SHAYA POTTER AND JASON NIEH

breaking the ties that bind

APPLICATION ISOLATION AND MIGRATION

Shaya Potter is a Ph.D. student in the Computer Science Department at Columbia University. His research focuses on virtualization and process migration technologies to improve the way users and administrators use their computers.

spotter@cs.columbia.edu

Jason Nieh is an associate professor of computer science at Columbia University and director of Columbia's Network Computing Laboratory. He is also the technical advisor for nine states on the Microsoft Antitrust Settlement. His current research interests are in systems, including operating systems, thin-client computing, utility computing, Web and multimedia systems, and performance evaluation.

nieh@cs.columbia.edu

AS COMPUTERS BECOME MORE ubiquitous in large corporate, government, and academic organizations, the cost of owning and maintaining them is becoming unmanageable. Computers are increasingly networked, which only complicates the management problem given the myriad of viruses and other attacks commonplace in today's networks. Security problems can wreak havoc on an organization's computing infrastructure. To prevent this, software vendors frequently release patches that can be applied to address security and maintenance issues that have been discovered. This becomes a management nightmare for administrators who take care of large sets of machines.

Even when software security or maintenance updates are applied, they commonly result in system disruptions. Patching an operating system can cause the entire system to be down for extended periods, and a system administrator who chooses to fix an OS security problem immediately risks upsetting his users because of loss of data. Therefore, a system administrator must schedule downtime in advance and in cooperation with all the users, leaving the computer vulnerable until repaired. If the operating system is patched successfully, the system downtime may be limited to just a few minutes during the reboot. Even then, users are forced to incur additional inconvenience and delays in restarting applications and in attempting to restore their sessions to the state they were in before shutdown.

AutoPod is a system we have built at Columbia University that provides an easy-to-use autonomic infrastructure for operating system self-maintenance. AutoPod uniquely enables unscheduled operating system updates of commodity operating systems while preserving application service availability during system maintenance [1]. AutoPod provides this functionality without modifying, recompiling, or relinking applications or operating system kernels. This is accomplished by combining three key mechanisms: a lightweight virtual machine isolation abstraction that can be used at the granularity of individual applications; a checkpoint-restart mechanism that operates across operating system versions with different security and maintenance patches; and an autonomous system status service that monitors for system faults and security updates.

AutoPod is based on a virtual machine abstraction called a pod (PrOcess Domain) [2, 3]. A pod looks just like a regular machine and provides the same application interface as the underlying operating system, but it also provides a complete secure virtual machine abstraction with heterogeneous migration functionality. Pods can be used to run any application, privileged or otherwise, without modifying, recompiling, or relinking applications. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment. Unlike a traditional operating system, the pod abstraction provides a self-contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted.

A pod does not run an operating system instance but, rather, offers a virtualized machine environment by providing a host-independent virtualized view of the underlying host operating system. This is done by giving each pod its own virtual private namespace. All operating system resources are only accessible to processes within a pod through the pod's virtual private namespace.

A pod namespace is private in that only processes within the pod can see the namespace. It is private in that it masks out resources that are not contained within the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional IPC mechanisms. Processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory or signals.

A pod namespace is virtual in that all operating system resources, including processes, user information, files, and devices, are accessed through virtual identifiers within a pod. These virtual identifiers are distinct from host-dependent resource identifiers used by the operating system. Since the pod namespace is distinct from the host's operating system namespace, the pod namespace preserves resource-naming consistency even if the underlying operating system namespace changes, as is the case in migrating processes from one machine to another.

The pod private virtual namespace enables secure isolation of applications by providing complete mediation to operating system resources. Pods can restrict what operating system resources are accessible within a pod by not providing identifiers to such resources within its namespace. A pod only needs to provide access to resources that are needed for running those processes within the pod. It does not need to provide access to all resources to support a complete operating system environment. An administrator can configure a pod in the same way she configures and installs applications on a regular machine. Pods enforce secure isolation to prevent exploited pods from being used to attack the underlying host or other pods on the system. Similarly, the secure isolation allows one to run multiple pods from different organizations, with different sets of users and administrators on a single host, while retaining the semantic of multiple distinct and individually managed machines.

Many ways have been proposed for isolating applications on a single system. These systems, such as VMware's and Xen's virtual machine technology, Solaris's Zone virtual servers, and FreeBSD's jails, differ from AutoPod in a fundamental way. They restrict a running process to a single kernel instance. AutoPod is the only system that enables an administrator to checkpoint a generic set of processes running on one kernel with known security problems and restart those processes on a machine running an updated kernel. By providing each pod with its own virtual private namespace, AutoPod has advantages over systems that just prevent applications from making use of specific global resources. Those systems only restrict what a process can do to the namespace, instead of providing each pod with its own complete virtual private namespace to work with. AutoPod provides isolation without requiring multiple operating system instances, and implements all of its functionality without any invasive kernel support.

AutoPod provides this functionality using a virtualization architecture that operates between applications and the operating system, without requiring any changes to applications or the operating system kernel. This virtualization layer is used to translate between the pod namespaces and the underlying host operating system namespace. It protects the host operating system from dangerous privileged operations that might be performed by processes running within pods, and it protects those processes from processes outside of the pods.

Pods are supported using virtualization mechanisms that translate between pod virtual resource identifiers and operating system resource identifiers. Every resource that a process in a pod accesses is through a *virtual private name* that corresponds to an operating system resource identified by a *physical name*. When an operating system resource is created for a process in a pod, such as with process or IPC key creation, instead of returning the corresponding physical name to the process, the pod virtualization layer catches the physical name value and returns a virtual private name to the process. Similarly, any time a process passes a virtual private name to the operating system, the virtualization layer catches it and replaces it with the appropriate physical name. The key pod virtual-

ization mechanisms used are a system call interposition mechanism and the chroot utility, with file system stacking to provide each pod with its own file system namespace, which can be separate from the regular host file system.

Pod virtualization uses system call interposition to virtualize operating system resources, including process identifiers, keys, and identifiers for IPC mechanisms, such as semaphores, shared memory, message queues, and network addresses. System call interposition wraps existing system calls to check and replace arguments that take virtual names with the corresponding physical names before calling the original system call. Similarly, wrappers are used to capture physical name identifiers that the original system calls return, and return corresponding virtual names to the calling process running inside the pod. The pod's virtual names are maintained consistently as the pod migrates from one machine to another and are remapped appropriately to underlying physical names, which may change as a result of migration.

To enable processes within a pod to run with root privilege, AutoPod interposes on select system calls that could allow a privileged process to break the virtualized namespace. By selectively controlling how specific system calls are used, AutoPod is able to enable processes to run with privilege, while preventing them from using that privilege to break out of the pod's context. Specifically, AutoPod disables certain system calls that do not make sense within a pod, drops a process's privileges for other system calls, and filters the arguments for system calls.

Because commodity operating systems are not built to support multiple namespaces, one security issue that pod virtualization must address is that there are many ways to break out of a standard chrooted environment, especially if one allows the chroot system call to be used by processes in a pod. Pod file system virtualization enforces the chrooted environment and ensures that the pod's file system is only accessible to processes within the given pod, by using a simple form of file system stacking to implement a pod-aware barrier directory. The barrier directory provides a file system permission function that denies access to all processes that are running within a pod context, even if they are running as root. By preventing any process within a pod context from accessing it, the processes cannot walk past it. This prevents a process that breaks out of the chroot context—which is simple if one allows root processes and the chroot system call to be used—from gaining access to any files outside of the pod's virtualized file system view.

To support migration across different kernels, AutoPod uses a checkpoint-restart mechanism that employs an intermediate format to represent the state that needs to be saved on checkpoint. On checkpoint, the intermediate format representation is saved and digitally signed to enable the restart process to verify the integrity of the image. Although the internal state that the kernel maintains on behalf of processes can be different across different kernels, the high-level properties of the process are much less likely to change. We capture the state of a process in terms of higher-level semantic information specified in the intermediate format, rather than kernel-specific data in native format, to keep the format portable across different kernels. Open network connections are preserved as a pod moves between computers based on network address virtualization [3, 4].

AutoPod provides an autonomic system status service to control when and where pods are checkpointed and restarted. Many operating system vendors provide their users with the ability to automatically check for system updates and to download and install them when they become available. Examples of these include Microsoft's Windows Update service and the Debian distribution's security repositories. AutoPod monitors these security repositories and determines whether a system reboot is required to install security updates. If so, it checkpoints the pods running on the system and migrates them to other systems to be restarted, ensuring that no state is lost and minimizing application downtime.

We've implemented AutoPod in Linux as a loadable kernel module and user-level utilities. We've used AutoPod to migrate applications across operating system maintenance and security updates as well as across major kernel changes, including Linux 2.4 and 2.6 kernels. Our experiences using AutoPod on a wide range of everyday desktop and server applications demonstrate that it imposes very little virtualization overhead and can provide fast, subsecond checkpoint and restart times [2, 3, 5].

As an example of the benefits of AutoPod and how easy it is to set up and use, let us consider AutoPod in the context of email delivery. Email delivery services such as Exim are often run on the same system as other Internet services, to improve resource utilization and simplify system administration through server consolidation. However, services such as Exim have been easily exploited by the fact that they have access to system resources, such as a shell program, that they do not need to perform their job.

AutoPod can isolate email delivery to provide a significantly higher level of security in light of the many attacks on mail transfer agent vulnerabilities that have occurred. Using AutoPod with Exim, Exim can execute in a resource restricted pod, which isolates email delivery from other services on the system. In particular, the Exim pod can be configured with no shell,

preventing the common buffer overflow exploit of getting the privileged server to execute a local shell. If a fault is discovered in the underlying host machine, the email delivery service can be moved to another system while the original host is patched, preserving the availability of the email service.

Setting up AutoPod to provide the Exim pod on Linux is straightforward and leverages the same skill set and experience system administrators already have on standard Linux systems. AutoPod is started by loading its kernel module into a Linux system and using its user-level utilities to set up and insert processes into a pod.

Creating a pod's file system is the same as creating a chroot environment. Administrators who have experience creating a minimal environment, just containing the application they want to isolate, do not need to do any extra work. However, many administrators do not have such experience and therefore need an easy way to create an environment to run their application in. Debian's debootstrap utility enables a user to quickly set up an environment that's the equivalent of a base Debian installation. An administrator would do a debootstrap stable /pod to install the most recently released Debian system into the directory. While this will also include many packages that are not required by the installation, it provides a small base to work from. An administrator can remove packages, such as the installed mail transfer agent, that are not needed.

To configure Exim, an administrator edits the appropriate configuration files within the /pod/etc/exim4/ directory. To run Exim in a pod, an administrator does mount -o bind /pod /autopod/exim/root to loopback mount the pod directory onto the staging area directory where AutoPod expects it. autopod add exim is used to create a new pod named exim which uses /autopod/exim/root as the root for its file system. Finally, autopod addproc exim /usr/sbin/exim4 is used to start Exim within the pod by executing the program, which is actually located at /autopod/exim/root/usr/sbin/exim4.

To manually reboot the system without killing the processes within this Exim pod, an administrator can first checkpoint the pod to disk by running autopod checkpoint exim -o /exim.pod, which tells AutoPod to checkpoint the processes associated with the exim pod to the file /exim.pod. The system can then be rebooted, potentially with an updated kernel. Once it comes back up, the pod can be restarted from the /exim.pod file by running autopod restart exim -i /exim.pod.

Standard Debian facilities for installing packages can be used for running other services within a pod. Once the base environment is set up, an administrator can run chroot /pod to continue setting it up. By editing the /etc/apt/sources.list file appropriately and running apt-get update, an administrator will be able to install any Debian package into the pod. In the Exim example, Exim does not need to be installed since it is the default MTA and already included in the base Debian installation. If one wanted to install another MTA, such as Sendmail, one could run apt-get install sendmail, which will download Sendmail and all the packages needed to run it. This will work for any service available within Debian. An administrator can also use the dpkg –purge option to remove packages that are not required by a given pod. For instance, in running an Apache Web server in a pod, one could remove the default Exim mail transfer agent, since it is not needed by Apache.

The AutoPod system provides an operating system virtualization layer that decouples process execution from the underlying operating system, by running the process within a pod. Pods provide an easy-to-use lightweight virtual machine abstraction that can securely isolate individual applications without the need to run a full operating system instance in the pod. Furthermore, AutoPod can transparently migrate isolated applications across machines running different operating system kernel versions. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of application services. For more information, see http://www.ncl.cs.columbia.edu/research/migrate/.

REFERENCES

[1] Shaya Potter and Jason Nieh, "AutoPod: Unscheduled System Updates with Zero Data Loss," Abstract in Pro*ceedings of the Second IEEE International Conference on Autonomic Computing (ICAC 2005), Seattle, WA, June 13–16, 2005, pp. 367-368.*

[2] Ricardo Baratto, Shaya Potter, Gong Su, and Jason Nieh, "MobiDesk: Mobile Virtual Desktop Computing," *Proceedings of the 10th Annual ACM International Conference on Mobile Computing and Networking (MobiCom 2004),* Philadelphia, PA, September 26–October 1, 2004, pp. 1–15.

[3] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02),* Boston, MA, December 9–11, 2002, pp. 361–376.

[4] Gong Su, "MOVE: Mobility with Persistent Network Connections," Ph.D. Thesis, Department of Computer Science, Columbia University, October 2004.

[5] Shaya Potter and Jason Nieh, "Reducing Downtime Due to System Maintenance and Upgrades," *Proceedings of the 19th Large Installation System Administration Conference (LISA '05),* San Diego, CA, December 4–9, 2005.