Secure Isolation of Untrusted Legacy Applications

Shaya Potter, Jason Nieh, and Matt Selsky – Columbia University

ABSTRACT

Existing applications often contain security holes that are not patched until after the system has already been compromised. Even when software updates are available, applying them often results in system services being unavailable for some time. This can force administrators to leave system services in an insecure state for extended periods. To address these system security issues, we have developed the PeaPod virtualization layer. The PeaPod virtualization layer provides a group of processes and associated users with two virtualization abstractions, pods and peas. A pod provides an isolated virtualized environment that is decoupled from the underlying operating system instance. A pea provides an easy-to-use least privilege model for fine grain isolation amongst application components that need to interact with one another. As a result, the system easily enables the creation of lightweight environments for privileged program execution that can help with intrusion prevention and containment. Our measurements on real world desktop and server applications demonstrate that the PeaPod virtualization layer imposes little overhead and enables secure isolation of untrusted applications.

Introduction

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 issues that have been discovered. However, software patches need to be applied to be effective. It is not uncommon for systems to continue running unpatched applications long after a security exploit has become well-known [25]. This is especially true of the growing number of server appliances intended for very low-maintenance operation by less-skilled users. Furthermore, by reverse engineering security patches, attackers have been able to release exploits less than a month after the vulnerability is patched [16].

This impacts system administrators, as even with security patches being released, one cannot always apply them in a timely manner. First, many security patches require that the system service being patched be taken off-line, thereby making it unavailable. Patching an operating system can result in the entire system having to be down for some period of time. If a system administrator chooses to fix an operating system security problem immediately, he 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. Furthermore, just because a security patch is released, does not mean it will apply successfully to one's system. If the system service is patched successfully, the system downtime may be limited to just a few minutes during the reboot. However, if the patch is not successful, downtime can extend for many hours while the problem is diagnosed and a solution is found. Therefore, a system administrator will have to delay applying the security patch until one is sure that it will cause only a minimum amount of downtime.

Second, many system services in use today are supplied as *appliances*. Just like one's physical appliances are simple single task machines, computing appliances, be they commercial appliances, such as a TiVo or a NetApp Filer, or a simplified appliance a corporation deploys internally, such as a web or mail server appliance, are simplified single task systems. A primary advantage of computing appliances is that they can be deployed very easily by less-skilled users. However, this can result in them being set up, left running, and forgotten about since they "just work." As with all software systems, they will suffer from bugs, some of which can have large security implications. Since these appliances are meant to be put into use by people who are not skilled in system administration, one can end up deploying a large number of systems that are vulnerable to be taken over and used maliciously, without the owner of the appliance having any knowledge that this has occurred. Today, actively used personal machines are being actively taken over and used as part of large bot-nets without any knowledge of the owners of the machines [6]. In the future where large numbers of computing appliances will be deployed, this problem will become significantly worse.

There are many principles that are used to increase the security of a software system and limit the damage that can occur if security is breached [26]. One of the most important is ensuring that one operates in a *Least Privilege* environment. Least Privilege environments requires that a user or a program only have access to the resources that are required to complete their job. Even if the user or service's environment is exploited, the attacker will be constrained. For a system with many distinct users and uses, designing a least privilege system can prove to be very difficult as many independent application systems can be used in many different and unknown ways. On the other hand, securing a single service, such as a software appliance, is more tractable due to the limited nature of what the service accesses.

A common approach to providing least privilege environment to a single service is a sandbox container environment. Many sandbox container environments have been developed to isolate untrusted applications, however, many of these approaches have suffered from being too complex and too difficult to configure to use in practice, and have often been limited by an inability to work seamlessly with existing system tools and applications. Virtual machine monitors (VMMs) offer a more attractive approach by providing a much easier-to-use isolation model of virtual machines, which look like separate and independent systems apart from the underlying host system. However, because VMMs need to run an entire operating system instance in each virtual machine, the granularity of isolation is very coarse, enabling malicious code in a virtual machine to make use of the entire set of operating system resources. Multiple operating instances also need to be maintained, adding administrative overhead.

A primary problem with a sandbox container that attempts to isolate a single service is that many services are composed of many interdependent programs. Each individual application that makes up the service has their own set of requirements. However, since they will all be run within the same sandbox container, each individual application will end up with access to the superset of resources that are needed by all the programs that make up the service, thereby negating the least privilege principle. One cannot divide the programs into distinct sandbox container environments since many programs are interdependent and expect to work from within a single context.

We present PeaPod, a virtualization layer that provides an easy-to-use abstraction that can be used at the granularity of individual applications. The PeaPod virtualization layer provides virtual machine isolation without the need to run multiple operating system instances. PeaPod further enables fine-grain isolation among application components that may need to interact within a single machine environment. PeaPod provides its functionality without modifying, recompiling, or relinking applications or operating system kernels.

PeaPod combines two key virtualization abstractions in its virtualization layer. First, it leverages the pod (PrOcess Domain) [20, 22] to provide a sandbox container for entire services to run within. A pod is a lightweight environment that mirrors the underlying operating system environment. PeaPod isolates processes in pods from underlying system by associating virtual identifiers with operating system resources and only allowing access to resources that are made available within the pod virtualized namespace. Since the pod virtualization layer provides a virtual machine like environment, it also defines its own set of users, which can be distinct from those supported by the underlying system. Since it does not run an operating system instance, a pod prevents malicious code from making use of an entire set of operating system resources. Second, it introduces peas (Protection and Encapsulation Abstraction). A pea is an easy-to-use least privilege mechanism that enables further isolation among application components that need to share limited system resources within a pod. It can prevent compromised application components from attacking other components within the same pod. A pea provides a simple resource-based model that restricts access to other processes, IPC, file system, and network resources within a pod.

PeaPod improves upon previous approaches by not requiring any operating system modifications, as well as avoiding the *time of check, time of use* race conditions that affect many of them [31]. For instance, unlike other approaches that perform file system security checks at the system call level and therefore do not check the actual file system object that the operating system uses, PeaPod leverages stackable file system to integrate directly into the kernel's file system security framework. PeaPod is designed to avoid the time of check, time of use race conditions that affect previous approaches by performing all file system security checks within the regular file system security paths and on the same file system objects that the kernel itself uses.

This paper describes how the PeaPod system can isolate applications to limit their ability to attack a system. The next section describes the PeaPod's virtualization abstractions in further detail followed by the virtualization architecture to support PeaPod. The next two sections provide a security analysis of the PeaPod system as well as examples of how to use PeaPod. Then the experimental results evaluating the overhead associated with PeaPod and measures the system performance of providing secure isolation for several application scenarios are presented followed by related work. Finally, we present some concluding remarks.

PeaPod Model

The PeaPod model is based on a virtualization abstraction called a pod (PrOcess Domain). A pod looks just like a regular machine and provides the same application interface as the underlying operating system. Pods can be used to run any application, privileged or otherwise, without modifying, recompiling, or relinking applications. This is essential for both easy-of-use and protection of the underlying system, since applications not executing in a pod offer an opportunity to attack the system. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment.

A pod does not run an operating system instance, it instead provides a virtualized machine environment by providing a host-independent virtualized view of the underlying host operating system. This is done by providing each pod with its own private, virtual namespace. All operating system resources are only accessible to processes within a pod through the pod's private, virtual 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. Other 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. Instead, processes outside the pod can only interact with processes inside the pod using network communication and shared files that are normally used to support process communication across machines.

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. The pod virtual namespace provides a host-independent view of the system by using virtual identifiers that remain consistent throughout the life of a process in the pod, regardless of whether the pod moves from one system 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 simply 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 one configures and installs applications on a regular machine.

For example, if one had a web server that just serves static content, one can easily setup a web server pod to only contain the files the web server needs to run and the content it wants to serve. The web server pod could have its own IP address, decoupling its network presence from the underlying system. It could also limit network access to client-initiated connections. If the web server application gets compromised, the pod limits the ability of an attacker to further harm the system since the only resources he has access to are the ones explicitly needed by the service. Furthermore, there is no need to carefully disable other network services commonly enabled by the operating system that might be compromised within the pod since there is no operating system running in the pod.

Pods can be used in conjunction with peas (Protection and Encapsulation Abstraction). While pods separate processes into separate machine environments, a pea can be used in a pod to provide fine-grain isolation among application components that may need to interact within a single machine environment, such as using interprocess communication mechanisms, including signals, shared memory, IPC messages and semaphores, and process forking and execution.

A pea is an abstraction that can contain a group of processes and restrict those processes in interacting with processes outside of the pea, and limit their access to only a subset of system resources. Unlike a pod, which achieves isolation by controlling what resources are located within the namespace, a pea achieves isolation levels by controlling what system resources within a namespace its processes are allowed to access and interact with. For example, a process in a pea can see file system resources and processes available to other peas within a single pod, but can be restricted from accessing them. Unlike processes in separate pods, processes in separate peas in a single pod share the same namespace and can be allowed to interact using traditional interprocess communication mechanisms. Processes can also be allowed to move from one pea to another in the same pod. However, by default processes in separate peas cannot access any resource that is not made available to its pea, be it a process pid, IPC key or file system entry.

Peas can support a wide range of resource restriction policies. By default, processes contained in a pea can only interact with other processes in the same pea. They have no access to other resources, such as file system and network resources or processes outside of the pea. This provides a set of fail safe defaults, as any extra access has to be explicitly allowed by the administrator.

The pea abstraction allows for processes running on the same system to have varying levels of isolation, by running in separate peas. Many peas can be used side by side to provide flexibility in implementing a least privilege system for programs that are composed of multiple components that must work together, but do not all need the same level of privilege. One usage scenario would be to have a severely resource limited pea in which a privileged process executes but allows the process to use traditional UNIX semantics to work with less privileged programs that are in less resource restricted peas.

For example, peas can be used to allow a web server appliance the ability to serve dynamic content via CGI in a more secure manner. Since the web server and the CGI scripts need separate levels of privilege, as well as different resource requirements, they should not have to run within the same security context. By configuring two separate peas for a web service, one for the web server to run within, and a separate for the specific CGI programs it wants to execute, one limits the damage that can occur if a fault is discovered within the web server. If one manages to execute malicious code within the context of the web server, one can only make use of resources that are allocated to the web server's pea, as well as only execute the specific programs that are needed as CGIs. Since the CGI programs will also only run within their specific security context, the ability for malicious code to do harm is severely limited.

Peas and pods together provide secure isolation based on flexible resource restriction for programs as opposed to restricting access based on users. Peas and pods also do not subvert underlying system restrictions based on user permissions, but instead complement such models by offering additional resource control based on the environment in which a program is executed. Instead of allowing programs with root privileges to do anything they want to a system, PeaPod enables a system to control the execution of such programs to limit their ability to harm a system even if they are compromised.

PeaPod Virtualization

To support the PeaPod virtualization abstraction design of secure and isolated namespaces on commodity operating systems, we employ a virtualization architecture that operates between applications and the operating system, without requiring any changes to applications or the operating system kernel. This thin virtualization layer is used to translate between the PeaPod namespaces and the underlying host operating system namespace. It protects the host operating system from dangerous privileged operations that might be performed by processes within the PeaPod, as well as protecting those processes from processes outside of the PeaPod on the host. It also enables program-based resource restriction for file access, device access, network access, root privileges, process interactions, and process transitions among peas.

Pod Virtualization

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 name*, which 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, creates a shadow identifier with a private virtual name that maps to the physical name and returns the private virtual name to the process. Similarly, any time a process passes a virtual name to the operating system, the virtualization layer catches it, and replaces it with the appropriate physical name. The key pod virtualization 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 that can be separate from the regular host file system.

Pods can take advantage of the regular user identifier (UID) security model to support multiple security domains on the same system running on the same operating system kernel. For example, since each pod can have its own private file system, each pod can have its own /etc/passwd file that determines its list of users and their corresponding UIDs. Since the pod file system is separate from the host file system, a process running in the pod is effectively running in a separate security domain from another process with the same UID that is running directly on the host system. Although both processes have the same UID, each process is only allowed to access files in its own file system namespace. Similarly, multiple pods can have processes running on the same system with the same UID, but each pod effectively provides a separate security domain since the pod file systems are separate from one another. Since each pod provides a separate security domain, it needs to be viewed as if it is a distinct machine. For instance, if two physical machines share a writable file system, an attacker could leverage flaws in one machine to get programs on the shared file system that can be used to exploit the second one. While there is value in sharing file system data between pods, one has to use the same care in verifying the shared file system data with multiple pods as one would with multiple independent machines.

Because the root UID 0 is privileged and treated specially by the operating system kernel, pod virtualization also treat UID 0 processes inside of a pod in a special way to prevent them from breaking the pod abstraction, accessing resources outside of the pod, and causing harm to the host system. While a pod can be configured for administrative reasons to allow full privileged access to the underlying system, we focus on the case of pods for running application services that do not need to be used in this manner. Pods do not disallow UID 0 processes, which would limit the range of application services that could be run inside pods. Instead, pods provide restrictions on such processes to ensure that they function correctly inside of pods [22].

While a process is running in user space, the UID it runs as does not have any effect. Its UID only matters when it tries to access the underlying kernel via one of the kernel entry points, namely devices and system calls. Since a pod already provides a virtual file system that includes a virtual /dev with a limited set of secure devices, the device entry point is already secured. The only system calls of concern are those that could allow a root process to break the pod abstraction. Only a small number of system calls can be used for this purpose [22].

Pea Virtualization

Peas are supported using virtualization mechanisms that label resources and enforce a simple set of configurable permission rules to impose levels of isolation among process running within a single pod. For example, when a process is created via the fork() and clone() system calls, its process identifier is tagged with the identifier of the pea in which it was created. Pea's leverage the pod's shadow pod process identifier and also place it in the same pea as its parent process. A process's ability to access pod resources is then dictated by the set of access permissions rules associated with its pea. Like pod virtualization, the key pea virtualization mechanisms used are a system call interposition mechanism and file system stacking for file system resources.

Pea virtualization employs system call interposition to wrap existing system calls to enforce restrictions on process interactions by controlling access to process and IPC virtual identifiers. Since each resource is labeled with the pea in which it was created, the system call interposition mechanism checks if the pea labels of the calling process and the resource to be accessed are the same. For example, if a process in one pea would try to send a signal to another process in a separate pea by using the kill system call, the system would return an error value of EPERM, as the process exists, but this process has no permission to signal it. On the other hand, a parent is able to use the wait system call to clean up a terminated child process's state, even if that child process is running within a separate pea since wait does not modify a process by affecting its execution. This is analogous to a regular user being able to list the meta data of a file, such as owner and permission bits, even if the user has no permission to read from or write to the file.

When a new process is created, it executes in the pea security domain of its parent. However, when the process executes a new program, one wants the ability to transition the pea security domain the new program is executing within. Therefore, peas support a single type of pea access transition rule that lets a pea determine how a process can transition from its current pea to another. This transition rule is specified by a program filename and pea identifier. A pea is able to have multiple pea access transition rules of this type. The rule specifies that a process should be moved into the pea specified by the pea identifier if it executes the program specified by the given filename. This is useful when it is desirable to have that new program execution occur in an environment with different resource restrictions. For example, an Apache web server running in a pea may want to execute its CGI child processes in a pea with different restrictions. Pea transitioning is supported by interposing on the exec system call and transitioning peas if the process to be executed matches a pea access transition rule for the current pea. Note that pea access transition rules are one-way transitions that do not enable a process to return to its previous pea unless its new pea explicitly provides for such a transition.

System call interposition is also used to control network access for processes inside the pea. Peas provide two networking access rule types, one to allow processes in the pea to make outgoing network connections on a pod's virtual network adapters, the other to allow processes in the pea to bind to specific ports on the adapter to receive incoming connections. Pea network access rules can allow complete access to a pod network adapter, or only allow access on a per port basis. Since any network access occurs through system calls, peas simply check the options of the networking system call, such as bind and connect, to ensure that it is allowed to perform the specified action.

Pea virtualization employs a set of file system access rules and file systems stacking to provide each pea with its own permission set on top of the pod file system. To provide a least privilege environment, processes should not have access to file system privileges they do not need. For example, while Sendmail has to write to /var/spool/mqueue, it only has to read its configuration from /etc/mail and should not need to have write permission on its configuration. To implement such a least privilege environment, peas enable files to be tagged with additional permissions that overlay the respective underlying file permissions. File system permissions determine access rights based on the user identity of the process while pea file permission rules determine access rights based on the pea context in which a process is executed. Each pea file permission rule can selectively allow or deny use of the underlying read, write and execute permissions of a file on a per pea basis. The underlying file permission is always enforced, but pea permissions can further restrict whether the underlying permission is allowed to take effect. The final permission is achieved by performing a bitwise AND operation on both the pea and file system permissions. For example, if the pea permission rule allowed for read and execute, the permission set of r-x would be triplicated to r-xr-xr-x for the three sets of UNIX permissions and the bitwise AND operation would mask out any write permission that the underlying file system allow. This prevents any process in the pea from opening the file to modify it.

Enforcing on disk labeling of every single file, such as supported through access control lists provided by many modern file systems, is too inflexible if a single underlying file system is going to be used for multiple disparate pods and peas. Since each pea in each pod might make use of similar underlying files but have different permission schemes, storing the pea permission data on disk is not feasible. Instead, peas support the ability to dynamically label each file within a pod's file system based on two simple path matching permission rules, *path specific permission rules* and *directory default permission rules*. A path specific permission matches an exact path on the file system. For instance, if there is a path specific permission for /home/user/file, only that file will be matched with the appropriate permission set. On the other hand, if there is a directory default permission for the directory /home/user/ any file under that directory in the directory tree can match it, and inherit its permission set.

Given a set of path specific and directory default permissions for a pea, the algorithm for determining what permission matches to what path starts with the complete path and walks up the path to the root directory until it finds a matching permission rule. The algorithm can be described in four simple steps:

1. If the specific path has a *path specific permission*, return that permission set.
2. Otherwise, choose the path's directory as the current directory to test.
3. If the directory being tested has a *directory default permission*, return that permission set.
4. Otherwise set its parent as the current directory to test and go back to step 3.

If there is no *path specific permission*, the closest *directory default permission* to the specified path becomes the permission set for that path. Since, by default, peas give the root directory "/" a *directory default permission* denying all permissions, the default for every file on the system, unless otherwise specified is deny. This ensures the pea's have a fail safe default setup and do not allow access to any files unless specified by the administrator.

The semantics of pea file permission are based on file path name. If a file has more than one path name, such as via a hard link, both have to be protected by the same permission, otherwise depending on what order the underlying file is accessed the permission set it gets will be determined simply based on the path name that was accessed initially. This issue only occurs on creating the initial set of pea file access permissions. Once the pea is setup, any hard links that are created will obey the regular file system permissions. For instance, one is not allowed to create a hard link to a file that one does not have permission to. On the other hand, if one has permission to access the file, a *path specific permission* rule will be created for the newly created file that corresponds to the permission of the path name it was linked to.

The pea architecture makes use of the pod's stackable file system to integrate the pea file system namespace restrictions into the regular kernel permission model, thereby avoiding *time of check, time of use* race conditions. It accomplishes this by stacking on top of the file system's lookup function, which fills in the respective file's inode structure, and the permission function, which makes use of the stored permission data to make simple permission determinations. A file system's permission function is a standard part of the operating system's security infrastructure, so no kernel changes are necessary.

Pea Configuration Rules

File System

Many system resources in UNIX, including normal files, directories, and system devices, are accessed via files so controlling access to the file system is crucial. Each pea must be restricted to those files used by its component processes. This control is important for security, because processes that work together do not necessarily need the same access rights to files. All file system access is controlled by path specific and directory default rules, which specify a file or directory and an access right, such as read, write, and execute.

The access values for file rules are *read*, *write*, *execute*, similar to standard UNIX permissions. For convenience, we also define *allow* and *deny*, which are aliases for all three of read, write, and execute and cannot be combined with other access values in the same rules. When a path specific or directory default rule gives access to a file, it implicitly gives execute, but not read or write, access to all parent directories of the file, up to the root directory. On the other hand, if a path specific rule denies access to a directory, then access to both the directory and the directory contents, including subdirectories and files, will be denied, even if a separate rule would give access to subdirectories or files due to it being the best match.

```
pod mailserver {
    pea sendmail {
        path /etc/mail/aliases    read
        path /etc/mail/aliases.db read
    }
    pea newaliases {
        path /etc/mail/aliases    read
        path /etc/mail/aliases.db read,write
    }
}
```

Rule 1: Example of Read/Write rules.

Consider the case of the Sendmail mail daemon and the *newaliases* command with regard to the system-wide aliases file. Sendmail runs as the root user and needs to be able to read the aliases file in order to know to where it should forward mail or otherwise redirect it. *newaliases* is a symbolic link to *sendmail* and typically also runs as the root user in order to update the aliases file and convert it into the database format used by the Sendmail daemon. In our example, *newaliases* runs in its own pea and is able to read from */etc/mail/aliases* and read from and write to */etc/mail/aliases.db*. Meanwhile *sendmail* runs in another pea and is able to read both files, but not write to them. We use two path specific rules to express these access rules as described in Rule 1.

```
pod music {
    pea play {
         path /dev/dsp     write
    }
    pea rec {
         path /dev/dsp     read
    }
}
```

Rule 2: Protecting a device.

Similar rules can protect a device like /dev/dsp. When a user logins into a system locally, via the console, they are typically given control of local devices, such as the physical display and the sound card. Any application that the user runs has access to read from and write to these local devices, even though this privilege is not necessary. For example, we want to restrict playing and recording of sound files to the play and rec applications, which are part of SoX [27]. Rule 2 describe the rules that provide the appropriate access to the device.

The other file system rule is *dir-default*. It uses the same access values as path, but it is used to specify the default access for files below a directory. Any file or sub-directory will inherit the same access flags since access is determined by matching the longest possible path prefix. Unlike path specific rules, directory default rules can deny access to a directory in general, while still allowing access to specific files. Rule 3 describes a pea that denies access to all files in /bin, while only allowing access to /bin/ls.

```
pod fileLister {
    pea onlyLs {
        dir-default /bin   deny
        path /bin/ls       allow
     }
}
```

Rule 3: Directory default rule.

Transition Rules

In the Sendmail/Procmail use case, sendmail forks off and executes a procmail process to deliver the mail to the user's spool. Procmail needs different security settings, so it must transition from a Sendmail pea to a Procmail pea. Rules must be defined that state to which pea a process will transition upon execution. When a process calls the execve system call, we examine the file name to be executed and perform a longest prefix match on all the transition rules. For instance, by specifying a directory for a transition, PeaPod will cause a pea transition to occur for any program executed that is located in that directory, unless there's a more specific transition rule available.

Rule 4 creates a pea for Sendmail and Procmail, and specifies that a process should transition when the procmail program is executed.

```
pod mailserver {
  pea sendmail {
    transition /usr/bin/procmail   procmail
  }
  pea procmail {
  }
}
```

Rule 4: Transition rules.

PeaPod does not provide the ability for a process to transition to another pea besides by executing a new program. If it could, a process could open an allowed file in one pea and then transition to another pea where access to that file was not allowed and thus circumvent the security restrictions.

Networking Rules

PeaPod provides two rules that define the network capabilities a pea exposes to the processes running within it. First, peas are able to restrict a process from instantiating an outgoing connection. Second, peas are able to limit what ports a process can bind to and listen for incoming connections. By default, peas do not let processes make any outgoing connections or bind to any port. While a full network firewall is an important part of any security architecture, it is orthogonal to the goals of PeaPod and therefore belongs in its own security layer.

Continuing the simplified Sendmail/Procmail usage case, an administrator would want to easily confine the network presence of processes running within Sendmail/Procmail peas. By allowing sendmail to make outgoing connections, to enable it to send messages, as well as bind to port 25, the standard port for receiving messages, Sendmail can continue to work normally. On the other hand, processes run within the procmail pea, which will be less restricted, are not allowed to bind to any port for this same reason. On the other hand, programs run from within the procmail pea are allowed to initiate outgoing network connections. This allows programs, such as spam filters that require checking network based information, to continue to work.

```
pod mailserver {
    pea sendmail {
        outgoing   allow
        bind       tcp/25
    }
    pea procmail {
        outgoing   allow
    }
}
```

Rule 5: Networking rules.

Shared Namespace Rules

PeaPod provides a single namespace rule for enabling processes to access the pod's virtual private identifiers that do belong to its personal pea. PeaPod enables peas to be configured to only have access to resources tagged with specific pea identifiers or with the special global pea identifier that enables access to every virtual private resource in the pod. A common usage of this rule is to enable the creation of a global pea with access to all the resources of a pod, for instance to be enable a process to startup and shutdown services run within a resource restricted pea. Rule 6 describes a pod that has a global pea that is able to access every private virtual identifier in the pod, as well as pea that is able to access the virtual identifiers that belong to one of its sibling peas.

```
pod service {
    pea global_access {
        namespace       global
    }
    pea test1 {
        namespace       test2
    }
    pea test {
    }
}
```

Rule 6: Namespace access rules.

Managing Rules

To make it simpler for administrators to create peas in a pod, we allow groups of rules to be saved to a file and included in the main configuration file for a given PeaPod configuration. These groups of rules would typically describe the minimum resources necessary for a single application. Application packagers can include rule group files in their package and administrators can share rule groups with each other.

```
path /usr/bin/gcc               read,execute
dir-default /usr/lib/gcc-lib    read,execute
path /usr/bin/cpp               read,execute
path /usr/lib/libiberty.a       read
path /usr/bin/ar                read,execute
path /usr/bin/as                read,execute
path /usr/bin/ld                read,execute
path /usr/bin/ranlib            read,execute
path /usr/bin/strip             read,execute
```

Rule 7: Compiler rules.

A rule group, such as Rule 7 for a compiler, would be stored in a central location. An administrator uses an *include* rule to reference the external file as part of a development PeaPod. Rule 8 contains the tools necessary to build a Linux kernel from source; and permits access to the source code itself and a writable directory for the binaries.

```
pod workstation {
    pea kernel-development {
        include "stdlibs"
        include "compiler"
        include "tar"
        include "bzip2"
        dir-default /usr/local/src/   read
        dir-default /scratch/binaries allow
    }
}
```

Rule 8: Set of multiple rule files.

These management rules demonstrate PeaPod's ability to distinguish the minimal needs of a program service in order to execute, while enabling an administrator to define a local policy that can restrict what local resources the program service has access to. The knowledge needed to build a set of rules for a program service that provides the minimal needed set of resources to execute is not always readily available to users of security systems. However, this knowledge is available to the authors and distributors of the system. PeaPod's management rules enable the creation and distribution of rule files that define the minimal set of resources needed to execute a program service, while enabling the local administrator to further define the resources restriction policy.

Security Analysis

Saltzer and Schroeder [26] describe several principles for designing and building secure systems. These include:

- *Economy of mechanism*: Simpler and smaller systems are easier to understand and ensure that they do not allow unwanted access.
- *Fail safe defaults*: Systems must choose when to allow access as opposed to choosing when to deny.
- *Complete mediation*: Systems should check every access to protected objects.
- *Least privilege*: A process should only have access to the privileges and resources it needs to do its job.
- *Psychological acceptability*: If users are not willing to accept the requirements that the security system imposes, such as very complex passwords that the users are forced to write down, security is impaired. Similarly, if using the system is too complicated, users will misconfigure it and end up leaving it wide open.
- *Work factor*: Security designs should force an attacker to have to do extra work to break the system. The classic quantifiable example is when one adds a single bit to an encryption key, one doubles the key space an attacker has to search.

PeaPod is designed to satisfy these six principles. PeaPod provides economy of mechanism using a thin virtualization layer based on system call interposition and file system stacking that only adds a modest amount of code to a running system. The largest part of the system is due to the use of a null stackable file system with 7000 lines of C code, but this file system was generated using a simple high-level file system language [33], and only 50 lines of code were added to this well tested file system to implement the PeaPod file system security. Furthermore, PeaPod changes neither applications nor the underlying operating system kernel. The modest amount of code to implement PeaPod makes the system easier to understand. Since the PeaPod security model only provides resources that are explicitly stated, it is relatively easy to understand the security properties of resource access provided by the model.

PeaPod provides fail safe defaults by only providing access to resources that have been explicitly given to peas and pods. If a resource is not created within a pea, or explicitly made available to that pea, no process within that pea will be allowed to access it. While a pea can be configured to enable access to all resources of the pod, this is an explicit action an administrator has to take.

PeaPod provides for complete mediation of all resources available on the host machine by ensuring that all resource accesses occur through the pod's virtual namespace. Unless a file, process, or other operating system resource was explicitly placed in the pod by the administrator or created within the pod, PeaPod's virtualization will not allow a process within a pod to access the resource.

PeaPod's provide a least privilege environment in two ways. First, pods provide a least privilege environment by enabling an administrator to only include the data necessary for each service. PeaPod can provide separate pods for individual services so that separate services are isolated and restricted to the appropriate set of resources. Even if a service is exploited, PeaPod will limit the attacker to the resources the administrator provided for that service. While one can achieve similar isolation by running each individual service on a separate machine, this leads to inefficient use of resources. PeaPod maintains the same least privilege semantic of running individual services on separate machines, while making efficient use of machine resources at hand. For instance, an administrator could run MySQL and Sendmail mail transfer services on a single machine, but within different pods. If the Sendmail pod gets exploited, the pod model ensures that the MySQL pod and its data will remain isolated from the attacker. Furthermore, PeaPod's peas are explicitly designed to enable least privileged environments by restricting programs in an environment that can be easily limited to provide the least amount of access for the encapsulated program to do its job.

PeaPod provides psychological acceptability by leveraging the knowledge and skills system administrators already use to setup system environments. Because pods provide a virtual machine model, administrators can use their existing knowledge and skills to run their services within pods. Furthermore, peas use a simple resource based model that does not require a detailed understanding of any underlying operating system specifics. This differs from other least privilege architectures that force an administrator to learn new principles or complicated configuration languages that require a detailed understanding of operating system principles.

Similar to least privilege, PeaPod increases the work factor that it would take to compromise a system by simply not making available the resources that attackers depend on to harm a system once they have broken in. For example, since PeaPod can provide selective access to what program are included within their view, it would be very difficult to get a root shell on a system that does not have access to any shell program.

Usage Examples

We briefly describe three examples that help illustrate how the PeaPod virtualization layer can be used to improve computer security and application availability for different application scenarios. The application scenarios are e-mail delivery, web content delivery, and desktop computing. In the following examples we make extensive use of PeaPod's ability to compose rule files in order to simplify the rules. Instead of listing every file and library necessary to execute a program, we isolate them into a separate rule file to place the focus on the actual management of the service the pea is trying to protect.

E-mail Delivery

For e-mail delivery, PeaPod's virtualization layer can isolate different components of e-mail delivery to provide a significantly higher level of security in light of the many attacks on Sendmail vulnerabilities that have occurred. Consider isolating a Sendmail installation that also provides mail delivery and filtering via Procmail. E-mail delivery services are often run on the same system as other Internet services to improve resource utilization and simplify system administration through server consolidation. However, this can provide additional resources to services that do not need them, potentially increasing the damage that can be done to the system if attacked.

```
pod mail-delivery {
  pea sendmail {
    include "stdlibs"
    include "sendmail"
    dir-default /etc                read
    dir-default /var/spool/mqueue   allow
    dir-default /var/spool/mail     allow
    dir-default /var/run            allow
    path /usr/bin/procmail     read, execute
    transition /usr/bin/procmail    procmail
    bind                            tcp/25
    outgoing                        allow
  }
  pea procmail {
    dir-default /                   allow
    outgoing                        allow
  }
}
```

Rule 9: E-Mail delivery configuration.

As shown in Rule 9, using PeaPod's virtualization layer, both Sendmail and Procmail can execute in the same pod, which isolates e-mail delivery from other services on the system. Furthermore, Sendmail and Procmail can be placed in separate peas, which allows necessary interprocess communication mechanisms between them while improving isolation. This pod is a common example of a privileged service that has child helper applications. In this case, the Sendmail pea is configured with full network access to receive e-mail, but only with access to files necessary to read its configuration and to send and deliver email. Sendmail would be denied write access to file system areas such as /usr/bin to prevent modification to those executables, and would only be allowed to transition a process to the Procmail pea if it is executing Procmail, the only new program its pea allows it to execute. On mail delivery, Sendmail would then exec Procmail, which transitions the process into the Procmail pea. The Procmail pea is configured with a more liberal access permission, namely allowing access to the pod's entire file system, enabling it to run other programs, such as SpamAssassin. While an administrator could configure programs Procmail executes, such as SpamAssassin, to run within their own Peas, this case keeps them all within a single pea to demonstrate how simple a system can be. As a result, the Sendmail/Procmail pod can provide full e-mail delivery service while isolating Sendmail such that even if Sendmail is compromised by an attack, such as a buffer overflow, the attacker would be contained in the Sendmail pea and not even be able to execute processes, such as a root shell, to further compromise the system.

Web Content Delivery

For web content delivery, PeaPod's virtualization layer can isolate different components of web content delivery to provide a significantly higher level of security in light of common web server attacks that may exploit CGI script vulnerabilities. Consider isolating an Apache web server front end, a MySQL database back-end, and CGI scripts that interface between them. While one could run Apache and MySQL in separate pods, since they are providing a single service, it makes sense to run them within a single pod that is isolated from the rest of the system. However, since both Apache and MySQL are within the pod's single namespace, if an exploit is discovered in Apache, it could be used to perform unauthorized modifications to the MySQL database.

To provide greater isolation among different web content delivery components, Rule 10 describes a set of three peas in a pod: one for Apache, a second for MySQL, and a third for the CGI programs. Each pea is configured to contain the minimal set of resources needed by the processes running within the respective pea. The Apache pea includes the apache binary, configuration files and the static HTML content, as well as a transition permission to exec all CGI programs into the CGI pea. The CGI pea contains the relevant CGI programs as well as access to the MySQL daemon's named socket, allowing interprocess communication with the MySQL daemon to perform the relevant SQL queries. The MySQL pea contains the mysql daemon binary, configuration files and the files that make up the relevant databases. Since Apache is the only program exposed to the outside world, it is the only process that can be directly exploited. However, if an attacker is able to exploit it, the attacker is limited to a pea that is only able to read or write specific Apache files, as well as exec specific CGI programs into a separate pea. Since the only way to access the database is through the CGI programs, the only access to the database an attacker would have is what is allowed by said programs. Consequently, the ability of an attacker to cause serious harm to such a web content delivery system running with PeaPod's virtualization layer is significantly reduced.

```
pod web-delivery {
  pea apache {
    include "stdlibs"
    path /usr/sbin/apache       read,execute
    path /usr/sbin/apachectl    read,execute
    dir-default /var/www        read,execute
    transition /var/www/cgi-bin cgi
    bind                        tcp/80
  }
  pea cgi {
    include "stdlibs"
    include "perl"
    dir-default /var/www/data   allow
    path /tmp/mysql.sock        allow
  }
  pea mysql {
    include "stdlibs"
    path /usr/sbin/mysqld read,  execute
    path /tmp/mysql.sock         allow
    dir-default /usr/share/mysql read
    dir-default /var/lib/mysql   allow
  }
}
```

Rule 10: Web delivery rules.

Desktop Computing

For desktop computing, PeaPod's virtualization layer enables desktop computing environments to run multiple desktops from different security domains within multiple pods. Peas can also be used within the context of such a desktop computing environment to provide additional isolation. Many application used on a daily basis, such as mp3 players and web browsers, have had security holes. These holes enable attackers to execute malicious code or gain access to the entire local file system [12, 13]. Rule 11 describes a set of PeaPod rules that are used to contain a small set of desktop applications being used by a user with the /home/spotter home directory.

To secure an mp3 player, a pea can be created within the desktop computing pod that restricts the mp3 player's ability to make use of files outside of a special mp3 directory. Since most users store their music within its own subtree, this isn't a serious restriction. Most mp3 content should not be trusted, especially if one is streaming mp3s from a remote site. By running the mp3 player within this fully restricted pea, a malicious mp3 cannot compromise the user's desktop session. This mp3 player pea is simply configured with four file system permissions. A path specific permission that provides access to the mp3 player itself is required to load the application. A directory default permission that provides access to the entire mp3 directory subtree is required to give the process access to the mp3 file library. A directory-default permission to a directory meant to store temporary files so the mp3 player can be used as a helper application. Finally, a path specific permission that provides access to the /dev/dsp audio device is required to allow the process to play audio.

```
pod desktop {
  pea firefox {
    include "firefox"
    dir-default /home/spotter/.mozilla allow
    dir-default /home/spotter/tmp      allow
    dir-default /home/spotter/download allow
    transition /usr/bin/mpg123       mpg123
    transition /usr/bin/acroread    acroread
  }
  pea mpg123 {
    include "stdlibs"
    path /usr/bin/mpg123       read, execute
    path /dev/dsp                      write
    dir-default /home/spotter/tmp      allow
    dir-default /home/spotter/music    allow
  }
  pea acroread {
    include "stdlibs"
    include "acroread"
    dir-default /home/spotter/tmp      allow
  }
}
```

Rule 11: Desktop application rules.

To secure a web browser, a pea can be created within a desktop computing pod that restricts the web browser's access to system resources. Consider the Mozilla Firefox web browser as an example. A Firefox pea would need to have all the files Firefox needs to run accessible from within the pea. Mozilla dynamically loads libraries and stores them along with its plugins within the /usr/lib/firefox directory. By providing a directory default permission that provides access to that directory, as well as another directory default permission that provides access to the user's .mozilla directory, the Firefox web browser can run as normal within this special Firefox pea. Users also want the ability to be able to download and save files, as well as launch viewers, such as for postscript or mp3 files, directly from the web browser. This involves a simple reconfiguration of Firefox to change its internal application.tmp_dir variable to be a directory that is writable within the Mozilla pea. By creating such a directory, such as download within the users home directory, and providing a directory default permission allowing access, we enable one to explicitly save files, as well as implicitly save when one wants to execute a helper application. Similarly, just like Mozilla is configured to run helper applications for certain file types, one would have to configure the Mozilla pea to execute those helper applications within their respective peas. As shown, for an mp3 player, configuring such a pea for these process is fairly simple. The only addition one would have to make is to provide an additional pea transition permission to the Mozilla pea that tells the PeaPod's virtualization layer to transition the process to a separate pea on execution of programs such as the mpg123 mp3 player or the Acrobat Reader PDF viewer.

Experimental Results

We implemented PeaPod's virtualization layer as a loadable kernel module in Linux that requires no changes to the Linux kernel source code or design. We present some experimental results using our Linux prototype to quantify the overhead of using PeaPod on various applications. Experiments were conducted on two IBM Netfinity 4500R machines, each with a 933 Mhz Intel Pentium-III CPU, 512 MB RAM, 9.1 GB SCSI HD and a 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for the PeaPod on the other client system. The client ran Debian stable with a 2.4.21 kernel.

Name	Description
getpid	average getpid runtime
ioctl	average runtime for the FIONREAD ioctl
shmget-shmctl	IPC Shared memory segment holding an integer is created and removed
semget-semctl	IPC Semaphore variable is created and removed
fork-exit	process forks and waits for child that calls exit immediately
Apache	Runs Apache 1.3 under load and measures average request time
Make	Linux Kernel 2.4.21 compile with up to 10 processes active at one time
Postmark	Use Postmark Benchmark to simulate Sendmail performance
MySQL	"TPC-W like" interactions benchmark that uses Tomcat 4 and MySQL 4

Table 1: Application benchmarks.

To measure the cost of PeaPod's virtualization layer, we used a range of micro benchmarks and real application workloads and measured their performance on our Linux PeaPod prototype and a vanilla Linux system. Table 1 shows the seven micro-benchmarks and four application benchmarks we used to quantify PeaPod's virtualization overhead. To obtain accurate measurements, we rebooted the system between measurements. Additionally, the system call micro-benchmarks directly used the TSC register available on Pentium CPUs to record time-stamps at the significant measurement events. Each time-stamp has an average cost of 58 ns. The files for the benchmarks were stored on the NFS Server. All of these benchmarks were performed in a chrooted environment on the NFS client machine running Debian Unstable. Figure 1 shows the results of running the benchmarks under both configurations, with the vanilla Linux configuration normalized to one. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmarks results.

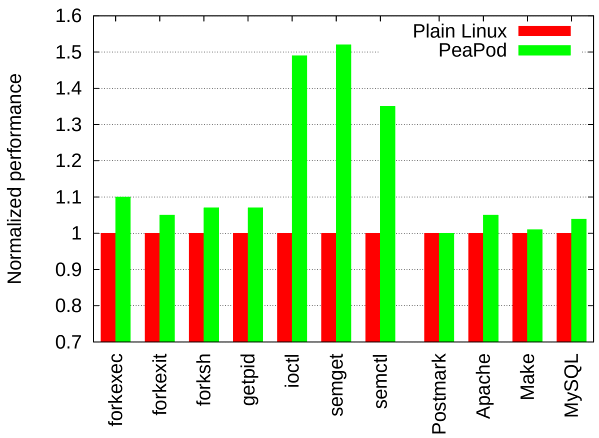

Figure 1: PeaPod virtualization overhead.

The results in Figure 1 show that PeaPod's virtualization overhead is small. PeaPod incurs less than 10% overhead for most of the micro-benchmarks and less than 4% overhead for the application workloads. The overhead for the simple system call getpid benchmark is only 7% compared to vanilla Linux, reflecting the fact that PeaPod virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup.

The most expensive benchmarks for PeaPod is semget+semctl, which took 51% longer than vanilla Linux. The cost reflects the fact that our untuned PeaPod prototype needs to allocate memory and do a number of namespace translations. The ioctl benchmark also has high overhead, because of the 12 separate assignments it does to protect the call against malicious root processes. These assignments correspond to saving the four variables that store UID state, assigning them a non privileged UID, and then restoring the original state. This is large compared to the simple FIONREAD ioctl that just performs a simple dereference. However, since the ioctl is simple, we see that it only adds 200 ns of overhead over any ioctl.

For real applications, the most overhead was only four percent, which was for the Apache 1.3 workload, where we used the http_load benchmark [21] to place a parallel fetch load on the server with 30 clients fetching at the same time. Similarly, we tested MySQL as part of a web-commerce scenario outlined by TPC-W with a bookstore servlet running on top of Tomcat 4 with a MySQL 4 back-end. The PeaPod overhead for this scenario was less than 2% versus vanilla Linux. These results are directly comparable to the virtualization results in AutoPod [22] and are effectively the same, demonstrating the additional overhead needed to confine processes into distinct peas is minimal.

Related Work

Many systems have been developed to isolate untrusted applications. NSA's Security Enhanced Linux [19], which is based upon the Flask Architecture [28], implements a policy language that one can use to implement models that enable one to enforce privilege separation. The policy language is very flexible but also very complex to use. The example security policy is over 80 pages long. There is research into creating tools to make policy analysis tractable [2], but the fact that the language is so complex makes it difficult for the average end user to construct an appropriate policy.

System call interception has been used by systems such as Janus [30, 10], Systrace [24], MAPbox [1], Software Wrappers [15], and Ostia [11]. These systems can enable flexible access controls per system call, but they have been limited by the difficulty of creating appropriate policy configurations. TRON [5], SubDomain [7] and Alcatraz [17] also operate at the system call level but focus on limiting access to the underlying file system. TRON allows transitions between different isolation units but requires application modifications to use this feature, while SubDomain supports an implicit transition on execution of a new child process. These systems provide a model somewhat similar to the file system approach used by PeaPod peas. However, peas are designed based on a full-fledged stackable file system that integrates fully with regular kernel security infrastructure and provides much better performance. Similarly, the PeaPod's virtualization layer provide a complete process isolation solution that is not just limited to file system protection.

Safer languages and run-time environments, most notably Java, have been developed to prevent common software errors and isolate applications in language-based virtual machine environments. These solutions require applications to be rewritten or recompiled, often with some loss in performance. Other language-based tools [8, 3] have also been developed to harden applications against common attacks, such as buffer overflow attacks. PeaPod's virtualization layer complements these approaches by providing isolation of legacy applications without modification.

Virtual machine monitors (VMMs) have been used to provide secure isolation [29, 32, 4]. Unlike PeaPod's virtualization layer, VMMs decouple processes from the underlying machine hardware, but tie them to an instance of an operating system. As a result, VMMs provide an entire operating system instance and namespace for each VM and lack the ability to isolate components within an operating system. If a single process in a VM is exploitable, malicious code can make use of it to access and make use of the entire set of operating system resources. Since PeaPod's virtualization layer decouples processes from the underlying operating system and its resulting namespace, they are natively able to limit the separate processes of a larger system to the appropriate resources needed by them. Furthermore, VMMs require more administrative overhead due to requiring administration of multiple full operating system instances as well imposing higher memory overhead due to the requirements of the underlying operating system.

A number of other approaches have explored the idea of virtualizing the operating system environment to provide application isolation. FreeBSD's Jail mode [14] provides a chroot like environment that processes cannot break out of. However, Jail is limited in what it can do, such as the fact that it doesn't allow IPC within a jail [9], and therefore many real world application will not work. More recently, Linux Vserver [18] and Solaris Zones [23] offer a similar virtual machine abstraction as PeaPod pods, but require substantial in-kernel modifications to support the abstraction. While these system's share the simplicity of the Pod abstraction. they do not provide finer-granularity isolation as provided with peas.

Conclusions

The PeaPod system provides an operating system virtualization layer that enables secure isolation of legacy applications. The virtualization layer supports two key abstractions for encapsulating processes, pods and peas. Pods provide an easy-to-use lightweight virtual machine abstraction that can securely isolate individual applications without the need to run an operating system instance in the pod. Peas provide a fine-grain least privilege mechanism that can further isolate application components within pods. PeaPod's virtualization layer can isolate untrusted applications, preventing them from being used to attack the underlying host system or other applications even if they are compromised.

PeaPod secure isolation functionality is achieved without any changes to applications or operating system kernels. We have implemented PeaPod in a Linux prototype and demonstrated how peas and pods can be used to improve computer security and application availability for a range of applications, including e-mail delivery, web servers and databases, and desktop computing. Our results show that PeaPod's virtualization layer can provide easily configurable and secure environments that can run a wide range of desktop and server Linux applications in least privilege environments with low overhead.

Acknowledgments

This work was supported in part by NSF grants CNS-0426623 and CNS-0717544, a DOE Early Career Award, and an IBM SUR Award.

Author Biographies

Shaya Potter is a Ph.D. candidate in Columbia University's Computer Science department. His research interests are focused around improving computer usage for users and administrators through virtualization and process migration technologies. He received his B.A. from Yeshiva University and his M.S. and M.Phil degrees from Columbia University, all in Computer Science. Reach him electronically at spotter@cs.columbia.edu .

Jason Nieh is an Associate Professor of Computer Science and and Director of the Network Computing Laboratory at Columbia University. He received his B.S. from MIT and his M.S. and Ph.D. from Stanford University, all in Electrical Engineering.

Matt Selsky earned his BS in Computer Science from Columbia University. He has been working at Columbia University since 1999, most recently as an engineer in the UNIX Systems Group. He works on e-mail-related services and is currently pursuing an MS in Computer Science from Columbia University. Reach him electronically at selsky@columbia.edu .

Bibliography

[1] Acharya, A. and M. Raje, ''MAPbox: Using Parameterized Behavior Classes to Confine Applications,'' *Proceedings of the 9th USENIX Security Symposium*, Aug., 2000.

[2] Archer, M., E. Leonard, and M. Pradella, ''Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux,'' Technical Report NRL/MR/5540-02-8629, NRL, 2002.

[3] Baratloo, A., N. Singh, and T. Tsai, ''Transparent Run-Time Defense Against Stack Smashing Attacks,'' *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[4] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield, ''Xen and the Art of Virtualization,'' *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct., 2003.

[5] Berman, A., V. Bourassa, and E. Selberg, ''TRON: Process-specific File Protection for the UNIX Operating System,'' *Proceedings of the 1995 USENIX Winter Technical Conference*, Jan., 1995.

[6] CNN Technology, ''Expert: Botnets No. 1 Emerging Internet Threat,'' Jan, 2006, http://edition.cnn.com/2006/TECH/internet/01/31/furst/index.html .

[7] Cowan, C., S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, ''SubDomain: Parsimonious Server Security,'' *Proceedings of the 14th USENIX Systems Administration Conference*, New Orleans, LA, Dec., 2000.

[8] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, ''StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,''

Proceedings of the 7th USENIX Security Conference, San Antonio, Texas, Jan., 1998.

[9] FreeBSD Project, *Developer's Handbook*, http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/secure-chroot.html .

[10] Garfinkel, T., "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," *Proceedings of the 2003 Network and Distributed Systems Security Symposium*, Feb., 2003.

[11] Garfinkel, T., B. Pfaff, and M. Rosenblum, "Ostia: A Delegating architecture for Secure System Call Interposition," *Proceedings of the 2004 Network and Distributed Systems Security Symposium*, Feb., 2004.

[12] GOBBLES Security, *Local/Remote mpg123 Exploit*, http://www.opennet.ru/base/exploits/1042565884_668.txt.html .

[13] GreyMagic Security Research, *Reading Local Files in Netscape 6 and Mozilla*, http://sec.greymagic.com/adv/gm001-ns/ .

[14] Kamp P.-H., and R. N. M. Watson, "Jails: Confining the Omnipotent Root," *Proceedings of the 2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May, 2000.

[15] Ko, C., T. Fraser, L. Badger, and D. Kilpatrick, "Detecting and Countering System Intrusions Using Software Wrappers," *Proceedings of the 9th Usenix Security Symposium*, Aug., 2000.

[16] LaMacchia, B., Personal Communication, Jan, 2004.

[17] Liang, Z., V. Venkatakrishnan, and R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs," *Procedings of the 19th Annual Computer Security Applications Conference*, Dec., 2003.

[18] *Linux VServer Project*, http://www.linux-vserver.org/ .

[19] Loscocco, P. and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June, 2001.

[20] Osman, S., D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec., 2002.

[21] Poskanzer, J., http://www.acme.com/software/http_load/ .

[22] Potter, S. and J. Nieh, "Reducing Downtime Due to System Maintenance and Upgrades," *Proceedings of the 19th Large Installation System Administration Conference*, San Diego, CA, Dec., 2005.

[23] Price, D. and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," *Proceedings of the 18th Large Installation System Administration Conference*, Nov., 2004.

[24] Provos, N., "Improving Host Security with System Call Policies," *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug., 2003.

[25] Rescorla, E., "Security Holes... Who Cares?" *Proceedings of the 12th USENIX Security Conference*, Washington, D.C., Aug., 2003.

[26] Saltzer, J. H., and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the 4th ACM Symposium on Operating System Principles*, Oct., 1973.

[27] *SoX – Sound eXchange*, http://sox.sourceforge.net/ .

[28] Spencer, R., S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," *Proceedings of the 8th USENIX Security Symposium*, Aug., 1999.

[29] VMware, Inc., http://www.vmware.com .

[30] Wagner, D., *Janus: An Approach for Confinement of Intrusted Applications*, Master's thesis, University of California, Berkeley, 1999.

[31] Watson, R. N. M., "Exploiting Concurrency Vulnerabilities in System Call Wrappers," *Proceedings of the First USENIX Workshop on Offensive Technologies*, Boston, MA, Aug., 2007.

[32] Whitaker, A., M. Shaw, and S. D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec., 2002.

[33] Zadok, E. and J. Nieh, "FiST: A Language for Stackable File Systems," *Proceedings of the 2000 Annual USENIX Technical Conference*, June, 2000.