Securing Commercial Operating Systems

Since the discovery of the reference monitor concept during the development of Multics, there have been many projects to retrofit existing commercial operating systems with a true reference monitor implementation. Successful, commercial operating systems can have a large customer base and a variety of popular applications. As a result, those customers with strong secrecy and integrity requirements (e.g., US Government) often encourage the construction of secure versions of existing commercial operating systems. Many such systems have been retrofitted over the years.

In this chapter, we explore some of the commercial systems that have been retrofitted with reference monitors. The aim is not for completeness, as there are far too many systems, but we want to capture the distinct movements in creating a secure operating system from an existing commercial system.

Converting an existing code base to one that implements a reference monitor is a challenging task. In order to be a secure operating system, the resulting code base must achieve the three reference monitor guarantees, but this is difficult because much of the code was not developed with these guarantees in mind. This contrasts markedly with the security kernel approach in Chapter 6 where the system design considers mediation, tamperproofing, and verification from the outset.

After outlining the tasks involved in retrofitting a commercial operating system with a reference monitor, we examine a variety of different retrofitted systems. We group these systems by a trend motivating their construction. We examine the resultant system architectures in detail for two systems: Solaris Trusted Extensions in Chapter 8 and the Linux operating system in Chapter 9.

7.1 RETROFITTING SECURITY INTO A COMMERCIAL OS

To retrofit a commercial operating system into a secure operating system, the resultant operating system must be modified to implement a secure operating system that implements the reference monitor concept, see Definitions 2.5 and 2.6. The reference monitor concept requires guarantees in complete mediation, tamperproofing, and verifiability. There are challenges in each of these areas.

Complete mediation requires that all the security-sensitive operations in the operating system be identified, so they can be authorized. Identifying security-sensitive operations in a complex, production system is a nontrivial process. Such systems have a large number of security-sensitive operations covering a variety of object types, and many are not clearly identified. As we will see, a significant number of security-sensitive operations are embedded deep inside the kernel code. For example, in order to authorize an open system calls, several authorizations may be necessary for
directories, links, and finally the target file (i.e., inode) itself. In addition to files, there are many such objects in modern operating systems, including various types of sockets, shared memory, semaphores, interprocess communication, etc. The identification of covert channels (see Chapter 5) is even more complex, so it is typically not part of retrofitting process for commercial operating systems. As a result, complete mediation of all channels is not ensured in the retrofitted operating systems we detail.

Tamperproofing the reference monitor would seem to be the easiest task in retrofitting an existing system, but this also has proven to be difficult. The obvious approach is to include the reference monitor itself in the kernel, so that it can enjoy the same tamper-protection that the kernel has (e.g., runs in ring 0).

There are two issues that make guaranteeing tamper-protection difficult. First, commercial operating systems often provide a variety of ways to update the kernel. Consider that UNIX kernels have a device file that can be used to access physical memory directly /dev/kmem. Thus, processes running outside of the kernel may be able to tamper with the kernel memory, even though they run in a less-privileged ring. Modern kernels include a variety of other interfaces to read and write kernel memory, such as /proc, Sysfs file systems, and netlink sockets. Of course, such interfaces are only accessible to root processes, but there are many processes in a UNIX system that run as root. Should any one get compromised, then the kernel may be tampered. In effect, every root process must be part of a UNIX system’s trusted computing base to ensure tamper-protection.

But the biggest challenge for retrofitting an operating system is providing verification that the resultant reference monitor implementation enforces the required security goals. We must verify that mediation is implemented correctly, that the policy enforces the expected security goal, that the reference monitor implementation is correct, and that the rest of the trusted computing base will behave correctly. Verifying that the mediation is done correctly aims to address the problems discussed above. Typically, the mediation interface is designed manually. While tools have been developed that find bugs in mediation interfaces [149, 351], proving the correctness of a reference monitor interface in an operating system is intractable in general because they are written in nontype safe languages, such as C and various assembly languages.

Policy verification can also be complex as there are a large number of distinct authorization queries in a commercial operating system, and there are a large number of distinct processes. Some retrofitted commercial operating systems use a multilevel security (MLS) model, such as Bell-LaPadula [23], but many use access matrix mandatory access control (MAC) models, such as Type Enforcement [33]. The latter models are more flexible, but they also result in more complex policies. A Bell-LaPadula policy is fixed size, but an access matrix policy tends to grow with the number of distinct system programs. Such models present a difficult challenge in verifying that each system is enforcing the desired security goals.

Finally, the implementation of a commercial operating system and the remaining trusted computing base is too complex to verify whether the overall system protects the reference monitor. Commercial operating systems are large, there are often several developers of the trusted computing
base software, and the approaches used to build the software are not documented. The best that we can hope for is that some model of the software can be constructed after the fact. As described in Chapter 6, the verification of Scomp's correctness required an evaluation that the design model enforced system security goals and that the source correctly implemented the design. Many believe that it is not possible to build a sufficiently precise design of a commercial system and a mapping between this design and the system's source code necessary to enable such verification. Clearly, current technologies would not support such a verification.

7.2 HISTORY OF RETROFITTING COMMERCIAL OS'S

In this section, we examine the evolution of retrofitting security into commercial operating systems. We organize this section by identifiable eras in the construction of secure operating systems. As the lessons from the Multics project were being disseminated, many companies examined ways to retrofit Multics-style security into their existing commercial operating systems during the **commercial era**. The invention of the microkernel systems led to several attempts to retrofit security in the smaller, microkernel architectures, which resembled security kernels (see Chapter 6), during the **microkernel era**. Gradually, the focus returned to UNIX systems, which had become the de facto server operating system (although there were many distinct UNIX systems maintained by competing entities by then). Some of the novel ideas of the commercial and microkernel era were transferred to UNIX-style systems in this most recent era, the **UNIX era**.

Each of the eras focused on particular themes. The commercial era work focused on either emulation of commercial systems on security kernels or retrofitting by adding orthogonal features to existing code bases. The result of this era was systems that enforce multilevel secrecy policies in UNIX. The microkernel era focused on adding security via independent server processes, but as the work proceeded, more invasive modifications, lower in the software stack were deemed necessary. Also, innovative security models emerged that aimed to address both secrecy and integrity comprehensively. The UNIX era composed the mature solutions of the first two eras with a renewed focus on system integrity. Both Solaris Trusted Extensions (see Chapter 8) which resulted from the commercial era and the SELinux (see Chapter 9) which resulted from the UNIX era have adopted many similar solutions, although there are significant differences and some challenges remain open to future research.

7.3 COMMERCIAL ERA

In the late 1970s and early 1980s, it became clear that Multics provided some fundamental security features, see Chapter 3, but it was too large and slow to be effective. A variety of competing vendors saw improvements in security as a potential advantage for their systems. A goal became to capture Multics security features in their commercial systems. The chief question was how to marry the security enforcement of Multics with the application interface of these commercial system.
Emulated Systems: Data Secure UNIX and KSOS  Some projects focused on the construction of a security kernel, see Chapter 6, that ran an emulator for the UNIX API. UCLA Data Secure UNIX [248] and KSOS [198, 97], fall into this category. In both cases, the performance of the emulated systems was poor, so later security kernels, such as Scomp, dropped the idea of an emulator.

These systems did not really integrate security into the existing operating system, but rather tried to slide a secure environment under the existing system. However, in addition to problems in performance, insecure features of the UNIX interface, such as permitting the passing of file descriptors on process creation regardless of the relationship between the processes, also presented security problems that could cause incompatibilities with the security kernel.

KVM/370  The KVM/370 system adds a layer between the virtual machine monitor (VMM) and individual virtual machine (VM) to mediate inter-VM communication. The design of this new layer is a retrofit of the existing VM/370 code base with multilevel security features. The retrofit resulted in performance overhead of about 25% of a typical VM/370 virtual machine. This was partly due to the additional layer between the virtual machines and the VMM and partly due to required reuse of VM/370 code in the new KVM/370 system, which introduced extra effort in indirection. Virtual machine-based secure operating environments are discussed in Chapter 11.

VAX/VMS  DEC and Sandia Labs retrofitted VAX/VMS with multilevel security enforcement [180]. In addition, improvements in auditing were developed and a number of security vulnerabilities were fixed. Because of the retrofit of the existing code base, the VAX/VMS system aimed only for modest assurance levels, B1 or B2 in the Orange Book [304]. This work was a prototype and performance impact was not discussed.

Secure Xenix  Somewhat later than the work above, IBM retrofitted Microsoft’s Xenix with access control and auditing features [111]. This work was influenced the UNIX retrofit of Kramer [173], but aimed to provide a comprehensive and effective implementation of Multics security features [280] (see Chapter 3) in Xenix. Two key issues among several addressed by the Secure Xenix work were compatibility and trusted path. First, the Secure Xenix system included both the retrofitting of a variety of UNIX services with security-aware function, so that UNIX applications could be run without modification. Further, compatibility mechanisms, such as hidden subdirectories, were invented to enable multiple processes at different security levels to “shared” directories without introducing information leakage. This mechanism is the basic idea behind polyinstantiated file systems discussed in Section 8.2 and used in several systems now. Second, Secure Xenix also introduced the notion of a trusted path. A trusted path is a mechanism to communicate directly with the system’s trusted computing base. A trusted path is often implemented via a “secure attention sequence” that can only be caught by the trusted computing base (e.g., Control-Alt-Delete). Thus, the user can be certain that she is communicating with trusted code. Secure Xenix was successfully evaluated at the US
government B2 rating based on the Orange Book [304]. Secure Xenix was later renamed Trusted Xenix, when its development was shifted to Trusted Information Systems.

By 1990, a variety of UNIX variants had been extended with security mechanisms, particularly those aiming at MLS enforcement [339]. One of these systems, SunOS MLS was introduced in 1989, but it ultimately established itself as the market leader in MLS systems. It has continued to evolve over the last twenty years, so we present an overview of the current version, called Trusted Solaris Extensions, in Chapter 8.

### 7.4 MICROKERNEL ERA

In the 1980s, microkernel systems emerged. Microkernel systems were similar to security kernels in that they aimed for minimal functionality in the kernel, but microkernel systems focused on providing system abstractions for building complete systems more easily and more efficiently, rather than more securely. The hope was that microkernel systems would be more effective at running UNIX systems (i.e., perform better), while preserving the economy of size and potential for verification offered by the security kernels.

The emergence on the Mach microkernel [348, 116] in the 1980s was the source of interest in microkernel systems. Mach aimed to be a minimal kernel while provided abstractions to enable complete operating system construction, including mechanisms for message passing between components and multi-threaded process support. However, fundamental operating systems services, such as memory managers, file systems, and network servers, are not implemented in the microkernel, so user-level servers must be designed to implement such functions. Thus, the system’s trusted computing base consists of the microkernel and those user-levels servers that must be trusted. While these microkernel architectures appear similar to the security kernel architectures of Chapter 6, typically microkernel systems were built to improve nonsecurity dimensions of operating systems, such as ease of development, flexibility, and even performance.

Several projects that used Mach as the base for a secure operating system were inspired by that architecture. These included Trusted Mach (TMach) [36, 35, 196], Distributed Trusted Mach (DTMach) [282, 96], the Distributed Trusted Operating System (DTOS) [313, 213], and the Flask system [191, 295]. TMach was built by Trusted Information Systems (TIS) and implemented multilevel security (MLS) servers for files, memory, etc. that would provide function for single-level operating system personalities, such as UNIX or Windows. Thus, TMach provides trusted services for MLS computing where each instance of a traditional operating system runs as a single-level system. DTMach was built by Secure Computing Corporation (SCC) and the National Security Agency (NSA). DTMach supports a hybrid access control model that uses both MLS labels for secrecy control and Type Enforcement (TE) [33] labels for integrity. TE is an access matrix-based, mandatory access control policy with a fixed set of subject and object labels, called types, and the policy defines which subject types may perform which operations on which object types. TE was first applied in the LOCK system [293]. When a new file or process is defined, it is labeled from the type set and inherits the policy defined by the TE matrix.
The DTMach architecture is similar to TMach, but it also includes additional servers for networking DTMach systems and providing general security policy server support. DTOS was the SCC/NSA/University of Utah followup to the DTMach system. The DTMach project found security limitations in the Mach microkernel mechanisms that the DTOS project aimed to fix. The Mach architecture was found to have significant performance issues, so the DTOS architecture was migrated to another microkernel system for the Flask project. We explore some of the issues in DTMach, DTOS, and Flask projects below.

**DTMach**  DTMach extended Mach with a separate security server, a reference monitor outside the kernel that responds to authorization queries. As file, network, and interprocess communication (IPC) are invoked by sending messages to Mach ports, DTMach authorization queries are invoked on port access. For example, when a process opens a file, it sends a message to a port of the file server hosting that file. The security server is invoked to ensure that the process has the necessary permissions to access the file.

The DTMach security server represents permissions in two forms, MLS permissions and TE permissions. MLS permissions enforce secrecy using the traditional Bell-LaPadula model [23]. TE permissions were used to protect the integrity of the system. TE was used in DTMach to define limited mandatory domains for users and particular system services. TE policies in DTMach limit code installation and modification to administrators only, limit the code that can be executed by system subjects, prevent servers from having unnecessary rights to system objects, ensure that only authorized downgraders could relabel certain data, etc.

However, Mach ports suffered from some limitations that prevented correct enforcement of TE policies. For example, a `send` right on a Mach port implies that a process with that right can send arbitrary messages to the port, but we may want to limit the set of messages that untrusted processes can send to ports served by trusted processes. Consider when an untrusted process asks for a file to be mapped into its address space. In this case, the process must have a `send` permission to the memory pager to ask for the file to be mapped. However, this right permits the untrusted process to send any message to the pager, increasing the complexity of pager. DTMach defines more nuanced `send` rights to only allow file mapping requests. There are other specific cases where the meaning of a `send` permission to a port must be limited, and these are all handled by extending the port authorization mechanism.

**DTOS**  The changes to Mach and its server to address port control resulted in several ad hoc changes to the Mach microkernel. In the DTOS project, the aim was to construct a true reference monitor in the Mach microkernel [283, 213]. To address the problems caused by `send` port permissions above, DTOS defined a richer set of operations for operating on ports. The DTOS Mach microkernel managed the labeling of subjects and kernel objects and provided access control over each kernel operation by querying the security server itself. This resulted in complete mediation of kernel operations with the richness necessary to limit access to trusted servers in a tamperproof Mach
microkernel. The DTOS project also focuses on verifiability through assurance of the microkernel and its trusted computing base.

**Fluke/Flask** So-called second generation microkernels made significant improvements in the IPC performance [184, 185, 83] making Mach obsolete, so the DTOS architecture was ported to a second generation microkernel called Fluke from the University of Utah [98]. The resulting security architecture, called Flask, retained many of the elements of the DTOS architecture, in particular the microkernel reference monitor, trusted servers (called object managers), and a separate security server [295]. At this time, the focus shifted from MLS to the TE mechanism, as the latter is more general and integrity protection became the focus. As with all the second generation microkernels, Fluke did not attract a large user community, and another UNIX-based system was emerging in popularity, Linux, see Chapter 9.

### 7.5 UNIX ERA

By the early 1990s, a variety of different approaches to retrofitting UNIX systems has been explored [339]. In the process of constructing these systems, a variety of technical challenges were discovered and solutions were proposed. This ultimately resulted in successful deployment of UNIX systems that supported MLS policies. However, the research community continued to explore comprehensive UNIX retrofitting that would address both integrity and security in concert. While the UNIX MLS systems did not ignore integrity, integrity was addressed more implicitly, so it was left to the administrators to ensure that the inputs to their high secrecy systems were also high integrity. We examine two systems in detail here, IX and DTE, which retrofit UNIX with integrity and secrecy mechanisms.

#### 7.5.1 IX

AT&T Research built an experimental UNIX prototype that enforces multilevel (MLS) secrecy and integrity, called IX [200]. IX provides a reference monitor over file access that implements a mandatory access control policy that provides secrecy and integrity protections. Care is also taken in the definition of the trusted computing base to prevent tampering. Verification of security guarantees is not a focus in IX. This is partly due to the complexity of verifying correctness on an existing kernel, and partly due to the desire for more flexible labeling.

The IX mandatory access control policy enforces information flow secrecy. Processes have labels in a secrecy lattice that ensure that they may only read data at their secrecy level or lower. Unlike traditional MLS systems, such as Multics [23, 280], IX uses dynamic labeling to provide more flexible information flow control, however. The label of a file may change if it receives information from a process at a higher secrecy level in the lattice. For example, a file may start with a low secrecy label, but its label is changed to high secrecy when a high secrecy process updates it.
IX also includes a transition state that enables relabeling of processes and objects. Process labels change based on the secrecy labels of the files that they have read. However, an *ceiling* is defined on process labels to limit the level that a process may reach. Ceilings may also be associated with file systems to limit the secrecy of data written to that file system.

IX also supports a separate integrity lattice with dynamic labels as well. IX uses the LO-MAC [27, 101] semantics for integrity labels. The label of an entity (process or file) is equal to the lowest lattice label of an input to that entity. IX uses label *floors* to limit the degradation of integrity that is possible for particular processes or files. A process may not read a file whose integrity is lower than its floor which ensures that the integrity of a process is at least at the label of its floor.

The use of dynamic labeling requires authorization on each data transfer. For example, if a file is opened for writing when the process is at high integrity, it may no longer be written after the process's integrity has decreased. Thus, authorization must be performed on all reads and writes, not just at the time the file is opened. Such semantics precludes the use of memory-mapped files, because file accesses are implemented by memory operations, rather than system calls. The risk is that a process could map a high integrity file into its address space, load potentially malicious code, and write to the file via the mapped memory without mediation by the reference monitor. Thus, memory-mapped files must be prohibited in IX.

IX provides mechanisms for establishing trusted paths (called *private paths* in IX) between trusted processes. IX defines a *pex*, a process exclusive access to a file or pipe, that prevents interference from other processes. Further, a pex provides labels of the processes on each end of the pex, such that an *assured pipeline* [33] of processes can be constructed. Figure 7.1 demonstrates an assured pipeline. *Process 1* is the only one that can read the input data (labeled *0th*) and generate the output data (labeled *1st*). *Process 2* then reads the *1st* data and then outputs the next data *2nd* that is used by *Process 3*. Since an assured pipeline can be used to ensure that a computation spanning multiple processes is high integrity, each of its stages must be performed only by trusted code.

![Image](image_url)

**Figure 7.1:** An assured pipeline chains together processes to perform a secure computation where each process can only communicate with its predecessor and successor. *Process 1* is the only one that can read the input data labeled *0th*, and outputs data of label *1st* that can only be read by *Process 2*.

### 7.5.2 DOMAIN AND TYPE ENFORCEMENT

Trusted Information Systems (TIS) retrofitted UNIX with a reference monitor that implements an extension of the Type Enforcement (TE) [33] policy, called Domain and Type Enforcement.
UNIX ERA

UNIX has many trusted (i.e., root) processes that had been found to be vulnerable (and often still continue to be vulnerable) to malicious network inputs. The DTE approach aims to confine UNIX processes to protect the trusted computing base from other root processes.

Strictly speaking, DTE UNIX runs as a server on a TMach system [35, 15]. However, we consider DTE to be a retrofit of UNIX because the reference monitor is added to the UNIX server (OSF/1), not to TMach.

DTE Policy Model

DTE extends the TE model by distinguishing subject types from object types and adding transition states. In classical TE [33], there is only one set of types covering all subjects and objects. In a DTE policy, types are assigned to objects, and domains are assigned to subjects (i.e., processes). A domain is a tuple consisting of three parts: (1) access rights to object types; (2) access rights to subjects in other domains (e.g., signals); and (3) an entry point program, a file that when executed triggers the domain. DTE domains describe how a process accesses files, signals processes in other domains, and creates processes in another domain.

First, DTE UNIX defines limited protection domains for root processes [323]. Instead of complete access, a least privilege policy is defined for such processes. Thus, should a network-facing daemon be compromised, the extent of the damage that the attacker could cause may still be limited. For example, it may not be possible for a compromised root process to install a rootkit under the confinement of the TE policy. Also, the definition of domain transitions (see below) limits which domains may be invoked by any process, also limiting possible malicious actions should a process be compromised.

Second, signals are a mechanisms for the operating system or other processes to notify a process. A signal interrupts the target process and forces it to handle the signal immediately. Signals may be used for a variety of purposes, such as terminating or resuming a process, but the process's signal handler defines the effect. If an untrusted process can submit a signal to other processes, then it can cause unauthorized execution resulting in termination or incorrect behavior.

The third element in domain specification defines the transition state. For each domain, DTE enables us to control transitions using the third element of a domain specification. In DTE, we limit domain transitions to the execution of a specific file corresponding to that domain and limit the domains that can cause that transition. In UNIX, the conditions under which a process starts also depends on environment variables, input arguments, file descriptors not set to close-on-exec, etc., so controlling the file used for a domain transition is only a start. Compare this to a ring transition in Multics which also uses gatekeepers to verify arguments, see Chapter 3.

Labeled Networking

A key innovation that appeared around this time was labeled networking. The idea is that each machine labels its network packets so that the receiver may authorize delivery to its processes. For example, two machines may have both secret and confidential processes. When a secret process sends a packet, the sending operating system adds the label to the packet header. When the packet is received, the receiving operating system extracts the packet’s label, and
authorizes delivery to a receiver's socket based on its label. Thus, if the secret packet is targeted for a socket created by a confidential process, the receiving system can deny authorization. Thus, authorization can span processes on multiple machines.

Labeled networking was enabled by the addition of IP security options to the IP header and protocol [153], later revised in IETF RFC 1108 [165]. IP security options enabled transmission of the sensitivity level of the packet. Implementations that used the IP security options header were developed. Ones based on RFC 1108 directly are referred to as revised IP security options (RIPSO) system. Since the information is simply stored in the IP header, external mechanisms were still needed to protect the header in transit over an untrusted network (e.g., IPsec [168, 166, 167]).

A later revision to the approach, called the commercial IP security option (CIPSO), was proposed and implementations were built to support it, although it was never standardized (there is a draft [53]). CIPSO has generally superseded RIPSO. It consists of a domain of interpretation, defining the meaning of the labels, and a specification for labels (tags) that can include levels and categories. Both secrecy and integrity levels are supported. Implementations often support both RIPSO and CIPSO. Solaris Trusted Extensions (see Chapter 8) supports RIPSO, CIPSO, and its own variant called Trusted Systems Information Exchange for Restricted Environments (TSIX) [301].

DTE also includes a form of labeled networking [15]. In this case, DTE includes the labels of the data being written (type) and the process doing the writing (domain) inside the packet. IP security options are also used to encapsulate the DTE labels. The labeling of both process and data domains means that the system must be able to reliably label the data being sent. For example, NFS servers were modified to use DTE networking, and they are responsible for labeling the file data that they deliver to clients. The range of labels that may be specified by a particular NFS server may be limited by DTE, so there is not a need for fully trusted servers.

7.5.3 RECENT UNIX SYSTEMS
A variety of UNIX systems now include a significant set of security features. We focus on two such UNIX systems in this book, Solaris Trusted Extensions in Chapter 8 and SELinux in Chapter 9, but there are several others of note. In particular, security has been a major focus of several of the BSD variants. The BSD systems derived from the Berkeley Software Distribution, a derivative of UNIX developed at the University of California, Berkeley [201]. We briefly examine them here, but recommend a detailed examination of their security features, as well.

Within the FreeBSD [103] community, the Trusted BSD project [327, 328, 312] aims to implement trusted operating system extensions for FreeBSD. TrustedBSD includes services for mandatory access control, auditing, and authentication, necessary to implement a mandatory protection system (see Definition 2.4 in Chapter 2). TrustedBSD includes a mandatory access control framework that implements a reference monitor interface, analogous to the Linux Security Modules framework [342] for Linux (see Chapter 9). The TrustedBSD framework enables the development of reference monitor modules that can enforce mandatory access control policies. One such module is SEBSD, a version of the SELinux module designed for BSD. Another FreeBSD security project
7.6. SUMMARY

Adding security features to an existing operating system, with its existing customer base and applications, has been a popular approach for building secure systems. Unfortunately, retrofitting security into existing, insecure systems leads to a variety of issues. Many programs are designed and configured such that they will not work in the more restrictive environment of a secure system. The operating systems themselves have complex interfaces that may be difficult to mediate.

In this chapter, we surveyed a variety of systems where security is retrofitted. We describe the security features that are added to these systems, the challenges in ensuring that the reference monitor concept is achieved, and the decisions that were taken to address these challenges. In general, these efforts show that it is practical to add a reference monitor interface to an existing system, but that it difficult to ensure the reference monitor guarantees are actually achieved. The complexity
and dynamics of these commercial systems prevent security professionals from developing models necessary to verifying mediation, tamperproofing, or correctness. We examine these challenges in detail for Solaris Trusted Extensions in Chapter 8 and for Linux in Chapter 9.