

Final Thesis

**Comparative Study of Containment Strategies in  
Solaris and Security Enhanced Linux**

by

**Magnus Eriksson  
Staffan Palmroos**

LITH-IDA-EX-ING--07/004--SE

2007-06-04



Final Thesis

**Comparative Study of Containment Strategies in  
Solaris and Security Enhanced Linux**

by

**Magnus Eriksson  
Staffan Palmroos**

LITH-IDA-EX-ING--07/004--SE

2007-06-04

Supervisor: Prof. Dr. Christoph Schuba  
Examiner: Prof. Dr. Christoph Schuba



# Abstract

To minimize the damage in the event of a security breach it is desirable to limit the privileges of remotely available services to the bare minimum and to isolate the individual services from the rest of the operating system. To achieve this goal there are a number of different containment strategies and process privilege security models that may be used. Two of these mechanisms are Solaris Containers (a.k.a. Solaris Zones) and Type Enforcement, as implemented in the Fedora distribution of Security Enhanced Linux (SELinux). This thesis compares how these technologies can be used to isolate a single service in the operating system.

As these two technologies differ significantly we have examined how the isolation effect can be achieved in two separate experiments. In the Solaris experiments we show how the footprint of the installed zone can be reduced and how to minimize the runtime overhead associated with the zone. To demonstrate SELinux we create a deliberately flawed network daemon and show how it can be isolated by writing an SELinux policy.

We demonstrate how both technologies can be used to achieve isolation for a single service. Differences between the two technologies become apparent when trying to run multiple instances of the same service where the SELinux implementation suffers from lack of namespace isolation. When using zones the administration work is the same regardless of the services running in the zone whereas SELinux requires a separate policy for each service. If a policy is not available from the operating system vendor the administrator needs to be familiar with the SELinux policy framework and create the policy from scratch. The overhead of the technologies is small and is not a critical factor for the scalability of a system using them.



# Table of Contents

<b>1 Introduction.....</b>	<b>1</b>
1.1 Objectives.....	1
1.1.1 Solaris 10.....	1
1.1.2 Fedora: SELinux.....	2
1.2 Related Work.....	2
1.3 Typographical Conventions.....	3
<b>2 Problem Statement.....</b>	<b>5</b>
<b>3 Technology Background.....</b>	<b>7</b>
3.1 Solaris Background.....	7
3.1.1 Zones.....	7
3.1.2 SMF.....	8
3.1.3 Resource Management.....	8
3.1.4 Privileges.....	9
3.1.5 Multi-Level Security (MLS).....	10
3.2 SELinux Background.....	10
3.2.1 SELinux MAC.....	10
3.2.2 Reference policy.....	13
3.3 Related Containment Technologies.....	14
3.3.1 chroot(2).....	14
3.3.2 Jails.....	15
3.3.3 Systrace.....	15
3.3.4 AppArmor.....	16
3.3.5 Linux Vserver, Virtuozzo, OpenVZ.....	16
3.3.6 Xen, VMWare .....	17
<b>4 Experimentation.....</b>	<b>19</b>
4.1 Solaris.....	19
4.1.1 Zone creation.....	20
4.1.2 Zone start-up.....	32
4.1.3 Limitations.....	40
4.1.4 Conclusion.....	41
4.2 SELinux: Exploit simulation.....	42
4.2.1 Exploit walk-through.....	42
4.2.2 Policy syntax.....	43
4.2.3 Writing the policy module for exploit.....	44
4.2.4 Analysis.....	47
<b>5 . Evaluation.....</b>	<b>49</b>
5.1 Administration.....	49
5.2 Validation.....	50
5.3 Scalability/Overhead.....	50
5.4 Achieved protection.....	52
<b>6 Conclusions and Future Work.....</b>	<b>55</b>
6.1 Conclusions.....	55
6.2 Methodology Limitations.....	56
6.3 Future Work.....	57

<b>7 References.....</b>	<b>59</b>
<b>Appendix A – Glossary.....</b>	<b>61</b>
<b>Appendix B – Solaris Resources.....</b>	<b>63</b>
<b>Appendix C – Linux Resources.....</b>	<b>77</b>

## Table of Listings

Listing 4.1 – Zone creation.....	21
Listing 4.2 – Zone configuration file.....	22
Listing 4.3 – Regular zone installation.....	22
Listing 4.4 – Zone installation with inherit-pkg-dir.....	23
Listing 4.5 – Zone installation with lucreatezone.....	26
Listing 4.6 – Installed packages with lucreatezone.....	26
Listing 4.7 – Apache httpd dependencies.....	28
Listing 4.8 – Installation by copying.....	28
Listing 4.9 – Benchmarked installation by copying.....	29
Listing 4.10 – Zone processes after regular boot.....	34
Listing 4.11 – Setup of SMF repository for Apache http.....	35
Listing 4.12 – Zone processes with customized SMF repository.....	36
Listing 4.13 – Zone processes after customized boot.....	36
Listing 4.14 – Booted zone with replaced /sbin/init.....	38
Listing 4.15 – Loaded modules.....	44
Listing 4.16 – Defining the application domain.....	44
Listing 4.17 – The log file type.....	45
Listing 4.18 – The network port type.....	45
Listing 4.19 – Labeling a network port.....	46
Listing 4.20 - The pseudo-terminal type.....	46
Listing 4.21 – Miscellaneous permissions.....	46
Listing 4.22 – Extra permissions.....	46
Listing 4.23 – Commands used for testing.....	47
Listing 4.24 – Testing /bin/ls /.....	47
Listing 4.25 – Testing /bin/bash.....	47
Listing 4.26 – Log file after /bin/bash.....	48
Listing 4.27 – Testing /bin/touch /tmp/foo.....	48

## Table of Figures

Figure 4.2: Installation states.....	21
Figure 4.3: Zone boot states.....	32
Figure 4.4: Zone boot flow.....	37



# 1 Introduction

This paper will examine and compare the security features of Security Enhanced Linux (SELinux) [11] and Solaris 10, focusing on containment strategy. A containment strategy is the method an operating system uses to isolate unrelated services and makes them independent of each other. With a perfect containment strategy a flaw in one service will not affect other services in any way. In other words, the system should act as if every service was running on its own dedicated machine. This is desirable in a number of different scenarios such as a service provider hosting multiple customers on the same physical hardware or when running an internal and an external web server on the same host system. By isolating individual services on a single host system an organisation will be able to increase the utilization of underutilized hardware.

## 1.1 Objectives

To illustrate the different implementations we have chosen different methods to examine the two systems since they are fundamentally different in how they provide the containment features.

### 1.1.1 Solaris 10

For the containment on Solaris we make use of the zone facilities provided by Solaris. In order to get to know the zone implementation we experiment with the zone installation and start up procedures. Our goal is to create a zone with as little overhead as possible with respect to disk space, installation time and processing power to

illustrate that the zone facility can be used to create small isolated environments for a single service.

For illustration we use the Apache HTTP daemon as a sample application to contain in our isolated environment. This is a typical example of a service exposed to public networks and is a perfect candidate for isolating.

Solaris provides other means of limiting the privileges of an application but the experimentation in this thesis is focused on the use of the zone facilities.

### **1.1.2 Fedora: SELinux**

For the SELinux implementation, Fedora Core 5 was chosen. There are newer versions of Fedora and other SELinux implementations, but much of the reference documentation of SELinux is based on FC5 or on Red Hat Enterprise Linux 4, which is based on FC5.

To experiment with the containment effect in SELinux a program was developed to simulate a severely flawed telnet daemon. Unprotected, this simulation allows an attacker to remotely execute an arbitrary command as the root user. We shall see that when protected with SELinux this application can only do what is explicitly allowed by policy. The SELinux policy will only allow the daemon to run a small subset of commands from the /bin directory.

This experiment illustrates that the SELinux policy mechanism allows very fine-grained control over services, but at the same time it shows that a user must be very careful when writing the policy to avoid unintended side effects, resulting in security breaches.

## **1.2 Related Work**

Glenn Faden has compared the Multi Level Security features of Solaris with the SELinux in an article on the Sun Big Admin website [18].

A more general comparison of different containment implementations is available in a Sun Blueprint [2].

## 1.3 Typographical Conventions

*Fixed width italic font* is used to indicate path names, e.g. */sbin/init*.

Fixed width font is used in console and code listings. User input in console listings is emphasized with **bold fixed width font**. Console and code listings are enclosed in frames.

Example:

```
$ date
Mon May  7 11:05:37 CEST 2007
$
```

The ‘#’ character in console listings is used to denote a shell prompt for the root account and ‘\$’ is used to indicate a regular user.

System utilities and library functions with manual references are written with fixed width font as `fork(2)` where the entry enclosed by parentheses is a references to the manual page section. Which system we are referring to is explained unless it is obvious from the context in which the reference appears.

References to functions without a manual reference are written with fixed width font and a pair of empty parentheses as `foo()`.



## 2 Problem Statement

With the traditional model of running remotely available services in a UNIX operating system a security breach of one of these services will leave the attacker with access to the whole operating system which may be used as a trampoline for further attacks, backdoored, DDOS node etc.

To minimize the damage in the event of a security breach it is desirable to limit the privileges of remotely available services to the bare minimum and to isolate the individual services from the rest of the operating system. To achieve this isolation there are a number of different containment strategies and process privilege security model that may be used. Two of these are Zones in Solaris and SELinux policies in Fedora.

This study aims to compare the security features available for process containment and to limit process privileges in Solaris and Fedora with SELinux extensions with respect to:

- Administration - How does one handle the system with regards to installing and ongoing maintenance?
- Validation - How does one verify that the set of policies are really secure? Allow all and remove privileges vs. Deny all and explicit allow.
- Scalability/overhead - Cost of the features in terms of memory, disk and processing power. How does it handle large workloads?
- Achieved protection - Which security threats are handled and to what extent.



## **3 Technology Background**

### **3.1 Solaris Background**

These sections present features of the Solaris operating systems that are of use when isolating a process or service. It is intended to give the reader an introduction to features that will be used during the analysis part of the thesis. References to more in depth descriptions of the technologies are provided.

#### **3.1.1 Zones**

Solaris Zones provides the ability to create multiple isolated application environments inside a single instance of the operating system. The goal of the zone implementation is to provide a lightweight partitioning technology, compared to hardware partitioning and virtual machines [1]. Since there is only one instance of the Solaris kernel it is not possible to run different versions of the operating system in the zones.

There are two types of zones; global and non-global. The global zone is the environment that is running when the system is booted. This is basically the same as the version of the operating system before the introduction of zones. Non-global zones are created and administered from the global zone and have their root directory configured as a directory in one of the global zone's file systems. By looback mounting directories into the zones root directory tree a zone may share e.g. the /usr directory with the global zone to lessen the number of files that need to be copied when creating the zone.

Bringing a zone up is called booting the zone. This choice of word is no accident since the way a zone is started is very similar to when booting the base operating system i.e., the global zone. Bringing a zone down is called halting the zone.

When used in combination with Solaris resource management, Zones are sometimes referred to as containers [5]. This paper does not use this term.

[1] gives in depth information about the thoughts that went into the design process of the zone implementation. A comparison between different containment technologies are provided in [2].

### **3.1.2 SMF**

Solaris 10 brings a new facility for managing system services, Services Management Facility (SMF) [3]. It is a replacement for the old way of bringing the system up by using a set of customized scripts in the `/etc/rc.*` directories which are executed in sequence by the `/sbin/init` process. The dependencies were determined by a somewhat arbitrarily number in the name of the script. The approach taken by SMF is to maintain a database of services which have well defined dependencies on other services. A service and it's dependencies is specified by a XML manifest file. By constructing a graph of the service dependencies SMF are able to start independent services in parallel, thus making the system's boot sequence more efficient.

Another SMF feature of interest is the automatic restarting of failed services which is part of the Solaris 10 Predictive Self-Healing technologies [4]. As the name suggests SMF monitors the state of a service and makes sure it is running at all times by restarting it if the process exits in an uncontrolled fashion.

### **3.1.3 Resource Management**

Solaris has the ability to partition system resources through the use of resource pools. This feature allows the administrator to assign system resources in the form of CPU time, memory and disk space to different resource categories. This feature is not covered in depth by this thesis but it's existence will be considered when evaluating different systems. A thorough treatment on how to use resource management in zones is available in [9].

### 3.1.4 Privileges

In the UNIX world the administrator is identified as the user with the user id 0, traditionally named root. This account has privileges that allows it to perform certain special operations like changing system parameters, binding to TCP and UDP ports below 1024, bypassing DAC controls on files etc. The problem with this approach is that the root account has full privileges over the system whereas normal users have no special privileges at all. Since each running process has privileges based on the user that owns the process this translates to all running processes on the system.

If a user needs to run programs that requires special privileges special actions must be taken. Traditionally this problem has been solved by setting the set-user-ID bit on the binary that requires the special privileges [10]. When this bit is set the operating system will execute the binary with the privileges of the file's owner, not the user that is running the program.

The problem with this approach is that there is no way to explicitly allow only certain operations, i.e., to bind to a port below 1024. The implementers of the software can try to limit the damage that may be done by dropping the privileges when they are no longer needed but from the kernels perspective there is no difference between a SUID root process and a regular root process. If the user manages to make the process execute his own code before dropping the privileges he has full administrator privileges on the system.

To remedy this weakness Solaris 10 provides a more fine grained privilege model where the administrator may assign individual privileges to a user and in consequence to running processes. For instance, the HTTP server needs to bind to TCP port 80 but for security reasons one would not want to run this server with root privileges. With the Solaris privilege model a special privilege, *net\_privaddr*, is given to the user running the HTTP server and the process is able to bind to port 80 without running as a SUID process.

In addition to setting privileges on a per-user, per-role, basis privileges may be configured for each zone.

### **3.1.5 Multi-Level Security (MLS)**

Solaris provides MLS support through the Trusted Extensions packages. It has been implemented using the zone facilities and is no longer provided through a separate distribution but are implemented in the standard Solaris 10 installation. It is activated by installing the Trusted Extension packages.

## **3.2 SELinux Background**

SELinux started out as a research project by the United States National Security Agency (NSA) as a security framework for the FLASK micro-kernel based operating system. It was ported to the Linux kernel to demonstrate how Linux would benefit from an improved security model and how to implement such a model.

Initially SELinux was created and distributed as a separate set of patches to the 2.4.x Linux kernel. Maintaining these patches was troublesome so it became desirable to integrate SELinux into the mainline kernel. But since not all users of the kernel need or want these patches and since they also incur some overhead that might not be desirable, Immunix developed a subsystem called the Linux Security Modules (LSM) [12]. LSM is a pluggable architecture that allows the user to insert modules into the Linux kernel that implements additional security models to the regular discrete access control. The framework provides hooks (pointers to functions that can disallow access based on some internal calculation) deep in the kernel where important kernel structures are modified.

### **3.2.1 SELinux MAC**

SELinux implements and mixes three models of mandatory access control: Role-Based Access Control (RBAC), Type Enforcement (TE) and Multi-Level Security (MLS)

#### **3.2.1.1 Type Enforcement**

The primary security model of SELinux is type enforcement. A type enforced system has three basic entities: domains, types and permissions. Processes in a system, usually referred to as subjects, are labeled with a domain. All other units in the system, like files, pipes, network ports and more, are referred to as objects and are labeled with a

type. Domains and types are both stored as short text strings ending with `_t`, like `user_home_dir_t` for the home directories of a system.

Permissions are the operations that can be performed on objects. The set of permissions that can be applied on an object is called access vector.

It is the purpose of type enforcement to determine and enforce which permissions can be applied to objects in the system by which subject. By default all permissions are denied, the user must create a set of rules that explicitly states which permissions should be allowed for every combination of subject and object. This rule set is called a policy. Setting up a policy from scratch is quite complex, the normal procedure is to start with a reference policy and then modify it to suit whatever needs there are.

SELinux requires file system support for extended attributes, which limits which file systems can be used with SELinux. This requirement makes it problematic to use with NFS for example. In such cases a generic type can be applied to the specific mount, but that will work poorly with the protection features.

### **3.2.1.2 Object classes and permissions**

The objects in the system are grouped in object classes. The object classes correspond to the kernel structures of the objects. Objects of these classes can be manipulated in different ways, a file can be written to for example. These operations on kernel objects are called permissions. The set of permissions of an object class is called an access vector.

The set of object classes directly reflects the features of the kernel. It is not possible to add custom object classes to a policy. Research is ongoing to make it possible to allow adding or removing object classes at runtime, but nothing is finalized yet.

### **3.2.1.3 Types and Domains.**

In SELinux there is not really any difference between a type and a domain. A domain is just a type that has been marked 'domain' with an attribute. The distinction between a type and a domain is purely decided by policy. In the reference policy there are rules that say that only processes may be labeled with types that has the attribute 'domain'. A file

can not be relabeled to use a type attributed as a domain, there simply is no rule that allows it.

An interesting effect of this lack of distinction can be seen in the /proc directory. /proc is a pseudo-file system that lists runtime statistics of all running processes in the system. In this directory all files are labeled with domains and they cannot be relabeled with non-domain types. The reason for this is that the files in the /proc directory does not actually exist, they only provide a view of the system processes.

#### 3.2.1.4 Roles

SELinux also provides a Role Based Access Control security model. The RBAC model defines a set of roles (using the suffix \_r) and explicitly allows which roles can enter which domains. This security model is not very well developed in the strict policy, only a few standard roles are defined:

system_r	This role is used by applications started by the system itself, for example during boot.
user_r	A role used for regular users with no right to do any system administration.
staff_r	This role has the same permissions as user_r, but is also allowed to transition to the sysadm_r role
sysadm_r	This role is used for all administration tasks. When you log in or su to the root user you are still in the staff_r role. You must transition to the sysadm_r to be able to actually do something.

In the Fedora Core 5 experimental MLS policy two more roles are defined:

secadm_r	This role must be used when using the SELinux tools.
auditadm_r	This role must be used when handling the auditing subsystem.

### 3.2.1.5 Users

Users in SELinux is different from the regular Unix User ID:s. While the user id changes when setuid applications (such as sudo) is run, the SELinux user identity is not changed. Early versions of the standard SELinux policy used a separate SELinux user for every Linux user. This scheme has been abandoned in later versions, instead there are generic user classes for different types of users.

In Fedora Core 5, these users are defined:

root	When a user logs in directly to the root account, this is the SELinux user that will be used.
system_u	Applications started automatically by the system runs under the system_u SELinux user, despite whatever their Linux user might be.
user_u	Regular Linux users without system administration rights gets the user_u class.
staff_u	The users allowed to do system administration tasks belong are in the staff_u user class.
sysadm_u	Users in the sysadm_u class gets administration rights directly on login.

SELinux users are granted access to a number of roles. The newrole command is used to switch roles.

### 3.2.1.6 Multi-Level Security (MLS)

SELinux does support an MLS context, but in Fedora Core 5 it was still experimental so it has not been examined for this paper.

## 3.2.2 Reference policy

The initial example policy from NSA was complex and hard to understand and modify. Tresys Technology [13] set out to rewrite this policy to make it modular and easier to adapt. This rewritten policy is commonly referred to as the reference policy [14] and is available at [sourceforge.net](http://sourceforge.net) [15].

Fedora Core 5 delivers three different policies, all derived from this reference policy:

- Strict is the policy closest to the reference policy. It is intended only for use on servers.
- Targeted is a simplified policy where only selected - targeted - daemons and applications are protected. Other applications run in a special domain `unconfined_t` which is for the most part unrestrained. This policy allows SELinux to be used in a desktop environment, which would otherwise require an extremely detailed policy.
- MLS is an experimental policy which depends on an additional 4th field in the SELinux label. This policy is an attempt to implement the Bell-LaPadula model in SELinux.

### 3.3 Related Containment Technologies

This section gives an overview of other virtualization technologies.

#### 3.3.1 chroot(2)

The `chroot()` system call is a simple form of containment that changes the root directory of the current process and its child processes. Careful attention must be made that there are no way to get out of the `chroot()`ed file system view, for example by a hard link to a directory outside the directory subtree.

An independent directory tree is set up somewhere in the file system where a restricted set of the standard root file system layout is added. This directory usually includes an `/etc`, a `/dev` and a `/var`. The `/etc` is limited to the configuration file of the daemon that is `chroot()`ed. `/dev` contains a `/null` device and any other device that the daemon will explicitly ask for. The `/var` is used for the runtime data of the daemon, usually `/var/<daemon>/var/run/<daemon>.pid` and `/var/log/<daemon>.log`

The only thing that `chroot()` protects against is access to files outside the directory. `chroot()` does nothing to protect the process space for example, so it is perfectly possible to interfere with processes outside of the `chroot()`ed environment. For example, a `chroot()`ed daemon could be used in a multi-stage attack where an attacker first gets access to the `chroot` environment by remotely exploiting a weakness, and from there

launches a local root exploit against another system service to gain access outside the `chroot()` jail.

The limitations of `chroot()` might be explained by a paper from the FreeBSD project [6], which explains that the `chroot()` system call was not intended as a security mechanism but as part of the build process for 4.2BSD.

### **3.3.2 Jails**

Jails is a feature available in the FreeBSD operating system since release 4.0. Like Solaris Zones it virtualizes the application environment and as a consequence it is not possible to run different versions of the operating system. The original motive for Jails was to limit the privileges of the root user by partitioning the system into isolated environments, of which each has their own root account with no privileges to access resources outside of that particular environment[6].

In a similar way that Solaris differs between global and non-global zones, FreeBSD differs between the host environment and jail environments. As with Solaris Zones and `chroot(2)` a jail is configured as a directory tree in one of the host environments file system.

### **3.3.3 Systrace**

Systrace is a process confinement technology that uses system call interposition to provide privilege limiting features for the operating system [7]. It is available on OpenBSD, NetBSD and Linux based operating systems.

By requiring approval from a policy engine before executing a system call the user can limit the privileges for a running application which in turn limits the damages that can be done as a result of bugs in the application.

Systrace enables the user, not only the administrator, to create policies for individual binaries which makes it highly configurable on a per-user basis.

Some concerns has been raised with regards to system call interpositions technologies [8]. These issues have been accounted for in the design of Systrace [7].

### **3.3.4 AppArmor**

AppArmor [16] is a technology similar to SELinux. It uses the same LSM mechanism as SELinux but provides a much simplified security model: AppArmor only checks POSIX.1e capabilities and name based file access whereas SELinux checks against a large policy tree containing all possible manipulations of the system structures. Because of this simplified model AppArmor is more lightweight than SELinux but at the same time not as fine-grained. AppArmor does not provide extended features such as Role-Based Access Control or Multi-Level Security. Neither does AppArmor lock down the entire system, only specific applications are policed. In that respect it can be compared to the targeted SELinux policy. There are tools that 'learns' what the application normally does and creates a policy from that.

AppArmor was initially developed by commercial Linux vendor Immunix. In May 2005 Novell acquired Immunix and with them, AppArmor. AppArmor 2.0 was integrated into Novell's SuSE 10.1 (later renamed OpenSuSE 10.1) and SuSE Linux Enterprise 10.

### **3.3.5 Linux Vserver, Virtuozzo, OpenVZ**

These are OS-level virtualization technologies for the Linux kernel. They are the Linux equivalents of FreeBSD Jails or Solaris Zones. Unfortunately, they are not part of the mainline kernel but instead distributed as separate patch sets.

OpenVZ adds an abstraction layer for a number of kernel features: Every virtual environment has its own private process id namespace, private IPC namespace, network interfaces and more. These resources can be accessed from the host environment but not from other VE:s.

OpenVZ does not require that all VE:s run the same Linux distribution, but since there is only one kernel in the system the distribution must support that version.

OpenVZ allows live migration, which means that virtual servers can be moved between physical hosts. The migration is completely transparent to a user of a VE, it will only be noticeable as a sudden, small (couple of seconds) delay.

Virtuozzo is a proprietary product based on OpenVZ.

Linux-Vserver is a similar product but somewhat simpler in implementation. It adds a number of flags and/or fields in various kernel structures and then check for these in appropriate places. The project does not seem to move along as fast as OpenVZ, whose developers are frequent contributors to the mainline Linux kernel. Virtuozzo is a proprietary product based on OpenVZ.

### **3.3.6 Xen, VMWare**

Xen and VMWare provide the ability to run full installations of different operating systems on the same physical hardware. Unlike some of the previously described technologies, Xen and VMWare have multiple kernels running, one for each operating system instance. These systems differs between the host operating system and VMWare provides full virtualization which allows an operating system to be installed without any modifications to the guest operating system. Xen provides paravirtualization where modifications to the guest operating system are needed.

Common for these technologies is that they use more system resources than the technologies that provide virtualization support within the same kernel.



## 4 Experimentation

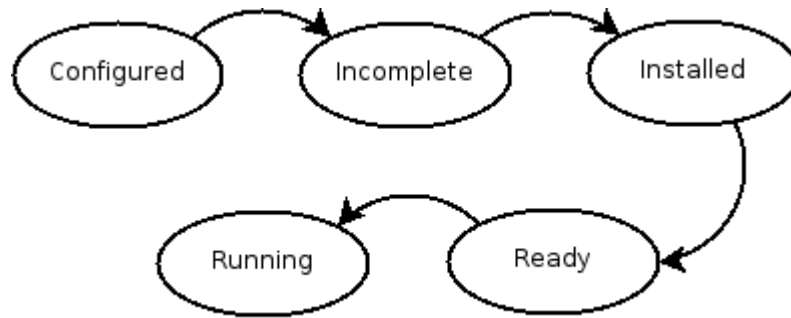
### 4.1 Solaris

Solaris 10 offers two features for creating an isolated environment for a process group, `chroot` (2) and `zones` (5). The disadvantages of `chroot` (2) was described in section 3.3.1. This evaluation makes use of the zone facility to create an isolated environment.

As was described in section 3.1.1 Solaris Zones offer a virtual environment in which processes can be run completely isolated from processes in other zones. We use this feature to confine an exposed network service to limit the damages that can be done by taking advantage of a possible vulnerability in this service. The service used in the evaluation is the Apache web server that is available in the Solaris 10 distribution through the packages in **SUNWapchr\***.

The goal of the experiments is to minimize the overhead introduced by running a service in a zone with regards to disk space, memory and processing power. In addition to this goal we are interested in removing all unnecessary binaries in the zone. Having unnecessary binaries in the installation may add additional attack vectors for a potential adversary that has managed to get access to the zone by exploiting flaws in the exposed service.

During the lifetime of a zone it goes through different states in a well defined state machine [1]. The states dealt with in this thesis are shown in Figure 4.1. The graph only shows the states that are observable in our experiments.



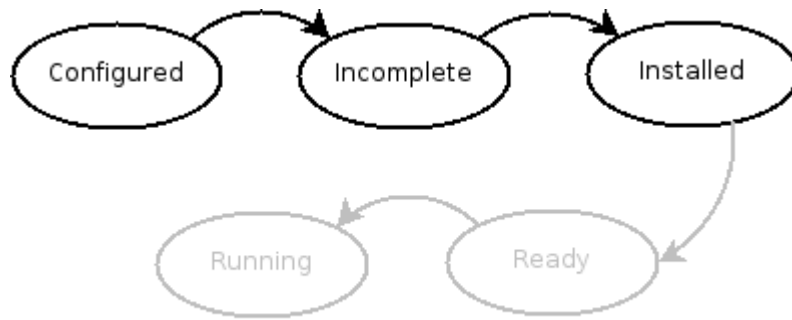
**Figure 4.1: Zone state model**

The experiment is divided into two parts. The first part present different ways of configuring and installing a zone. We start from a standard zone installation and proceed to evaluate different installation methods in order to reduce the installation footprint. The time it takes to install a zone will also be part of the evaluation and we try to reduce the installation time as much as possible.

The second part presents different ways of bringing the zone into a running state i.e., to start the Apache service in the isolated environment. As with the installation part we start out with the conventional way of booting a zone and try to improve it by cutting down the number of processes running in the zone.

#### **4.1.1 Zone creation**

In this section we will present different ways of creating a zone. The parts of the zone state machine that are dealt with in this section are highlighted in Figure 4.2. We start out with the standard procedure for creating a zone with the standard Solaris 10 zone utilities. We then try to improve the zone creation process by cutting down the creation time and the zone’s disk usage. The `time(1)` utility is used to give estimation of zone creation time and `du(1)` is used to measure disk usage. These utilities are considered accurate enough to prove the points. To assure accurate values from the runs, every experiment was run multiple times and the values presented in the text are those of a typical run.



**Figure 4.2: Installation states**

#### 4.1.1.1 Regular installation

In this experiment we create a zone by using `zonecfg(1M)` and `zoneadm(1M)` in the way described by the Solaris documentation [9]. Doing this will give us an understanding about what happens when you create a zone and the drawbacks that we will try to improve in the following sections. We proceeded to create a zone which will reside under `/zone/apache` and have the IP address `192.168.0.1`. The configuration for this zone is shown in Listing 4.1.

```

# zonecfg -z apache
apache: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:apache> create
zonecfg:apache> set zonepath=/zone/apache
zonecfg:apache> add net
zonecfg:apache:net> set physical=bge0
zonecfg:apache:net> set address=192.168.0.1
zonecfg:apache:net> end
zonecfg:apache> commit
zonecfg:apache> exit
  
```

**Listing 4.1 – Zone creation**

This set of commands will create a zone by adding an entry to `/etc/zones/index`:

```
apache:configured:/zone/apache:
```

and creating a new file, `/etc/zone/apache.xml`, with the contents shown in Listing 4.2.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE zone PUBLIC "-//Sun Microsystems Inc//DTD Zones//EN"
file:///usr/share/lib/xml/dtd/zonecfg.dtd.1">
<!--
    DO NOT EDIT THIS FILE.  Use zonecfg(1M) instead.
-->
<zone name="apache" zonepath="/zone/apache" autoboot="false">
  <inherited-pkg-dir directory="/lib"/>
  <inherited-pkg-dir directory="/platform"/>
  <inherited-pkg-dir directory="/sbin"/>
  <inherited-pkg-dir directory="/usr"/>
  <network address="192.168.0.1" physical="bge0"/>
</zone>

```

### Listing 4.2 – Zone configuration file

Note that the create on line 4 in Listing 4.1 implies the argument `-t SUNWdefault`<sup>1</sup> which will use the `/etc/zones/SUNWdefault.xml` as a template of the zone, hence the `inherited-pkg-dir` entries in Listing 4.2.

```

# time zoneadm -z apache install
Preparing to install zone <apache>.
Creating list of files to copy from the global zone.
Copying <9010> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <975> packages on the zone.
Initialized <975> packages on the zone.
Zone <apache> is initialized.
The file </zone/apache/root/var/sadm/system/logs/install_log> contains a
log of the zone installation.

real    5m29.402s
user    0m38.508s
sys     0m53.594s

# pkginfo | wc -l
975
# pkginfo -d /zone/apache/root/var/sadm/pkg | wc -l
975
# du -ms /zone/apache
120    /zone/apache

```

### Listing 4.3 – Regular zone installation

The next step is to install packages in the configured zone which is shown in Listing 4.3 along with some data of the newly created zone. The statistics gathered with `time(1)`, `pkginfo(1)` and `du(1)` are used when comparing this method to the ones in the following sections. Note that all the packages from the global zone are installed in the new zone and that the zone occupies 120 MB of disk space.

When the zone is installed it has transitioned to the *installed* state. The `/etc/zones/index` now contains the line:

<sup>1</sup> See `create_func()` in <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/zonecfg/zonecfg.c>

```
apache:installed:/zone/apache:36f5e18e-58b3-c6cd-9a62-9c46ae16c46e
```

When using `zoneadm(1M)` there is currently no way to choose which packages are installed when creating a new zone. All packages from the global zone is installed and instantiated in the new zone<sup>2</sup>. By specifying additional *inherited-package-dir* entries in the zone configuration the number of copied files can be reduced. All packages still have to be initialized in the zone even if we add *inherited-package-dir* entries.

```
# zonecfg -z apache
apache: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:apache> create
zonecfg:apache> set zonepath=/zone/apache
zonecfg:apache> add net
zonecfg:apache:net> set physical=bge0
zonecfg:apache:net> set address=192.168.0.1
zonecfg:apache:net> end
zonecfg:apache> add inherit-pkg-dir
zonecfg:apache:inherit-pkg-dir> set dir=/opt
zonecfg:apache:inherit-pkg-dir> end
zonecfg:apache> commit
zonecfg:apache> ^D
# time zoneadm -z apache install
Preparing to install zone <apache>.
Creating list of files to copy from the global zone.
Copying <1921> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <975> packages on the zone.
Initialized <975> packages on zone.
Zone <apache> is initialized.
The file </zone/apache/root/var/sadm/system/logs/install_log> contains a
log of the zone installation.

real    5m29.052s
user    0m38.547s
sys     0m54.809s

# pkginfo -d /zone/apache/root/var/sadm/pkg | wc -l
  975
# du -ms /zone/apache
50    /zone/apache
```

#### Listing 4.4 – Zone installation with inherit-pkg-dir

As can be seen in Listing 4.4 the number of copied files is reduced but the number of initialized packages is the same. Since the time it takes for `zoneadm(1M)` has not decreased this suggest that the majority of the time is spent in the package initialization part. This is confirmed by watching the console feedback during the installation process.

Since there is currently no way to exclude packages from the installation of the zones by using `zonecfg(1M)` and `zoneadm(1M)` we will have to resort to other methods of

<sup>2</sup> When installing packages in the global zone using `pkgadd` with the `-G` flag packages are flagged as global zone-only packages and will not be installed in non-global zones.

installing a zone to cut down the installation time and disk footprint. These methods will be explored in the following sections.

The pros of using this method of installation are that it is the way supported by Sun so it will remain compatible during system upgrades.

The drawbacks are the installation time, disk footprint and that we have no way of excluding packages.

#### 4.1.1.2 Installation using lucreatezone

In this section we try to improve the zone creation process by using a non-supported method when installing a zone. As was described in the last section the use of `zoneadm(1M)` does not allow the administrator to specify which packages are installed during the installation process. As of 2007-03-24 the installing of packages in the zone when we issue

```
# zoneadm -z apache install
```

is done by the utility `/usr/lib/lu/lucreatezone`<sup>3</sup>. The processes leading up to the invocation of `lucreatezone` is somewhat different after the introduction of branded zones, BrandZ<sup>4</sup>, but the result is the same; `zoneadm(1M)` invokes `lucreatezone` as:

```
/usr/lib/lu/lucreatezone -z apache
```

This invocation causes `lucreatezone` to install *all* packages from the global zone into the new zone.

`lucreatezone` is part of the Live Upgrade suite from Sun<sup>5</sup> and is at the time of this writing not released as part of the OpenSolaris project. This deficiency, combined with a lack of documentation makes analysis of its inner workings harder. A posting in the OpenSolaris bugs database<sup>6</sup> suggest that the `-P` flag may be used to exclude packages from being installed by `lucreatezone`. It was verified that this flag indeed works but side effects from using this approach can not be analyzed due to the missing source code. An analysis of the `zoneadm(1M)` and `zoneadmd(1M)` source code showed that

---

<sup>3</sup> `/usr/lib/lu/lucreatezone` is actually a symbolic link to `/etc/lib/lu/ludo` which is the real binary

<sup>4</sup> Section 4.1.1.4 gives more information on BrandZ

<sup>5</sup> <http://www.sun.com/software/solaris/liveupgrade> access 2007-03-24

<sup>6</sup> [http://bugs.opensolaris.org/bugdatabase/view\\_bug.do?bug\\_id=4963323](http://bugs.opensolaris.org/bugdatabase/view_bug.do?bug_id=4963323) accessed 2007-03-24

apart from running `lucreatezone` the zone's state is changed to *incomplete* before the installation begins and to *installed* after a successful installation. If the installation fails the zone remains in the *incomplete* state and has to be reverted to *configured* state manually by the administrator.

In the following installation the use of `zoneadm(1M)` is abandoned. Instead, `lucreatezone` is executed directly to allow us to use the `-P` flag to exclude packages that are not needed in the zone. By using this flag we expect to lessen the installation time, disk usage and available utilities in the zone.

When using `lucreatezone` directly the zone's state will have to be updated manually. This can be done by changing the entries in `/etc/zones/index`, however, when the zone is transitioned to the *incomplete* state the entry in `/etc/zones/index` should contain a UUID and if the entries are modified by hand a UUID for the zone needs to be generated. Performing these steps manually is somewhat cumbersome and does not ensure future compatibility.

`zoneadm(1M)` use the `zone_set_state()` function from the `/usr/lib/libzonecfg.so` library to change the zone state<sup>7</sup> and we decided to do the same by writing a small utility, `zone_set_state`<sup>8</sup>, that make use of the functions from the `libzonecfg.so` library to help with the state transitions during the installation. It takes the name of the zone as the first argument and one of the three states *configured*, *incomplete* or *installed* as the second argument and updates the zone's state accordingly.

To bring the new zone into the *configured* state we repeat the procedures with `zonecfg(1M)` from Listing 4.1 The next task is to prepare a file with packages to exclude from the installation. In a typical Solaris 10 installation there are hundreds of installed packages of which we only want a fraction to be installed into our new zone. Unfortunately there are no flags available to `lucreatezone` that takes a list of packages to install. To remedy this absence, a script `create_packages.pl`<sup>9</sup>, was written that takes the packages we want to install as arguments, calculates the dependencies for

---

<sup>7</sup> <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/zoneadm/zoneadm.c?r=3777> row 3231 and 3274

<sup>8</sup> See Appendix B.2

<sup>9</sup> See Appendix B.5

those packages and generates a list of packages to exclude by removing these from the full set of packages installed in the global zone.

```
# /opt/thesis/zone_set_state apache incomplete
# perl /opt/thesis/create_packages.pl SUNWapch2u SUNWapch2r > exclude.pkg
# time /usr/lib/lu/lucreatezone -z apache -P exclude.pkg
Preparing to install zone <apache>.
Creating list of files to copy from the global zone.
Copying <455> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <16> packages on the zone.
Initialized <16> packages on zone.
Zone <apache> is initialized.
The file </zone/apache/root/var/sadm/system/logs/install_log> contains a
log of the zone installation.

real    0m5.206s
user    0m2.998s
sys     0m1.062s

# /opt/thesis/zone_set_state apache installed; rm exclude.pkg
# pkginfo -d /zone/apache/root/var/sadm/pkg | wc -l
    16
# du -ms /zone/apache
    2    /zone/apache
```

**Listing 4.5 – Zone installation with lucreatezone**

Listing 4.5 displays a zone installation using the techniques described above. As can be seen the installation time has decreased significantly to about 5 seconds from the previous 5 minutes and the disk footprint is about 2 MB compared to 50 MB. This is a significant improvement from the values we got in Listing 4.4

```
# pkginfo -d /zone/apache/root/var/sadm/pkg/
system    SUNWapch2r    Apache Web Server V2 (root)
system    SUNWapch2u    Apache Web Server V2 (usr)
system    SUNWcakr      Core Solaris Kernel Architecture (Root)
system    SUNWcar       Core Architecture, (Root)
system    SUNWckr       Core Solaris Kernel (Root)
system    SUNWcnetr     Core Solaris Network Infrastructure
(Root)
system    SUNWcsd      Core Solaris Devices
system    SUNWcsl      Core Solaris, (Shared Libs)
system    SUNWcsr      Core Solaris, (Root)
system    SUNWcsu      Core Solaris, (Usr)
system    SUNWkvm      Core Architecture, (Kvm)
system    SUNWlibms    Math & Microtasking Libraries (Usr)
system    SUNWlibmsr   Math & Microtasking Libraries (Root)
system    SUNWopenssl-libraries OpenSSL Libraries (Usr)
system    SUNWperl584core Perl 5.8.4 (core)
system    SUNWperl584usr Perl 5.8.4 (non-core)
```

**Listing 4.6 – Installed packages with lucreatezone**

Listing 4.6 shows a list of the installed packages in the zone. The additional packages in this list were identified as dependencies to **SUNWapchr2r** and **SUNWapchr2u** by the

`create_packages.pl` script. The number of installed packages in the zone has gone from 975 to 16. This is also a significant improvement from the previous installation.

To be able to boot the zone the zone's root directory must have a DAC permission set of 0700, and since `/usr/lib/lu/lucreatezone` creates the directory with permissions 0755 we need to change the permissions in order to make our zone bootable:

```
# chmod 0700 /zone/apache
```

At this point the zone is ready to be brought up.

After verifying the installation steps used in this test manually a script, `install_with_lucreatezone.pl`<sup>10</sup>, was created that may be used to perform these steps based on values derived from a manifest file. An example manifest file for the Apache web service, `apache_manifest.pkg`<sup>11</sup>, was created that is used in section 4.1.1.4 where we explore the use of the BrandZ framework.

The advantage of executing `lucreatzone` directly is that we decreased installation time, disk footprint and the available utilities and we are still compatible with the Solaris packaging system.

The disadvantages are that we moved outside of Sun's officially supported method of installing zones which might break future compatibility. As a result of installing whole packages we still got a lot of unwanted files in the zone.

#### 4.1.1.3 cp(1) installation

The installations in the previous section significantly decreased the installation time and the disk footprint of the installed zone. However, there is still room for improvement. By installing packages we still get a lot of files that are of no use for the isolated process that are running in the zone. In this section we try to limit the installed files to only those necessary for the running service, i.e., Apache, to function properly.

To achieve this minimal installation we manually copy the necessary files from the global zone to our newly created zone. This means that we no longer use the `lucreatezone` utility and hence lose the integration with the Solaris packaging system.

---

<sup>10</sup> See Appendix B.7

<sup>11</sup> See Appendix B.8

After configuring the zone as shown in Listing 4.1 we proceed to copy the necessary files from the global zone. This way of setting up the new environment is similar to the way one would prepare a `chroot(2)` or FreeBSD jail environment where there are no administrative utilities like `zoneadm(1M)` or `lucreatezone`. We start out by determining the binaries that should be run in the isolated environment, in this case that would be `/usr/apache2/bin/httpd`. The dependencies for this binary are then determined using the `ldd(1)` utility in Solaris. The result from running `ldd(1)` on `/usr/apache2/bin/httpd` is shown in Listing 4.7.

```
# ldd /usr/apache2/bin/httpd
libssl.so.0.9.8      => /usr/sfw/lib/libssl.so.0.9.8
libcrypto.so.0.9.8  => /usr/sfw/lib/libcrypto.so.0.9.8
libdl.so.1          => /lib/libdl.so.1
libaprutil-0.so.0   => /usr/apache2/lib/libaprutil-0.so.0
libexpat.so.0       => /usr/apache2/lib/libexpat.so.0
libapr-0.so.0       => /usr/apache2/lib/libapr-0.so.0
libsendfile.so.1    => /lib/libsendfile.so.1
libm.so.2           => /lib/libm.so.2
libsocket.so.1      => /lib/libsocket.so.1
libnsl.so.1         => /lib/libnsl.so.1
libresolv.so.2      => /lib/libresolv.so.2
libpthread.so.1     => /lib/libpthread.so.1
libc.so.1           => /lib/libc.so.1
libmp.so.2          => /lib/libmp.so.2
libmd.so.1          => /lib/libmd.so.1
libscf.so.1         => /lib/libscf.so.1
libuutil.so.1       => /lib/libuutil.so.1
libcrypto_extra.so.0.9.8 => (file not found)
#
```

**Listing 4.7 – Apache httpd dependencies**

The dependencies we get from running `ldd(1)` might have their own dependencies and to aid us in the process of determining the whole dependency graph a script, `calc_bin_deps.pl`<sup>12</sup>, was written. This script takes one or more binaries as input and outputs a full list of library dependencies for these binaries.

We use this script to copy all the necessary files from the global zone to the application zone by issuing the commands in listing 4.8.

```
# for f in `perl calc_bin_deps.pl /usr/apache2/bin/httpd \
                /usr/apache2/libexec/*.so`; \
do mkdir -p `dirname /zone/apache/root/$f`; \
  cp -Ppr $f /zone/apache/root/$f; \
done
# cp -Ppr /lib/ld.so.1 /zone/apache/root/lib/
```

**Listing 4.8 – Installation by copying**

<sup>12</sup> See Appendix B.6

Note that this is a simplified example, in a real world installation the permissions of the directories created by `mkdir(1)` need to be altered to match those of the global zone. `ld.so.1(1)` is the runtime linker that handles mapping of libraries to dynamically linked binaries and are necessary for running most binaries.

In addition to these files there are a few other files and directories that needs to be present for the zone to boot. They were identified by trial and error and are `/proc`, `/system/contract`, `/system/object`, `/etc/svc/volatile` and `/etc/mnttab`. These files and directories need to be created in order for the zone to be transitioned from the *ready* state to the *running* state. By using our own branded zone type the need for these special files can be avoided. This method is discussed in 4.1.1.4.

In order to get a time estimate we put all the commands in a script, `install_by_copying.sh`<sup>13</sup>, and proceed to create the zone. The result can be seen in Listing 4.9. As can be seen the installation time using this method is below one second. The disk footprint is 8 MB which is 6 MB larger than the installation from section 4.1.1.2. This is a result of not using the *inherited-package-dir* directives in our zone configuration. The reason for not mounting directories from the global zone is that this causes unwanted files to be available in our zone.

```
# time sh install_by_copying.sh
real  0m0.737s
user  0m0.159s
sys   0m0.537s
# du -ms /zone/apache
8     /zone/apache
# find /zone/apache -type f | wc -l
65
```

**Listing 4.9 – Benchmarked installation by copying**

The advantage of using this method is that the files available in the zone are those necessary for the contained service, the small disk footprint, and the installation time.

The disadvantages are that the installed files will no longer be part of the Solaris packaging system. If the Apache packages are upgraded the binaries will have to be removed manually and the dependencies, which might have changed, needs to be resolved again. Only the files necessary for running the Apache service was copied. Because of this the zone boot process will have to be chosen based on the fact that the

---

<sup>13</sup> See Appendix B.9

SMF framework is not available. The zone boot process will be explored in section 4.1.2.

#### 4.1.1.4 BrandZ

From Solaris Express snv-49 and Solaris 10 Update 4 there is a new feature called Brand Zones<sup>14</sup>, BrandZ for short. BrandZ enables the user to use zones as containers for other operating systems than Solaris. In the first release there is support for Linux based operating systems through a branded zone type called *lx*. The original zones are now termed *native* zones. This section explores the use of BrandZ to provide a customized zone installation and boot up process. A thorough description of the BrandZ implementation is available at the OpenSolaris BrandZ community homepage<sup>15</sup>.

Before the introduction of BrandZ the binaries used for zone installation, `/usr/lib/lu//lucreatezone`, and the `init(1M)` process, `/sbin/init`, was hardcoded into the zone administration utilities. When support for BrandZ was added, `zoneadm(1M)` and `zonecfg(1M)` was modified to support different installation procedures for different types of zones. In the new system the previously hardcoded binaries are identified through a set of configuration files located in directories under `/usr/lib/brand`. The configuration for different BrandZ can be accessed from convenience functions in the library `libbrand.so`.

One of the problems identified in section 4.1.1.2 was that the use of `lucreatezone` was hardcoded into the `zoneadmd(1M)` binary. With the use of the BrandZ framework we can solve this problem by creating a brand type of our own that represents a minimal zone installation for a particular service. In this way we can continue to use the Solaris utilities `zonecfg(1M)` and `zoneadm(1M)` in the normal way when creating a zone and still benefit from using the `-P` flag with `lucreatezone`. We would still use the scripts created in section 4.1.1.2 but from the viewpoint of an administrator creating the zone would be done in the same way as when creating a native zone.

We demonstrate this use of the BrandZ framework by creating a new brand which we call *apache*, which will represent a minimal zone for the Apache web server. We start out with a copy of the native brand configuration from `/usr/lib/brand/native` and

---

<sup>14</sup> <http://www.opensolaris.org/os/community/brandz/> Accessed 2007-04-15

<sup>15</sup> <http://www.opensolaris.org/os/community/brandz/design/> Accessed 2007-05-03

customize it to fit our purpose. The new configuration is put in `/usr/lib/brand/apache`. At this point there is really only one change we want to do from the way native zones function; the behaviour when we issue the command:

```
zoneadm -z apache install
```

Analyzing the source for `zoneadmd(1M)` showed that it uses the function `brand_get_install()` from `libbrand.so` to get the install command, `/usr/lib/lu/luzonecreate`. Looking in `platform.xml` we see the line:

```
<install>/usr/lib/lu/lucreatezone %z</install>
```

which, excluding the `<install>`-tags, is what is returned by a call to the `brand_get_install()` function. We change this line to:

```
<install>/usr/bin/perl /opt/thesis/install_with_lucreatezone.pl /opt/thesis/apache.pkg  
%z</install>
```

Apart from this line we need to change the brand name from

```
<brand name="native"> to <brand name="apache"> in platform.xml and config.xml.
```

We can now create an apache zone using the `brand` directive when we configure the zone with `zonecfg(1M)`:

```
zonecfg:apache> set brand=apache
```

The definition of the native brand is that it is named “*native*” so even though the zones created in this way are in most way identical to native zones from the view of the utilities our brand is non-native. In some cases *native* zones are handled differently from non-native zones. An analysis of the source code for `zoneadm(1M)` and `zoneadmd(1M)` showed that these special cases does not affect the way we are using the BrandZ framework. This might of course change in the future and using non-native BrandZ in this way should be done with caution.

There is no difference in installation time and disk usage between a zone created by utilizing the BrandZ framework and the zones we created in the previous sections. The resulting installation is dependent on the method chosen by the `<install>`-directive in `config.xml`.

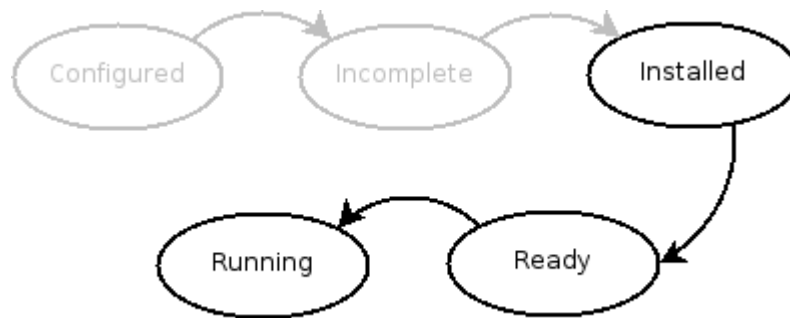
The advantage of using a branded zone is that the administrator may still use `zoneadm(1M)` when installing a zone.

## 4.1.2 Zone start-up

Our goal is to minimize the number of running processes in the running zone. The reason for this is to have as little overhead as possible when using a zone to contain a single service. By tailoring the way a zone is brought up the processes running in the zone can be kept to the bare essentials. This section gives a short description of what happens during the booting of a zone and proceeds to present different ways to tailor the boot process in order to achieve as little overhead as possible. The introduction is not a complete guide to what happens during the boot process. For a thorough discussion of the boot process see [1].

### 4.1.2.1 Background

After configuring and installing a zone in one of the ways described in section 4.1.1 the zone is in the *installed* state. This section deals with the rest of the state machine as depicted in Figure 4.3.



**Figure 4.3: Zone boot states**

During the boot process the zone's state is transitioned from *installed*, via *ready*, to *boot*. The administrator requests for the zone to be booted by issuing the commands:

```
# /usr/sbin/zoneadm -z <zone> ready
```

```
# /usr/sbin/zoneadm -z <zone> boot
```

The first command for transferring the zone to the *ready* state may be skipped as this transfer is done implicitly if the zone is in the *installed* state when being told to boot.

When transitioning from *installed* to *ready* the zone's environment is set up to prepare it for the user space boot sequence. When the zone is in the *ready* state all file systems are mounted into the zone's root directory, devices are configured and an instance of the kernel process *zsched* is created. The *zsched* process will be the root node in the process tree for the zone. At this point no user space processes exist in the zone.

When transitioning from state *ready* to state *running* the *zsched* process spawns the first user space process that will be responsible for bringing the system up. For native zones this transition is done in the traditional UNIX fashion by executing `/sbin/init`<sup>16</sup>. This process is responsible for bringing the user space part of the system up.

#### **4.1.2.2 Booting with the standard SMF repository**

Solaris 10 brings a new facility for managing system services, Services Management Facility (SMF), which has replaced the use of runlevels and rc scripts as a way to bring up services. It is an attempt to provide a more uniform way to manage services and their dependencies. A general introduction to SMF is provided in Chapter 3.

In addition to service start-up SMF provides a feature for restarting failed services. Since we are only running one service in the zone the service dependency management facility of SMF is not of much use to us. The restarter, however, is a feature that would increase the availability of the service. Because of this we can benefit from using SMF in the boot-up process even though our zone will only contain a single service. The downside of using SMF is that we get additional processes running in the zone.

The SMF runtime revolves around two processes. The first is `svc.startd(1M)` which is responsible for startup, restart and monitoring of system services. Apart from acting as a restarter it is responsible for handling the service dependency graph and start services in the right order based on these dependencies.

The second process is `svc.configd(1M)` which handles the service configurations and makes sure that configurations are persistent between system, or zone, restarts.

---

<sup>16</sup> This can be altered by using the `-i` flag when using `zoneadm` to boot the zone. This is explored in section 4.1.2.4

SMF replaces the use of runlevels and the responsibility of `init(1M)` is now to start the `svc.startd(1M)` process which is responsible for starting the services required for the system to be brought to a desired state. The former run levels are now replaced by what SMF calls *milestones*. These milestones represent a known, predictable state of the system similar to that of a runlevel.

In order to boot a zone using SMF we need a zone which has the SMF binaries installed. We choose the system we installed in section 4.1.1.2 where we installed the zone by running `lucreatezone` manually.

After booting the zone we end up with the process tree shown in Listing 4.10.

```
# ps -fz apache
  UID  PID  PPID  C   STIME TTY      TIME CMD
  root 18672  1    0 23:06:58 ?        0:00 /usr/sbin/nscd
  root 18799  1    0 23:06:59 ?        0:00 /usr/lib/inet/inetd start
  root 18733 18484  0 23:06:59 ?        0:00 /usr/lib/saf/sac -t 300
daemon 18658  1    0 23:06:58 ?        0:00 /usr/lib/crypto/kcfd
  root 18737 18733  0 23:06:59 ?        0:00 /usr/lib/saf/ttymon
  root 18722  1    0 23:06:59 ?        0:00 /usr/lib/utmpd
  root 18726  1    0 23:06:59 ?        0:00 /usr/sbin/cron
  root 18469  1    0 23:06:48 ?        0:00 zsched
  root 18481 18469  0 23:06:48 ?        0:00 /sbin/init
  root 18745  1    0 23:06:59 ?        0:00 /usr/sbin/syslogd
  root 18484  1    0 23:06:48 ?        0:01 /lib/svc/bin/svc.startd
  root 18865 18484  0 23:07:27 zoneconsole 0:00 /usr/bin/login
  root 18486  1    0 23:06:48 ?        0:05 /lib/svc/bin/svc.configd
#
```

**Listing 4.10 – Zone processes after regular boot**

Since the Apache service is not running yet all processes in the listing are overhead processes consuming resources. By using the SMF configuration utilities most of these processes may be removed since they are not needed by the service.

#### 4.1.2.3 Minimal SMF repository

Even though a zone is conceptually equal to a stand alone system and has support for using the same facilities for booting there are differences between the boot process of the global zone and that of the non-global zones. As an example, the plumbing of network interfaces for a non-global zone is done from the global zone when we transition to the *ready* state, not from the start-up scripts in the non-global zone. Since the global zone and the non-global zones use the same service manifests and start-up scripts there are checks to determine if the zone is the global zone or not. For example,

the scripts for plumbing the physical network interface, */lib/svc/method/net-physical*, and the loopback interface, */lib/svc/method/net-loopback*, contains the check:

```
smf_is_global_zone || exit $SMF_EXIT_OK
```

at the beginning of the scripts which have them exit immediately if not run from the global zone. This test combined with the fact that we are not using the zones as complete operating systems but as an isolated environment for a single service, enables us to create an Apache service manifest without any dependencies and as the single service in the SMF service graph.

To try SMF out we create a service manifest, *apache\_http\_jailed.xml*<sup>17</sup>, which is a modified version of *apache\_http.xml* from the default installation where we have removed the network and file system dependencies. The manifest is installed into the svc repository by issuing the commands in Listing 4.11.

```
# rm /zone/apache/root/etc/svc/repository.db
# svccfg
svc:> repository /zone/apache/root/etc/svc/repository.db
svc:> import /opt/thesis/apache2-httpd-jailed.xml
```

#### Listing 4.11 – Setup of SMF repository for Apache http

When we boot the zone we have five processes running as shown in Listing 4.12. The `sulogin(1M)` process is running since we are in single user mode. The `<defunct>` process is a zombie child process of `sulogin(1M)` which have not been collected by a call to one of the `wait(3C)` family functions. These two are processes that we do not need but they are created by the `init(1M)` process over which we have no control. To get rid of them we need to replace the `/sbin/init` binary which will be explored in 4.1.2.4. We can still get rid of the `sulogin(1M)` functionality by replacing the `/sbin/sulogin` binary with a dummy program that does nothing, but the number of running processes will still be the same except for the zombie process. Removing `sulogin(1M)` would rid us of the console login functionality when using the `-C` flag to `zlogin(1)`.

---

<sup>17</sup> See Appendix B.1

```

# zoneadm -z apache boot
# ps -fz apache
  UID    PID  PPID   C    STIME TTY          TIME CMD
  root 17978 17977   0      - ?           0:00 <defunct>
  root 17970     1   0 21:09:42 ?           0:00
/lib/svc/bin/svc.startd
  root 17954     1   0 21:09:42 ?           0:00 zsched
  root 17977 17970   0 21:09:42 zoneconsole 0:00 sulogin
  root 17972     1   0 21:09:42 ?           0:00
/lib/svc/bin/svc.configd
  root 17966 17954   0 21:09:42 ?           0:00 /sbin/init
#

```

**Listing 4.12 – Zone processes with customized SMF repository**

There are no Apache processes running at this point. The reason for this is that the Apache service could not be started due to lack of configuration. In a real setup we would configure the Apache service to serve our desired pages. In this setup however, we are content with getting the service up and serving the default web pages. The process of preparing, and starting the service is shown in Listing 4.13.

```

# cp /zone/apache/root/etc/apache2/httpd.conf-example \
  /zone/apache/root/etc/apache2/httpd.conf
# mkdir /zone/apache/root/var/run/apache2
# echo "apache 192.168.0.1" >> /zone/apache/root/etc/hosts
# zlogin svcadm clear apache2
# ps -fz apache
  UID    PID  PPID   C    STIME TTY          TIME CMD
  root 18125     1   0 21:25:05 ?           0:00 /usr/apache2/bin/httpd -k
start
webservd 18130 18125   0 21:25:06 ?           0:00 /usr/apache2/bin/httpd -k
start
webservd 18129 18125   0 21:25:06 ?           0:00 /usr/apache2/bin/httpd -k
start
  root 17978 17977   0      - ?           0:00 <defunct>
  root 17970     1   0 21:09:42 ?           0:00 /lib/svc/bin/svc.startd
  root 17954     1   0 21:09:42 ?           0:00 zsched
webservd 18126 18125   0 21:25:06 ?           0:00 /usr/apache2/bin/httpd -k
start
webservd 18128 18125   0 21:25:06 ?           0:00 /usr/apache2/bin/httpd -k
start
  root 17977 17970   0 21:09:42 zoneconsole 0:00 sulogin
  root 17972     1   0 21:09:42 ?           0:00 /lib/svc/bin/svc.configd
webservd 18127 18125   0 21:25:06 ?           0:00 /usr/apache2/bin/httpd -k
start
  root 17966 17954   0 21:09:42 ?           0:00 /sbin/init
#

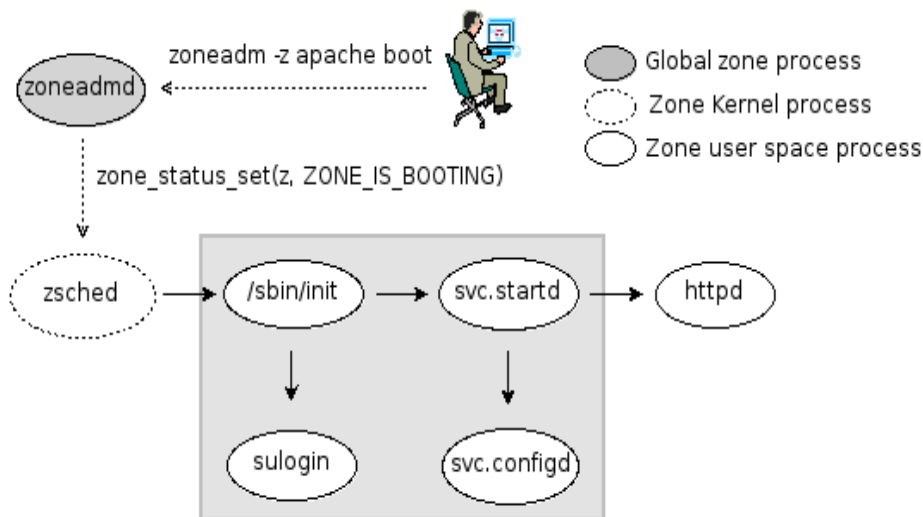
```

**Listing 4.13 – Zone processes after customized boot**

Now the Apache service, */usr/apache2/bin/httpd*, is running. This is the best that can be done without sacrificing the SMF restarting feature and altering the *sulogin(1M)* or *init(1M)* binaries.

#### 4.1.2.4 Replacing /sbin/init

Using the method in the previous section there are at least four processes in addition to the Apache processes; `init(1M)`, `svc.startd(1M)`, `svc.configd(1M)` and `sulogin(1M)`. The start-up sequence of these processes is shown in Figure 4.4. Dashed arrows indicate a subject performing some action affecting an object. Solid arrows indicate process creation. In this section we try to get rid of the processes in the grey box. This will decrease the number of running processes at the expense of losing the restarting feature of SMF.



**Figure 4.4: Zone boot flow**

As was described earlier a zone is booted by the administrator using the `zoneadm(1M)` utility. The `zsched` process was started when the process entered the *ready* state. When `zoneadm(1M)` changes the zone state<sup>18</sup> to `ZONE_IS_BOOTING` by invoking the `zone()` system call with the `ZONE_BOOT` command<sup>19</sup> the `zsched` kernel process, which was asleep waiting for this event, is awoken. This will cause `zsched` to create a new kernel process which, eventually, will call the `start_init_common()` function. This is interesting because this is the rendez-vous point in the kernel code paths taken when creating the `init(1M)` process for the global zone, i.e., the PID 1 process, and when creating the `init(1M)` process in a non global zone. From this point on they are

<sup>18</sup> Note that this is the zone's kernel state. This is not the same states that are observable in user land, e.g. the one we see when we issue `zoneadm list -vi`.

<sup>19</sup> See `src/uts/common/cmd/zoneadm/zoneadm.c` and `src/uts/common/os/zone.c`.

essentially doing the same thing provided that the `/sbin/init` binary and the SMF configuration are equal in the global and the non-global zones<sup>20</sup>.

In order to remove the processes in the grey box we need to make the `zsched` process invoke the Apache binaries instead of `/sbin/init`. It turns out that there is a simple solution to this. When running `zoneadm(1M)` we can supply a path to a binary to be executed in place of `/sbin/init`. This is done by issuing

```
# zoneadm -z apache -- -i /path/to/binary
```

This is similar to the way you would start a process in a `chroot(2)`ed or FreeBSD `jail` environment.

In order to test this command without being burdened by the complexity of Apache a small program, `prog41`<sup>21</sup>, was written that writes some data to `/output.txt` and goes to sleep.

A booted zone with this program as the init process is shown in Listing 4.14. The text in `/output.txt` suggests that the program is running as expected. It shows that there are only two processes running of which `prog41` is the only process running in user space. In addition to this the `zoneadmd(1M)` associated with the zone is running in the global zone. This test demonstrate that it is indeed possible to only have the essential processes running in the zone.

```
# /usr/sfw/bin/gcc -o prog41 prog41.c
# cp prog41 /zone/apache/root/
# zoneadm -z apache boot -- -i /prog41
# cat /zone/apache/root/output.txt
Hello zone!
# ps -fz apache
  UID  PID  PPID   C   STIME TTY          TIME CMD
  root 21746 21734   0 19:19:27 ?           0:00 prog41
  root 21734     1   0 19:19:27 ?           0:00 zsched
#
```

**Listing 4.14 – Booted zone with replaced `/sbin/init`**

There are still some issues that need to be resolved. When using `zoneadm(1M)` there is no way to pass arguments to the binary that are to be run instead of `/sbin/init`. For some programs this is no problem, but for others it is a requirement. Apache for instance

---

<sup>20</sup> As was described in 4.1.2.1 there are checks in the start up scripts that makes them behave differently when run from non global zones so this is not entirely true. From a kernel perspective this is not relevant.

<sup>21</sup> See Appendix B.3

is usually started by the `/usr/apache2/bin/apachectl` script. This script use `/bin/sh` as its interpreter, hence we need to execute the `/bin/sh` and pass “`/usr/apache2/bin/apachectl start`” as arguments. This problem can be solved by using a small helper application that reads the binary path and its arguments from a file and executes the real binary, with arguments, by calling one of the `exec(2)` family functions. A small program was written that reads an argument list from the file `/prog42.cfg`. These arguments, of which the first is the absolute path to the binary, are extracted and passed to `execv(2)` for execution<sup>22</sup>.

The `/usr/apache2/bin/apachectl` script will execute the `/usr/apache2/bin/httpd` binary which will `fork(2)` and have the parent process return. This will cause `apachectl`, and as a consequence `/bin/sh`, to return. From the kernel's point of view `/bin/sh` is the zone's `init(1M)` process<sup>23</sup> and mandates that this process is running at all times. When the process that are recorded as the zone's `init(1M)` process exits the kernel will restart it, thus creating another `prog42` process. This will go on until the zone is halted with a massive amount of kernel warning messages in the system log. To prevent this behaviour we added functionality to `prog42` that, if the binary is renamed to `prog42_fork`, will `fork(2)` before calling `execv(2)` in the child process. The parent will be put in an infinite loop calling `wait(3C)` at regular intervals. This keeps the `init` process running which solves the problem of the process restarts.

It turns out that calling `execv(2)` in the `init` process is not a good idea for other reasons; In Solaris the `init` process becomes the parent of all orphaned processes, i.e., processes whose parent has exited before the child process. In order for other processes to be able to get the exit status of terminated ones the processes are not removed when they exit. To be removed from the kernel a different process needs to collect their exit status by calling one of the `wait*()` functions. This is normally done by the default `/sbin/init` process and to prevent orphaned processes from staying around forever, occupying system resources, we need to call this function in our own customized `init` process. As was described earlier this is the responsibility of the `prog42_fork` parent process after it `fork(2)`s.

---

<sup>22</sup> See Appendix B.4

<sup>23</sup> This means that the PID of the `/bin/sh` process is set in `zone_proc_initpid` in the zone kernel structure.

One interesting possibility not covered in these experiments is to use the fact that the kernel restarts the init process when it exits as a restarting feature for the service. This would certainly be possible if the program is design with this in mind. For instance, the main process of the service should remain running during the service lifetime.

The advantage of using the techniques described in this section is that we have eliminated all but two overhead processes.

The disadvantages are that we have sacrificed the restarting feature of SMF and that we are using undocumented behaviour of the operating system. Since this is not a supported way of using zones we have to resort to small hacks to get things working, e.g. the need for helper applications to allow argument passing when executing a binary. One might argue that the `prog42_fork` program is a crippled version of the SMF `svc.startd(1M)` program without the restarting feature.

### **4.1.3 Limitations**

In the experiments we have not considered the use of ZFS pools when creating the zones. Using the ZFS clone feature [9] may improve the zone installation time drastically when a full zone installation is performed. This, however, assumes that there is a previously installed system to clone.

Using the number of processes as a general guideline for memory and processing consumption is not very accurate. For instance, if there are multiple processes running the same program the operating system will load only one instance of the binary into physical memory and map this same memory to each process. Thus, having a second instance of the same binary will not add as much overhead as the first instance. As most binaries are dynamically linked this sharing of physical memory is extended to the libraries shared between processes running different binaries. This only affects the memory allocated for loading the actual binary multiple times. If the process uses a large amount of heap memory multiple instances of the same binary will still incur a larger overhead. Besides, two or more memory and processing conservative processes might consume less system resources than a single resource intensive process.

#### 4.1.4 Conclusion

With the experiments in this section we have demonstrated that the Solaris zone facility can be used to create light weight isolated environments. By using methods other than those officially supported we are able to determine which files will be available in the zone on a file basis providing an installation similar to `chroot(2)` or the FreeBSD jail environment. By using an alternative `/sbin/init` binary we are able to get the overhead processes associated with a zone down to `zsched` and `zoneadmd(1M)`. However, when taking this approach we need to consider what effect it has on our application. For instance, the process running as the `/sbin/init` replacement is not allowed to exit since the kernel mandates that this process is running at all times.

## 4.2 SELinux: Exploit simulation

To experiment with the SELinux framework a small application was written that simulates a severely flawed system daemon. This daemon allows a remote user that connects to a specific network port to execute any application on the target system without having to authenticate himself/herself first. Note that this application does not try to illustrate any common attack method, the purpose is only to show how the (mis-) functionality of the daemon can be cut down to do a minimal amount of damage.

### 4.2.1 Exploit walk-through

We shall start with a brief walk-through of the application to see what it does and then take a look at how an SELinux policy is set up for it.

#### 4.2.1.1 Log file

To be able to see what happens when the daemon is running the application logs interesting events and the client-server communication to a file `/var/log/exploit.log`. A root user in the `sysadm_r` role can monitor this file with the `tail -f` command:

```
[root@fc5 ~] # tail -f /var/log/exploit.log
```

#### 4.2.1.2 Daemonization

After opening the log file the application daemonizes by performing the usual `fork()-setsid()-fork()` sequence. This sequence detaches the daemon from the login session of the user which allows it to run in the background even after the user logs out.

#### 4.2.1.3 Establish server socket

The application then creates a socket and tries to bind it to a port. When experimenting with the application a user will start and stop the application frequently, sometimes by force (`kill -9`). Since this forced termination might render ports temporarily disabled (`time_wait` state) the port range 9990-9999 is tested. Normally, daemons only bind its server socket to one specific, known port number.

If the application finds a usable port the application enters the main loop.

#### **4.2.1.4 Main loop**

The main loop waits for incoming connections on its listening socket. On a successful connect a child process is created to handle the communication with the remote client. The main process will then go back to listening for more connections.

#### **4.2.1.5 Remote client handler**

The remote client handler starts by writing a prompt to the client asking for a program to run. The remote user can enter any command here that the regular root user can run, be it dangerous or not. For example, entering `/bin/bash` starts an interactive shell that can be used to enter further commands. In this state the security of a normal system is completely broken. It then prepares and opens a pseudo-terminal which will be used to communicate with the program after it is `exec()`ed.

#### **4.2.1.6 fork() again**

The client handler will now fork again. The parent side will go into a loop that moves data between the socket and pseudo-terminal descriptors. The data moved is also written to the log file so that the user can see what is going on.

#### **4.2.1.7 Another child**

The child process starts with splitting up the command line into a program plus arguments. It then opens the slave side of the pseudo-terminal and `dup()`s it to the `stdin`, `stdout` and `stderr` file descriptors so that all input and output from the command will go to the remote client (via the parent process). Finally, the program is `exec()`ed and the exploit is complete.

### **4.2.2 Policy syntax**

Because every interaction with the system an application needs to perform have to be specifically allowed, the policy for an application can consist of thousands of statements. Writing all these statements directly in the policy would be exhausting and error-prone. Therefore, a layered system of macros has been developed in the `m4` macro language to ease policy development. In this system, low-level services, system services and applications are collected into its own modules. These modules consists of three types of files: `module.te`, `module.if` and `module.fc`.

#### 4.2.2.1 module.te

This file contains the new domains and types that will be used by the module and the rules that define what the application is allowed to do. This file will use macros in lower-level modules instead of explicit allow definitions.

#### 4.2.2.2 module.if

The domains and types used in a policy are private to the module and should not be defined elsewhere. Because other applications may need access to these types this file contains macro functions that other policy modules can call.

#### 4.2.2.3 module.fc

This file contains the context labels of files related to the application. That includes the application executable itself and all files it can read or create, such as configuration or log files.

### 4.2.3 Writing the policy module for exploit

#### 4.2.3.1 exploit.te

All policy modules begin with the `policy_module()` macro. The parameters to this macro define the base name of the module and the version. The base name is used for handling the module in the kernel. A list of currently loaded policy modules can be seen by executing the command

```
[root@fc5 ~]# semodule -l
acct      1.0.0
alsa      1.0.1
amanda    1.2.1
...
```

**Listing 4.15 – Loaded modules**

#### 4.2.3.2 Execution domain

The next 3 lines defines the domain of the application, the entry point to that domain and a macro is called that sets up a working environment for the domain.

```
type exploit_t;
type exploit_exec_t;
init_daemon_domain(exploit_t, exploit_exec_t)
```

**Listing 4.16 – Defining the application domain**

This macro expands to a large number of access rights, for example access to the binary, access to safe devices (`/dev/null` and `/dev/zero`) and it also sets up the actual transition to the application's domain (`exploit_t`).

#### 4.2.3.3 Log file

The log file is defined to have the type `exploit_log_t`. Opening the log file requires specific permissions to access the `/var/log` directory and to create and write to files.

```
type exploit_log_t;
logging_log_file(exploit_log_t)
allow exploit_t exploit_log_t:dir rw_dir_perms;
allow exploit_t exploit_log_t:file create_file_perms;
logging_log_filetrans(exploit_t,exploit_log_t,{ file dir })
```

**Listing 4.17 – The log file type**

The `logging_log_filetrans` macro sets up a transition rule so that when a process in the `exploit_t` domain creates a file in a `var_log_t` directory the file will automatically be relabeled to `exploit_log_t`.

#### 4.2.3.4 Network access

The application needs permission to create a socket and bind it to a local port. The policy will allow the application only to bind to ports labeled `exploit_port_t`.

```
allow exploit_t self:tcp_socket create_stream_socket_perms;
allow exploit_t exploit_port_t:tcp_socket name_bind;
corenet_tcp_bind_inaddr_any_node(exploit_t)
corenet_tcp_sendrecv_all_if(exploit_t)
corenet_tcp_sendrecv_all_nodes(exploit_t)
corenet_tcp_sendrecv_all_ports(exploit_t)
corenet_non_ipsec_sendrecv(exploit_t)
```

**Listing 4.18 – The network port type**

Labeling the ports the application can use has two benefits:

- 1) It prevents other applications from using this application's port
- 2) It prevents this application to open unexpected ports, something that is sometimes used by real exploits for example to fetch a root kit from the attacker's server.

The following command is used to label the ports used by exploit:

```
[root@fc5 ~] # semanage port -a -t exploit_port_t -p tcp 9990-9999
```

#### Listing 4.19 – Labeling a network port

#### 4.2.3.5 Pseudo-terminal

The application needs to open and use a pseudo-terminal, so we need to allow that too:

```
type exploit_devpts_t;  
term_pty(exploit_devpts_t)  
term_create_pty(exploit_t,exploit_devpts_t)  
allow exploit_t exploit_devpts_t:chr_file { rw_file_perms setattr };
```

#### Listing 4.20 - The pseudo-terminal type

#### 4.2.3.6 Miscellaneous permissions

A few other permissions are also needed: The logging functions need access to the systems locale information and since the application is built with shared libraries, access to the dynamic linker is also needed

```
miscfiles_read_localization(exploit_t)  
libs_use_ld_so(exploit_t)  
libs_use_shared_libs(exploit_t)
```

#### Listing 4.21 – Miscellaneous permissions

#### 4.2.3.7 Extra permissions

Up to this point the base policy is finished. Now the application can start and run normally, but it cannot do anything useful. For the purpose of demonstration, the following lines are added to the policy to make it useful:

```
corecmd_search_bin(exploit_t)  
corecmd_exec_ls(exploit_t)  
corecmd_exec_bin(exploit_t)  
#corecmd_exec_shell(exploit_t)
```

#### Listing 4.22 – Extra permissions

The first line allows exploit\_t processes to search bin\_t directories. The bin\_t directories are /bin and /usr/bin.

The second line allows exploit to execute binaries labeled ls\_exec\_t, which is the ls command.

The third line does the same for binaries labeled `bin_t`, which are most other 'safe' binaries, for example the touch command.

Lastly, the commented fourth line allows exploit to execute a command shell. If this line is uncommented a remote user gets a shell to play with. This shell will however still be subject to the `exploit_t` policy restrictions.

#### 4.2.4 Analysis

The policy restricts in fine detail what the application can do. It is no longer allowed to execute any other application unless the policy explicitly allows it. This behaviour shall now be demonstrated with the help of a few system commands:

```
[root@fc5 ~]# ls -Z /bin/ls /bin/bash /bin/touch
-rwxr-xr-x root root system_u:object_r:shell_exec_t /bin/bash
-rwxr-xr-x root root system_u:object_r:ls_exec_t /bin/ls
-rwxr-xr-x root root system_u:object_r:bin_t /bin/touch
```

**Listing 4.23 – Commands used for testing**

Note that the prompt indicates that the root user is logged in and that these applications are allowed to be executed by the root user, as per the regular DAC. The policy for exploit however allows the application only to execute binaries labeled `bin_t` and `ls_exec_t`, but not others such as `shell_exec_t`.

```
$ telnet fc5 9990
Connected to fc5.
Escape character is '^]'.
Enter command > /bin/ls /
bin dev home lib64 media mnt opt root selinux sys usr
boot etc lib lost+found misc net proc sbin srv tmp var
Connection to fc5 closed by foreign host.
```

**Listing 4.24 – Testing /bin/ls /**

Listing the files in the root directory is no problem, since `/bin/ls` is labeled `ls_exec_t` and that is allowed by the policy.

```
$ telnet fc5 9990
Connected to fc5.
Escape character is '^]'.
Enter command > /bin/bash
Connection to fc5 closed by foreign host.
```

**Listing 4.25 – Testing /bin/bash**

Executing `/bin/bash` is not allowed, however. The log file will explain what just happened:

```
2007-04-13 12:35:31 Executing '/bin/bash'
2007-04-13 12:35:31 execvp:Permission denied
```

#### **Listing 4.26 – Log file after `/bin/bash`**

The log says that executing `/bin/bash` was not allowed, which is correct since executing `shell_exec_t` is not allowed by the policy.

```
$ telnet fc5 9990
Connected to fc5.
Escape character is '^]'.
Enter command > /bin/touch /tmp/foo
/bin/touch: cannot touch `/tmp/foo': Permission denied
Connection to fc5 closed by foreign host.
```

#### **Listing 4.27 – Testing `/bin/touch /tmp/foo`**

In this example, since `/bin/touch` is labeled `bin_t` and the policy allows executing such binaries the `execvp()` call was successful. But processes in the `exploit_t` domain are not allowed to write in the `/tmp` directory, so the command failed anyway.

## 5 . Evaluation

In the problem statement we set up four different criteria for the comparison. This section will evaluate the systems we have explored in the previous chapters based on these criteria.

### 5.1 Administration

Using zones in the way intended by the creators is easily done using well defined configuration utilities. In fact one of the goals of the zone implementation was to prioritize ease of use over increased flexibility [1]. However, these utilities do not allow the administrator to create customized zones such as those described in section 4.1.1.2 and 4.1.1.3. The administration work to achieve the minimal zone installation requires substantial work and validation from the responsible administrator.

Fedora Core 5 comes with policy modules for many common applications and more are continuously developed. For instance, the Apache HTTP server has a default policy that is automatically used if the service is started. However, if no module is available for a piece of software it must be written by an administrator. This is a fairly complex procedure and requires intimate knowledge of both the SELinux system and the application the module is for. There are a couple of useful tools that will tell the developer what rules are needed, but such a trial-and-error approach could easily produce policies that allows more than necessary.

Even if there is a policy module for the application the default rule set may need to be modified to fit the environment. The reference policy comes with a set of flags that

allow the administrator to change the behavior of the policy without rewriting it, potentially opening up unforeseen security holes.

Multiple instances of the same service all run in the same domain and are therefore not isolated from each other. To solve this problem the administrator will have to create a new policy for each instance. This would be done by replicating the original policy and changing the name of every domain and type.

SELinux does not provide namespace isolation which may be a problem when running multiple instances of the same application. For instance, if the application has a hardcoded path to a PID file in `/var/run/` there will be a resource conflict which in the worst case will prevent more than one instance of the application at the same time.

## 5.2 Validation

Assuming that zones function according to specifications the zone by definition provides an isolated environment. Thus, the isolation requirement is fulfilled. It is possible to have unintended information leaks between two zones by mounting the same directories in multiple zones but this is trivial to validate by observing the mounted file systems.

Writing SELinux policy modules is complicated and it is easy to give an application more access than absolutely necessary. However, the reference policy developed by Tresys Technology has improved this situation with its logical layered approach. These layers allow the policy writer to be very precise in the scope of the requested functionality. Using macros in the lower layers allows very narrow functionality to be enabled, such as the `dev_rw_null()` macro, which allows access to the `/dev/null` device. Using higher level macros enables more general functionality, such as the `init_daemon_domain()` macro that sets up all permissions commonly needed by daemonized applications.

## 5.3 Scalability/Overhead

Section 4.1 did extensive analysis on zone installation and booting with regards to disk usage and overhead processes running in the zone. It was shown that it is possible to

create environments that contain only the essential files for a service and that it is possible to keep the overhead processes in a zone down to the bare minimum.

Few reports have measured the system call overhead. The Fedora project's SELinux FAQ [17] claims the overhead to about 7%.

Both systems impose overhead processing for every process running in the system. A check is always made to make sure that a resource is accessible. In Solaris this is done by at most two integer comparisons<sup>24</sup>, adding a minimal overhead. The `zsched` kernel process goes to sleep after creating the zone's init process, waiting for the zone to be halted, and consumes no processing resources when the zone is in the *running* state. With SELinux the access control is more complex due to the fine-grained nature of the SELinux implementation. To search through the policy requires significant effort. This is accelerated by the Access Vector Cache but the lookup still incur larger overhead than the Solaris implementation as the estimated 7% indicates. Note that this does not mean that Solaris has better performance, the overhead is measured against the same system without the SELinux or zone implementation.

According to the Sun System Administration Guide [9] the disk space set aside for a sparse root zone installation should be around 100 MB. As was demonstrated in Chapter 4 this number can be reduced by customizing the installation procedure. With SELinux there is an initial cost where all system files are required to have a security label which is stored in an extended attribute associated with the inode of the file. However, this overhead is independent of the size of the policy.

The Sun System Administration Guide [9] suggests 40 MB of additional RAM per zone. In Chapter 4 we showed that it is possible to minimize the number of processes running in a zone, thus reducing the consumed memory resources. In the experiment where we replaced the `init(1M)` process the overhead memory consumption consists of the memory taken by the `zsched` process and the `zoneadmd(1M)` process. The SELinux binary strict policy file is just over 1 MB in size and requires a negligible amount of kernel memory together with the access vector cache<sup>25</sup>.

---

<sup>24</sup> See the `INGLOBALZONE` macro in <http://cvs.opensolaris.org/source/xref/onnv/aside/usr/src/uts/common/sys/zone.h?r=3792> line 494, accessed 2007-05-07

<sup>25</sup> See `security/selinux/avc.c` in the Linux 2.6 kernel source tree

## 5.4 Achieved protection

Zones, when combined with the Solaris resource management facilities [9], provides a way too protect a system that contains multiple services, each contained in a separate zone, from denial of service attacks by exhausting the system's resources. This is achieved by partitioning system resources with regards to the zones. The same is achieved in Fedora by using the `cpuset` to allocate processor and memory resources for a specific process.

Using the IP filtering feature of Solaris a zone's ability to reach the network can be limited. By using stateful packet inspection in the filtering software it is possible to prevent the zone from generating network traffic that is not associated by a state created by a connection attempt from the outside. In the Apache scenario for example, we are only interested in sending responses to HTTP requests, and perhaps resolve some addresses using DNS. By configuring the IP filtering software it is possible to prevent a exploited service in a zone to be used as a node in a distributed denial of service attack or as a trampoline system for attacks against other systems. Usually it is not advisable to run this filtering software on the same system as the service we are trying to confine. An attacker that manages to achieve administrator privileges, i.e., root access, through some privilege escalation method are able to just turn the filtering software off, thus circumventing the whole protection scheme. When running the service in a zone, achieving root access will not allow the attacker to turn the filtering software off since this is running in the global zone.

When an attacker has compromised a system the administrators of the system are faced with the possibility that the system has been back doored. This often result in reinstalling the system since proving the absence of back doored software can be very hard.

The fine-grained nature of SELinux have the potential of being very efficient in a correctly configured policy. If an attacker manages to upload code to the target machine that code will still not be able to do anything the policy does not allow.

SELinux does not protect against denial-of-service attacks. An attacker may be able to make a vulnerable application to go into an endless loop or crash. In such cases a watchdog/heart beat mechanism must be used.

SELinux does not only protect against outside threats. The flexibility of the SELinux framework provides several security models at once and allows for implementing Role-Based Access Control and Multi-Level Security to protect against internal threats.

Multiple instances of the same service all run in the same domain and are therefore not isolated from each other. To solve this problem the administrator will have to create a new policy for each instance. This would be done by replicating the original policy and changing the name of every domain and type.



## 6 Conclusions and Future Work

### 6.1 Conclusions

We have demonstrated that both Solaris 10 Zones and Fedora with SELinux policies can be used to isolate processes. An important difference becomes apparent when running several instances of the same application. In Solaris every instance runs in its own zone which isolates them from each other, when in the SELinux case every instance runs in the same domain and therefore can influence each other. To solve this problem a new policy has to be made for each instance of the application. Another problem when running multiple instances of the same application in SELinux is the lack of namespace isolation which may lead to conflicts in resource use.

If the default set-up mechanisms are sufficient, i.e., there is an SELinux policy and zones are used according to documentation, the administration of both systems does not require in-depth knowledge of the systems. However, if a customized configuration is needed the administrator needs extensive knowledge of the underlying mechanisms. When writing customized SELinux policies each program needs a different set of privileges whereas the Solaris zone configuration is done in the same way regardless of the services running in the zone. In addition to knowing how to write SELinux policies the developer needs to be very familiar with the software implementation. This can be a huge problem if one needs to develop a policy for a proprietary program where the developer does not have access to the source code. In this case the developer needs to develop the policy by trying to get the program to execute all branches in order to find

all requirements for the program. Solaris Zones supports proprietary software transparently without requiring any modifications.

Neither Zones nor SELinux policies imposes such overhead that they become critical for the scalability of the systems using them. As demonstrated in the experiments, zones installations can have a minimal footprint by using techniques such as loopback mounting or tailored installation procedures. Likewise, the number of SELinux domains and types has a very small impact.

The performance overhead of running applications within a zone compared to an application in the global zone is none since they are treated the same way. A small amount of memory is required for the processes needed for a running zone. With SELinux the overhead varies between applications. A computationally intensive application does not impose as much overhead as an I/O intensive one. The overhead also varies between different system calls.

## **6.2 Methodology Limitations**

In a way it would be more accurate to use the Apache HTTP service in both experiments but since the Apache policy was already written we chose to examine the policy writing process by creating our own application and the policy for it from scratch. To show that the policy worked, we would have had to find an exploitable bug in the Apache software and a way to take advantage of it. Using our own deliberately flawed application enabled us to test and verify the policy. In addition to this, the Apache policy is very complex because it supports interaction with several other software packages and would not be a good starting point to get to know the policy framework.

A benchmark would have been desired to get a more accurate measurement of the overhead using the containment strategies.

The Solaris experiments do not use the ZFS file systems which, in certain cases, could improve the zone creation time. We also did not make use of the resource management facilities for Solaris Zones.

This thesis only focuses on the containment aspect of securing an exposed service. When deciding which system to use, other aspects need to be considered such as performance, level of support and administrative staff experience.

### **6.3 Future Work**

There are system level virtualization software packages available for the Linux platform such as OpenVZ or the VServer kernel patch sets. Because these alternatives are much closer in scope and functionality to the Solaris Zones technology a comparative study could uncover interesting facts.

In more recent Fedora releases the support for MLS has improved. Future work would be a comparison between the MLS functionality of Solaris, which is implemented using zones, and MLS functionality in SELinux.

This thesis lacks a performance evaluation of the two systems. A future study could focus more on this aspect to get a more complete comparison of the two systems.

Both Solaris and SELinux have capabilities of role based access control. A future comparison could evaluate a user environment implementing these security features.



## 7 References

- [1] Price, Daniel and Tucker, Andrew. *Solaris Zones: Operating System Support for Consolidating Commercial Workloads*. Proceedings of USENIX LISA 2004, November 2004.
- [2] Foxwell, Harry J. and Rozenfeld, Isaac. *Slicing and Dicing Servers: A Guide to Virtualization and Containment Technologies*. Sun BluePrints, October 2005.
- [3] Romack, Rob. *Service Management Facility (SMF) in the Solaris™ 10 Operating System*. Sun BluePrints, February 2006.
- [4] Sun Microsystems. *Predictive Self-Healing in the Solaris™ 10 Operating System*, September 2004.
- [5] Victor, Jeff. *Solaris™ Containers Technology Architecture Guide*. Sun BluePrints, May 2006.
- [6] Kamp, Poul-Henning and Watson N.M. Robert. *Jails: Confining the Omnipotent root*.
- [7] Provos, Niels. *Improving Host Security with System Call Policies*, November 2002.
- [8] Garfinkel, Tal. *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools*. Proceedings of the Internet Society's 2003 Symposium on Network and Distributed System Security, San Diego, CA, February 2003.
- [9] Sun Microsystems. *System Administration Guide: Solaris Containers - Resource Management and Solaris Zones*. November 2006.
- [10] Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Reading Massachusetts: Addison Wesley Longman Inc, 1993. ISBN 0-201-56317-7.
- [11] NSA. SELinux Homepage. <http://www.nsa.gov/SELinux/>. Accessed 2007-05-08.

[12] Smalley, Stephen and Vance, Chris and Salamon, Wanye. *Implementing SELinux as a Linux Security Module*. 2006.

[13] Tresys Technology. Tresys homepage, <http://www.tresys.com>. Accessed 2007-05-08.

[14] PeBenito, Christopher J. and Mayer, Frank and MacMillan, Karl. *Reference Policy for Security Enhanced Linux*. 2006.

[15] SELinux reference policy. <http://serfpolicy.sourceforge.net>. Accessed 2007-05-08.

[16] Novell. OpenSuSE homepage. <http://en.opensuse.org/AppArmor>. Accessed 2007-05-08.

[17] Redhat. Fedora SELinux FAQ, <http://docs.fedoraproject.org/selinux-faq-fc5/#id2965028>. Accessed 2007-05-08.

[18] Faden, Glenn. *Comparing the Multilevel Security Policies of the Solaris Trusted Extensions and Red Hat Enterprise Linux Systems*. Sun BigAdmin, February 2007.

## Appendix A – Glossary

AVC	Access Vector Cache
DAC	Discretionary Access Control
DDOS	Distributed Denial of Service
DOS	Denial of Service
HTTP	Hyper Text Transport protocol
IP	Internet Protocol
IPC	Inter-process Communication
LSM	Linux Security Modules
MAC	Mandatory Access Control
MLS	Multi-Level Security
PID	Process ID
RBAC	Role Based Access Control
SELinux	Security Enhanced Linux
SMF	Solaris Management Facilities
SUID	Set User ID
TCP	Transport Control Protocol
UDP	User Datagram Protocol
UUID	Universally Unique ID
XML	Extensible Markup Language
ZFS	Zettabyte File System



## Appendix B – Solaris Resources

### B.1 apache\_http\_jailed.xml

```
<?xml version="1.0"?>
<!--
CDDL HEADER START

The contents of this file are subject to the terms of the
Common Development and Distribution License (the "License").
You may not use this file except in compliance with the License.

You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
or http://www.opensolaris.org/os/licensing.
See the License for the specific language governing permissions
and limitations under the License.

When distributing Covered Code, include this CDDL HEADER in each
file and include the License file at usr/src/OPENSOLARIS.LICENSE.
If applicable, add the following below this CDDL HEADER, with the
fields enclosed by brackets "[]" replaced with your own identifying
information: Portions Copyright [yyyy] [name of copyright owner]

CDDL HEADER END
-->

<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">
<!--
  Copyright 2006 Sun Microsystems, Inc. All rights reserved.
  Use is subject to license terms.

  ident      "@(#)http-apache2.xml      1.6      06/03/18 SMI"
-->

<service_bundle type='manifest' name='SUNWapch2r:apache'>
<service
  name='network/http'
  type='service'
  version='1'>
  <!--
    Because we may have multiple instances of network/http
    provided by different implementations, we keep dependencies
    and methods within the instance.
  -->

  <instance name='apache2' enabled='true'>
    <exec_method
```

```

        type='method'
        name='start'
        exec='/lib/svc/method/http-apache2 start'
        timeout_seconds='60' />

<exec_method
  type='method'
  name='stop'
  exec='/lib/svc/method/http-apache2 stop'
  timeout_seconds='60' />

<exec_method
  type='method'
  name='refresh'
  exec='/lib/svc/method/http-apache2 refresh'
  timeout_seconds='60' />

<property_group name='httpd' type='application'>
  <stability value='Evolving' />
  <propval name='ssl' type='boolean' value='false' />
</property_group>

<property_group name='startd' type='framework'>
  <!-- sub-process core dumps shouldn't restart
        session -->
  <propval name='ignore_error' type='astring'
    value='core,signal' />
</property_group>

</instance>

<stability value='Evolving' />

<template>
  <common_name>
    <loctext xml:lang='C'>
      Apache 2 HTTP server
    </loctext>
  </common_name>
  <documentation>
    <manpage title='httpd' section='8'
      manpath='/usr/apache2/man' />
    <doc_link name='apache.org'
      uri='http://httpd.apache.org' />
  </documentation>
</template>
</service>
</service_bundle>

```

## B.2 zone\_set\_state.c

```

/*
 * Because there is no /usr/lib/libzonecfg.so symlink to
 * /usr/lib/libzonecfg.so.1 we create one for ourselves
 * before trying to link our program:
 *
 * # ln -s /usr/lib/libzonecfg.so.1 ./libzonecfg.so
 * # /usr/sfw/bin/gcc zone_set_state.c -L. -lzonecfg -o zone_set_state
 * # rm libzonecfg.so
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <dlfcn.h>
#include <link.h>

/* Defines are from libzonecfg.h */
#define ZONE_STATE_CONFIGURED 0

```

```

#define ZONE_STATE_INCOMPLETE 1
#define ZONE_STATE_INSTALLED 2
#define Z_OK 0
typedef uint_t zone_state_t;

/*
 * There is no include file for the libzonecfg functions
 */
extern int zone_set_state(char *, zone_state_t);

void
usage(char *exec)
{
    fprintf(stderr,
            "Usage: %s <zone name> <configured | incomplete | installed>\n",
            exec);

    exit(255);
}

int
main(int argc, char *argv[])
{
    int err;
    zone_state_t state;

    if (argc != 3) {
        usage(argv[0]);
    }

    if (strcasecmp(argv[2], "installed") == 0) {
        state = ZONE_STATE_INSTALLED;
    } else if (strcasecmp(argv[2], "incomplete") == 0) {
        state = ZONE_STATE_INCOMPLETE;
    } else if (strcasecmp(argv[2], "configured") == 0) {
        state = ZONE_STATE_CONFIGURED;
    } else {
        usage(argv[0]);
    }

    err = zone_set_state(argv[1], state);
    if (err != Z_OK) {
        fprintf(stderr, "Could not set state");
        return 255;
    }

    return 0;
}

```

## B.3 prog41.c

```

#include <sys/fcntl.h>

#include <fcntl.h>
#include <unistd.h>

int
main()
{
    int fd;
    char output[] = "Hello zone!\n";
    char file[] = "/output.txt";

    fd = open(file, O_CREAT | O_WRONLY, 0600);
    if (fd < 0) {
        return -1;
    }
}

```

```

    write(fd, output, sizeof(output) - 1);
    close(fd);

    while(1) {
        sleep(10);
    }

    return 0;
}

```

## B.4 prog42.c

```

#include <sys/wait.h>

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct config {
    int    narg;
    char **args;
    char  *buffer;
};

/*
 * Read at most buflen bytes from file
 */
int
read_config(char *file, char *buf, int buflen)
{
    int nread;
    int fd;

    fd = open(file, O_RDONLY, 0);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    for (;;) {
        nread = read(fd, buf, buflen - 1);
        if (nread < 0) {
            if (errno == EINTR) {
                continue;
            }

            perror("open");
            return -1;
        }

        buf[nread] = '\0';
        break;
    }

    return nread;
}

#define STATE_CHAR 0
#define STATE_WS  1
#define IS_WS(x) (x == ' ' || x == '\t' || x == '\n' || x == '\r')

/*
 * Tokenize arguments from a string
 * Utilizes a very simple state machine to extract
 * the tokens separated by one or more white space
 * characters

```

```

*/
int
build_args(char *file, struct config *conf)
{
    int i;
    int len;
    int argsi;
    int narg;
    int state;
    char line[256];

    if ((len = read_config(file, line, sizeof(line))) < 0) {
        return -1;
    }

    state = STATE_WS;
    narg = 0;
    for (i = 0; i < len; i++) {
        switch (state) {
            case STATE_CHAR:
                if (IS_WS(line[i])) {
                    state = STATE_WS;
                }
                break;
            case STATE_WS:
                if (!IS_WS(line[i])) {
                    narg++;
                    state = STATE_CHAR;
                }
                break;
            default:
                return -1;
        }
    }

    conf->narg = narg;
    /*
     * Since execv() takes a NULL terminated (char *)-vector as argument
     * we allocate memory for one extra (char *)-entry to fit the terminating
     * NULL
     */
    conf->args = malloc(sizeof(char *) * (narg + 1));
    conf->buffer = malloc(sizeof(char) * len);

    if (conf->args == NULL || conf->buffer == NULL) {
        return -1;
    }

    memcpy(conf->buffer, line, len);
    conf->buffer[len] = '\0';

    state = STATE_WS;
    for (i = 0, argsi = 0; i < len; i++) {
        switch (state) {
            case STATE_CHAR:
                if (IS_WS(conf->buffer[i])) {
                    conf->buffer[i] = '\0';
                    state = STATE_WS;
                }
                break;
            case STATE_WS:
                if (!IS_WS(conf->buffer[i])) {
                    conf->args[argsi++] = &conf->buffer[i];
                    state = STATE_CHAR;
                }
                break;
            default:
                return -1;
        }
    }

    conf->args[argsi] = NULL;

```

```

    return 0;
}

int
main(int __attribute__((unused)) argc, char *argv[])
{
    char *binary;
    struct config conf;

    if (build_args("/prog42.cfg", &conf) || !conf.narg) {
        printf("build_args() failed\n");
        return 1;
    }

    binary = strchr(argv[0], '/');
    if (binary == NULL) {
        binary = argv[0];
    } else {
        binary++;
    }

    /*
     * If the binary is called "prog42_fork" we fork-exec() and start
wait()ing
     * in the parent process
     */
    if (strcmp(binary, "prog42_fork") == 0) {
        pid_t pid;

        pid = fork();
        if (pid < 0) {
            perror("fork");
            /*
             * Sleep in case the fork failed to prevent a process creation
             * loop from consuming all the system resources
             */
            sleep(1);
            return 2;
        } else if (pid) { /* Parent */
            int dummy;

            while (1) {
                /*
                 * Since we're acting as init(1) all orphaned processes
                 * will report back to us. Call wait() to get rid of
                 * these zombie processes.
                 */

                wait(&dummy);
                sleep(1);
            }

            /* Child */
        }

        execv(conf.args[0], conf.args);

        /*
         * Only reached if the call to execv() failes
         */
        perror("execv");

        return 0;
    }
}

```

## B.5 create\_packages.pl

```
#!/usr/bin/perl

# -- Configuration --

$sadm_pkg_root = "/var/sadm/pkg";
$pkginfo_bin = "/usr/bin/pkginfo";

# -- Configuration end --

# -- Variables --

my %pkg_hash;
my @pkg_array;
my $pkg_index;

my %global_pkg_hash;

# -- Variables end --

if ($#ARGV < 0) {
    print "Usage: $0 <package> [<package> ... <package>]\n";
    exit 1;
}

init_global_packages();

for ($i = 0; $i <= $#ARGV; $i++) {
    pkg_dep_calc($ARGV[$i]);
    add_pkg($ARGV[$i]);
}

foreach $key (keys %pkg_hash) {
    delete $global_pkg_hash{$key};
}

foreach $key (keys %pkg_hash) {
    print "$key\n";
}

sub fatal {
    local $msg = shift;

    print STDERR "$msg\n";

    exit 255;
}

sub add_pkg {
    local $pkg = shift;

    if (!defined($pkg_hash{$pkg})) {
        $pkg_hash{$pkg} = $pkg_index;
        $pkg_array[$pkg_index++] = $pkg;
        return 1;
    }

    return ;
}

sub pkg_dep_calc {
    local $pkg = shift;
    local $pkg_depend = "$sadm_pkg_root/$pkg/install/depend";
    local @deps;
    local $i;
    local $j;
```

```

if (! -d "$sadm_pkg_root/$pkg") {
    fatal("Package $pkg does not exist, add package and retry");
}

if (! -f "$pkg_depend") {
    # Assume we got no dependencies for this package, not that the file is
    # is missing
    return 0;
}

open DEPEND, "$pkg_depend" or fatal("Open of $pkg_depend failed: $!");

$i = 0;
while (<DEPEND>) {
    if (/^P\s+(\S+)\s+/) {
        $deps[$i++] = $1;
    }
}
close DEPEND;

for ($j = 0; $j < $i; $j++) {
    if (add_pkg($deps[$j])) {
        pkg_dep_calc($deps[$j]);
    }
}
}

sub init_global_packages {
    open PKGINFO_PIPE, "$pkginfo_bin|" or fatal("Spawning of $pkginfo_bin
failed: $!");

    while (<PKGINFO_PIPE>) {
        if (/^\S+\s+(\S+)\s/) {
            $global_pkg_hash{$1} = 1;
        }
    }
}

```

## B.6 calc\_bin\_deps.pl

```

#!/usr/bin/perl

# -- Configuration --

$lld_bin = "/usr/bin/lld";

# -- Configuration end --

# -- Variables --

my %file_hash;
my @file_array;
my $file_index;

# -- Variables end --

if ($#ARGV < 1) {
    print "Usage: $0 </zone/path/root> <package> [<package> ... <package>]\n";
    exit 1;
}

for ($i = 0; $i <= $#ARGV; $i++) {
    add_file($ARGV[$i]);
    file_dep_calc($ARGV[$i]);
}

foreach $key (keys %file_hash) {
    print $key . "\n";
}

sub fatal {

```

```

    local $msg = shift;

    print STDERR "$msg\n";

    exit 255;
}

sub add_file {
    my $file = shift;
    my $link = shift;

    if (!defined($file_hash{$file})) {
        $file_hash{$file} = $link;
        $file_array[$file_index++] = $file;
        return 1;
    }

    return ;
}

sub file_is_added {
    local $file = shift;

    return defined($file_hash{$file});
}

sub file_dep_calc {
    local $file = shift;
    local $i;
    local $j;
    local %deps = ();
    local $key;

    #
    # Since symlinks doesn't always have absolute paths
    # we calculate the absolute path by taking the directory
    # of the symlink and adds it to the file pointed to
    # before adding it to the dependency list
    #

    if (-l $file) {
        local $symlink = readlink $file;

        if ($symlink =~ m#[^/]*#) {
            if ($file =~ m#(.*[/])[^/]+$#) {
                $symlink = $1 . $symlink;
            } else {
                fatal("$file is not a absolute path");
            }
        }

        if (!file_is_added($file)) {
            $deps{$file} = $symlink;
        } else {
            $file_hash{$file} = $symlink;
        } else {
            fatal("$file is not a absolute path");
        }
    }

    if (!file_is_added($file)) {
        $deps{$file} = $symlink;
    } else {
        $file_hash{$file} = $symlink;
    }

    if (!file_is_added($symlink)) {

```

```

        $deps{$symlink} = 1;
    }
} else {
    open LDD_PIPE, "$ldd_bin $file|" or fatal("Failed to spawn ldd\n");

    $i = 0;
    while (<LDD_PIPE>) {
        #
        # Extract the argument following '=>' from the ldd output, i.e.,
the
        # absolute path to the library
        #
        if (/=>\s+([\S]+.)$/) {
            $lib = $1;

            #
            # We demand that libraries begin with the '/' character
            # All other strings are discarded
            #
            if ($lib =~ m#^/#) {
                if (!file_is_added($lib)) {
                    $deps{$lib} = 1; # This will eat duplicates
                }
            }
        }
    }

    close LDD_PIPE;

    foreach $key (keys %deps) {
        add_file($key, $deps{$key});
        file_dep_calc($key);
    }
}

```

## B.7 install\_with\_lucreatezone.pl

```

#!/usr/bin/perl

use File::Temp ':mktemp';

$lucreatezone = "/usr/lib/lu/lucreatezone";

if ($#ARGV == 0) {
    read_manifest_file($ARGV[0]);
} else {
    if ($#ARGV < 5) {
        print "Usage: $0 <zone name> <iface> <IP>[/<netmask>] <svc_profile>
<postprocess> <package> [<package> ... <package>]\n";
        print "Usage: $0 <parameter file>\n";
        exit 1;
    }

    $name          = $ARGV[0];
    $iface         = $ARGV[1];
    $ipnm         = $ARGV[2];
    $svc_profile  = $ARGV[3];
    $postprocess  = $ARGV[4];
}

if ($ipnm =~ /(\d+\.\d+\.\d+\.\d+)/(\d+)/) {
    $ip = $1;
} else {

```

```

    $ip = $ipnm;
}

for ($i = 5; $i <= $#ARGV; $i++) {
    $packages .= "$ARGV[$i] ";
}

$tempdir = mkdtemp("/tmp/exjobb_XXXXX") or die "mkdtemp: $!\n";
print "Work dir: $tempdir\n";

open ZONECMD, ">$tempdir/zone.cmd" or die "open: $!\n";

print ZONECMD <<END
create -b

    set zonepath=/zone/$name

    add net
        set address=$ipnm
        set physical=$iface
    end

    add inherit-pkg-dir
        set dir=/usr/perl5
    end

commit
END
;

close ZONECMD;

open SVCCMD, ">$tempdir/svc.cmd" or die "open: $!\n";

print SVCCMD "repository /zone/$name/root/etc/svc/repository.db\n";
@svcps = split(/\s+/, $svc_profiles);
foreach $svc (@svcps) {
    print SVCCMD "import $svc\n";
}
;

close SVCCMD;

print("----- Creating zone -----\n");
system("/usr/sbin/zonecfg -z $name -f $tempdir/zone.cmd");

print("----- Building package list -----\n");
system("/usr/bin/perl /opt/thesis/create_packages.pl $packages >
$tempdir/exclude.pkg");

print("----- Installing zone -----\n");
system("/opt/thesis/zone_set_state $name incomplete");
system("$lucreatezone -z $name -P $tempdir/exclude.pkg");
system("chmod 0700 /zone/$name");
system("/opt/thesis/zone_set_state $name installed");

if ($svc_profiles) {
    print("----- Importing SVC profiles -----\n");
    system("/usr/bin/rm /zone/$name/root/etc/svc/repository.db");
    system("/usr/sbin/svccfg -f $tempdir/svc.cmd");
} else {
    print("----- Skipping SVC profiles -----\n");
}

if ($postprocess) {
    print("----- Post processing -----\n");
    system("$postprocess $ip $name");
} else {
    print("----- Skipping post processing -----\n");
}

```

```

system("/usr/bin/rm -rf $tempdir");

sub read_manifest_file {
    local $file = shift;
    local %entries;

    open PARAMS, $file or die "open: $!\n";

    while (<PARAMS>) {
        if (/^\s*(\S+)=\s*(.+)/) {
            $entries{$1} = $2;
        }
    }

    # Mandatory parameters
    foreach $entry ('zonename',
                   'interface',
                   'ip',
                   'packages') {

        if (!$entries{$entry} || $entries{$entry} =~ /\s*/) {
            print "Missing $entry\n";
            exit 2;
        }
    }

    # Optional parameters:
    # 'svc_profiles'
    # 'postprocess'

    $name          = $entries{'zonename'};
    $iface         = $entries{'interface'};
    $ipnm          = $entries{'ip'};
    $svc_profiles  = $entries{'svc_profiles'};
    $postprocess   = $entries{'postprocess'};
    $packages      = $entries{'packages'};
}

```

## B.8 apache\_manifest.pkg

```

zonename=      apache
interface=     bge0
ip=            192.168.0.1/24
svc_profiles=  http-apache2-jailed.xml
postprocess=   perl apache2_pp.pl
packages=      SUNWapch2d SUNWapch2r SUNWapch2u SUNWcslr SUNWopenssl-libraries
SUNWlibms     SUNWgss

```

## B.9 install\_by\_copying.sh

```

#!/bin/sh
zonecfg -z apache '
create -b;
  set zonepath=/zone/apache;
  add net;
    set address=192.168.0.1/24;
    set physical=bge0;
  end;
commit'

/opt/thesis/zone_set_state apache incomplete

for f in `perl /opt/thesis/calc_bin_deps.pl /usr/apache2/bin/httpd
/usr/apache2/libexec/*.so`; do

```

```
mkdir -p `dirname /zone/apache/root/$f`;
cp -Ppr $f /zone/apache/root/$f;
done

cp -Ppr /lib/ld.so.1 /zone/apache/root/lib/

/opt/thesis/zone_set_state apache installed
```



## Appendix C – Linux Resources

### C.1 client.c

```
/* $Id$
 *
 * client.c - Handles incoming clients.
 *
 * Staffan Palmroos, March 2007
 *
 * Starts with asking for a command to execute. It then sets up a
 * pseudo-terminal which is used to communicate with the child
 * process that is executing the command.
 */

#define _XOPEN_SOURCE

#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#include "exploit.h"

/* Max number of arguments the user is allowed to send */
#define MAX_ARGS 20

/** Handles connected clients
 *
 * Reads a command from the connected socket and then executes it.
 */
void
handle_client(int clientfd) {

    char buf[1024];
    int master_fd;
    int childpid;
    char *slavename;
    char *prompt = "Enter command > ";
```

```

/* grantpt result is undefined if sigchld is caught. */
signal(SIGCHLD, SIG_DFL);

if (-1 == write(clientfd, prompt, strlen(prompt)))
    log_exit("write(prompt)");

if (-1 == read(clientfd, buf, sizeof(buf)))
    log_exit("read(command)");

/** open pty master side **/

if (-1 == (master_fd = open("/dev/ptmx", O_RDWR|O_NOCTTY)))
    log_exit("open /dev/ptmx");

if (-1 == grantpt(master_fd))
    log_exit("grantpt");

if (-1 == unlockpt(master_fd))
    log_exit("unlockpt");

if (NULL == (slavename = ptsname(master_fd)))
    log_exit("ptsname");

log_msg("slave name is: %s", slavename);

childpid = fork();

if (childpid == -1) { // error
    log_exit("fork");
} else if (childpid) { // parent

    fd_set set;
    FD_ZERO(&set);

    while(1) { // move data fd <-> tty
        int size;

        FD_SET(clientfd, &set);
        FD_SET(master_fd, &set);

        if (-1 == select(10, &set, NULL, NULL, NULL)) {
            if (errno == EINTR)
                continue;
            log_exit("select");
        }

        if (FD_ISSET(clientfd, &set)) {
            size = read(clientfd, buf, sizeof(buf));
            if (size == -1) {
                if (errno == EAGAIN)
                    continue;
                break;
            }
        }
        if (size == 0) {
            kill(childpid, SIGHUP);
            break;
        }
        log_msg("<- Read %d bytes from client: '%.*s'", size, size, buf);
        write(master_fd, buf, size);
    } else {
        size = read(master_fd, buf, sizeof(buf));
        if (size == -1) {
            if (errno == EAGAIN)
                continue;
            break;
        }
    }
    if (size == 0) break;
    log_msg("-> Read %d bytes from child: '%.*s'", size, size, buf);
    write(clientfd, buf, size);
}

```

```

    }
    wait(NULL);
    log_msg("Child %d exited", childpid);
} else { // child

    char *argv[MAX_ARGS+1];
    int argp;

    buf[strlen(buf)-2] = 0;

    log_msg("Executing '%s'", buf);

    argv[0] = strtok(buf, " ");

    for (argp = 1; argp < MAX_ARGS; argp++) {
        argv[argp] = strtok(NULL, " ");
        if (argv[argp] == NULL) break;
    }
    argv[argp] = NULL; // for safety, if nargs > 20

    close(master_fd);
    close(clientfd);
    if (-1 == setsid())
        log_exit("setsid");

    if (-1 == (clientfd = open(slavename, O_RDWR))) // open slave tty
        log_exit("open slave tty");

    if (clientfd != 0) {
        if (-1 == dup2(clientfd, 0))
            log_exit("dup2(0)");

        close(clientfd);
    }

    if (-1 == dup2(clientfd, 1)) // spread to stdout
        log_exit("dup2(1)");

    if (-1 == dup2(clientfd, 2)) // spread to stderr
        log_exit("dup2(2)");

    if (-1 == execvp(argv[0], argv)) // exec command
        log_exit("execvp");
}

exit(EXIT_SUCCESS);
}

```

## C.2 exploit.c

```

/* $Id$
 *
 * Entry point to exploit.
 *
 * Staffan Palmroos, March 2007
 *
 * The this program is to simulate a severely broken network daemon. The
 * purpose of this is to examine how SELinux can be used to limit the
 * possible damage such daemons can do.
 *
 * The application tries to bind to a network port in the range 9990-9999 and
 * then listens for incoming connections. When a client connects the daemon
 * asks for a command to execute and then forks off a child process that
 * executes this command. Input and output of this command is sent to the
 * client. Basically, it works like the regular telnet daemon, with the
 * exception that a telnet daemon always executes the /bin/login program,
 * while this program execs whatever the user sent.
 */

```

```

#define _XOPEN_SOURCE

#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <unistd.h>

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

#include "exploit.h"

/** Signal handler taking care of zombies.
 */
static void
zombiekiller(int dummy) {

    dummy = errno;

    while(waitpid(-1, 0, WNOHANG) > 0);

    errno = dummy;
} /* zombiekiller */

/** Program entry point.
 */
int
main(void) {

    struct sockaddr_in sa;
    struct sigaction sigact;

    int on = 1;
    int client, sock;
    int i;

    init_log();

    log_msg("Exploit 1.0 starting");

    bzero(&sigact, sizeof(sigact));
    sigact.sa_handler = zombiekiller;

    if (-1 == sigaction(SIGCHLD, &sigact, 0))
        log_exit("sigaction");

    /** Daemonize application **/

    if (fork()) exit(0);
    close(0);
    close(1);
    close(2);
    setsid();
    if (fork()) exit(0);

    /** Set up a server socket **/

    if (-1 == (sock = socket(PF_INET, SOCK_STREAM, 0)))
        log_exit("socket");

    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;

```

```

sa.sin_addr.s_addr = INADDR_ANY;

for (i = 9990; i < 9999; i++) {
    sa.sin_port = htons(i);

    if (-1 == bind(sock, (struct sockaddr *)&sa, sizeof(sa)))
        continue;

    log_msg("Got port: %d", i);
    break;
}

if (i == 9999)
    log_exit("bind");

if (-1 == setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)))
    log_exit("setsockopt");

if (-1 == listen(sock, 1))
    log_exit("listen");

/** Wait for clients **/

while(1) {
    struct sockaddr_in raddr;
    socklen_t raddr_len = sizeof(raddr);

    client = accept(sock, (struct sockaddr *) &raddr, &raddr_len);

    if (client == -1) {
        if (errno == EINTR) // interrupted because of signal, just continue
            continue;

        log_exit("accept");
    }

    log_msg("Got client from %s:%d", inet_ntoa(raddr.sin_addr),
raddr.sin_port);

    if (fork() == 0) {
        close(sock);
        handle_client(client);
    }

    close(client);
}

return 0;
}

```

### C.3 exploit.h

```

/* $Id$
*
* exploit.h - Common log file for exploit daemon
*
* Staffan Palmroos, March 2007
*
*/

/* functions in client.c */

void handle_client(int clientfd);

/* functions in log.c */

```

```

void init_log(void);
void log_msg(char const * msg, ...);
void log_exit(char const * func);

```

## C.4 log.c

```

/* $Id$
 *
 * log.c - Handles logging
 *
 * Staffan Palmroos, March 2007
 *
 * Writes log messages to a log file in /var/log
 */

#define _GNU_SOURCE

#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>

#include "exploit.h"

#define LOG_FILE "/var/log/exploit.log"

/** Descriptor to log file. If == -1 the log file is not opened. */
static int log_fd = -1;

/** Try to open the log file.
 *
 * The SELinux policy requires the log file to be tagged
 * exploit_log_t. This is not visible here, though.
 */
void
init_log(void) {
    if (log_fd > -1) // log file already open
        return;

    log_fd = open(LOG_FILE, O_CREAT|O_APPEND|O_WRONLY, 00640);
    if (log_fd == -1)
        exit(EXIT_FAILURE);

    if (log_fd < 3) { // log descriptor must be >= 3
        int newfd;

        newfd = fcntl(log_fd, F_DUPFD, 3);
        if (newfd == -1)
            exit(EXIT_FAILURE);

        close(log_fd);
        log_fd = newfd;
    }

    // We do not want to give the log file descriptor to our child, so

```

```

    // we close it on exec()
    if (-1 == fcntl(log_fd, F_SETFD, FD_CLOEXEC))
        exit(EXIT_FAILURE);
}

/** Write a timestamp to the log file.
 * @return -1 on error
 */
static int
timestamp() {
    char buf[32];
    int len;

    time_t mytime = time(NULL);
    struct tm *mytm = localtime(&mytime);

    len = strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S ", (struct tm const *)mytm);

    if (len < 1)
        return -1;

    if (-1 == write(log_fd, buf, strlen(buf)))
        return -1;

    return 0;
}

/** Output a message to the log file.
 * @param msg An error message in regular printf() style.
 */
void
log_msg(char const *msg, ...) {
    if (log_fd > -1) {
        char *msgbuf;
        int len;
        va_list ap;

        if (-1 == timestamp())
            exit(EXIT_FAILURE);

        va_start(ap, msg);
        len = vasprintf(&msgbuf, msg, ap);
        if (len == -1) {
            return;
        }
        write(log_fd, msgbuf, len);
        free(msgbuf);
        va_end(ap);

        write(log_fd, (void const *) "\n", 1);
    }
}

/** Output an error to the log file and then exit.
 * @param func The function where the error occurred.
 */
void
log_exit(char const *func) {
    if ((log_fd > -1) && (func != 0)) {
        char const *errstr = strerror(errno);

        if (-1 == timestamp())

```

```

        exit(EXIT_FAILURE);

        write(log_fd, (void const *) func, strlen(func));
        write(log_fd, (void const *) ":", 1);
        write(log_fd, errstr, strlen(errstr));
        write(log_fd, (void const *) "\n", 1);
    }

    exit(EXIT_FAILURE);
}

```

## C.5 exploit.te

```

policy_module(exploit,1.0.114)

# Exploit shall run in the exploit_t domain. Make exploit_t an actual domain
#
type exploit_t;
type exploit_exec_t;
init_daemon_domain(exploit_t, exploit_exec_t)

#
# We log to exploit_log_t files
#
type exploit_log_t;
logging_log_file(exploit_log_t)
allow exploit_t exploit_log_t:dir rw_dir_perms;
allow exploit_t exploit_log_t:file create_file_perms;
logging_log_filetrans(exploit_t,exploit_log_t,{ file dir })

#
# The application will be allowed to bind sockets to exploit_port_t ports
#
type exploit_port_t;

#
# Create a socket and bind it to inaddr_any, port exploit_port_t
#
# Remember to tag appropriate ports:
# semanage port -a -t exploit_port_t -p tcp 9990-9999
#
allow exploit_t self:tcp_socket create_stream_socket_perms;
allow exploit_t exploit_port_t:tcp_socket name_bind;
corenet_tcp_bind_inaddr_any_node(exploit_t)
corenet_tcp_sendrecv_all_if(exploit_t)
corenet_tcp_sendrecv_all_nodes(exploit_t)
corenet_tcp_sendrecv_all_ports(exploit_t)

#
# Allows unlabeled packets.
# non-IPSEC packets are not labeled in FC5
#
corenet_non_ipsec_sendrecv(exploit_t)

#
# Our pseudo terminal device is labeled exploit_devpts_t
#
type exploit_devpts_t;
term_pty(exploit_devpts_t)
term_create_pty(exploit_t,exploit_devpts_t)
allow exploit_t exploit_devpts_t:chr_file { rw_file_perms setattr };

#
# Needed for the localtime() call in the log functions
#
miscfiles_read_localization(exploit_t)

#

```

```

# Needed for applications built with shared libs
#
libs_use_ld_so(exploit_t)
libs_use_shared_libs(exploit_t)

#
# Here is the place to allow or disallow exploit what exploit can do
#
corecmd_search_bin(exploit_t) # Can run programs from bin directories
corecmd_exec_ls(exploit_t)    # Allow running ls
corecmd_exec_bin(exploit_t)   # Allow running bin_t apps, such as /bin/echo
corecmd_exec_shell(exploit_t) # Allow exec shell

```

## C.6 exploit.if

```

## <summary>Exploit policy</summary>
## <desc>
##     <p>
##         A simulation of a flawed daemon.
##     </p>
## </desc>

#####
## <summary>
##     Execute a domain transition to run exploit.
## </summary>
## <param name="domain">
##     <summary>
##         Domain allowed to transition.
##     </summary>
## </param>

interface(`exploit_domtrans`,`
    gen_requires(`
        type exploit_t, exploit_exec_t;
    `)

    domain_auto_trans($1,exploit_exec_t,exploit_t)

    allow $1 exploit_t:fd use;
    allow exploit_t $1:fd use;
    allow exploit_t $1:fifo_file rw_file_perms;
    allow exploit_t $1:process sigchld;
)

#####
## <summary>
##     Read exploit log files.
## </summary>
## <param name="domain">
##     <summary>
##         Domain allowed to read the log files.
##     </summary>
## </param>

interface(`exploit_read_log`,`
    gen_requires(`
        type exploit_log_t;
    `)

    logging_search_logs($1)
    allow $1 exploit_log_t:file r_file_perms;
)

```

## C.7 exploit.fc

```
/bin/exploit      gen_context(system_u:object_r:exploit_exec_t,s0)
/var/log/exploit\.log  gen_context(system_u:object_r:exploit_log_t,s0)
/root/exploit      gen_context(system_u:object_r:exploit_exec_t,s0)
/root/exploit\.log  gen_context(system_u:object_r:exploit_log_t,s0)
```