Vsys: A programmable sudo

Sapan Bhatia*, Giovanni Di Stasi†, Thom Haddow‡, Andy Bavier*, Steve Muir♯, Larry Peterson*
*Princeton University †University of Napoli ‡Imperial College ♯Juniper Networks

We present Vsys, a mechanism for restricting access to privileged operations, much like the popular `sudo` tool on UNIX. Unlike `sudo`, Vsys allows privileges to be constrained using general-purpose programming languages and facilitates composing multiple system services into powerful abstractions for isolation. In use for over three years on PlanetLab, Vsys has enabled over 100 researchers to create private overlay networks, user-level file systems, virtual switches, and TCP-variants that function safely and without interference. Vsys has also been used by applications such as whole-system monitoring in a VM. We describe the design of Vsys and discuss our experiences and lessons learned.

1 Introduction

One of the key challenges we have faced when operating PlanetLab [1, 11] is helping researchers to implement and evaluate new ideas while maintaining a reasonable level of isolation between experiments (each of which runs in a separate *slice*). PlanetLab users may require the ability to sniff a subset of network traffic for diagnostic purposes, gain access to certain log data restricted to administrators, view global system state that is typically hidden from users, reserve TCP and UDP ports, create IP-level rules, and so on. We have received these requests frequently and continue to do so today [13]. Our goal is to grant such privileges to enable research, while simultaneously preserving isolation and the principle of least privilege to the extent possible.

Service isolation can be imposed at multiple levels in any system. A current trend is to equate virtual machines with service isolation, but different degrees of isolation can be enforced by the hardware, virtual machines, operating system, and user-space tools. On a PlanetLab node, Linux-Vservers [15] run each experiment in a `chroot` environment that prevents cross-domain actions between slices, and that provides a "superuser" account with limited privileges. However, since all slices share a single OS kernel, it is possible to grant additional OS privileges to a particular slice, for example, to let the "superuser" bind privileged ports or add routes to the kernel's IP forwarding table. But it is these sorts of these privileges that, if misused, can unacceptably impact other slices. We would like to limit a particular user to *only* bind port 53 to run his DNS service, and to *only* change routes on virtual devices that he controls. The problem is that the abstractions the OS gives us do not support granting privileges to users while imposing narrow limits on how they are used.

In this paper we describe Vsys, a framework that allows users to invoke privileged operations via scripts called *extensions* that precisely specify how these operations can be accessed. Vsys is inspired by the UNIX philosophy of creating new system services by combining simple OS primitives: Vsys enforces security policies and achieves isolation through a combination of existing OS primitives. For example, packet filters can block a subset of IP traffic from a service, virtual devices combined with bridging can be used to filter Ethernet traffic, `grep` can filter access to files, IP policy routing can instantiate key-based routes for packets, and so on.

Vsys began with the modest goal of being a `sudo` compatible with `chroot` jails. The `sudo` [17] tool allows users to run programs with the privileges of another user. It enables coarse-grained admission control via an access control list of commands that each user is allowed to run, along with limited predicates on the arguments. Vsys is designed with three primary goals that make it an improvement over `sudo`: (1) ease of assembling new extensions from existing OS abstractions and tools, using arbitrary programming languages; (2) ease of accessing extensions from the UNIX command line or within arbitrary programs; and (3) maintaining a fine-grained level of control over exactly what extensions users are able to invoke and how. With Vsys, simple tools can be used to rapidly develop extensions that multiple services can access safely. Unlike modifications to the virtual machine or OS layers of the system, a new Vsys extension can be developed and deployed on PlanetLab in a matter of days and enhanced incrementally over time.

This paper makes three contributions. First, we discuss the design of Vsys, an important feature of which is an Access Control Policy (ACP). Vsys ACPs insert policy code between the user, the Vsys extension, and the OS to ensure that the invocation of a given building-block command is consistent with the privilege granted to the user. Second, with the help of four heavily-used Vsys extensions, we show that existing OS primitives can be composed into powerful isolation abstractions, enabling functions such as virtual networking. While variants of the security mechanisms underlying Vsys have been explored before, Vsys is novel both in its details, and in its scope—for instance, in how extensions crosscut multiple OS subsystems (packet filtering, rout-

ing, sockets, file systems, etc.). Furthermore, Vsys has been used on a large scale for several years. Finally, we describe our experiences with Vsys and draw some lessons on creating new abstractions and fostering an active user community. We hope that these lessons and experiences will be helpful to designers of systems services and frameworks.

2 User View

In this section we describe Vsys from the standpoint of how extensions are added and invoked. In the next section we explain how Vsys works.

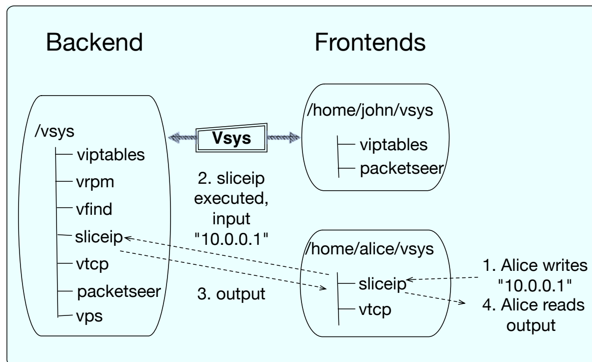

Figure 1: Basic use of Vsys

Figure 1 illustrates the basic operation of Vsys. Vsys extensions are executable scripts placed in a *backend* directory. Vsys monitors the backend directory and detects when new extensions are copied in. For each extension in the backend directory, Vsys also expects a corresponding ACL of the *contexts* authorized to use the extension. A context is just a system identifier like a slice or user ID. Vsys then creates special files (a pair of FIFO pipes or a UNIX domain socket) in the *frontend* directories for contexts listed on the extension's ACL. Each pair of FIFOs (for extensions with text input and output) or UNIX domain socket (for passing arbitrary data types) in a frontend directory maps to a specific extension in the backend directory.

Vsys requires that the special files it creates can only be read and written by contexts authorized to access the Vsys extension. In the `chroot` environment used on PlanetLab, each frontend directory is only visible within one slice's filesystem and so access to the FIFOs or socket is limited to the slice's users. On a standard UNIX system, the frontend could be any directory (e.g., a subdirectory of the user's HOME directory), and file system permissions limit access to a specific user.

In order to use an extension, a user simply opens the FIFO pipes or connects to the UNIX socket bearing the name of the extension and writes some arguments to it (e.g., to use a Vsys extension named `sliceip`, the user opens FIFOs called `/vsys/sliceip.in` and `/vsys/sliceip.out`). Vsys reads the arguments, runs the appropriate executable script on these arguments with sufficient privileges, and returns the output to the user through the pipe or socket.

3 Vsys Design

Vsys is intended to dispatch requests from non-privileged users to privileged extensions in a controlled manner. While there could be many approaches to implementing this functionality, we started with three design requirements. First, the Vsys framework should leverage existing UNIX primitives where possible. The philosophy of reusing OS building blocks when creating services inspired us to create Vsys; the Vsys design should also follow this philosophy. Second, users should be able to invoke Vsys using native operations on UNIX and on the command line, rather than via a new API or protocol. Third, one needed to be able to develop Vsys extensions using native code in any programming language. Our goal was to bundle Vsys as close to the OS as possible, not tying it with proprietary libraries, and to encourage users and administrators to contribute extensions by letting them program in their preferred programming environment.

These requirements led us to model our interface after the *everything-is-a-file* idiom as in Plan9 [12]. Users see Vsys extensions as special files in a */vsys* directory, and the Vsys daemon dispatches events back and forth between these special files and processes running extensions. The files that users interact with can be FIFO pipes or UNIX domain sockets. While the former are convenient to use, the latter support sending and receiving objects such as file and socket descriptors.

Vsys extensions are associated with access control policies (ACPs). An ACP is a program that defines a filter on the arguments passed to an operation, admitting a caller into the guarded operation only if the combination of the arguments and the current calling context is allowed by its policy. Each privileged operation wrapped by a Vsys extension is associated with two ACPs: an *invocation ACP* and a *syscall ACP*. The invocation ACP is run before the Vsys extension is executed and filters the arguments passed to the extension. The syscall ACP is triggered every time the extension makes a system call.

Figure 2 provides an overview of how Vsys works. Referring to the circled numbers:

1. A client process writes arguments into the input FIFO or UNIX domain socket corresponding to a particular Vsys extension. Vsys leverages UNIX file permissions and chroot to limit access to the FIFO or socket to a particular context (e.g., to a UID or a PlanetLab slice).

2. The Vsys daemon reads the arguments from the input FIFO or socket and looks up the corresponding

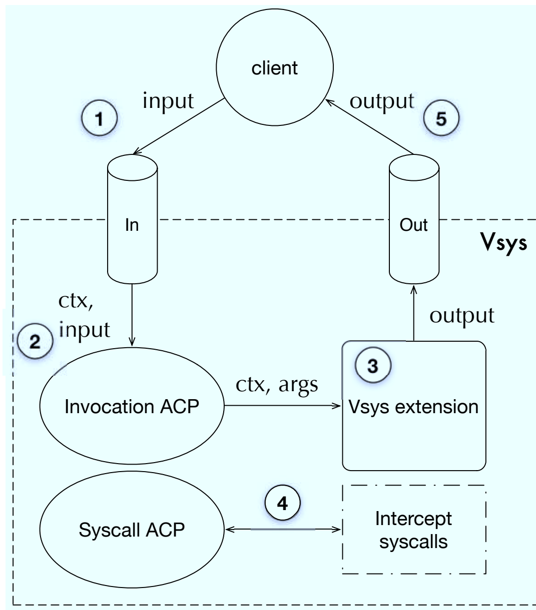

Figure 2: Vsys control flow

context. It passes the context ID and the arguments to the extension's invocation ACP. The invocation ACP runs and returns either success or failure.

3. If the invocation ACP returns success, the Vsys daemon executes the appropriate extension, passing in the original arguments and the calling context.

4. Vsys uses UNIX's `ptrace` facility to intercept system calls made by the extension. For each system call, Vsys executes the corresponding syscall ACP to allow or deny the call. This enables Vsys to limit the extension to touching specific resources (e.g., only opening certain ports).

5. Vsys writes output from the extension to the output FIFO or UNIX domain socket, from which it can be read by the client process.

Just like Vsys extensions themselves, ACPs are executables written in arbitrary programming languages. Sticking to our belief in re-using existing tools, we did not invent a new language to implement ACPs. In most cases, policies can be encoded with the help of regular expressions. Alternatively, lexing and parsing libraries such as GNU lex and yacc may also be used.

The design of Vsys relies on our assumed threat model: that extension developers are trusted, even though end users are not. On PlanetLab, all Vsys extensions are vetted by an administrator before they are deployed. We believe that the Vsys design provides a reasonable amount of security against the threat of malicious users. The Vsys daemon is simple and does little more than read and write FIFOs and execute processes. It is written in Ocaml [21], a type-safe functional language that adds robustness to this simplicity. The boundary between end users and Vsys extensions is stringent and cannot be circumvented as Vsys extensions run in a separate process address space. Extensions themselves are controlled using `ptrace`, which is a weak security mechanism [5]. However, security at that layer focuses on narrowing the interface to system calls that may be called with tainted data—i.e., inputs from an end user. It does not protect against malicious extension developers. Finally, we expect Vsys extensions to follow good coding practices by checking user-provided data and composing provably correct inputs to sensitive operations, as opposed to passing such tainted data directly or a transformed version of it.

4 Vsys Extension Library

An active user community has contributed a number of powerful extensions to Vsys over the years. This section presents several extensions that have been deployed and used on PlanetLab.

4.1 sliceip

`sliceip` enables users to create service-specific route tables. It is invoked with the same syntax as the `ip` command that creates and manages routes on Linux.

`sliceip` implements isolation through IP policy routing, a mechanism that extends the definition of the hash key used in routing to include fields other than the destination IP address. `sliceip` uses a *packet tag*, which associates packets with the sending or receiving user, as part of this route key. Thus, even when two users define separate routes to a single destination address, the tag determines whether a packet should take the route defined by the first user, the second user, or whether it should take the default route. There are many ways to set this packet tag to associate packets with users. The easiest way is to use the intermediate step of a network interface. The `pl_tuntap` extension discussed below lets users create and manage isolated virtual interfaces. Since local packets hold a record of the interface that emitted them, the name or ID of this interface can be used as the packet tag to identify the user.

Combined with `pl_tuntap`, `sliceip` enables users to create virtual overlay networks—one problem that the Linux community tackled by implementing the `netns` module for the Linux kernel. In contrast to `netns` which took over four years to develop and is still under active development, the deployment timeline for `sliceip` was of the order of months.

4.2 fusemount

FUSE [10] is a Linux-based framework for implementing and managing filesystems in userspace. A new filesystem can be developed by implementing standard filesystem operations such as directory and file lookups,

and exporting these operations via the FUSE userspace library. An in-kernel component implemented as a kernel module and the FUSE library communicate via a file descriptor obtained by opening a special character device (`/dev/fuse`). The obtained file descriptor is subsequently passed to the mount system call, to match up the descriptor with the mounted filesystem.

FUSE facilitated the development and deployment of the WheelFS [16] wide-area distributed filesystem on PlanetLab. WheelFS is implemented as a FUSE module that can be instantiated by PlanetLab users via Vsys. The authors of this work have made it possible for PlanetLab users to create their own shared filesystem as well as share it with other users [20].

Unlike `sliceip`, the `fusemount` extension is accessed via a UNIX domain socket. The caller (i.e., the user creating the filesystem) first obtains a file descriptor and uses it to populate a virtual filesystem via FUSE. Since at this point the filesystem has not been mounted, the operation of obtaining and using the file descriptor is safe. Next, the user connects to `fusemount` by opening the corresponding UNIX domain socket, and passes the aforementioned file descriptor over this connection. `fusemount` then performs the mount operation via the received file descriptor and passes the file descriptor back to the caller. The ensures the restrictions of the mount operation, such as by making sure that the mount point is owned by the caller.

4.3 socketops

`socketops` is a collection of extensions that lets users create privileged sockets for operating large TCP or UDP buffers, viewing low-level packet headers, etc. Rather than granting coarse-grained administrative access to the network as facilitated by the `CAP_NET_ADMIN` capability on Linux, Vsys allows users to access these operations selectively. Similar to `fusemount`, all of the above operations are accessed via UNIX domain sockets. Callers open the domain socket corresponding to the desired extension and pass parameters, such as a buffer size for bmsocket. The Vsys extension then returns a socket descriptor with the requested properties, and can be used by the caller independent of Vsys.

4.4 vtuntap

`vtuntap` lets users create and manage virtual devices without giving them administrative access to the network. This extension is a wrapper around *tun/tap*, a popular virtual point-to-point network device on Linux. On Linux, the tun/tap device is used via a file descriptor obtained by opening a special character device (`/dev/tun`). The file opening operation causes the tun/tap kernel module to create a new network interface. The device can then be configured using tools such as *ifconfig*. Once configured, the kernel serializes all packets sent to the device as a raw stream of packet data to the aforementioned file descriptor, and receives data written to the file descriptor as packets on the device.

`vtuntap` arbitrates both steps in this operation via two extensions. The `pl_tuntap` extension uses the UNIX socket interface to create the tun/tap file descriptor and send it to the caller. In addition, the `vif_up` extension lets users configure devices with parameters such as the MTU, transmit queue length and the IP address. `vif_up` is a wrapper around the `ifconfig` command and takes the same set of parameters. Through an ACP, Vsys verifies that the user is restricted to a set of allowed IP addresses and other authorized parameters.

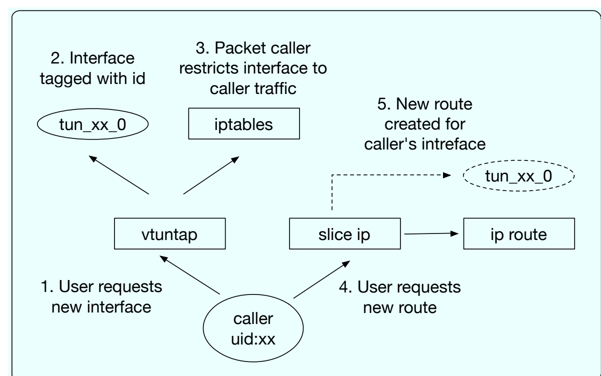

Figure 3: The vtuntap extension

Figure 3 shows how a user can combine `vtuntap` with `sliceip` to create an overlay topology. The user first calls `vtuntap` to create a new network interface, tag it an ID field unique to the user, and restrict the traffic to that interface. This configures the overlay's data plane. Then `sliceip` can add and remove routes on the control plane of the overlay.

5 Experiences and Lessons

This section articulates some of our experiences building Vsys and the conclusions to which they have led us.

Creating new OS abstractions is hard. The goal of the VINI project was to build a new testbed, using PlanetLab software, that would combine isolated virtual network topologies and PlanetLab slices [2]. At the start of the project, we considered two alternatives. The first was building virtual topologies using existing Linux networking primitives (e.g., IP policy routing) and userspace tools. We initially developed Vsys to explore this space. The second alternative was to leverage Linux network namespaces (`netns`), a new abstraction for the Linux kernel intended specifically for isolating the network subsystem. Over a summer we developed an initial prototype using `netns` that satisfied the requirements of

VINI and deployed it on the public VINI testbed [3].

Our subsequent experiences with netns were less positive. On the VINI testbed, incremental improvement of our netns-based prototype as well as bug fixes to netns continued for about two years. By this point the Vsys-based approach for building virtual topologies, which we had deployed on PlanetLab, was mature and had been used extensively by researchers. Most of the issues we encountered on VINI involved interactions between netns and other modules. For example, PlanetLab and VINI use Linux-Vserver to assign IP addresses to network slices, but netns would hide network devices from Vservers. The tools associated with netns unexpectedly also added filesystem and namespace isolation to processes when only network isolation was requested. Upon modifying the tools, we realized that there were dependencies between these isolations that required further kernel modifications. Such problems were hard to diagnose because of the lack of debugging tools for the new abstractions.

The lesson is that the continued predominance of old abstractions (e.g., pipes, file descriptors, and sockets) is no coincidence. Since fundamental OS abstractions are global and can affect all modules and processes in the system, changes cause side effects that are very hard to predict, especially when these side effects cut across modules. Had we foreseen our problems with netns, we would probably have focused our efforts on the Vsys approach from the start.

Flexibility drives innovation in development. Though we invite the PlanetLab user community to contribute code, we receive few contributions. Vsys extensions have been the exception to this rule: all but one Vsys extension were submitted by developers other than the authors of the Vsys framework. Vsys is a success story in our efforts to engage PlanetLab users in helping to develop the platform. This success is all the more surprising given that such user contributions are unusual for security mechanisms.

We attribute this to the use of standard abstractions and the ability to use the programming language of one's choice. In Vsys, an extension is an executable script to which inputs are passed explicitly as arguments and via standard input. This explicit and simple data flow adds developer confidence to the reliability of a script and enables him or her to develop scripts on a standard installation of Linux even if it does not run the PlanetLab environment. The ability to use any programming language also helps contributors reuse their existing code, regardless of the language it is written in.

The lesson is that even security mechanisms can attract external developers if you provide a flexible and easy-to-learn development environment. In our experience, skilled developers also have very strong tastes for using specific programming tools and standard environments, which it helps to support.

Reusing standard abstractions simplifies interfaces between components. Despite having been developed independently by over a dozen individuals, many of PlanetLab's Vsys scripts depend on one another. For instance, the fd_tuntap and reserve_eth scripts allocate network endpoints for users, and vifconfig configures the parameters of these devices and sets up routes and network address translations. Similarly, sliceip sets up tunnels and makeswitch connects the interfaces to virtual OpenFlow switches. The lesson is that the use of standard primitives—files, file descriptors, pipes, directories, network interfaces, packet filtering rules, network routes, and sockets—simplifies interfaces and facilitates program reuse.

6 Related Work

There is a great deal of work related to Vsys; we will focus on UNIX-centric mechanisms based on processes and other standard OS primitives. Vsys is similar to existing sandboxing tools [17, 6, 7, 5, 19, 4] but is novel in both details and scope. Furthermore, unlike those systems, Vsys has been deployed and used at scale for several years. In the process it has attracted numerous contributions from an active user community, validating our design goal of flexibility through the use of grassroots abstractions.

Vsys is similar to the interposition agents introduced by Jones [9] to insert policy between privileged operations and untrusted user code. Jones implemented a library of object-oriented abstractions that could be used to intercept system calls and modify their behavior, such as by tracing them or filtering their arguments. Vsys divides policy code between extensions and ACPs. Thus rather than one, there are two interposition sites, the first between calling clients and the underlying OS (i.e. the extensions themselves), and the second between the extensions and the underlying OS (i.e. the syscall ACPs).

SLIC [6], Janus [7] and Ostia [5] are sandboxing frameworks that use system call filtering and delegation to grant untrusted processes access to system resources. In Vsys, isolation is implemented in the form of Vsys extensions, which compose multiple system abstractions such as file descriptors, sockets and packet filtering rules. Like Janus, Vsys uses system call filtering with the help of ptrace to reinforce the limitations that the extension developer places on clients. Ostia uses system call delegation to protect against time-of-check-to-time-of-use bugs. The delegation mechanism executes the system call on the client's behalf immediately after authenticating it, eliminating the window of opportunity for attackers. The Vsys design assumes that extension developers are not malicious and so such mechanisms

are unnecessary; the goal of system call filtering in Vsys is to narrow the interface to extensions, as a backup to incomplete checks by script developers. Jain and Sekar's framework [8] also uses system calls for containment. Finally, Systrace [14] enables administrators to define system call access policies in much the same way as UNIX permissions define file access policies. In this way, fine-grained control can be imposed on processes, and the privileges of programs can be elevated without the use of potentially dangerous suid binaries.

The Proper ("PRivileged OPERations") daemon was the precursor to Vsys. It let PlanetLab users run privileged operations by passing file descriptors between privileged and non-privileged contexts. Users invoked primitive operations—socket creation, file opening and closing, execution, etc.—proxied by the Proper daemon. Vsys inherits Proper's use of file-descriptor passing from privileged to non-privileged contexts.

A more advanced form of `sudo` is `sus` [18], which extends the access control list to include predicates on objects such as files and users. Calife [4] is another variant of `sudo` with usability enhancements and privileged command logging. `SSU` [19] handles the remote execution of privileged operations over `ssh` sessions.

In contrast to these tools and their variants, the goal of Vsys goes beyond defining ACLs for privileged commands. Vsys is meant to facilitate the composition of existing tools to build isolated operations. The relationship between `sudo` scripts and Vsys extensions can be compared to that between assembly language and high-level programming languages. The former is a low-level mechanism and the latter provides convenient abstractions such as ACPs, context identifiers and file descriptor transfers making use of the mechanism.

Perhaps the best known OS mechanism for privilege allocation is UNIX file permissions and `setuid` bits. Vsys sets permissions on pipes and sockets so that they can only be opened by users authorized to access the corresponding extensions. Vsys is also able to penetrate `chroot` jails and filesystem containers, letting users invoke functionality that they do not have direct access to in the filesystem.

7 Conclusion

It has been demonstrated many times over that the rich library of abstractions available on OSes can go a long way in solving problems for which dedicated OS extensions were developed. Our experiences with Vsys have reinforced this belief by showing that simple compositions of existing UNIX tools can be used to implement powerful isolation. The `sliceip` and `pl_tuntap` extensions enable network isolation comparable to that found in dedicated approaches such as VINI and Linux network namespaces. The `fuse` extension lets users create userspace filesystems in a collaborative manner, letting other users on the system mount and use deployed filesystems. Our design choice of grassroots abstractions and an unconstrained development environment has also been validated by the continuous contributions of Vsys extensions by an active user community.

References

[1] BAVIER, A., BOWMAN, M., CULLER, D., CHUN, B., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating System Support for Planetary-Scale Network Services. In *Proc. 1st NSDI* (San Francisco, CA, Mar 2004).

[2] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REXFORD, J. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. SIGCOMM 2006* (Pisa, Italy, Sep 2006).

[3] BHATIA, S., MOTIWALA, M., MUHLBAUER, W., MUNDADA, Y., VALANCIUS, V., BAVIER, A., FEAMSTER, N., PETERSON, L., AND REXFORD, J. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proc. ACM CoNEXT 2008* (Madrid, Spain, December 2008).

[4] Calife: how to become root (or another user) with ones own password. http://www.keltia.net/programs/calife/.

[5] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. 2004 Symposium on Network and Distributed System Security* (2004).

[6] GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., AND ANDERSON, T. E. Slic: an extensibility system for commodity operating systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1998), ATEC.

[7] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (1996).

[8] JAIN, K., AND SEKAR, R. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *In Proc. Network and Distributed Systems Security Symposium* (1999).

[9] JONES, M. B. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (1993), SOSP.

[10] Filesystem In Userspace. http://fuse.sourceforge.net.

[11] PETERSON, L., BAVIER, A., FIUCZYNSKI, M., AND MUIR, S. Experiences Building PlanetLab. In *Proc. 7th OSDI* (Seattle, WA, Nov 2006).

[12] PIKE, R., PRESOTTO, D., THOMPSON, K., AND TRICKEY, H. Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference* (1990), pp. 1–9.

[13] Questions on PlanetLab Development Mailing List. http://lists.planet-lab.org/pipermail/devel/2009-December/004014.html http://lists.planet-lab.org/pipermail/devel/2009-June/003470.html http://lists.planet-lab.org/pipermail/users/2010-November/003756.html.

[14] PROVOS, N. Improving host security with system call policies. In *In Proceedings of the 12th Usenix Security Symposium* (2002).

[15] SOLTESZ, S., POTZL, H., FIUCZYNSKI, M., BAVIER, A., AND PETERSON, L. Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proc. EuroSys 2007* (Lisbon, Portugal, Mar 2007).

[16] STRIBLING, J., SOVRAN, Y., ZHANG, I., PRETZER, X., LI, J., KAASHOEK, M. F., AND MORRIS, R. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (April 2009).

[17] Man page for sudo. http://www.gratisoft.us/sudo/sudo.man.html.

[18] Sus privilege elevation tool. http://pdg.uow.edu.au/sus/.

[19] THORPE, C. Ssu: Extending ssh for secure root administration. In *Proceedings of the 12th USENIX conference on System administration* (1998).

[20] WheelFS Installation Instructions for PlanetLab. http://pdos.csail.mit.edu/wheelfs/doku.php?id=documentation#planetlab_nodes.

[21] XAVIER LEROY. The objective caml system, release 1.07, documentation and user's manual., 1997. http://caml.inria.fr/pub/distrib/ocaml-1.07/ocaml-1.07-refman.txt.