent in `mapbox(%)`	ratio of calls intercepted
gcc (sec)	14.5	20.49(1.41)	4.412(1.28)	5489/13557
latex	2798.4	3058.3(1.09)	372.34(1.13)	2440/9050
dvips	2881.7	3264(1.13)	113.4(1.04)	5704/19669
ftp	1989.6	2325(1.17)	372.3(1.19)	375/1391
gzip 1MB	4260.1	4300.5(1.01)	92.168(1.02)	124/440
gzip 8KB	1518.6	2018(1.33)	228.83(1.15)	264/713
fgrep	2722.3	2755.7(1.01)	252.28(1.09)	1382/32204

Table 6.5: Results in millisecs for non-interactive applications. The numbers in parentheses are ratio of total time with and without `mapbox` and overhead added by `mapbox` Last column gives the ratio of number of calls intercepted vs. total number of calls made.

application	time spent in `mapbox`	total time
vi	86.25ms	138s
lynx	28.88ms	25s
pine	278.20ms	46s

Table 6.6: Results in milliseconds for interactive applications

the time spent in `mapbox` We tested the following cases: (1) using *vi* to type 100 words to a file, save and exit, (2) using *lynx* to open a site, follow three pointers, return to the main page and exit and (3) using *pine* to open the mailbox, read three messages, delete them, send three messages and quit. The values are given in table 6. To these values, even if we add context switch time for every intercepted call, they do not change by more than 1ms). We conclude that for interactive applications, time spent in `mapbox` is negligible.

6.5 Conclusions

From the measurements, we conclude that, for individual calls, the extra time spent by the operating system cost dominates the overhead. For the worst case, `getpid` which takes 3μs to execute, requires 100μs to be spent in the context switches. The percentage overhead for complex calls like `open` is less compared to that for `getpid`, but is still substantial. These measurements suggest that it would be expensive to intercept a large fraction of the calls made by applications. The results from the macro-benchmarks, however, show that for most of the applications only a small number of calls are intercepted (4% for `fgrep` to 40% for `gcc`). Accordingly, the overhead introduced by using `mapbox` varies from 1% for `fgrep` to 41% for `gcc`.

Applications which incur relatively higher overhead make lot of file related calls (`open, stat, access`, for example). Resolution of the paths using `resolvepath` takes about 65% of the total time spent inside `mapbox` for these calls. Another 25% of the time is spent in reading the pathname from the process's memory. If library re-linking or binary editing is used for interposition then the cost of context switches the path can be eliminated completely (reduction in the overhead from 41% to 30%). If `mapbox` is implemented inside the kernel, the cost of resolution can be reduced considerably (from 41% to 10%). In this case the extra context switches and the path resolution are avoided completely.

Chapter 7

Related Work

We divide the related work into four sections. The first section describes the interposing agents which extend the operating system. The second section covers the work done in confining applications. The third section describes the systems that detect intrusions. The last section describes the security model for Java.

7.1 Interposing Agents

Interposing agents have been developed for various reasons. Extending the operating system for value-added functionality is one of them. `mapbox`, in a similar way, can be viewed as functionality added to the operating system, so that the system can run untrusted applications safely. In this section, we discuss `Interposition Agents` and `SLIC`, which are mechanisms available for implanting user-level extensions, `SPIN` which is an extensible operating system and `Condor` and `Securelib` which use modified libraries for interposition.

`Interposition agents` [22], introduced by Mike Jones, is a toolkit that substantially increases the ease of interposing user code between clients and instances of the system interface by allowing the code to be written in terms of higher-level objects instead of the system calls themselves. This toolkit was developed on top of the Mach 2.5 system call interception mechanism, which forwards the system calls to extensions linked into the application's address space. The abstractions provided by the toolkit are pathnames, descriptors, processes, process groups, files, directories, symbolic links, pipes, sockets, signals, devices, users, groups, permissions and time. A major drawback of using such a tool is that the extension resides in the same address space as the application itself, and therefore is susceptible to memory corruption. Interposition agents, therefore, cannot enforce security guarantees.

`SLIC` [26] is a system developed at Berkeley for efficiently inserting trusted extension code into the operating system kernel with minor or no modification to the operating system source code. The goal of this project is to make it easier for software vendors to develop and deploy innovative operating system features. It provides security by protecting extensions and efficiency by running extensions in the kernel's address space. SLIC could be used to implement mapbox in the kernel.

`SPIN` [23] is a general purpose operating system that provides extensibility, safety and good performance. It uses a type-safe language, Modula-3. The language enforces the interface and address boundaries, thus ensuring that any extension written in Modula-3, when loaded into operating system space, stays separated from the kernel. This makes the system easily extensible and safe. Extensions like `mapbox`, efficient paging algorithms and high performance network protocols can be easily added to the operating system.

`Condor` [27], which is a high throughput computing system developed at University of Wisconsin, runs on a cluster of workstation and harnesses its computation power by scheduling jobs on idle CPUs. It requires that the applications have two abilities: check pointing for migration and remote system calls. To migrate the calls made by applications, Condor system re-links the applications with its own libraries. In a similar approach, `securelib` [11], which is a shared library, replaces the C `accept, recvfrom` and `recvmsg` library calls by a version that performs address-based authentication. Wrapper libraries could be used as an interposition mechanism for `mapbox`, but it would require the untrusted applications to be relinked.

7.2 Confinement

There are many research groups which have addressed the confinement problem. `Janus, Deeds, Consh` and `PCC` confine untrusted applications. `DTE` confines accesses of root programs. `Software Fault Isolation` is a technique for memory protection in which untrusted process's address space is logically separated from the trusted process's address space.

`Janus` [1], which is a system developed at Berkeley, provides a secure environment to untrusted helper applications spawned by browsers like Netscape. Helper applications handle untrusted data and can be potential security threats. The language for `mapbox` is derived from that of Janus. Constructs such as `path, connect` and `putenv` are common between the two. Since it focuses on confining helper applications it does not have `accept, rename` and `childbox` in its language. For confining accesses to the display it uses `Xnest` which is not very stable. `Janus` ignores symbolic links.

`Deeds` [2] is a system that implements history-based access control for mobile codes. It maintains a selective history of accesses for an application and uses this information to grant or deny further accesses. The application is classified at runtime as opposed to classifying it at load time as in `mapbox`. For every application, there are pre-installed policies which specify the resource constraints. For every access, all relevant policies are inspected and access is denied if at least one policy denies it. The disadvantage is that it is hard to compose policies.

`Consh` [28], which stands for "confined shell", is a system that monitors the accesses to local resources and provides transparent accesses to remote resources. It uses `Janus` for monitoring local accesses. It provides access to the `ftp` and `http` file system. It migrates the network related system calls made by an untrusted application to its original host.

`Proof carrying code` [25] is a technique in which a producer of an untrusted code provides a proof that the code adheres to a predefined set of safety rules as chosen by the code consumer. The consumer, who runs the code, can, verify the proof at loadtime. The proof is checked statically, which eliminates the runtime overhead. Although verifying a proof is relatively easy, generating the proof is very difficult. The early experiments are focused exclusively on fine-grained memory safety and not system resources which are the focus of `mapbox`.

`Software Fault Isolation` [24] places an untrusted module into its own fault domain, a logically separate portion of application's address space. The object code for the untrusted module is modified to prevent it from writing or jumping to an address outside its fault domain. This mechanism is portable and programming language independent. It uses binary-rewriting technique which can be used in our approach as well. The disadvantage is that it ignores system-level security.

`Domain and Type Enforcements` (DTE) [21] is a configurable operating system access-control technology that can minimize the damage caused by subverted root programs. Configuring an appropriate DTE access control policy causes many root programs to be executed in restrictive domains that limit the accesses from each program to those needed for its responsibilities. DTE is an attractive and broadly applicable approach, but its main disadvantage is that it requires kernel modifications.

7.3 Intrusion Detection Systems

Intrusion Detection Systems [20] come in two basic varieties: anomaly detection and misuse detection. Anomaly detection assumes that users and their processes have consistent, recognizable behavior patterns with respect to things such as login time, applications used and so on. Anything falling outside this behavior is termed as intrusion. The main disadvantage of this approach is that it is subject to both false positives (benign behavior termed as potential attack) and false negatives (malicious behavior ignored). Misuse detection exploits vulnerabilities which are known to exist. There are no false positives: if an alert is raised, then an attack has happened. The disadvantage is that false negatives are inherent. Attacks which are not known cannot be detected.

USTAT [4] is an intrusion-detection system developed at Santa Barbara. In this system, penetrations are represented as state diagrams. The audit trails produced by the system are matched to find out possible attacks. A sequence of events which maps to an attack can be expressed using the language. The language is much more powerful compared to `mapbox` as it can express events based on predicate logic. The system is currently being modified to detect network-based attacks.

7.4 Java security

Java 1.0 classifies applications in two different categories. All applications loaded from the CLASSPATH are trusted and have all access rights of the user running them. All other applications are not trusted and do not have permissions to access files, permission to access third-party hosts and permission to create processes. This approach is very restrictive and prevents a lot of useful applications from executing.

Java 1.2 [6] takes a different approach. It classifies applications based on their code-base or URL. This means that applications coming from two different domains are treated differently. The policy specifications can be specified in a file which is read by the access controller. The file contains the mapping of specific permissions granted to the specific code sources. The basic entity that the access controller operates on is a *permission* object. There are different types of permission objects as

there are resources. These permissions include FilePermission, SocketPermission, PropertyPermission, RuntimePermission, NetPermission, SecurityPermission, SerializablePermission, ReflectPermission, UnresolvedPermission and AllPermission. These permissions are similar to the commands we define. Permissions to access files in /tmp/*, for example, can be created using the following line of code:
FilePermission p1 = new FilePermission("/tmp/*","read, write");

Now to grant this permission to a code loaded from *www.xyz.com* we have the following entry in the file:

```
grant codeBase http://www.xyz.com/ {
permission java.io.FilePermission "/tmp/*", "read, write" };
```

`Wallach` et al. [3] have proposed three implementation strategies for interposing flexible security policies in Java Virtual Machine. They are: Capabilities, Extended Stack Introspection and Type Hiding. All three strategies assume the presence of digital signatures to identify the principal. This principal is mapped to the security policy. Implementation of type hiding requires changes to the classloader and implementation of stack introspection requires complex changes to the virtual machine. Capabilities are quite simple , but developing a capability-based Java environment would require redesigning the interface for the Java class libraries.

Chapter 8

Discussion

8.1 Usage

We first discuss how some of the attacks, discussed earlier in the introduction, can be prevented. The `Internet worm` exploited weaknesses in `fingerd` and `sendmail` and used the fact that most of the encrypted password files were usually world-readable. If `mapbox` is used to confine `fingerd`, then it will ensure that it can only accept but can never connect. This will render the worm from spreading further. In a similar manner, `Back Orifice` running under `mapbox` can be prevented from accepting connections from other hosts, deleting files or monitoring user activities.

We see the system being used in two distinct ways. They are:

- For the programs that are run explicitly, for example, cgi-bins and network services, configuration environments are specified by the users. For example, in case of web servers, clients will upload their services or programs. In case of active networks different clients will provide services which are to be run on the intermediate network nodes. In both these cases the administrator of the server or the network node will have the knowledge of the type of application being installed, as specified by the service provider. The administrator will then use this information to ensure that those applications run with proper configuration environment. Other examples where the system can be used in this manner are confining daemon programs (preventing buffer attacks) and confining code that is `ftp'`ed over.

- For programs that are run implicitly, for example, executables sent as e-mail attachments, installable plug-ins, downloaded code in global computing systems and macros in word documents and postscript documents, we anticipate that the code will carry a tag indicating its type. The runtime system will then map this tag to an appropriate configuration environment. The mechanism is similar to the way Netscape and other MIME-aware applications use the `.mailcap` file to demultiplex downloaded data to helper applications.

8.2 Running applications under `mapbox`

We monitored the applications we studied with `mapbox` using an appropriate configuration environment for each of them. We did this to verify that existing applications can be confined using the behaviors we have defined. We ran the applications using `mapbox`. We also ran X-applications separately using `xbox`. For the experiments involving `mapbox`, we made the following observations:

- Solaris `finger client, telnet client` and `ftp client` currently run under `mapbox`.
- Currently `lynx` does not run as it tries to connect to the password database using `door` call. `Netscape` does not run as it requires information in `/etc/passwd` and `/etc/mnttab`.
- C compilers such as `cc` and `gcc` currently do not run as they make calls to find out the working directory by making a series of `stat` and `getdents` calls up the directory hierarchy, from the current directory up to the root. This sequence of system calls is described in Figure 2.6 as the `pwd` pattern. Getting the value of the working directory is central to the functioning of many applications. We recommend that `pwd` be provided as one system call. This will allow the applications to run in a safe manner. Other compiler applications such as `latex` and `dvips` currently run under `mapbox`.
- Transformers such as `gzip` and `gunzip` currently run under `mapbox`.
- The editor `emacs` fails trying to find the working directory (pattern discussed in `cc` and `gcc` example). Other editors such as `vi` and `pico` currently run under `mapbox`.
- Shells `sh` and `ksh` currently run under `mapbox`; `csh`, however, does not run. It scans all the directories in its PATH environment variable, a behavior which

is disallowed for a shell. As a part of our policy, we have allowed only those accesses that are required as a part of core functionality. As the presence or absence of files can be a piece of information which can be used to compromise the security, we have disallowed scanning of directories in PATH.

- Viewer `ghostview` currently runs under `mapbox`.

- `xterm` currently fails trying to invoke `pt_chmod` which is a `setuid` root program. We cannot trace `setuid` root programs using `/proc` interface and therefore disallow them.

- Mail clients `pine` and `elm` currently do not run under `mapbox`. Both applications try to access `/dev/ticosord`, access to which is denied.

For the experiments involving `xbox`, we made the following observations:

- Several applications such as `xcalc, xclock, xterm` and `idraw` currently run under `xbox`.

- Several other X applications currently unable to run for various reasons. For example, `xv` is interrupted when it invokes `XQueryTree` on the root window; `ical` and `ghostview` fail trying to invoke `XChangeProperty` on the root window; `xfig` fails trying to allocate a colormap not owned by itself; `pageview` trying to change an attribute of a window not owned by itself; `Netscape` fails trying to get the selection owner (`X_GetSelectionOwner`).

8.3 Assumptions

Mapbox makes the following assumptions:

- Path related information provided by NFS is consistent. While resolving the path, however, NFS might have inconsistent information. This might accidently give an application access to a file originally disallowed by its configuration.

- IP addresses are not spoofed. An application could take advantage of the services available to certain hosts by spoofing the IP address (ACM digital library, for example).

The `mapbox` can effectively prevent trojan horse attacks by confining applications to their specified behaviors. However one has to be careful while designing configurations. Two applications, if accidently given access to a common directory (`/tmp`, for example) can still break into the system by sharing information. However, accesses must be explicitly granted and applications must be aware of each other's existence.

Chapter 9

Conclusions

We make the following conclusions: First, behaviors of applications are identifiable. We have identified fourteen different behaviors. They are: `filters, transformers, mail clients, compilers, viewers, editors, shells`, `upload clients, download clients, browsers, info providers`, `servers, games` and `applets`. These behaviors form a hierarchy. For example, `compiler` is a superset of `transformer`, which indeed is a superset of `filter`. The set of behaviors is not exhaustive. However, it covers many of the applications that we use today. These behaviors can be refined or the new behaviors can be introduced, as needed.

Second, configuration environments defined for different behavior classes can be used to confine untrusted binaries. Many of the existing applications used in the study, run under `mapbox` when monitored using appropriate confinement environment. Many other applications do not run as they try to access resources disallowed by their behavior classes.

Third, the `mapbox` is *usable* as well as *configurable*. End-user need to specify only the behaviors which she wishes to allow. For example, end-user might not want to allow applications which make network connections, but might allow `filters, transformers` and `compilers`. We expect that end-user will not have to deal with configuring the configuration environment. If required, system administrators can configure new behaviors or refine existing behaviors using the confinement language. Configurations are also *portable*. All site-specific constants are defined in a single file, which encapsulates all platform-dependencies.

Fourth, the study of applications has revealed that many applications, in particular proprietary applications designed to run on a single platform, access more resources than their functionality demands. However, they may or may not make use of these *extra* resources. For example, accesses from Solaris `finger client` to `/etc/passwd` can be replaced by accesses to some dummy file. On the other hand, the Sun `cc` compiler connects to a license server and will not run without it.

Fifth, the overhead introduced by `mapbox` depends upon the nature and frequency of the system calls intercepted. For compute-intensive applications, the overhead introduced by `mapbox` can be as low as 1%. For file-intensive applications it can be as high as 41%. For file-intensive applications, this overhead can be roughly distributed as: 65% for the path resolution; 25% for the context switches; remaining 10% for filename search. Performance can be improved, at the cost of generality, by using alternative interposition techniques where the interposed code is in the same address space as the process being monitored (eg: *library re-linking* and *binary editing*). This eliminates the cost of the extra context switches. Theoretically, in the best case, overhead can be reduced from 41% to 30%. Further improvements can be made at the cost of kernel modifications, if `mapbox` is directly implemented inside the kernel. In that case the overhead can be as low as 10%, as this would eliminate the cost of *path resolution* (65% of the handler time) in addition to eliminating the cost of extra context switches per intercepted system call.

Sixth, the policy of *confining applications to their own windows* can effectively confine untrusted X applications. Several application such as `xcalc, xclock, xterm` and `idraw` can run under `xbox`. Many of the regular behaviors such as keyboard input, output, pointer input, menus, colormaps are supported. Other behaviors such as selection, grabbing the server are disallowed.

Appendix A

Handling system calls

In this appendix, we list all system calls for Solaris 2.6 and the corresponding action that we take for each of them. The data is organized into three sections. Section one has all the calls that are allowed (not intercepted). Section two lists all the calls that are denied. Section three (Table) lists all the calls that are selectively allowed or denied based on their arguments.

The following system calls are not intercepted: read, write, close, time, brk, lseek, getpid, getuid, alarm, pause, nice, sync, dup, pipe, times, profil, getgid, signal, fdsync, ulimit, getdents, poll, signalhandlers, context, mincore, mmap, mprotect, munmap, fpathconf, readv, writev, setrlimit, getrlimit, memcntl, uname, sysconfig, sigtimedwait, lwp calls, gettimeofday, getitimer, setitimer, lwp calls, pread, pwrite, llseek, processor_bind, processor_info,
sigqueue, clock_gettime, clock_getres, timer functions, lwp stuff, ntp_gettime, wait.

The following system calls are denied: chown, mount, umount, stime, ptrace, stty, gtty, statfs, fstatfs, pgrpsys, xenix, plock, msgsys, syssun, sysi86, sysppc, acct, shmsys, semsys, uadmin, utssys, umask, chroot, sysfs, setgroups, getgroups, fchown, evsys, evtrapret, xstat, lxstat, fxstat, xmknod, clocal, lchown, adjtime, systeminfo, vtrace, modctl, fchroot, vhangup, inst_sync, kaio, tsolsys, auditsys, p_online, clock_settime, setregid, install_utrap, signotify, schedctl, pset, signotifywait, rpcsys, sockconfig, ntp_adjtime.

System call	Module(s)
ioctl	*connect*' *accept*
execve	*path rename*
fcntl	fcntl_hook
rmdir	*path rename*
mkdir	*path rename*
getmsg, putmsg	*connect accept*
lstat	*path rename*
symlink	*path rename*
readlink	*path rename*
fchmod	*path rename*
pathconf	pathconf_hook
vfork	fork_hook
fchdir	*path*, *rename*
getpmsg	*accept*
putpmsg	*connect*'
rename	*path*, *rename*
utimes	*path*, *rename*
acl	acl_hook
facl	acl_hook
door	door_hook
64-bit calls	similar to their 32-bit counterparts.
socket calls	*connect*' *accept*
getpeername	*connect*', *accept*
getsockname	*connect*', *accept*
getsockopt	*connect*', *accept*
setsockopt	*connect*', *accept*
syscall	will not be seen by mapbox.

System call	Module
open	*path*, *rename connect*' *accept*
creat	*path rename*
link	*path*, *rename*
unlink	*path rename*
exec	*path rename*
chdir	*path rename*
mknod	*path rename*
chmod	*path rename*
stat	*path rename*
fstat	*path rename*
utime	*path rename*
access	*path rename*
kill	kill_hook
ioctl	ioctl_hook *connect accept*
fcntl	fcntl_hook
fork, fork1	fork_hook
setuid	setuid_hook
seteuid	seteuid_hook
setgid	setgid_hook
setegid	setegid_hook
resolvepath	resolvepath_hook

Table A.1: System calls and function/modules handling them

Appendix B

Configurations for Applications

This appendix contains the constants and configurations for the behavior classes we have identified. The configurations are defined for Solaris 2.6. Figure 1 contains the file which holds all the site specific constants. Subsequent figures are the configurations for various classes.

```
define _COMMON_PATH /usr/bin:/usr/local/bin:/usr/openwin/bin:
/fs/net/solaris/bin:/opt/SUNWspro/bin:_APP_HOME:/usr/dt/bin:.

define _COMMON_LD_LIBRARY_PATH /usr/lib:/lib:/fs/solaris/lib:
/usr/openwin/lib:/usr/ucblib

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

define _COMMON_TERM xterm

define _COMMON_DISPLAY unix:4

define _BIN_UTILS /usr/bin/ispell /usr/bin/tee /usr/bin/troff

define _MISC_LIBS /usr/lib/libthread.so.1 /usr/lib/libICE.so.6
/usr/lib/libSM.so.6 /usr/lib/libw.so.1 /usr/ucblib/*

define _COMMON_X_READ /usr/openwin/lib/* /usr/openwin/share/*
/usr/openwin/bin/*  /fs/net/solaris/lib/*

define _NETWORK_READ /etc/netconfig /etc/nsswitch.conf
/etc/.name_service_door

define _NETWORK_LOAD /usr/lib/libsocket.so.1 /usr/lib/libnsl.so
/usr/lib/nss_compat.so.1

define _COMMON_TTY /usr/share/lib/terminfo/x/xterm
```

Figure B.1: File defining constants for solaris 2.6

```
# configuration file for application filter.
# syntax:  filter

# can set the application home to save states after the exec.
# if not specified... it will be /tmp by default.
set _APP_HOME /tmp

putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
```

Figure B.2: Configuration for the filter class

```
# configuration file for application shell.
# syntax:  shell (mapfile, list-file)

# can set the application home to save states after the exec.
# if not specified... it will be /tmp by default.
set _APP_HOME /tmp

putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read $mapfile $listfile
```

Figure B.3: Configuration for the shell class

```
# configuration file for application browser
# syntax: browser
set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH
putenv TERM=_COMMON_TERM
putenv LD_LIBRARY_PATH=_DEFAULT_LD_LIBRARY_PATH
putenv DISPLAY _COMMON_DISPLAY

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read _COMMON_X_READ

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

#should be allowed to connect to ip addresses
# specified in the definition.
connect allow tcp $ip1 $ip2 $ipn

#allowed to connect to the x-server.
connect allow display

#all the children must belong to class viewer
childbox viewer

#should rename /etc/passwd
rename read /etc/passwd /tmp/foo
```

Figure B.4: Configuration for browser class

```
# configuration file for application compiler
# syntax: compiler (idir, odir, libdir)
# reads files from idir, generates output in odir
# using libraries in libdir.

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH:$dir
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH

# includes the default and the current working directory.
path allow read _COMMON_READ $idir $odir $libdir
path allow write _COMMON_WRITE  $odir
path allow read,exec _COMMON_EXEC

# allowed to spawn filters
childbox filter
```

Figure B.5: Configuration for the compiler class

```
# configuration file for application getinfo
# syntax: getinfo (IP address, dir)

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH:$dir
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE $dir
path allow read,exec _COMMON_EXEC

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

#should be allowed to connect to ip addresses
# specified in the definition.
connect allow tcp $ip
```

Figure B.6: Configuration for the getinfo class

```
# configuration file for application info provider
# syntax: infop (dir)
# dir is the name of the directory from which app
# uploads files. (must be absolute path).

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH:$dir
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY__PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read $dir

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

# accpet allowed from specific machines.
accept allow * $ip1 $ip2 $ip3:NON_SYSTEM_PORT
```

Figure B.7: Configuration for the info provider class

```
# configuration file for application coordinating server.
# syntax: server (ip1, ip2, ..., ipn)
# list of ip addresses where clients or servers are located.

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY__PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read _COMMON_X_READ

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

# accpet allowed from specific machines.
accept allow * $ip1 $ip2 $ipn:NON_SYSTEM_PORT

#allowed to connect to the x-server.
connect allow display

#allowed to spawn trabsform
childbox transform
```

Figure B.8: Configuration for the server class

```
# configuration file for application transform.
# syntax:  transform (inputfile, outputfile)
# absolute path is specified.

set _APP_HOME /tmp

#includes the default and the current working directory.
putenv PATH=_COMMON_PATH
putenv HOME=_COMMON_HOME
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read $inputfile
path allow read, write $outputfile
```

Figure B.9: Configuration for the transformer class

```
# configuration file for application upload
# syntax: upload (IP address, dir)
# dir is the name of the directory from which app
# uploads files. (must be absolute path).

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH:$dir
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY__PATH

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read $dir

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

# connect allowed only to 'machine name'.
connect allow * $machine-name:NON_SYSTEM_PORT
```

Figure B.10: Configuration for the upload class

```
# configuration file for application viewer.
# syntax:  viewer (inputfile1, inputfile2, .., inputfilen)

set _APP_HOME /tmp

#includes the default and the current working directory.
putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH
putenv DISPLAY=_COMMON_DISPLAY
putenv TERM=_COMMON_TERM

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC
path allow read _COMMON_X_READ
path allow read _NETWORK_LOAD
path allow read $inputfilei1, $inputfile2, $inputfilen

#will need these libraries
path allow read _MISC_LIBS

#allowed to do x-stuff.
connect allow display
```

Figure B.11: Configuration for viewer class

```
# configuration file for application editor.
# syntax:  editor (inputfile1, inputfile2, .., inputfilen)

set _APP_HOME /tmp

#includes the default and the current working directory.
putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH
putenv DISPLAY=_COMMON_DISPLAY
putenv TERM=_COMMON_TERM

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC _BIN_UTILS
path allow read _COMMON_X_READ
path allow read _NETWORK_LOAD
path allow read,write $inputfilei1, $inputfile2, $inputfilen

#will need lot of other libraries...
path allow read _MISC_LIBS

#allowed to do x-stuff.
connect allow display

# need to spawn utilities like spell-check.
childbox transformer
```

Figure B.12: Configuration for the editor class

```
# configuration file for application applet
# syntax: applet (IP address)

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH
putenv DISPLAY=_DEFAULT_DISPLAY

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC

# network read set for accessing network.
path allow read _NETWORK_READ _NETWORK_LOAD

#should be allowed to connect to ip addresses
# specified in the definition.
connect allow tcp $ip
connect allow display
```

Figure B.13: Configuration for the applet class

```
# configuration file for application game
# syntax: game

set _APP_HOME /tmp

# includes the default and the current working directory
putenv PATH=_COMMON_PATH
putenv LD_LIBRARY_PATH=_COMMON_LD_LIBRARY_PATH
putenv DISPLAY=_DEFAULT_DISPLAY

# includes the default and the current working directory.
path allow read _COMMON_READ
path allow write _COMMON_WRITE
path allow read,exec _COMMON_EXEC

# allowed to be an x-client.
connect allow display
```

Figure B.14: Configuration for the game class

Appendix C

Handling X protocol requests

In this appendix, we list all X protocol requests and the corresponding action that we take for each of them. Only requests are filtered. Replies, errors and events are not filtered. If a particular request is to be disallowed then the request is converted to an illegal request usually be changing the resource ID to an illegal resource ID. Selected X extension requests are handled by the X filter. The requests and the corresponding action are listed in the tables 2, 3, 4, 5, 6, 7 and 8. All other extensions are currently disallowed.

X protocol request	Action
X_CreateWindow	ALLOW as long parent is own window or ROOT
X_ChangeWindowAttributes	ALLOW as long as own window
X_GetWindowAttributes	ALLOW as long as own window
X_DestroyWindow	ALLOW as long as own window
X_DestroySubwindows	ALLOW as long as own window
X_ChangeSaveSet	DENY

X protocol request	Action
X_ReparentWindow	ALLOW as long as own window
X_MapWindow	ALLOW as long as own window
X_MapSubwindows	ALLOW as long as own window
X_UnmapWindow	ALLOW as long as own window
X_UnmapSubwindows	ALLOW as long as own window
X_ConfigureWindow	ALLOW as long as own window
X_CirculateWindow	ALLOW as long as own window
X_GetGeometry	ALLOW as long as own drawable
X_QueryTree	ALLOW as long as own window
X_InternAtom	ALLOW
X_GetAtomName	ALLOW as long as own window
X_ChangeProperty	ALLOW as long as own window
X_DeleteProperty	ALLOW as long as own window
X_GetProperty	ALLOW as long own window
X_ListProperties	ALLOW as long as own window
X_SetSelectionOwner	DENY
X_GetSelectionOwner	DENY
X_ConvertSelection	DENY
X_SendEvent	DENY
X_GrabPointer	DENY
X_UngrabPointer	ALLOW
X_GrabButton	ALLOW
X_UngrabButton	ALLOW
X_ChangeActivePointerGrab	DENY
X_GrabKeyboard	DENY
X_UngrabKeyboard	ALLOW
X_GrabKey	ALLOW

X protocol request	Action
X_UngrabKey	ALLOW
X_AllowEvents	ALLOW
X_GrabServer	DENY
X_UngrabServer	DENY
X_QueryPointer	ALLOW
X_GetMotionEvents	ALLOW as long as own window
X_TranslateCoords	ALLOW as long both src and dst windows are own
X_WarpPointer	DENY
X_SetInputFocus	DENY
X_GetInputFocus	DENY
X_QueryKeymap	ALLOW
X_OpenFont	ALLOW
X_CloseFont	DENY
X_QueryFont	ALLOW
X_QueryTextExtents	ALLOW
X_ListFonts	ALLOW
X_ListFontsWithInfo	ALLOW
X_SetFontPath	DENY
X_GetFontPath	ALLOW
X_CreatePixmap	ALLOW as long as own window
X_FreePixmap	ALLOW as long own pixmap
X_CreateGC	ALLOW as long own window
X_ChangeGC	ALLOW as long as own GC
X_CopyGC	ALLOW as long the dst GC is your own
X_SetDashes	ALLOW as long as own GC
X_SetClipRectangles	ALLOW as long as own GC
X_FreeGC	ALLOW as long own GC
X_ClearArea	ALLOW as long own drawable

X protocol request	Action
X_CopyArea	ALLOW when src and dst drawables are own
X_CopyPlane	ALLOW when src and dst drawables are own
X_PolyPoint	ALLOW as long as own drawable
X_PolyLine	ALLOW as long as own drawable
X_PolySegment	ALLOW as long as own drawable
X_PolyRectangle	ALLOW as long as own drawable
X_PolyArc	ALLOW as long as own drawable
X_FillPoly	ALLOW as long as own drawable
X_PolyFillRectangle	ALLOW as long as own drawable
X_PolyFillArc	ALLOW as long as own drawable
X_PutImage	ALLOW as long as own drawable
X_GetImage	ALLOW as long as own drawable
X_PolyText8	ALLOW as long as own drawable
X_PolyText16	ALLOW as long as own drawable
X_ImageText8	ALLOW as long as own drawable
X_ImageText16	ALLOW as long as own drawable
X_CreateColormap	ALLOW when window is own or ROOT
X_FreeColormap	ALLOW when colormap is your own
X_CopyColormapAndFree	ALLOW when colormap is your own
X_InstallColormap	DENY
X_UninstallColormap	DENY
X_ListInstalledColormaps	ALLOW when it is your own window or ROOT
X_AllocColor	ALLOW when colormap is your own
X_AllocNamedColor	ALLOW when colormap is your own
X_AllocColorCells	ALLOW when colormap is your own
X_AllocColorPlanes	ALLOW when colormap is your own
X_FreeColors	ALLOW when colormap is your own
X_StoreColors	ALLOW when colormap is your own

X protocol request	Action
X_StoreNamedColor	ALLOW as long as colormap is your own
X_QueryColors	ALLOW
X_LookupColor	ALLOW
X_CreateCursor	ALLOW
X_CreateGlyphCursor	ALLOW
X_FreeCursor	ALLOW as long as own cursor
X_RecolorCursor	ALLOW as long as own cursor
X_QueryBestSize	ALLOW
X_QueryExtension	ALLOW
X_ListExtensions	ALLOW
X_ChangeKeyboardMapping	DENY
X_GetKeyboardMapping	ALLOW
X_ChangeKeyboardControl	DENY
X_GetKeyboardControl	ALLOW
X_Bell	ALLOW
X_ChangePointerControl	DENY
X_GetPointerControl	ALLOW
X_SetScreenSaver	DENY
X_GetScreenSaver	ALLOW
X_ChangeHosts	DENY
X_ListHosts	DENY
X_SetAccessControl	DENY
X_SetCloseDownMode	ALLOW
X_KillClient	DENY
X_RotateProperties	ALLOW as long as own window
X_ForceScreenSaver	DENY
X_SetPointerMapping	DENY
X_GetPointerMapping	ALLOW
X_SetModifierMapping	DENY
X_GetModifierMapping	ALLOW
X_NoOperation	ALLOW

Table C.1: Protocol requests and the corresponding actions.

Shape extension request	Action
X_ShapeQueryVersion	ALLOW
X_ShapeRectangles	ALLOW as long as own window
X_ShapeMask	ALLOW as long as own window
X_ShapeCombine	ALLOW as long as own window
X_ShapeOffset	ALLOW as long as own window
X_ShapeQueryExtents	ALLOW as long as own window
X_ShapeSelectInput	ALLOW as long as own window
X_ShapeInputSelected	ALLOW as long as own window
X_ShapeGetRectangles	ALLOW as long as own window

Table C.2: Shape extension requests and the corresponding actions.

MIT-SHM request	Action
X_ShmQueryVersion	ALLOW
X_ShmAttach	ALLOW
X_ShmDetach	ALLOW
X_ShmPutImage	ALLOW as long as own drawable
X_ShmGetImage	ALLOW as long as own drawable
X_ShmCreatePixmap	ALLOW as long as own drawable

Table C.3: MIT-SHM requests and the corresponding actions.

mit screen-saver requests	Action
X_ScreenSaverQueryInfo	DENY
X_ScreenSaverSelectInput	DENY
X_ScreenSaverSetAttributes	DENY
X_ScreenSaverUnsetAttributes	DENY

Table C.4: MIT screen-saver requests and the corresponding actions.

Double Buffer request	Action
X_DbeGetVersion	ALLOW
X_DbeAllocateBackBufferName	ALLOW as long as own drawable
X_DbeDeallocateBackBufferName	ALLOW as long as own buffer
X_DbeSwapBuffers	ALLOW as long as all windows are own
X_DbeBeginIdiom	ALLOW
X_DbeEndIdiom	ALLOW
X_DbeGetVisualInfo	ALLOW as long as all windows are own
X_DbeGetBackBufferAttributes	ALLOW as long as own buffer

Table C.5: Double buffer extension requests and the corresponding actions.

Multi-buffer request	Action
X_MbufGetBufferVersion	ALLOW
X_MbufCreateImageBuffers	ALLOW as long as own window
X_MbufDestroyImageBuffers	ALLOW as long as own window
X_MbufDisplayImageBuffers	ALLOW as long as own window
X_MbufSetMBufferAttributes	ALLOW as long as own window
X_MbufGetMBufferAttributes	ALLOW as long as own window
X_MbufSetBufferAttributes	ALLOW as long as own buffer
X_MbufGetBufferAttributes	ALLOW as long as own buffer
X_MbufGetBufferInfo	ALLOW as long as own window
X_MbufCreateStereoWindow	ALLOW as long as own windowparent
X_MbufClearImageBufferArea	ALLOW as long as own buffer

Table C.6: Multibuffer extension requests and the corresponding actions.

Xtest request	Action
X_TestFakeInput	DENY
X_TestGetInput	DENY
X_TestStopInput	DENY
X_TestReset	DENY
X_TestQueryInputSize	DENY

Table C.7: Xtest extensions and the corresponding actions.

sun allplanes request	Action
X_AllPlanesQueryVersion	ALLOW
X_AllPlanesPolyPoint	ALLOW as long as own drawable
X_AllPlanesPolyLine	ALLOW as long as own drawable
X_AllPlanesPolySegment	ALLOW as long as own drawable
X_AllPlanesPolyRectangle	ALLOW as long as own drawable
X_AllPlanesPolyFillRectangle	ALLOW as long as own drawable

Table C.8: Sun allplanes extensions and the corresponding actions.

Bibliography

[1] Ian Goldberg, David Wagner, R. Thomas and Eric Brewer: A Secure Environment for Untrusted Helper Applications. In *USENIX security symposium* 1996.

[2] Anurag Acharya, Guy Edjlali, Vipin Choudhary: History-based Access Control for Mobile Code. To appear in *ACM CCCS-98*.

[3] Dan Wallach, Dirk Balfanz, Drew Dean, Edward Felten: Extensible Security Architectures for Java. In *Technical Report 546-97, Dept. of Computer Science, Princeton University* 1997.

[4] Koral Ilgun: A Real-time Intrusion Detection System for UNIX. *Masters Thesis, Dept. of Computer Science, UCSB* 1992.

[5] Adrian Nye: Xlib Programming Manual. O'Reilly and Associates, Inc. 1992.

[6] Scott Oaks: Java Security. O'Reilly and Associates, Inc. 1998.

[7] C. Ko, G. Fink and K. Levitt: Automated Detection of Vulnerabilities in privileged programs by execution monitoring. In *Proceeding. 10th Annual Computer Security Applications Conference*, pages 134-44, 1994.

[8] Robert Wahbe, et al.: Efficient software-based fault isolation. In *Proc. of the Symp. on Operating System Principles*, 1993.

[9] Nick Lai and Terrance Gray.: Strengthening discretionary access controls to inhibit Trojan Horses. In *Summer 1988, USENIX conference*, pages 275-286.

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

[11] William LeFevre.: Restricting network access to system daemons under SunOS. In *UNIX Security Symp. III Proc.*, pages 93-103, USENIX 1992.

[12] David G. Korn and Eduardo Krell.: The 3-D file system. In *Proc. of the 5th USENIX UNIX Security Symp.*, 1995.

[13] Glenn S. Fowler, et al.: A user-level replicated file system. In *Summer 1993 USENIX Conf. Proc.*, pages 279-290, USENIX 1993.

[14] M. Blaze, J. Feigenbaum, and J. Lacy.: Decentralized trust management. In *Proc. of the 17th Symp. on Security and Privacy*, pages 164-73, 1996.

[15] C. Gunter and T. Jim.: Design of an application-level security infrastructure. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[16] T. Jeaeger, A. Prakash, and A. Rubin.: Building systems that flexibly control downloaded executable context. In *Proc of the 6th USENIX Security Symp.*, 1996.

[17] S. Jajodia, et al.: A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD Int'l. Conf. on Mgmt. of Data,* pages 134-44, 1994.

[18] Armando Fox, et al.: TranSend: Transformational proxy service. University of California, Berkeley.

[19] Peter Cappello, Mike Neary, et al.: Javelin: Internet based global computing using Java. ACM workshop on Java for Science and Engineering Computation, Las Vegas, 1997.

[20] Steven Eckmann.: Language and Taxonomy for State Transition Representation of Scenarios in Intrusion Detection Systems. PhD proposal, Univ. of California, Santa Barbara, 1998.

[21] Kenneth Walker, Daniel Sterne, M Lee Badger, et al.: Confining Root Programs with Domain and Type Enforcement (DTE). In *Sixth USENIX UNIX Security Symposium, San Jose* 1996.

[22] Mike Jones: Interposing Agents: Transparently Interposing User Code at the System Interface In *SIGOPS* 1993.

[23] Brian Bershad, Stephen Savage, Susan Eggers, et al.: Extensibility, Safety and Performance in the SPIN Operating System In Proc. of the 15th ACM Symp. on Operating System Principles, pages 164-73, 1996.

[24] Robert Wahbe, Steve Lucco, T. Anderson, Susan Graham: Efficient Software-Based Fault Isolation In Proc. of the 15th ACM Symp. on Operating System Principles, pages 164-73, 1993.

[25] Peter Lee and George Necula: Proof-Carrying Code CMU Technical Report.

[26] Doug Ghormley, Steve Rodrigues, Dave Petrou, T. Anderson: SLIC: An Extensibility System for Commodity Operating Systems *To appear in 1998 USENIX conference.*

[27] M. Litzkow, M. Livny, and M. W. Mutka: The Condor System In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, June, 1988.

[28] Paul Kmiec: Consh: Confined Execution Environment for Internet Applications Masters Thesis, University of California Santa Barbara, 1998.

[29] Albert Alexandrov, Max Ibel, Klaus Schauser: Extending the Operating System at User Level: the UFO global file system In *Proceedings of the USENIX 1997 Annual Technical Conference, pp. 77-90, Anaheim, CA, January 6-10,* 1997.