Trusted Linux : A Secure Platform for Hosting Compartmented Applications

Author: Tse Huong Choo

Hewlett Packard Laboratories
Filton Road, Stoke Gifford,
Bristol BS34 8QZ,
United Kingdom

Telephone Number: +44-117-3128966
Fax Number: +44-117-3128985
Email: tchoo@hpl.hp.com

Introduction

Our research work over the last few years has shown that Trusted Operating Systems provide specific properties that we believe are critical for building secure Internet gateways and Application service platforms. One of the most significant properties they offer over normal operating systems is *containment*. Containment allows us the assurance that even if an application or service is broken (non-maliciously or by an attacker) the amount of damage that can be done via that compromised application or service is tightly controlled and possibly negligible.

This document describes the architecture and design of a Trusted Operating System based on Linux that provides the containment property. We believe it is significantly more usable, maintainable and application friendly than traditional Trusted Operating Systems.

We first describe in more detail the concept of containment and why it is useful. We then briefly describe the traditional approach used by Trusted Operating Systems to achieve containment, and some of the major problems associated with that approach. Following that we outline our alternative, more lightweight, approach to achieving containment. We describe in detail our overall architecture and design, our user level support requirements, our kernel extensions and the general kernel modifications we have made. We then show our approach to hosting services on the platform, application integration and administration of the enhanced functionality. Finally, we discuss some outstanding issues and some of our future research aims.

Containment

An application (or service) is contained if it has strict controls placed on which resources it can access (and what type of access to those resources it has, e.g. read-only) even if the application has been compromised. Containment also implies that an application that is contained should be protected from external attack and interference. Containment applies to file system, process and network resources.

Containment is particularly useful on systems used as network gateways and application hosting platforms between the Internet and an internal network. For example, it is possible to guarantee that if a service on the gateway machine is attacked and compromised from the Internet, the attacker can’t then go on and gain access to whole the internal network. Similarly, on a platform hosting multiple applications, we can be guaranteed that one process can not interfere with the running of another process or get access to it’s perhaps sensitive data.

Traditional Trusted OS approach to containment

Typically, Trusted Operating Systems achieve containment through a combination of Mandatory Access Controls (MAC), and Privileges. MAC protection schemes enforce a particular policy of access control to the system resources such as files, processes and network connections. This policy is enforced by the kernel and cannot be overridden by a user or compromised application. Most current trusted operating systems use the Bell-LaPadula policy model. This is a formal model developed on behalf of military organizations in which the flow of information around the system is predictable

Whilst appropriate for the military domain, we have found Bell-LaPadula MAC controls too restrictive when trying to deploy Internet applications on trusted operating systems. However, privileges can be given to applications so they can override some of the mandatory access controls on the system. This is typically done so that applications can communicate with other applications on the machine (and over the network) or get access to selected files, which the MAC policy prevents. However, the allocation of privileges has to be carefully managed so as not to render the MAC controls ineffective.

For application to application communication, an alternative approach is to use privileged user space relays between applications. Whilst potentially more secure than granting the applications themselves privileges, this often requires the applications to be modified (and the user space relays have to be provided).

Overall, whilst trusted operating systems based on the Bell-LaPadula model do usefully provide containment, they provide too much. This leads to often severe application integration issues and a significant management overhead. The end result is that whilst these systems are considerably more secure than ordinary operating systems they have not used extensively in practice outside of the military domain.

Our approach to containment

Similar to the traditional trusted operating system approach we achieve the property of containment by kernel level mandatory protection of processes, files and network resources. However, our mandatory controls are somewhat different to those found on traditional trusted operating systems. We have attempted to reduce some of the application integration and management problems associated with traditional trusted operating systems.

The key concept of our trusted operating system is the *compartment*. Services and applications on the machine are run within separate compartments.

We use simple mandatory access controls and process labeling to create the concept of a compartment. Each process is given a label; processes with the same label belong to the same compartment. We enforce kernel level mandatory checks to ensure that processes from one compartment cannot interfere with processes from another compartment. The access controls are very simple, labels either match or they don't. There is no hierachial ordering of labels within the system such as there is in the Bell-LaPadula model.

Filesystem protection is also mandatory. Unlike traditional Trusted operating system, we do not use labels to directly control access to the filesystem; each compartment has a section of file system associated with it. This section of file system is a chroot of the main filesystem. Processes running within a particular compartment only have access to that section of the filesystem. Importantly, via kernel controls, we remove the ability of a process to transition to root from within a compartment so that the chroot cannot be escaped. We also have the ability to make selected files within the chroot immutable.

Flexible communication paths between compartments and network resources are provided via narrow, kernel level controlled interfaces to TCP/UDP plus most IPC mechanisms. Access to these communication interfaces is governed by rules specified by the security administrator on a per compartment basis. Unlike traditional trusted operating systems we don't have to override our mandatory access controls with privilege or resort to the use of user level trusted proxies to allow communication between compartments / network resources.

These features together give us as a system that both offers containment but also has enough flexibility to make application integration relatively straightforward. This in turn reduces the management overhead and pain of deploying and running a trusted system. The rest of this document describes our design and architecture in detail.

Baseline Architecture

The Trusted Linux kernel is a layered extension of the standard Linux kernel (with user level support). It has the containment property that we believe is essential in guarding against application compromise amongst other things.

The Trusted Linux platform consists of modifications to the base Linux kernel to support containment of user-level services e.g. HTTP-servers. These changes can be broadly categorized as such:

1. Kernel modifications in the areas of:
   - TCP/IP networking
   - Routing-tables and routing-caches
   - System V IPC – Message queues, shared memory and semaphores
   - Processes and Threads
2. Kernel configuration interfaces in the form of:
   - Dynamically loadable kernel modules
   - Command-line utilities to communicate with these kernel modules
3. User-level scripts to administer/configure individual compartments:
   - Scripts to start/stop compartments

Figure 1 shows the architecture of our Trusted Linux host, including the major areas where these changes occur in the kernel and the addition of a series of compartments in user-space implementing Web-servers capable of executing CGI-binaries in configurable chroot-jails.

From the diagram it can be seen how we have modified some of the Linux kernel subsystems (TCP/IP, SysV IPC, etc.) to make call outs to our kernel level security module. The security module makes access control decisions and is responsible for enforcing our concept of a compartment. This is how we provide containment.

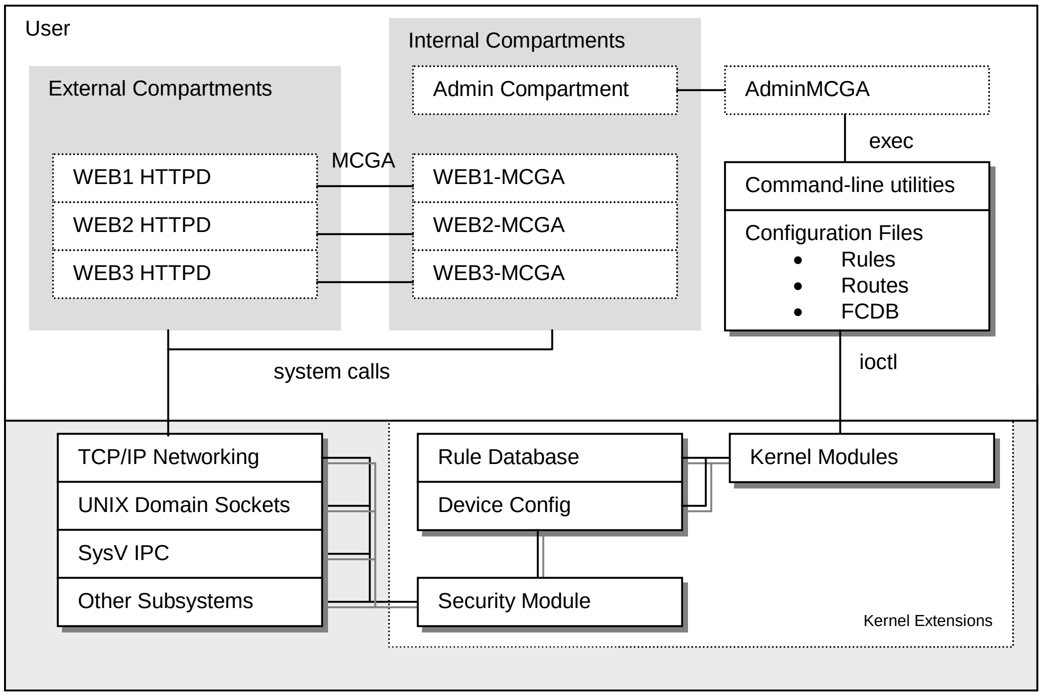

Figure 1. Baseline Architecture for a Trusted Linux host

The security module additionally consults a rule database when making a decision. The rule database contains information about allowable communication paths between compartments and is how we provide the narrow well controlled interfaces in and out of a compartment.

The diagram also shows how our kernel extensions (security module, rule database, etc.) are administered from user space via a series of ioctl commands. These ioctl take two forms; ones to manipulate the rule table and others to run processes in particular compartments and configure network interfaces.

User space services such as the web servers shown in the diagram are run unmodified on the platform but have a compartment label associated with them via the command line interface to our security extensions. The security module is then responsible for applying our mandatory access controls to them based on their applied compartment label. Importantly, this mechanism ensures that we can contain our user space services without having to modify those services.

The following section deals with major components of the architecture in detail. First the format of the rules used for implementing our containment mechanism. Secondly, the loadable modules that implement this functionality within the kernel and the command line utilities required to configure and administer aspects of our security extensions such as the communication rules and process compartment labels.

Access Control Rules

Each security check consults a table of rules. Each rule has the form:

```
source -> destination method m [attr]
                               [netdev n]
where:
source/destination is one of:
  COMPARTMENT (a named compartment)
  HOST        (a fixed IPv4 address)
  NETWORK     (an IPv4 subnet)
   m:         supported kernel mechanism
              e.g. tcp, udp,  msg (message queues),
              shm (shared-memory) etc.
attr:         attributes further qualifying the method m
   n:         a named network-interface if
              applicable eg eth0
```

An example of such a rule which allows processes in the compartment named “WEB” to access shared-memory segments (e.g. using *shmat/shmdt()*) from the compartment named “CGI” would look like:

```
COMPARTMENT:WEB -> COMPARTMENT:CGI
    METHOD shm
```

Present also are certain implicit rules, which allow some communications to take place within a compartment e.g. a process is allowed to see the process identifiers of processes residing in the same compartment. This is to allow a bare-minimum of functionality within an otherwise unconfigured compartment. An exception is compartment 0, which is relatively unprivileged and where there are more restrictions applied. Compartment 0 is typically used to host kernel-level threads (such as the swapper).

In the absence of a rule explicitly allowing a cross-compartment access to take place, all such attempts fail. The net effect of these rules is to enforce mandatory segmentation across individual compartments, except for those that have been explicitly allowed to access another compartment’s resources.

LNS Kernel Module

The current prototype uses a kernel module called LNS to implement custom *ioctl()s* that enable the insertion/deletion of rules and other functions e.g. labeling of network interfaces. The main client of this module is the *lcu* command-line utility described below.

1. a calling process to switch compartments
2. individual network interfaces to be assigned a compartment number
3. utility functions such as process listing with compartment numbers and the logging of activity to kernel-level security-checks.
4. add/delete rules including the translation between higher-level simplified rules into primitive forms more readily understood by kernel lookup-routines.

LCU Command-line Utility

The *lcu* command-line utility provides an interface to the LNS kernel-module. Its most important function is to provide various admininstration-scripts with the ability to spawn processes in a given compartment and to set the compartment number of network interfaces. Example of its usage are:

1. `lcu setdev eth0 0xFFFF0000`
   Sets the compartment number of the eth0 network interface to 0xFFFF0000
2. `lcu setprc 0x2 –cap_mknod bash`
   Switches to compartment 0x2, removes the cap_mknod capability and invokes bash.

General Kernel Modifications

Modifications were made to the kernel sources to introduce a tag on various datatypes and for the addition of access-control checks made around such tagged datatypes. Each tagged datatype contains an additional struct csecinfo data-member which is currently used to hold a compartment number (see below), but may be extended in future to hold other security attributes. In general, the addition of this member is typically done at the very end of a data-structure to avoid issues related to the common practice casting pointers between two or more differently named structures which begin with common entries.

```
struct csecinfo {
       unsigned long sl;
};
struct sock {
…
#ifdef TLINUX
    /* contains compartment number */
    struct csecinfo csi;
#endif /* TLINUX */
};
```

Figure 2. Example of modified datatype

The net effect of tagging individual kernel resources is to very simply implement a compartmented system where processes and the data they generate/consume are isolated from one another. This isolation is not meant to be strict as many covert channels exist (see the discussion about processes below) – the isolation exists to protect the obvious forms of conflict and/or interaction between logically different groups of processes.

There exists a single function cnet_chk_attr() that implements a yes/no security-check for the subsystems that are protected in the kernel. Calls to this function are made at the appropriate points in the kernel sources to implement the compartmented behaviour desired. This function is predicated on the subsystem concerned and may implement slightly different defaults or rule-conventions depending on the subsystem of the operation being queried at that time. For example, most subsystems implement a simple partitioning where only objects/resource having exactly the same compartment number result in a positive return value. However, in certain cases the use of a no-privilege compartment 0 and/or a wildcard compartment –1L can be used e.g. compartment 0 as a default 'sandbox' for unclassified resources/services; a wildcard compartment for supervisory purposes, like listing all processes on the system prior to shutting down.

Networking

To understand better the changes made to the TCP/IP stack, a short explanation of how networking works in Linux is given. Each process or thread is represented by a struct task_struct variable in the kernel. A process may create sockets in the AF_INET domain for network-communication over TCP/UDP. These are represented by a pair of struct socket and struct sock variables, also in the kernel.

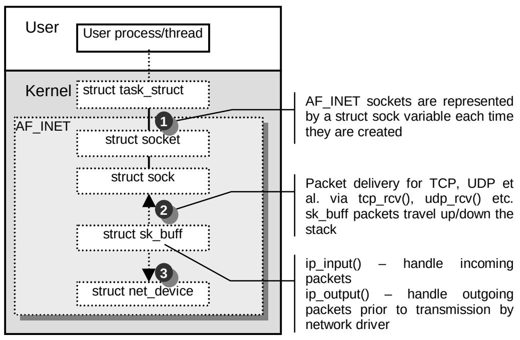
User
User process/thread
Kernel
struct task_struct
AF_INET
struct socket
struct sock
struct sk_buff
struct net_device
1
2
3
AF_INET sockets are represented by a struct sock variable each time they are created
Packet delivery for TCP, UDP et al. via tcp_rcv(), udp_rcv() etc. sk_buff packets travel up/down the stack
ip_input() – handle incoming packets
ip_output() – handle outgoing packets prior to transmission by network driver

Figure 3. Major networking datatypes in Linux IP networking

The struct sock datatype contains amongst other things queues for incoming packets represented by struct sk_buffs. It may also hold queues for pre-allocated sk_buffs for packet transmission. Each sk_buff represents an IP packet and/or fragment travelling up/down the IP-stack. They either originate at a struct sock (more specifically, from its internally pre-allocated send-queue) and travel downwards for transmission, or their originate from a network-driver and travel upwards from the bottom of the stack starting from a struct net_device which represents a network interface. When travelling downwards, they effectively terminate at a struct net_device. When travelling upwards, they are usually delivered to a waiting struct sock (actually, its pending queue).

Struct sock variables are created by indirectly by the socket()-call[1] and can usually be traced to an owning user-process i.e. a task_struct. There exist a struct net_device variable for each configured interface on the system, including the loopback interface. Localhost and loopback communications appear not to travel via a fastpath across the stack for speed, rather they travel up and down the stack as one would expect for remote host communications. At various points in the stack, calls are made to registered netfilter-modules for the purposes of packet interception.

By adding an additional csecinfo data-member to the most commonly used datatypes in Linux IP networking, it becomes possible to trace ownership and hence read/write-dataflows of individual IP packets for all running processes on the system, including internal kernel-generated responses.

Modified Networking Datatypes

Most of the data-structures modified are related to networking and occur in the networking stack and socket-support routines. The tagged network data structures serve to implement a partitioned IP stack. The following data structures were modified to include a struct csecinfo:

1. struct task_struct - processes (and threads)
2. struct socket - abstract socket representation
3. struct sock - domain-specific socket
4. struct sk_buff - IP packets or messages between sockets
5. struct net_device - network interfaces e.g. eth0,lo etc.

Once the major datatypes were tagged, the entire IP-stack was checked for points at which these datatypes were used to introduce newly initialised variables into the kernel. Having identified these, code was inserted to ensure that the inheritance of the csecinfo structure was carried out. The next section describes how the csecinfo structure is propagated throughout the IP networking stack.

[1] Well, almost. There are private per-protocol sockets owned by various parts of the stack within the kernel itself that cannot be traced to a running process. See the discussion later about this.

Propagation of struct csecinfo in IP networking

There are two named sources of struct csecinfo data members – per-process task_structs and per-interface net_devices. Each process inherits its csecinfo from its parent, unless explicitly modified by a privileged ioctl(). Currently, the init-process is assigned a compartment number of 0. Therefore, every process spawned by init during system startup will inherit this compartment number unless explicitly set otherwise. During system startup, init-scripts are typically called to explicitly set the compartment numbers for each defined network interface. The diagram below illustrates how csecinfo data-members are propagated for the most common cases.

All other data structures inherit their csecinfo structures from either a task_struct or a net_device. For example, if a process creates a socket, a struct socket and/or struct sock may be created which inherit the current csecinfo from the calling process. Subsequent packets generated by calling write() on a socket generate sk_buffs which inherit their csecinfo from the originating socket.

Incoming IP packets are stamped with the compartment number of the network interface on which it arrived, so sk_buffs travelling up the stack inherit their csecinfo structure from the originating net_device. Prior to being delivered to a socket, each sk_buff's csecinfo structure is checked against that of the prospective socket.

Example Propagation

Kernel functions that create/copy/clone the modified networking datatypes are modified to also handle the propagation of the struct csecinfo data-member. The code-fragment below shows how skb_clone() which clones an input struct sk_buff copies over the struct csecinfo member.

```
struct sk_buff * skb_clone( struct sk_buff * skb,
                            int gfp_mask )
{
        struct sk_buff *n;
        n = skb_head_from_pool();

        // other initialization steps
        // ...
        n->cloned = 1;
        n->next = n->prev = NULL;
        n->list = NULL;
        // ...

#ifdef TLINUX /* copy security info */
        n->csi = skb->csi;
#endif
        return n;
}
```

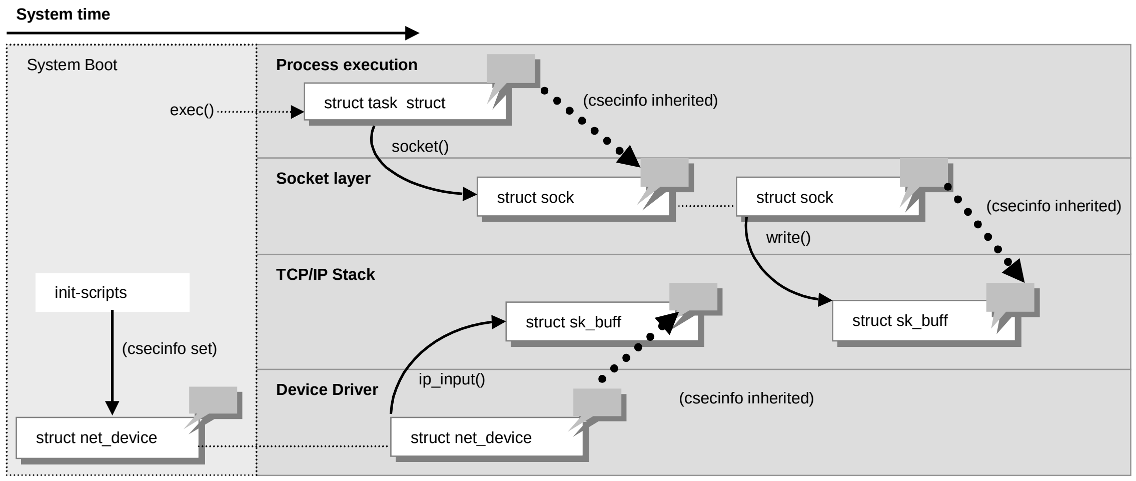

Figure 4. Propagation of struct csecinfo data-members for IP-networking

Handling Loopback communications

Special care has to be taken for non-remote networking. We need to handle the case when a connection is made between compartments X and Y through any number of the configured network interfaces and is allowed by a rule of this form:

```
COMPARTMENT X -> COMPARTMENT Y METHOD tcp
```

Because the security checks happen twice for IP-networking (once on output and once on input) we may naively end up looking for the existence of these rules instead:

```
COMPARTMENT X -> HOST a.b.c.d METHOD tcp
(for output)
HOST a.b.c.d -> COMPARTMENT X METHOD tcp
(for input)
```

which although are valid may not be used in preference to the rule specifying source and destination compartments directly. To cater for this, packets sent to the loopback device retain their original compartment numbers and are simply 'reflected' off it for eventual delivery. Note that the actual security check happens on delivery and not transmission. Upon receipt of an incoming local packet on the loopback interface, we avoid overwriting the compartment number of the packet with that of the network-interface and allow it to travel up the stack for the eventual check on delivery. Once there, we perform a check for a rule of the form:

```
COMPARTMENT X -> COMPARTMENT Y METHOD tcp
```
instead of
```
HOST a.b.c.d -> COMPARTMENT Y METHOD tcp
```

because of the presence on the sk_buff of a compartment number that is not of a form normally allocated to network interfaces.[2]

Dynamic Rules in TCP/IP

Because the rules are unidirectional, the TCP layer has to dynamically insert a rule to handle the reverse data flow once a TCP connection has been setup, either as a result of a connect() or accept(). This happens automatically in the current prototype and the rules are then deleted once the TCP connection is closed. Special handling occurs when a struct tcp_openreq is created to represent the state of a pending connection request, as opposed to one that has been fully set up in the form of a struct sock. A reference to the reverse-rule created is stored with the pending request and is also deleted if the connection request times out or fails for some other reason.

An example of this would be when a connection is made from compartment 2 to a remote host 10.1.1.1. The original rule allowing such an operation might have looked like this:

```
COMPARTMENT 2 -> NET
10.1.1.0/255.255.255.0 METHOD tcp
```

[2] Network interfaces as a general rule are allocated compartment numbers in the range 0xFFFF0000 and upwards and can therefore be distinguished from those allocated for running services.

As a result, the reverse rule would be something like this (abc/xyz being the specific port-numbers used):

```
HOST 10.1.1.1 PORT abc  -> COMPARTMENT 2
PORT xyz METHOD tcp
```

Per-Compartment Routing

To support per-compartment routing-tables, each routing-table entry is tagged with a csecinfo structure. The modified data-structures are:

1. struct rt_key
2. struct rtable
3. struct fib_rule
4. struct fib_node

Inserting a route using the route-command will cause a routing-table entry to be inserted with the csecinfo structure inherited from the calling context of the user-process i.e. if a user invokes the route-command from a shell in compartment N, the route added will be tagged with N as the compartment number. Attempts to view routing-table information (usually by inspecting /proc/net/route and /proc/net/rt_cache) are predicated on the value of the csecinfo structure of the calling user-process.

The major routines used to determine input and output routes a sk_buff should take are ip_route_output() and ip_route_input(). These have been expanded to include an extra argument consisting of a pointer to the csecinfo structure on which to base any routing-table lookup. This extra argument is supplied from either the sk_buff of the packet being routing for input or output.

Kernel-inserted routing-entries have a special status and are inserted with a wildcard compartment number (-1L). In the context of per-compartment routing, they allow these entries to be shared across all compartments. The main purpose of such a feature is to allow incoming packets to be routed properly up the stack. Any security-checks occur at a higher-level just prior to the sk_buff being delivered on a socket (or its sk_buff queue).

The net effect is that each compartment appears to have their individual routing tables which are empty by default. Every compartment shares the use of system-wide network-interfaces – there is currently no support for the concept of an IP-address per-compartment, although this can appear to be supported at a higher-level e.g. individual HTTP-servers bound to unique IP-addresses.

Subject to future revision, it is possible to restrict individual compartments to a strict subset of the available network-interfaces. This is because each network-interface is notionally in a compartment of its own (with its own routing table). In fact, to respond to an ICMP-echo request, each individual interface can optionally be configured with tagged routing-table entries to allow the per-protocol ICMP-socket to route its output packet.

Other Subsystems

- **UNIX Domain Sockets:** Each UNIX domain socket is also tagged with the csecinfo structure. As they also use sk_buffs to represent messages/data traveling between connected sockets, many of the mechanisms used by the AF_INET domain described above apply similarly. In addition, security-checks are also performed at every attempt to connect to a peer.

- **System V IPC:** Each IPC-mechanism listed above is implemented using a dedicated kernel structure that is similarly tagged with a csecinfo structure. Attempts to list, add, remove or add messages to these constructs are subject to the same security checks as individual sk_buffs. The security-checks are dependent on the exact type of mechanism used.

- **Processes/Threads:**
  Since individual processes i.e. task_structs are tagged with the csecinfo structure, most process-related operations will be predicated on the value of the process's compartment number. In particular, process listing (via the /proc interface) is controlled as such to achieve the effect of a per-compartment process-listing. Signal-delivery is somewhat more complicated as there remains issues around delivering signals to parent processes who may have switched compartments – thus constituting a 1-bit covert channel. Such cases have not been analysed in detail in the expectation that many other such covert channels (usually much wider) do exist – such as the consumption of CPU-time/disk-space as a signal in morse, for example.

System Defaults

Per-protocol Sockets

The Linux IP-stack uses special, private per-protocol sockets to implement various default networking behaviours such as ICMP-replies. These per-protocol sockets are not bound to any user-level socket and are typically initialised with a wildcard compartment number to enable ICMP et al. to behave normally.

Use of Compartment 0 as Unprivileged Default

The convention is to never insert rules that allow compartment 0 any access to other compartments and network-resources. This way, the default behaviour of initialised objects, or objects that have not been properly accounted for will fall under a sensible (and restrictive) default.

Default Kernel Threads

Various kernel threads may appear by default e.g. kswapd, kflushd and kupdate but to name a few. These threads are also assigned a csecinfo structure per-task_struct and their compartment numbers default to 0 to reflect their relatively unprivileged status.

Sealing Compartments Against Assumption of Root-identity

Individual compartments may optionally be registered as "sealed" to protect against processes in that compartment from successfully calling setuid(0) and friends, and also from executing any SUID-root binaries. This is typically used for externally-accessible services which may in general be vulnerable to buffer-overflow attacks leading to the execution of malicious code. If such services are constrained to initially run as a pseudo-user (non-root) and if the compartment it executes in is sealed, then any attempt to assume the root-identity either by buffer-overflow attacks and/or execution of foreign instructions will fail. Note that any existing processes running as root will continue to do so.

Layering User-level Services

The kernel modifications described previously serve to support the hosting of individual user-level services in a protected compartment. In addition to this, the layout, location and conventions used in adding or removing services in the current prototype are described here.

Individual services are generally allocated a compartment each. However, what an end-user perceives as a service may actually end up using several compartments. An example would be the use of a compartment to host an externally-accessible Web-server with a narrow interface to another compartment hosting a trusted gateway agent for the execution of CGI-binaries in their own individual compartments. In this case, one would need at least 3 compartments:

- One for the web-server processes
- One for the trusted gateway agent which executes CGI-binaries
- As many compartments as is needed to properly categorise each CGI-binary, as the trusted gateway agent will fork/exec CGI-binaries in their configured compartments

Directory Layout

Every compartment has a name and resides as a chroot-able environment under /compt. Examples used in the current prototype include:

Location	Description
`/compt/admin`	Admin HTTP-server
`/compt/omailout`	Externally visible HTTP-server hosting OpenMail CGIs
`/compt/omailin`	Internal compartment hosting OpenMail server processes
`/compt/web1`	Externally visible HTTP-server
`/compt/web1mcga`	Internal trusted gateway agent for web1's CGI-binaries

In addition, these subdirectories also exist:

1. /compt/etc/cac/bin – various scripts and command-line utilities for managing compartments
2. /compt/etc/cac/rules – files containing rules for every registered compartment on the system
3. /compt/etc/cac/encoding – configuration file e.g. compartment-name mappings

Layout of a Typical Compartment

To support the generic starting/stopping of a compartment, each compartment has to conform to a few basic requirements:

1. be chroot-able under its compartment location /compt/<name>
2. provide /compt/<name>/startup and /compt/<name>/shutdown to start/stop the compartment
3. Startup and shutdown scripts are responsible for inserting rules, creating routing-tables, mounting filesystems (e.g. /proc) and other per-service initialization steps.

In general, if the compartment is to be externally visible, the processes in that compartment should not run as root by default and the compartment should be sealed after initialization. Sometimes, this is not possible due to the nature of a legacy application being integrated/ported – in this case, one should remove as many capabilities as possible in order to prevent the processes from escaping the chroot-jail e.g. cap_mknod.

Administration Scripts

Due to the fact that the various administration scripts require access to each configured compartment's filesystem and that these administration-scripts are called via the CGI-interface of the administration Web-server, it is the case that these scripts then cannot reside as a normal compartment i.e. under /compt/<name>.

The approach currently taken is to enclose the chroot-able environment of the administration scripts around every configured compartment, but to ensure that the environment is a strict subset of the host's filesystem. The natural choice is to make the chroot-jail for the administration scripts to have its root at /compt.

Integrating Applications

Since compartments exist as chroot-ed environment under the /comp directory, application-integration will require the usual techniques used for ensuring that they work in a chroot-ed environment. A common technique is to prepare a cpio-archive of a minimally running compartment, containing a minimal RPM-database of installed software. One would normally install the desired application on top of this. In the case of applications in the form of RPMs, one could perform these steps:

```
root@tlinux# chroot /compt/app1
root@tlinux# rpm -install <RPM-package-filename>
root@tlinux# [Change configuration files are required
             e.g. httpd.conf]
root@tlinux# [Create startup/shutdown scripts in
             /compt/app1]
```

Figure 5. Steps performed to install RPM-packages in a compartment

The latter few steps may be integrated into the RPM-install phase. Reductions in disk-space can be achieved by inspection: selectively uninstalling unused packages via the rpm-command. Additional entries in the compartment's /dev-directory may be created if required, but /dev is normally left pretty bare in most cases. Further automation may be achieved by providing a Web-based interface to the process above to supply all the necessary parameters for each type of application being installed. No changes to the compiled binaries are needed in general, unless one wishes to install compartment-aware variants of such applications.

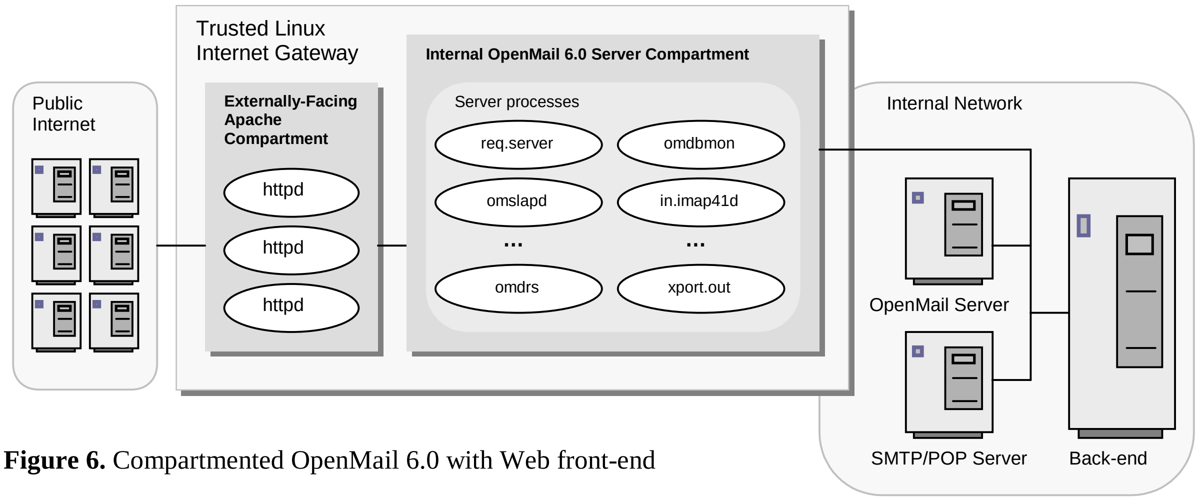

Figure 6. Compartmented OpenMail 6.0 with Web front-end

Example of Application Integration – OpenMail 6.0

The OpenMail 6.0 distribution for Linux consists of a large 160Mb+ archive of some unspecified format, and an install-script *ominstall*. To install OpenMail, we first chroot to an allocated bare-bones inner-compartment:

```
root@tlinux# chroot /compt/omailin
root@tlinux# ominstall
root@tlinux# [wait for OpenMail install to complete
              naturally ]
root@tlinux# [do additional configuration if
              required, e.g. setup mailnodes]
```

Since OpenMail 6.0 has a Web-based interface which we wish to also install, we allocated another bare-bones compartment (omailout), and install an Apache HTTP-server to handle the HTTP-queries.

```
root@tlinux# chroot /compt/omailout
root@tlinux# rpm –-install <apache-RPM-filename>
root@tlinux# [Configure Apache’s httpd.conf to handle
              CGI-requests as
              required by OpenMail’s installation
              instructions]
```

At this point we then have to install the CGI-binaries that come with OpenMail 6.0 so that they can be accessed by the Apache HTTP-server. There are a couple of methods available:

- Install OpenMail again in *omailout* and remove unnecessary portions e.g. server-processes
- Copy the OpenMail CGI-binaries from *omailin*, taking care to preserve permissions and directory structure.

In either case, the CGI-binaries typically are placed in the cgi-bin directory of the Apache Web-server. If disk-space is not an issue, the former approach is more brute-force and works well. The latter method can be used if one needs to be sure of exactly which binaries are to be placed in the externally-facing *omailout* compartment. Finally, we can start both compartments:

```
root@tlinux# comp_start omailout omailin
```

Figure 6 shows the final result – the only externally accessible processes are the Apache HTTP daemons which communicate via internal loopback to the internally-facing OpenMail server processes. These can optionally communicate with back-end servers.

Issues and Limitations

Capabilities in Linux

Capabilities do exist in Linux and most subsystems do check for them when performing operations that are considered to be privileged. However, the assignment of such capabilities on a per-user basis is not currently built-in into the standard distributions although the run-time propagation of capabilities from the init-process itself appears to have been thought out. There exist few tools to manipulate these capabilities (the *lcu*-utility is an example of one) and the the system include-files do not even include a function prototype for sys_capset/capget[3]!

Given the embryonic nature of capabilities in Linux, it is not possible to build a secure platform relying on the sole use of these capabilities for the execution of privileged operations. Whilst kernel-level code can be made to check for capabilities as opposed to zero-UIDs, all user-space binaries (except for lcu) have to remain unaware of this.

Handling IP fragments

It may be possible that IP fragments are received with different originating compartment numbers. In such a case, we should disallow the fragment re-assembly to proceed with fragments of differing compartment numbers. However, no checks are currently performed to implement this. The opposite happens when IP-fragments are generated in response to excedding the calculated MTU – each fragment inherits the compartment number of the original outsized packet.

Other Protocols

Support for various other network protocols needs to be added e.g. IPX/SPX etc. The effect of compartmentalisation on existing protocol IGMP, GRE etc. needs testing.

Filesystem Protection

A more comprehensive method for filesystem protection than chroot-jails needs to be developed. There exists utilities to verify ownership/permissions and MD5 hashes in the style of FCDBs, but these have not been

[3] Callers of the sys_capset/sys_capget system-calls typically resort to the _syscall*() interface to define equivalents to the sys_capset/sys_capget system calls.

integrated into the current prototype. Moreoever, they do not support the encoding of capabilities on a per-inode basis as no changes to the inode-structure have been made at this point. There exists spare bits in the standard ext2 inode, but any use of these will invariably clash with future changes to ext2 (and ext3!).

Kernel Compatibility

The current prototype is based on the 2.4.0 kernel. It is expected to be available for subsequent revisions when they become available. The size of nature of the patches required against the vanilla Linux source tree are such that tracking newer kernel revisions is relatively easy. There are no plans to back-port the changes for the 2.2 series of kernels.

Unsupported Configurations

Certain features of the Linux kernel are not supported, either because they bypass the kernel entirely or because no proper compartmentalization can be performed on them. Examples include:

- Hardware assisted card-to-card bridging
- NAT and IP-Masquerading
- NFS server configurations

Summary

The Trusted Linux kernel implements containment in a lightweight and flexible manner that allows a variety of services to be hosted in a secure manner. The enhancement of the vanilla kernel with mandatory access controls tuned for networked-applications such as web, mail and application-servers makes it ideal as a secure platform for an Internet gateway.

Application integration is relatively straightforward, following the normal constraints imposed by a chroot jail with the additional requirements that the Trusted Linux kernel provides itself. Relatively large binary-only packages, such as OpenMail, have been integrated without major difficulty.