Building Systems to be Shared Securely

POUL-HENNING KAMP, FREEBSD DEVELOPER, AND ROBERT WATSON, McAFEE RESEARCH

The history of computing has been characterized by continuous transformation resulting from the dramatic increases in performance and drops in price described by Moore's law. Computing "power" has migrated from centralized mainframes/servers to distributed systems and the commodity desktop. Despite these changes, system sharing remains an important tool for computing. From the multitasking, file-sharing, and virtual machines of the desktop environment to the large-scale sharing of server-class ISP hardware in collocation centers, safely sharing hardware between mutually untrusting parties requires addressing critical concerns of accidental and malicious damage.

There has been a strong continuing interest in access control and separation technologies to support safe yet efficient sharing of computing systems. Although the degree of sharing taking place has increased because of similarly dramatic changes in networking, there has not been a marked change in the nature of security concerns. Costs of adapting software, identifying policy goals, and minimizing administrative complexity remain a balancing act. The fundamental security primitives for separation have not changed, but our experience in attempting to apply them is substantially greater.

Want to securely partition VMs? One option is to put ’em in Jail.

Building Systems to be Shared Securely

In this discussion, we will review existing systems that provide strong separation via access control, virtualization, namespace management, and partitioning. We will then explore the FreeBSD Jail system, implemented by us, which adopts a hybrid approach to applying security primitives. We will pay particular attention to the implementation and administrative costs of modifying an existing system. We will also make recommendations for the design of new systems to facilitate the future introduction of security and separation features. These recommendations will reach the somewhat unsurprising conclusion that the path to security turns out to also be the path of improved and intentional software design and abstraction.

WHY SHARE SYSTEMS?

The sharing of systems refers to the use of a system to serve simultaneous functions with differing security properties. Sharing systems offers the following benefits:

- In environments with inadequate computing resources, especially early computing environments and modern clusters, sharing allows for a more efficient use of valuable hardware resources, as well as for a joint investment in hardware resources.
- The development of ubiquitous local-area and wide-area networking has facilitated large-scale inter-system communication, permitting computers to become meeting points between individuals and organizations.

Although the terms of sharing have changed, the requirements remain largely the same: tasks and data representing the interests of multiple users coexisting on the same hardware platform.

The means to share common hardware resources has been a fertile area for research and development for decades. From the days of early time sharing, operating systems have been used as platforms for sharing by scheduling jobs, providing resource mediation and virtualization, and later providing access control. As data communication spread, the desire to connect and share data became a strong driver for sharing systems, placing focus on safe sharing.

Shared systems introduce inefficiencies such as slower performance; they also create serious security concerns. Security cannot simply be a question of retaining control of the system in a technical sense, but must also address moral and legal requirements for separation and privacy.

HOW TO SHARE SYSTEMS

In this section we will discuss a number of existing systems generally representative of broad classes of similar systems allowing for resource sharing, as well as techniques that accomplish separation. We will first discuss a classic control-free environment and then consider the Unix process model, Unix discretionary access control, full virtual machine models, virtual machine-like execution environments, and classic trusted operating systems.

STRAWMAN

The earliest types of system sharing consisted of primitive multiprogramming kernels that time-sliced among a number of processes or programs.[1] In these "strawman" systems, no separation techniques are applied and the hardware may not even offer facilities to provide qualified separation. All users/programs of the system have access to all resources on the system (memory, files, etc.). Accidental or intentional malicious behavior cannot be prevented or reliably recorded. It can be argued that both the administrative and user overhead from this separation model are almost nonexistent, since little can be done. But this is like saying that castor oil works against the common cold: you dare not cough.

Today these systems survive mainly in the embedded market where response-time requirements, resource constraints, and tightly controlled software integration make this approach a viable if painful methodology.

UNIX PROCESSES

Unix processes provide a basic virtual machine primitive, running each instance of an application in its own sandbox. Although not a complete emulation of the actual hardware environment, the process provides basic isolation and consistent interfaces. Each application instance operates with its own address space, copy of registers, and independent references to common system resources. The kernel provides abstracted access, virtualization, and synchronization primitives for system resources, permitting controlled and intentional communication.

The Unix process model strongly resembles those found on many past and current systems, including Windows NT, BSD (Berkeley Software Distribution) and

Linux, and Mac OS X. Through the Posix standards, it has become the yardstick for application separation.

UNIX MULTI-USER SECURITY

In addition to process protections, Unix systems offer multi-user protection through a simple model of:

- Authenticated users
- Administrator-managed group system
- Simple discretionary access control lists for files/IPC objects
- Simple privilege model permitting the administrator to override protections and perform system management.

Properly configured, users, groups, and permissions may be used to provide integrity, confidentiality, and, to a lesser extent, availability protections among mutually untrusting users. The model scales poorly, however, requiring administrators to maintain the groups and requiring users to monitor and configure protections on all objects they own. The model lends itself well to discretionary protection in which users control access to objects they own, but poorly to mandatory protection, where the administrator requires controls that cannot be bypassed.

As with the Unix process model, many other systems have similar designs with similar benefits and pitfalls.

VMWARE

IBM's VM/SP and, more recently, VMware provide a classic form of system virtualization and containment by emulating an entire hardware platform and the separation offered by independent computers. The environment allocates physical (memory, CPUs, or CPU time-slicing) and virtual (memory backed to swap space, etc.) resources in a manner similar to operating-system resource allocation. The implementation efficiency of this model varies according to several important factors. For example, is the virtual hardware environment simply subsetting existing hardware into partitions—or is it multiplexing resources and performing more extensive virtualization? Also, does the hardware platform support nesting and easy virtualization in its instruction set or other features—or is substantial software support required (e.g., instruction rewriting)?

Resource management is an important consideration in the performance and cost of the model. This model, however, offers significant improvements in hardware utilization only if resources can be multiplexed and/or overcommitted.

From a sharing point of view, the advantage of this model is its ability to run multiple instances on a single hardware platform, which can offer firewall partitioning that would be difficult to implement using, for example, the Unix security model.

The degree to which communication between partitions is restricted has benefits and costs. Limiting communication to explicit channels such as virtualized or real networking provides well-understood existing mechanisms for control (packet filtering, proxies, and firewalls). It also reduces the efficiency and accessibility of communication, however. VMware users will be familiar with this phenomenon, as network file sharing must be used to move files between the host and guest environments.

JAVA VIRTUAL MACHINE

The JVM (Java Virtual Machine) and JRE (Java Runtime Environment) provide a variety of security and separation services to support the simultaneous execution of mutually untrusting code. Different programs in the same virtual machine run within the same address space and are protected using a combination of namespace protections (a type-safe environment forbidding direct pointer manipulation) and access control (capabilities to reference classes and objects controlled by policy).

The JVM model presents a mature and fascinating example of trade-offs in separation as it balances the desire for high portability and safety with the need for tight and fast communication between mutually untrusting code in execution. As a result of the tight integration of components in the same address space, interaction is facilitated in a manner not possible in virtual machines. The complexity of this approach, however, adds cost to implementing and using the system, as well as risk from incorrect implementation and use. Similar trade-offs are made in the balance of explicit access control for naming classes, followed by the use of a capabilities model that avoids the repetition of expensive policy calculations and cryptographic checksums.

CLASSIC TRUSTED SYSTEMS

Trusted operating systems attempt to address the issue of controlled separation through the introduction of mandatory security models, generally based on extensions to commercial Unix products. These policies complement Unix DAC (discretionary access control) by introducing protections set by the administrator and enforced for all users. Early trusted systems relied solely on MLS (multilevel security), which controlled the flow of information through a system to protect against unauthorized accidental or malicious sharing of sensitive information. MLS

Building Systems to be Shared Securely

assigns clearances to users and controls the interaction between users and various objects (such as files, IPC, and the network stack) in the system based on classifications on those objects.

In contrast to the discretionary nature of Unix access control lists, the MLS policy permits security administrators to reason about and control the flow of information in the system. With early requirements for trusted systems driven solely by military customers, trusted systems have slowly been expanded to include additional policies such as system integrity policies, role-based access control, and rule-based policies such as domain and type enforcement (http://www.networkassociates.com/us/nailabs/research_projects/secure_execution/dte_overview.asp) and type enforcement (http://www.nsa.gov/selinux/).

Trusted systems frequently offer separation based on combining two elements: a global policy, and either implicit or explicit security labels on subjects (processes) and objects. These security labels may hold clearance or classification data, domain or type information, rule sets, or other policy-specific content.

For sites willing to accept the trade-off of higher overhead of comprehensive labeling of every object in the system and implementation of the protection policy, trusted systems are a powerful tool for controlling the flow of information across security boundaries in a highly trustworthy manner.

THINKING ABOUT MENTAL MODELS FOR SEPARATION

Simply providing the technical means to accomplish separation is insufficient to produce a usable system: a philosophy of separation is necessary so that administrators can map their security requirements onto system primitives.

Strawman systems offer no separation and therefore little administrative complexity. Complete partitioning systems, such as VM/SP and VMware, that create isolation from strawman (or more complex) systems offer a comprehensible model, similar to introducing more independent systems. All resources are uniquely associated with a partition, communication channels are well defined, and protection follows naturally from well-understood resource-allocation processes.

For systems offering more fine-grained solutions, such as discretionary or mandatory access control, or permeable protections such as namespace subsetting, it is necessary to consider both the scope and the complexity of the administrative and auditing workload.

The *scope* of protection refers to the concept of protection that is provided by a protection mechanism. Even in systems with many controls, such as trusted systems, the scope of protection is usefully narrowed by a sensible mapping of macroscopic "business rules" to control points to constrain the degrees of freedom.

The *complexity* of protection corresponds to the quantity of work required to create, maintain, or audit the level and correctness of separation—and corresponds primarily to whether the addition of new controls and elements results in a simple accumulation of work, or a combinatorial increase. Although trusted systems offer relatively few policy choices, they require substantial administration as a result of labels on each system element.

Well-designed systems will minimize the scope and complexity in order to avoid any administration difficulty. To understand the impact of both poor scoping and high complexity, we consider the BSD *securelevel* facility. Securelevel is based on a simple notion of scope: as the securelevel is raised, privileges available in the system are reduced to limit the effect of compromise. As the system evolved, however, securelevel became a catch-all for a wide variety of policy controls, resulting in hundreds of poorly documented control points being affected by a single variable. Securelevel offers poor scoping by virtue of providing an inconsistent model of protection that cannot easily be applied as a tool to accomplish specific goals. It offers high complexity as a result of the many elements of their behavior.

It is important when designing separation facilities for operating systems to consider the total scope and abilities of the control points, as well as the ability of the human mind to comprehend their combined effect.

SEPARATION CASE STUDY: FREEBSD JAIL

The FreeBSD operating system is a widely used, production-quality operating system derived from BSD, developed at the University of California, Berkeley. In the ISP (Internet service provider) environment, FreeBSD finds primary use as a scalable hosting platform. The jail facility provides a lightweight partitioning system, forming the basis for a variety of virtual server environments—and presents an interesting example of a separation scheme introduced into an open source operating system.

In 1999 we added a partitioning facility to FreeBSD called jail(2).[2] It reuses the chroot(2) implementation, but prevents well-documented means to escape chroot confinement. Jail offers semi-permeable partitioning of the file system, process, and networking namespaces, and removes all super-user privileges that would affect objects not entirely inside the jail. In a Web-hosting environment, this is functionally similar to a full partitioning solution such as VMware, but it selectively abandons both the costs and benefits of complete flexibility—resulting in significantly lower overhead and performance impact.

Soon after jail(2) was made available, users of FreeBSD started to find novel uses for jails, many of which exploited the semi-permeable nature of the partitioning. Jail permits an administrator outside of the jail to inspect its full contents. If a service in the jail is compromised, the activities of the attacker will be constrained to the jail, but also will be fully visible to the administrator, at minimal risk to the administrator. This model offers substantially enhanced monitoring over dedicated hardware, or even fully virtual machines, that offer little reliable insight into their operation once compromised. Constraints on direct access to hardware, the kernel, and administrative functions greatly constrain the attacker in employing the normal suite of "rootkit" modifications that would normally prevent proper monitoring. Semi-permeable protections do, however, come with increased risk if information or control can flow out of the sandbox because of an administrator mistake.

HYBRID SECURITY AND SEPARATION MODELS

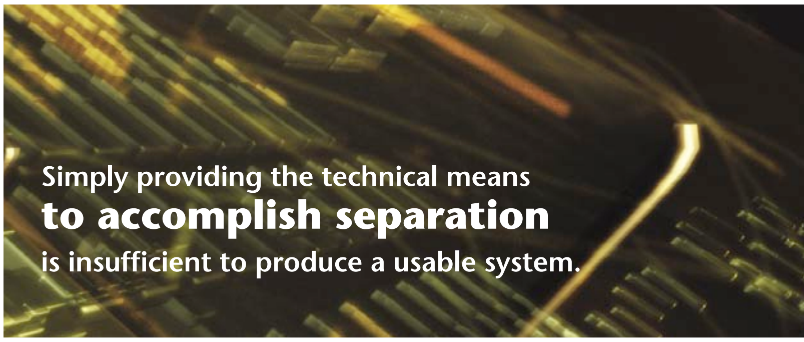
Hybrid separation models combine elements of each of the approaches that we discussed earlier to create a separation offering the correct balance of security, performance, cost, and complexity for the consumer. The FreeBSD Jail approach adopts a variety of elements—including the Unix process and access control model, namespace subsetting based on the chroot(2) facility, and elements of the virtual machine approach—to provide a high-performance virtual server environment. It also, however, adopts a pragmatic approach to combining these components at a low cost. The jail model substitutes namespace limits for the labeling of traditional strong security approaches, resulting in a simple implementation that meets the needs of a specific audience.

SOFTWARE REUSE

Complete virtual machine environments can require a substantial investment in new software to provide proper emulation, especially for complex architectures such as i386. The complete separation provided between guest environments requires running many instances of the operating system. Hybrid forms of partitioning can offer both reduced investment in software when reusing existing software extensively, as well as reduced investment in hardware when reducing redundant instances. The jail facility reuses a single kernel for all jails and takes advantage of existing operating system features to construct its protection policy—including FreeBSD's notion of privilege, file-system namespaces, and existing virtualization in the network stack.[3]

CODE COMPLEXITY

Software reuse as found in jail, or software modularity and object orientation as found in trusted operating systems, requires the existing software system to be modular and extensible. Code that has been built from the ground up as an object-oriented system lends itself to virtualization, because virtualization frequently consists simply of instantiating multiple instances of the object. Although the file system lends itself to subsetting and virtualization in FreeBSD, the network stack offered a much more



Simply providing the technical means **to accomplish separation** is insufficient to produce a usable system.

difficult target. Work to introduce virtualization and subsetting of the stack required a substantial investment of developer time to eliminate the use of global variables and data structures that prevented multiple instantiation—and to define mechanisms by which the instances could interact. Building from scratch with multiple instantiation in mind simplifies this task dramatically.

CAPABILITIES AND NAMING RIGHTS

The "capabilities" approach performs an access check upon first access, and then provides a reference to an object based on that check, which may be used indefi-

Building Systems to be Shared Securely

nitely in the future. A widely used example of the capability model is the Unix file descriptor. This permits the continued use of a file-system object following an initial lookup and access check. The model emphasizes performance and simpler application error handling—at the cost of revocation—and relies on the safety of local and global naming schemes.

Controlling access to system objects based on namespace is a related approach. By preventing a process from naming an object through namespace thinning, the system can prevent access to the object. Using policy to provide namespace limits can be a powerful mechanism for controlling access; it offers a relatively simple implementation and user-comprehensible behavior. In the world of Unix, the classic namespace limitation occurs via `chroot()`, which limits access to a subset of the namespace, thus preventing access to any objects that cannot be named.

Combining capabilities and namespace management avoids extensive explicit access control infrastructure, especially when namespace subsetting is facilitated by namespace structure—permitting a specific subject to see a masked view of the world. As the capability model dictates that operations may be performed only on an object that can be named, the namespace approach can be used to provide strong limits on object access.

In the JVM, type safety implements a simple capability scheme, in which executing code may name only those objects that it has been granted the right to access using a reference. All code executing in the virtual machine is loaded by a ClassLoader instance, which performs explicit access-control checks before granting a reference to a previously unreferenced class, combining explicit mandatory or discretionary access control with a capability scheme.

HIERARCHICAL NAMESPACES: SUBSETTING AND PROTECTIONS

Software systems frequently make use of hierarchical namespaces to organize information. Examples include file systems, MIBs (management information bases), and DNS (domain name system). Hierarchies are also a valuable tool for introducing separation using namespace subsetting, in which subjects in the system perform lookups in the namespace relative to a specific root in the namespace. In the `jail` model, file-system namespace subsetting using `chroot(2)` permits object naming to be constrained, a technique that may be applied easily in other systems with hierarchical namespaces.

Hierarchical namespaces also improve efficiency by permitting protections to be enforced recursively so that "container objects" protect objects placed in them. The canonical example is the "private" directory that helps users apply protections by simplifying the common-case act of protection to a namespace operation.

Hierarchical namespaces support other security models as well—including discretionary and mandatory protections—as they facilitate endowing sections of the namespace itself with trust. In a flat namespace, protections on the namespace are frequently poorly or inflexibly defined, permitting races to occur in establishing a service or object by name, or requiring an explicit allocation and the use of privilege to establish new names in the namespace.

The notion of a controlled and hierarchical namespace is particularly valuable in the JRE, in which the class namespace not only organizes the class hierarchy, but also may assist in determining policy for executing code loaded from parts of the namespace.

REVOCATION

A critical issue in security systems relates to the revocation of access to objects. Once access has been granted—and the security policy is modified to revoke this access—to what extent must the system identify and remove capabilities representing that right? In a strict partition scheme, in which communication between subjects (virtual machines) occurs only through a constrained set of interfaces, revocation will play little or no role. For labeled security systems, however, the issue of revocation is more important with discretionary policies such as Unix permissions and other permeable or hybrid security schemes.

For MLS or Unix discretionary protections, this issue is complicated by the cost of revocation. It's easy to block fresh access to an object by denying the granting of additional capabilities to the object. Existing capabilities may be scattered throughout the system, however, sometimes in ways that involve complex subsystem interaction. In such environments, the benefits of revocation will frequently be outweighed by the performance and implementation costs of bookkeeping and revocation, and will be omitted from the system in the name of expediency.

In the jail system, revocation occurs only during startup of the jail when a process must transition from the host environment to the jail environment—and may hold existing capabilities referring to objects outside the jail, which might be used to attack the jail containment. This circumstance is generally handled through the careful authoring of jail management tools to release capabilities prior to launching any untrusted code. This process has risks, however, and presents a challenge when any system requiring revocation must be addressed.

Similar challenges are faced by Unix multi-user security in which the notion of a user is defined purely in user-space libraries and applications (whereas the notion of a credential used for access control is defined in the kernel). Removing the user from the user database is insufficient to revoke the privileges of existing processes running on behalf of the user, requiring more expensive and failure-prone approaches to killing off user processes, deleting files, and removing services that they own.

INFORMATION FLOW

The specific security concerns to be addressed with partitioning systems (integrity, confidentiality, and availability) may often be reduced to concerns about the flow of data and control. Separation schemes address this differently: hard information flow controls implemented by the mandatory Biba and MLS policies do this explicitly, whereas virtual system tools do so implicitly. By understanding information flow as a specific concern, systems can make informed trade-offs. Systems might tolerate leakage of configuration information as long as it remains immutable, but they might not tolerate the unauthorized flow of confidential data. Techniques such as namespace subsetting provide powerful controls over the flow of information by placing efficiently expressed bounds on information access.

AVOID THE HARD PROBLEMS

Despite careful consideration and a broad set of tools for separation, there will occasionally be systems that simply do not lend themselves to strong separation without extensive virtualization. One such system is System V IPC, which uses flat namespaces combined with discretionary access controls. Protections on individual IPC objects are well understood and can generally be extended to provide mandatory protection. The namespace itself, however, poses a substantial challenge as it is nonhierarchical, and controls on the namespace are difficult to introduce while maintaining flexibility and safety. Virtualization of the namespace would involve complex modifications to the implementation and management tools.

In the jail system on FreeBSD, we opted simply to disable access to the IPC primitives to avoid the cost of introducing new namespaces for each system partition, while avoiding the risks of providing no controls. For the application environments targeted in the jail work, this has held up remarkably well, although we expect that it will be necessary to address this issue in the future. Avoiding hard problems to address a specific environment can be a powerful approach for introducing effective separation. A similar trade-off is made for administrative interfaces through jail, as system subsets frequently do not have to have the full capabilities of the total system.

RESOURCE SHARING AND SCHEDULING

One of the most difficult tasks in introducing separation is resource allocation and scheduling. In separated environments, resources come from a common pool. For operating systems, these resources typically consist of concrete objects or services such as CPU time, memory, network bandwidth, and disk storage. Less concrete resources such as latency to schedule services and other notions of fairness may be important in some environments. On most Unix systems, the primary model is timesharing. Resources should be allocated to balance concerns of throughput and response time based on the job and the expectation of resource contention. On other systems, the trade-offs may look different—ranging from cooperative multitasking to hard realtime systems that perform careful resource measurement and allocation to prevent overcommits.

The balancing act is complicated by implementation complexity. The control of resource allocation requires bookkeeping and enforcement. Both can become difficult to provide in more tightly integrated environments, where resources may be referenced by a changing set of separated components. In virtual environments permitting files to be shared among multiple sandboxes, identifying which sandbox to “bill” for the cost of the file may be difficult. Likewise, in the JVM, allocated memory may flow between security domains.

ADMINISTRATIVE COMPLEXITY

Administrative complexity is a critical consideration in the design of any system in which security will be important. Experience suggests that systems that are hard to manage securely won’t be used correctly—or at all.

Building Systems to be Shared Securely

Whereas some security systems, such as trusted operating-system access-control policies, are inherently complex because of the desired security results, identifying the right trade-off will generally be a question of minimalism: what is the simplest security solution that meets the requirements of the user?

Specific administrative complexity goals include the following:

- Avoid the costs of administration increasing linearly (or worse) with the number of objects being protected. Security policies should require more administration only as the goals of the policy become more complex.
- Avoid attributes managed by users, as each configuration setting or attribute offers the opportunity for user error, which must be minimized.
- Permit security policies to be expressed in the language of security goals, rather than in fine-grained primitives of the implementation. Some systems (most frequently, firewalls) require an expert knowledge of the implementation of the system—and make it difficult to convert policy requirements into practice.
- Build the system so that it's easy to determine if the protections are working correctly, and so that the implications of policy are clear and testable. The more elements that must be administered, the harder it may be to audit those settings and determine the overall system behavior.
- Avoid providing tunable resource limits instead of resource allocation policies as administrators will often be unable to select bounds in an informed manner. Also, default bounds and tunable bounds will rapidly become stale as hardware and application platforms evolve—potentially leading to damaging behavior.

HOW TO APPROACH NEW SYSTEMS

Throughout this article, we have discussed trade-offs in system design that facilitate (or impede) introducing separation efficiently and securely. In the end, the most important lesson when designing and implementing new systems is that this consideration of trade-offs must be explicit.

You can take the following concrete steps to prepare a system for use in shared environments, regardless of the security or separation model eventually adopted:

1. Object-oriented implementation works. The simplest systems to virtualize are ones in which the state of the system is encapsulated in a class that can be instantiated many times with little additional effort.

2. Avoid incestuousness in the implementation and adopt component-oriented design. By accomplishing separation in the implementation of different software components, you have increased leeway to adopt different separation approaches for the components, increasing flexibility. If a subsystem proves unnecessary in a particular environment, it can be disabled easily, thus lowering the workload.

3. Think about namespaces early. Effective use of hierarchical and protected namespaces permits trust to be placed in the namespace, and for techniques such as namespace subsetting and thinning to accomplish separation at a low cost.

4. Minimize the need for attribute and policy management by providing primitives for which policy can be expressed easily and in broad terms. The more fine-grained control the primitives permit, the more aspects of the primitives may require control in the future.

When introducing separation and protection, there are strong arguments for adopting a hybrid design that includes elements of many different separation approaches. This will frequently allow avoiding 90 percent of the costs while accomplishing 90 percent of the benefits for a specific environment:

- Avoid hard resource allocation requirements as they require extensive bookkeeping and are not always needed. This also improves scalability by increasing the level of resource sharing.
- Use lightweight namespace-based approaches to protection. They are cheap and easy to implement, and make sense to administrators.
- Avoid requiring extensive administration and management; specifically avoid approaches where the number of security labels or access control lists scales poorly. Allow users to make an explicit trade-off between security complexity and administrative complexity, rather than system utilization and administrative complexity.

COMPATIBILITY IS KEY

Attempts to introduce separation into existing systems almost always run up against a key concern: compatibility with the existing system. In 2000, Rob Pike gave a provocative talk about the state of operating-system research as a field in decline:[4]

"To be a viable computer system, one must honor a huge list of large, and often changing, standards: TCP/IP, HTTP, HTML, XML, Corba, Unicode, Posix, NFS, SMB, MIME, POP, IMAP, X, ...

"A huge amount of work, but if you don't honor the standards, you're marginalized.

"Estimate that 90-95 percent of the work in Plan 9 was directly or indirectly to honor externally imposed standards."

This is compounded by ad hoc, de facto, and design-by-committee interfaces that frequently prevent the integration of security features by precluding them. Imperfect partitioning offers some relief in this area, as it allows processes to be treated differently in certain aspects without impeding their interaction with other processes.

Sequent and Pyramid pioneered "dual-universe" Unix, which emulated both major variants of the Unix operating system on a process-by-process basis. FreeBSD follows in this tradition with the "Linux-o-lator," which is able to execute binary programs compiled for the Linux operating system—providing these processes with the environment of a real Linux system while appearing to be a native FreeBSD process. Systems such as Carnegie Mellon University's Mach (http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html) and later IBM's K42 (http://www.research.ibm.com/K42/) demonstrate that it's quite feasible to redesign the internals of an operating system while maintaining an application compatibility layer that meets most requirements.

CONCLUSION

Effective separation remains critical to the deployment of shared computing environments, driven by a desire to increase communication and lower investment in hardware. We have explored common implementations of separation, ranging from complete virtual machines to trusted operating systems, all tied together by their goals of providing for the sharing of hardware by mutually untrusting parties. By combining the diverse supporting technologies for separation, we have illustrated the use of hybrid approaches that offer many of the benefits of each underlying primitive—but with far less implementation cost and complexity.

Not surprisingly, tightly integrated separation technologies rely on consistent and clean implementation of services. Systems that are built with protection in mind—even if not the protection of eventual interest—will generally be easier to modify to add new protections. The mantras of careful software design, including modularity, object orientation, and intentional design, each support the integration of advanced security techniques and are often the best means by which to prepare a system for new security services. Since systems are inevitably used in environments that require communication and resource sharing, regardless of the expectations of the software developer, planning for security from inception is critical to allowing the system to be used safely. Q

REFERENCES

1. Hansen, P. B. RC 4000 Software: Multiprogramming System. RCSL No. 55-D140, Regnecentralen, Copenhagen, Denmark, 1969.
2. Kamp, P.-H., and Watson, R. Jails: Defining the Omnipotent Root. Sane 2000, Maastricht, The Netherlands.
3. See Reference 2.
4. Pike, R. System Software Research is Irrelevant. Bell Labs, Lucent Technologies, 2000; http://cm.bell-labs.com/cm/cs/who/rob/utah2000.pdf.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

POUL-HENNING KAMP (phk@FreeBSD.org) is one of the primary developers of the FreeBSD operating system, which he has worked on from the very beginning. He is widely unknown for his MD5-based password scrambler, which protects the passwords on Cisco routers, Juniper routers, and Linux and BSD systems. Some people have noticed that he wrote a memory allocator, a device file system, and a disk encryption method that is actually usable. Kamp lives in Denmark with his wife, his son, his daughter, about a dozen FreeBSD computers, and one of the world's most precise NTP (Network Time Protocol) clocks. He makes a living as an independent contractor doing all sorts of stuff with computers and networks.

ROBERT WATSON is a research scientist and DARPA principal investigator in the Host Intrusion Protection (HIP) Research Group at McAfee Research. He has led a variety of research and development projects relating to network and operating system security, ranging from product development to research into the security implications of active networking. His other research interests include access control, audit, tamper-resistant hardware, distributed file systems, network stack performance optimization and hardening, trusted operating systems, and windowing system security. Watson is a FreeBSD Core Team developer and founder of the TrustedBSD Project.