Communications of the ACM (ISSN: 0001-0782) Volume 44 , Issue 2 (2001)

An operating system approach to securing e-services
Page 58

Chris Dalton and Tse Huong Choo

Trusted Linux – an operating system approach to securing E-Services

More and more services are being turned electronic and exposed to the public world of the Internet. Many of these are and will become attractive and lucrative targets to would-be attackers. A large number of Internet security breaches take place via compromise of the applications forming the electronic services.

The applications that form electronic services are in general sophisticated and contain many lines of code. It is not surprising that there are bugs in some of this code. Indeed, with such large applications it is very difficult to guarantee otherwise. Offering a service over the Internet means exposing it to a large population of attackers capable of probing the service for vulnerabilities. It is not unlikely, and has been shown to be the case in the past, that some of these bugs can and will be exploited to lead to security violations.

Increasingly, single machines are being used to host multiple services concurrently (e.g. ISP, ASP, xSP service provision). It is becoming critically important that not only is the security of the host platform protected from application compromise attacks but also the applications are adequately protected from each other in the face of attack.

This article looks at some of the problems surrounding application compromise in more detail and puts forward our approach to solving those problems. We do not attempt to guarantee that the application services are bug free (a hard problem). Instead, we have found that the effects of this type of attack, and quite a few others, can be usefully mitigated by adding specific properties to the operating systems used to host those applications

Specifically, we take a look at Trusted Linux which is HP Laboratories' implementation of a secure version of Linux, and that we believe is an ideal platform for e-service application hosting.

Using the OS to Counter Application Compromise

Tackling application compromise at the OS level by kernel enforced controls is an attractive proposition. The controls implemented in the kernel can not be overridden or subverted from user space by any application or user. The controls apply to all applications irrespective of the individual application code quality. The downside of this approach is of course that the controls may be too broad to be useful, however in practice this appears not to be the case, certainly for a large class of Internet service type applications. We shall see evidence of this later in the detailed discussion of Trusted Linux and how it can be configured and deployed.

There are two basic requirements at the system level in order to protect against application compromise and its effects. Firstly the application should be protected from attack as much as possible in the first place. The exposed interfaces to the application should be as narrow as possible. Access to those interfaces should be as well controlled as possible. Secondly, there should be some guaranteed limit on the amount of damage that an application can do to the system and other applications on the system should it be compromised.

These two requirements are fairly well captured by the abstract property of containment. An application is contained if it has strict controls placed on which resources (e.g., file, process, network, ipc) it can access (and what type of access it has, e.g., read-only) even if the application has been compromised. Containment also protects an application from external attack and interference.

Once an application has been compromised (for example by a buffer overflow attack) it can be exploited in several ways by an attacker to breach the security of a system. The containment property has the potential to mitigate many of these exploits, in some cases the exploits are entirely eliminated. Some of the benefits of containment can only be achieved at the application design level, however if implemented correctly it can be usefully used to effectively secure a large number of existing applications without application modification.

The most common exploits following the compromise of an application can be roughly categorized as one of four types. (Though the consequences of a particular attack may be a combination of any or all of these).

1. **Misuse of privilege to gain direct access to protected system resources.** If an application is running with special privileges (e.g., an application running as root on a standard Unix OS) then an attacker can attempt to use that privilege in unintended ways. For example, the attacker can use that privilege to gain access to

protected OS resources or interfere with other applications running on the same machine.

2. **Subversion of application enforced access controls.** This type of attack gains access to legitimate resources (i.e., resources that are supposed to be exposed by the application) but in an unauthorized fashion. For example, a web server that enforces access control on its content before it serves it is an example of an application susceptible to this type of attack. Since the web server has uncontrolled direct access to the content, then so does an attacker if they can gain control of that web server.

3. **Supply of bogus security decision making information.** This type of attack is usually an indirect attack. Here the compromised application is not normally the main service but a support one such as an authorization service. That compromised security service can then be used to supply bogus or forged information and thus gain the attacker access to the main service. This is another way for the attacker to gain unauthorized access to resources legitimately exposed by the application.

4. **Illegitimate use of unprotected system resources.** This is where the attacker gains access to local resources of the machine that are not protected but nonetheless would not normally be exposed by the application. Typically, these local resources are then used to launch further attacks. For example, it may be possible for an attacker to gain shell access to the hosting system. From here staged attacks may then be launched on other applications on the machine or across the network, for example.

With containment, type (1) exploits are much less serious; even if the attacker makes use of the application privilege the resources that can be accessed are bounded by what has been made available in the application's container. This is also true for type (4) unprotected resource access, e.g., using containment we can block access to (or at least put very tight controls on) the network from a contained application. By using containment to limit to exposure of applications to attack we can help against exploits of type (3). We can guarantee that the only access to support services is from legitimate clients, i.e., the application services.

Type (2) exploits usually have to be solved at the application design, or at least configuration level. The design principle here is that large sophisticated applications (such as a web server) should not be trusted to carry out access control checks. This also implies that they should not have direct access to the resources that are being protected The code responsible for making access control decisions should be single purpose and as small and bug free as possible. Hopefully, then compromise of this code is unlikely. This code and the resources that it is designed to protect should be hosted in their own compartment. Using containment, we can arrange that access to protected resources from the large untrusted application must go through the smaller, more trusted application [2].

Theoretically, at least it seems that containment is a very useful property to have in an operating system.

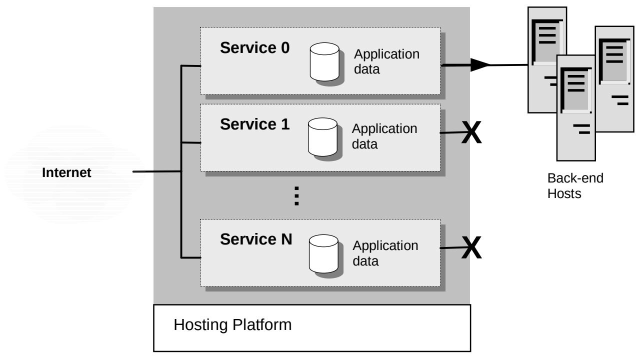

Figure 1 Example architecture for multi-service hosting on an operating system with the containment property.

Containment is used here to ensure that applications are kept separated from one another and critical system resources. An application can't interfere with the processing of another application or get access to its perhaps sensitive data. Containment is used to ensure that only the interfaces (input and output) that a particular application needs to function are exposed by the operating system. This limits both the scope for attack on a particular application and also the amount of damage that can be done should the application be compromised. Containment helps to preserve the integrity of the hosting platform as a whole not just individual applications.

Implementing Containment

Kernel enforced containment mechanisms in operating systems have been available for several years, typically in ones that were designed for the handling and processing of classified (military) information [4][3]. These types of operating systems are often termed *Trusted Operating* systems.

The containment property is usually achieved 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 [1]. This is a formal model developed on behalf of military organizations in which the flow of information around the system is predictable. Privileges break down the power of the superuser. With them an application can be given just the privilege it needs.

Despite offering the attractive property of containment, trusted operating systems have not been widely used outside of the classified information processing arena. The reasons for this seem two fold.

Firstly, attempts at adding trusted operating system features to conventional OSes have usually resulted in the underlying operating system personalities being lost. The operating systems have no longer supported standard applications or management tools. The operating systems can no longer be used or managed in standard ways. They are much more complicated than their standard counterparts.

Secondly, they have typically implemented a form of containment more akin to isolation, i.e., too strong - this has been found to be limited in scope in terms of it's ability to usefully and effectively secure [existing] applications without substantial and often expensive integration efforts. These are the two main issues we have attempted to address with our implementation of Trusted Linux.

The Trusted Linux Operating system

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

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 realise 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 hierarchical ordering of labels within the system such as there is in the Bell-LaPadula model[1].

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 and/or network resources.

These features together give us as a system that both offers containment and also has enough flexibility to make application integration relatively

[1] In most implementations of the Bell-Lapadula model, resource labels have 2 components to them; an ordered sensitivity level and zero or more compartment names. Our labels just have a single compartment name and no sensitivity level.

straightforward. This in turn reduces the management overhead and pain of deploying and running a trusted system.

Kernel Implementation

Within the kernel, each system-resource that we wish to protect has been extended with a tag indicating the compartment that the resource belongs to. Examples of such resources include data-structures describing

- individual processes,
- shared-memory segments,
- semaphores, message-queues,
- sockets, network packets, network-interfaces and routing-table entries.

The assignment of the tag occurs largely through inheritance, with the *init*-process initially being assigned to compartment 0. Any kernel objects created by a process inherit the current label of the running process. An example of the tag added to the kernel socket data-structure is shown below:

```
struct socket {
              ...
#ifdef TLINUX
              unsigned long compartment;
#endif
};
```

At appropriate points in the kernel, access-control checks are performed through the use of hooks to a dynamically loadable security-module that consults a table of rules indicating which compartments are allowed to access the resources of another compartment. This occurs transparently to the running applications.

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.

The rules are directional are in nature. One effect of the directional nature of the rules is that they match the connect/accept-behavior of TCP socket connections. Consider a rule used to specify allowable incoming HTTP connections of the form:

```
HOST:* -> COMPARTMENT X METHOD TCP PORT 80
```

This rule specifies that only incoming TCP connections on port 80 are to be allowed, but not outgoing connections.

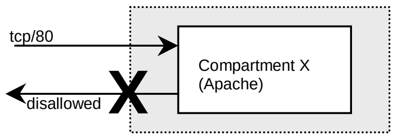

Figure 2 Only incoming TCP connections allowed

The directionality of the rules permits the reverse flow or packets to occur in order to correctly establish the incoming connection without allowing outgoing connections to take place.

Protocol Stack Virtualisation

Because most of the data structures used in the IP protocol stack are tagged, it is possible to partly virtualize its operation on a per-compartment basis. In particular, we can specify individual routing tables for each configured compartment. The segmentation of the data used within the IP stack covers high-level constructs like sockets all the way down to individual IP fragments and ARP cache entries.

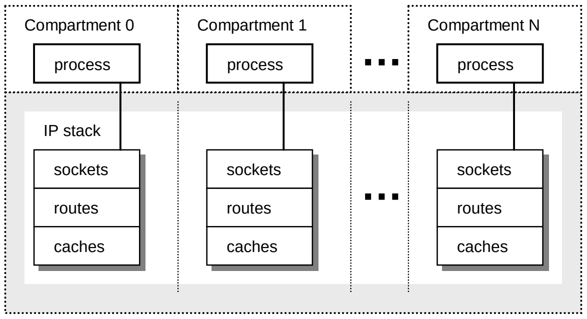

Figure 3 Semi-virtualized IP protocol stack

Other virtualized subsystems include the System V IPC mechanisms and the process table. When the *ipcs* command is invoked, it will only show the IPC objects in the current compartment. Similar restrictions are in place for process listings using *ps*. The virtualization occurs based on the context of the calling process and no user-level commands need to be modified to work in a compartmented fashion.

Installation & Configuration

Trusted Linux is designed to install in a layered fashion over standard RedHat installations. No change to the file system is required. The installation replaces the vanilla Linux kernel and any network services which remain will start in compartment 0 upon the next reboot.

Specific software packages are available for the use of Apache together with a Multi-Compartmented Gateway Agent (MCGA) which performs CGI-sandboxing by executing individual named CGI-binaries in separate compartments.

The diagram below shows a typical configuration hosting 3 HTTP servers each in their own compartments with another 3 compartments hosting the MCGA agents.

Each package can be installed in its own compartment, with the number of compartments limited by disk space (the tags are the size of a machine word each). This allows the possibility of hosting multiple instances of a web-server independently of each other and to also prevent security breaches in one affecting the others.

The applications that have been integrated without source-modifications so far include: Apache, Jakarta/Tomcat, HP OpenMail, and BEA WebLogic Server.

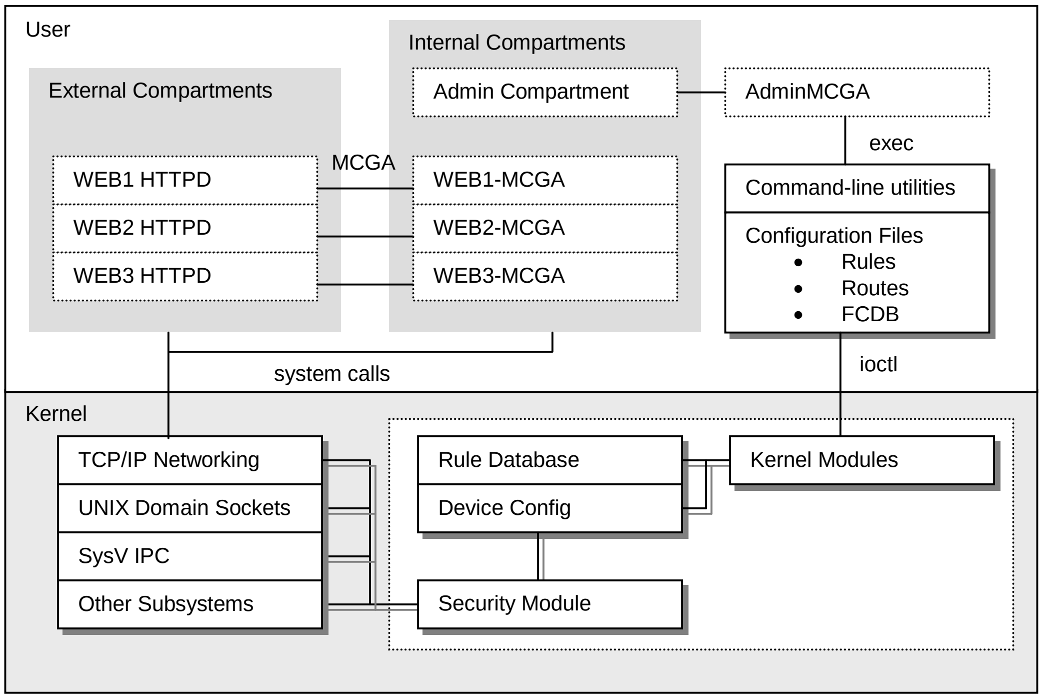

Figure 4 A typical secure Web-server configuration on Trusted Linux with CGI-sandboxing

Performance

Initial tests indicate that most aspects of performance fall within a few percentage points of a vanilla Linux kernel. A test system comprising twin Pentium-III 733Mhz CPUs, 256Mbytes RAM and dual Intel EtherExpress Pro/100Mbit NICs gave the following figures running Apache 1.3.12 using the HTTP GET operation on a static 1024-byte HTML file. The test program was the *ab*-utility from the Apache distribution, invoked as:

```
ab –n 100000 –c 50 [url]
```

Configuration	HTTP GET
Linux-2.4.0-test5	3298
Trusted Linux based on Linux-2.4.0-test5	3265 (1% penalty)

Configuration Example: Apache+Jakarta/VMs

This section illustrates how to compartmentalize a setup comprising an externally facing Apache Web-server configured to delegate the handling of Java servlets or the serving of JSP files to two separate instances of Jakarta/Tomcat each running in its own compartment. By default, each compartment uses a *chroot*-ed filesystem so as not to interfere with other compartments.

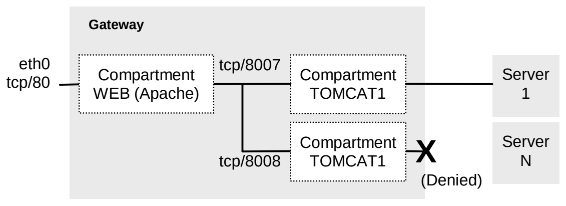

Figure 5 Configuration of Apache with 2 Tomcat VMs

The diagram above shows the Apache processes residing in one compartment (WEB). This compartment is externally accessible using the rule:

```
HOST:* -> COMPARTMENT WEB
     METHOD TCP PORT 80 NETDEV eth0
```

The presence of the `NETDEV` component in the rule specifies the network-interfaces which Apache is allowed to use. This is useful for restricting Apache to using only the external interface on dual/multi-homed gateway systems. This is to help prevent a compromised instance of Apache from being used to launch attacks on back-end networks through internally facing network interfaces.

The WEB compartment is allowed to communicate to two separate instances of Jakarta/Tomcat (TOMCAT1 and TOMCAT2) via two rules, which take the form:

```
COMPARTMENT:WEB -> COMPARTMENT:TOMCAT1
    METHOD TCP PORT 8007

COMPARTMENT:WEB -> COMPARTMENT:TOMCAT2
    METHOD TCP PORT 8008
```

The servlets in TOMCAT1 are allowed to access a back-end host called Server1 using this rule:

```
COMPARTMENT:TOMCAT1 -> HOST:SERVER1
    METHOD TCP …
```

However, TOMCAT2 is not allowed to access any back-end hosts at all. The kernel will deny any such attempt from TOMCAT2. This allows one to selectively alter the view of a back-end network depending on which services are being hosted, and to restrict the visibility of back-end hosts on a per-compartment basis.

It is worth emphasizing at this point that these four rules are all that is needed for this example configuration. In the absence of any other rules, the servlets executing in the Java VM can not initiate any outgoing connections, in particular it can not be used to launch attacks on the internal back-end network on interface *eth1*. In addition, it may not access resources from other compartments (e.g., shared-memory segments, UNIX-domain sockets), nor be reached directly by remote hosts. In this case, mandatory restrictions have been placed on the behavior of Apache and Jakarta/Tomcat without recompiling or modifying their sources.

Summary

We have described a platform based on Linux which implements the containment property to dynamically separate running services. The main use of the platform is to host (multiple) services in a compartmented fashion. This places guaranteed limits on the amount of damage that can be done via a compromised service. In particular, a compromised service can't be used to interfere with other services on the same platform. Likewise, the base integrity of the platform as a whole is protected from attack by a compromised service.

Trusted Linux represents a comfortable middle-ground between the heavyweight approach of traditional trusted operating systems and conventional operating systems by offering strict kernel-level access controls without requiring existing applications to be re-written to take advantage of new security-features.

It has been designed to incrementally secure vanilla Linux installations by layering a more secure kernel and leaving any unconfigured services in a relatively unprivileged compartment. The approach taken can be extended to other flavours of UNIX (e.g., HP-UX, Solaris) and can be broadened to cover more communications mechanisms.

One of the interesting possibilities of the current prototype is to extend the format of the access-control rules to cover compartments on remote hosts in a secure fashion, perhaps by implicitly securing all compartment-to-compartment communications through the use of IPSec.

References

[1] Bell, D. and Lapadula, L. (1975) Secure Computer Systems: unified Exposition and Multics Interpretation, *Mitre Technical Report MTR-1997, Mitre Corporation.*

[2] Dalton, C.I., Griffin, J.F. Applying Military Grade Security to the Internet - section 4.3 COMPUTER NETWORKS AND ISDN SYSTEMS – vol 29 Number 15.

[3] Hewlett-Packard Co. (1996) HP-UX 10.16 CMW Security Features Guide

[4] Sun Microsystems Corporation. Trusted Solaris Operating System. <URL: http://www.sun.com/trustedsolaris>