A flexible, virtualization-based approach for practical process isolation

Geoffrey Thomas

MEng Thesis Proposal

Abstract

Protecting the system from less-than-trustworthy processes and isolating processes from each other has long been a crucial component to computer security, but numerous weaknesses continue to plague current approaches. We demonstrate preliminary work towards a stronger isolation approach based on hardware virtualization support that is capable of integrating with existing operating systems. Our approach involves running individual processes inside separate, lightweight virtual machines with independent kernels, and providing limited but easily customizable access to the host system via userspace proxies to the file system and other resources. We argue that this approach addresses several issues with existing isolation and sandboxing approaches of many types, is practical, and is likely to be easily maintainable and secure as systems evolve.

Background and Previous Work

Isolating processes has a long and important history in the development of secure operating systems [1] [23]. Isolation allows multiple user accounts with different privileges to exist on the same system, with only a single program, the OS kernel, needing to be absolutely trusted to mediate these users. Typically, process isolation is enforced with some support from hardware and restricting what areas of memory and what devices are available to different programs: in particular, different programs cannot access each other's memory nor the kernel's, and all device access must be mediated through the OS kernel through the system call interface. This allows the kernel to respond to different user accounts with different responses for certain system calls, for instance, to prevent one user from reading files that were created by another without the permission of the second user. While this level of protection is generally effective, misconfigurations, vulnerabilities in individual programs, and vulnerabilities in the OS kernel can all cause a loss of system security.

We will describe several existing additional means of process isolation to investigate their security properties and their adoption: since the most secure system secures nothing if it isn't used, we would like to learn lessons from other attempts as to what yields or doesn't yield a well-used system before we design our own. By studying these approaches, we will come up with a short list of desired features for a new system, plus some ideas upon which to build that system.

`chroot` and `ptrace`

Beyond simple memory protection and user account protection are a number of additional approaches for isolating potentially problematic processes. One common approach is the UNIX `chroot` system call, which changes the root directory for a process and its children. This prevents a process from accessing files outside the new root directory through normal filesystem operations; the filesystem appears restricted to just the specified subdirectory. `chroot`-based isolation has long been a common security layer for many network services. For instance, FTP servers can be confined via `chroot` into a directory consisting of just the server software executables and the files to be served, thereby restricting the ability of remote users to browse other files on the host. However, there are several known methods for a determined attacker to break a `chroot` jail, making this more useful as a "defense in depth" strategy than as a complete restriction. For instance, a process running as the superuser, after being confined to a new root can arrange to run `chroot` again twice in a way that escapes the confinement [2], and even unprivileged processes can interact with the rest of the system in various ways, perhaps even by hijacking another process belonging

to the same user with the `ptrace` debugging interface [3]. For an audited FTP server running in its own user account which attempts to reject invalid requests from remote clients, `chroot` may provide some isolation, but by itself it cannot be considered sufficient to protect a machine from a determined attacker [4].

Various operating systems have developed methods based around the `chroot` concept, but with additional levels of isolation to allow them to serve as secure isolation techniques, whereas "`chroot` is not and never has been a security tool" [5]. At the simplest level, the third-party `grsecurity` patch set for the Linux kernel provides several restrictions on `chroot`'s behavior to restrict most common methods of escaping the jail, as well as the ability of processes confined by `chroot` to interact with other processes on the system [6]. FreeBSD jails add an additional system call based on `chroot` that isolate the process in several ways other than just the filesystem [36]. These solutions, and other similar ones such as Linux containers, overcome many of the weaknesses of `chroot`, but by their very isolation start moving towards requiring an entirely separate operating system installation, a significant overhead and perhaps leading towards a different design goal.

Another approach to securing a process beyond what `chroot` offers is to run the process under `ptrace`, the UNIX process-tracing interface typically used for debuggers. `ptrace` permits inspecting the process as it makes system calls and allows the tracer to refuse execution of certain system calls. For instance, an IRC bot on the Freenode network's C++ discussion channels, Geordi, permits execution of arbitrary snippets of C++ code, by restricting the number of system calls that can be made, even though it cannot assume benign users [7]. Geordi works largely by assuming that the evaluated code fragments demonstrate simple computation and programming-language features, and do not require much access to the host system; it therefore can directly deny system calls that, for instance, start new processes. Janus is a more full-featured system along these lines, starting from a basic assumption: "An application can do little harm if its access to the underlying operating system is appropriately restricted." Janus works by interposing itself in between an application and various system calls, and making access control decisions based on a policy file. While Janus is illustrative as to the properties a good isolation system should add to the standard OS security restrictions, unfortunately, system call interposition techniques have various weaknesses that leave such a system not completely impregnable [15].

Sandboxing in the Kernel: seccomp

A related and potentially more secure and performant approach is to let the kernel do system call restrictions. Linux offers a "secure computing" mode [13], which generally goes by the name of seccomp. A process can request the kernel enable seccomp via the `prctl` system call, such that any further activity from that processes is limited to reading and writing already open file descriptors (`read` and `write`), exiting normally (`_exit`), and returning from signal handlers (`sigreturn`). Any other system call will abort immediately without even parsing its arguments. The manual page documents secure computing mode as "useful for number-crunching applications that may need to execute untrusted byte code, perhaps obtained by reading from a pipe or socket," [9] but there are many more applications for this kind of protection. Many applications, once they have opened the files or other connections they need, can complete their work with just reads and writes and little more. Certainly in many cases, an application can spin off a child process to load a less-trusted shared library, enable secure computing mode, and jump into the shared library. For instance, many audio processing vulnerabilities could be avoided if the audio processor is not allowed to do anything other than read one format of addio on one end and produce another format on the other; the worst it could do is to produce garbled or incomplete output, but the input to the image processor was already trusted with the ability to specify an arbitrary output, and so assuming a simple output representation (such as one with no metadata), we can guarantee that no input can harm the system [10]. The same argument applies to image renderers, decompression software, and the like.

Note that this technique does not at all require us to prevent arbitrary code execution, as Janus noted. By using the existing protection domain functionality of UNIX, assuming the absence of kernel vulnerabilities, we can allow userspace to run any code, but by restricting its interaction with the system to reading from or writing to already-opened file descriptors that are known to be safe, or exiting, it cannot harm the rest of the system. To a large extent, even, this offers protection against kernel security bugs. Many of the particularly egregious bugs in Linux recently have been against system calls that are rarely used.

Consider the 2008 `vmsplice` local root exploit [11]: this system call was a variant of the `splice` system call, itself designed for particularly optimized data transfer between file descriptors that is traditionally just performed with a `read`/`write` loop. `vmsplice` allows optimized transfer between user memory and a pipe. This additional code missed some access checking, and allowed a caller to trick the kernel into overwriting kernel memory. The immediate fix that hit the web shortly after the news about the `vmsplice` vulnerability broke was simply to disable the system call entirely[1], as few programs in practice used it, and the functionality could be replicated with a simple `write`. Although there will always be newer, less-proofread system calls and demand for them, with secure computing mode enabled, even exploited programs simply do not have access to those system calls, so the exploit becomes harmless.

To examine the viability of a seccomp-based approach for applications beyond "number crunching", we investigated a common Linux PDF reader, xpdf, to see what system calls it uses. This is easy to determine with strace, a standard system call tracing utility. Table 1 lists the system calls used by xpdf once it has loaded a document, in the course of normal navigation. xpdf is a particularly useful example here because of its security track record [37].

The results indicate that restrictions similar to secure computing mode could be quite reasonable for running xpdf. `writev` is another optimized variant of write, select and poll are both mechanisms for waiting on file descriptors to be ready for reading or writing (again, providing essentially the same functionality in different interfaces), `_llseek` moves around within an open file, and `mmap2` and `munmap` are used either for memory allocation or mapping already-open files into the user's address space. Finaly, `restart_syscall` is a Linux implementation detail related to signal handling that only allows resuming existing system calls.

This leaves two calls to be examined, `stat64` and `gettimeofday`. The former gets file metadata by path; xpdf uses this to periodically check if a PDF has been changed and needs to be reloaded. For this functionality to work properly, it needs to examine the file path instead of the file handle, in case the document has been replaced with a new one. Under secure computing mode, though, xpdf would be unable to reopen the file, so for a first-order sandboxed PDF viewer we can simply remove this feature, assuming we are willing to recompile the program. But we can imagine a slightly more liberal secure computing mode that restricts stat64 or open to a particular set of arguments, or a trusted helper that implements the periodic check itself and somehow provides the new file descriptor to the program.

As for `gettimeofday`, there are known cryptographic side-channel attacks that are enabled by having access to accurate timing information [13], so by default secure computing mode disables access to the hardware timestamp counter [14]. So, providing microsecond-accurate timing information is presumably also unacceptable. However, we may be able to provide inaccurate times to the process as long as the values are still reasonable, either by introducing a bit of error or just by rounding the reported time.

From this exercise we can conclude that a secure-computing-style approach to xpdf is possible and retains its security properties, but implementation may be deeply tied to the behavior of the specific application and require customizing its code. We have not considered enabling access to a host of new system calls such as access to network sockets or executing new programs, which are both to attackers in allowing them to control a system and use it as a platform for new attacks, sending spam, etc. However, we would have to modify either xpdf or its libraries significantly in order to collapse all the I/O system calls into just `read` and `write` and disable the functionality that we can't permit, or we would risk requiring access to less-common variants of system calls and increase our attack surface for kernel vulnerabilities. For these reasons, we would like to see if an approach other than simple syscall restriction will work, preferably one that allows confining unmodified processes.

Google's Chromium project, for instance, does use an intricate sandbox based on the secure computing restriction. [14, 15] The setup is to use two threads, one untrusted and running under seccomp and one trusted, that share the same memory space, with communication between those threads over an RPC interface in a UNIX pipe. The two threads share their address space so that the trusted one can perform actions like allocating memory for the untrusted thread, but this means that the untrusted thread has to be hand-coded in assembly and not trust any writable memory – it cannot use any language that wants to store variables on a stack or heap. Meanwhile, a second trusted process in a separate address space handles verification for certain system calls like open,

[1]The de_exploit payload in [12], a variant of the exploit code itself, simply finds the address of `vmsplice` and replaces the first byte with an unconditional RET.

Table 1: A trace of xpdf's system calls in normal operation.

```
geofft@corndog:~/tmp$ strace -c -p 5344
Process 5344 attached - interrupt to quit
^CProcess 5344 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 52.54    0.005148           6       920           select
 38.32    0.003755           4       891           writev
  5.05    0.000495          55         9           munmap
  4.08    0.000400           0       981       682 read
  0.00    0.000000           0         1           restart_syscall
  0.00    0.000000           0       157           gettimeofday
  0.00    0.000000           0       120           _llseek
  0.00    0.000000           0       156           poll
  0.00    0.000000           0         9           mmap2
  0.00    0.000000           0         3           stat64
------ ----------- ----------- --------- --------- ----------------
100.00    0.009798                  3247       682 total
```

because of this memory restriction. In order to allow unmodified renderer code to execute under the sandbox, the initialization of the trusted thread disassembles the code for the untrusted one and replaces all system calls with RPCs before allowing the untrusted thread to execute. While there are some interesting advantages of the solution, it requires very careful coding of the trusted thread to reduce the risk of adding vulnerabilities there, as well as writing

The Chromium developers requested from the Linux kernel community an extension to the secure computing request that allows specifying which set of system calls to approve, instead of hard-coding just read and write. but this was not met with much enthuiasm [13]. One particular proposed alternative solution was to use a framework based on ftrace, a kernel instrumentation technique, which would allow informing the kernel of restrictions such as `{"sys_read", "fd == 0"}`. While this would allow more flexibility than the current secure computing sandbox and ideally eliminate the complexity of Chromium's trusted thread and trusted process, it appears that this approach would fall prey to the same issues that affect the system call interposition techniques that were problematic for Janus. On the kernel mailing list thread, James Morris pointed out two references, [15] (which we have already seen) and [16], as examples of why this approach is risky. We would like a solution that doesn't have the issues with these sorts of system call interposition techniques; relatedly, it seems optimal, if possible, to implement a technique that does not require new kernel features, such that we can start using it without having to convince the kernel development community of its merits.

Linux Security Modules

As mentioned on the kernel mailing list Linux already has a security module framework, LSM [17], that inspects data once it has been copied into kernelspace, which may be more suitable for the more complex policy decisions and avoids these issues. There are a couple of well-known security modules that use LSM, notably SELinux. Developed by the NSA, SELinux is a mandatory access control architecture for Linux that involves labeling all objects on the system and assigning to processes and applications that specify what labels they may access and how [38]. However, SELinux has seen a fair amount of resistance in terms of being used in production; although Red Hat Enterprise Linux and Fedora both default to SELinux enabled, a fair number of system administrators seem to disable it as a matter of course, and various software publishers require that it be disabled instead of adapting their software to be compatible with SELinux and providing appropriate profiles for their application [18] [19]. This process may also involve fixing various coding practices, such as text relocations resolved at runtime [20], which while not insecure of themselves,

prevents using some additional security techniques in SELinux such as not allowing executing memory that has ever been modified since it was mapped, which Fedora at least enables for even unconfined programs [19]. Also, if SELinux wants to refuse access to a real file, it can only return a failure code to the program, and cannot, say, divert the request to a different, safer directory so that it can appear to return success. This can be a common source of frustration when setting up a profile to confine an application. While SELinux can be used for tight sandboxing of individual programs to restrict most kinds of system access [21], in addition to just locking down abnormal behavior from a program with fairly liberal system access, the deployment history suggests that an approach based on or similar to SELinux will not have as much reach as we would like because of the demands it makes regarding modifying existing applications.

One final drawback of any of these approaches is that they require kernel modifications. While the LSM approach allows using a clean API and avoiding modifying the core kernel by loading a kernel module [17], the resulting code still runs with full kernel privileges, which has several limitations: code can only be developed in a low-level language such as C, does not have the support of standard userspace libraries, needs to run quickly, and increases the attack surface of the kernel. As an example of the latter, while most Linux systems default to not allowing users to map the zero page to make kernel null pointer vulnerabilities harder to exploit, the "Cheddar Bay" Linux local root exploit from summer 2009, which depended on such a vulnerability, was able to use SELinux to bypass this check, thereby [26]. The system-call interposition techniques discussed previously have the advantage of being able to run in userspace, avoiding "weakening default security," in addition to the practical flexibility advantages. Ideally we would like a solution that allows us to keep as much of our policy enforcement in userspace as possible.

Another system worth briefly addressing here is that of UserFS, which allows unprivileged users to create sub-user identities with a subset of their system access, and run programs as these new users instead of with their full privileges [35]. Specifically for the use case of preventing outside attacks from executing malicious code, UserFS is an elegant solution to preventing this malicious code from having the full privileges of a user's account. Although UserFS was in no small part inspiration for our present work, we reject its approach because it does not provide protection from kernel bugs, does not provide complete isolation, and also requires kernel modification. Arguably, though, UserFS operates on a different level and provides a different type of security than we are interested in providing, and we will see later that it can possibly integrate with our solution.

Virtual Machines

A completely different approach to confining possibly-malicious software is to run it within a virtual machine, and only make the necessary resources available to this virtual machine. This approach has been well-known since the 1970s and the days of IBM's VM line of operating systems: an architecture allowing for hardware virtualization can allow for essentially complete isolation between guest virtual machines, with security at least as strong as the interprocess protection in time-sharing operating systems [23]. The properties of a virtual machine as set out by Popek and Goldberg [24] are essentially the properties we want for a good sandboxing system for either individual software or an entire operating system: efficiency and equivalence guarantee that our scheme is reasonably usable with existing software, and the resource control property, that nothing in the guest can affect any system resources that were not explicitly allocated to it by the host, is sufficient to provide us with the security guarantees we want for sandboxing.

One simple although potentially cumbersome and high-overhead approach is to simply run multiple virtual machines with an existing off-the-shelf hypervisor, and keep tasks of different security profiles in different virtual machines. This is a variant of the traditional "air gap" approach, invovling multiple physical machines, but trades the ability to trust a hypervisor for consolidation into a single piece of physical hardware. One security researcher, Joanna Rutkowska, has detailed such a setup involving three levels of trustworthiness and indicates she uses it regularly as her standard computing platform, using VMware Fusion, and regularly resets the least-trustworthy but least-important image to a known clean state [25]. However, such an approach has not seemed to catch on, and the overhead of multiple VMs and things like file transfer make this less practical for users who are not themselves security researchers.

Recently, Rutkowska's firm has started developing an operating system based on this context, using the Xen hypervisor and a customized Linux distribution to isolate applications. Their setup, called Qubes OS [22], uses on the order of a half dozen security do-

mains ("banking", "social", etc.) for running individual applications. The privileged Xen domain (dom0) runs a non-networked GUI to display windows from the guests, with borders around the windows indicating which guest they are from; a special graphics driver using X compositing is available to the guests for rendering. Qubes allows strong isolation of standard off-the-shelf Linux software by running every application in a VM that is not the dom0, but its architecture requires it to be a full OS, as opposed to something usable on existing operating systems. Our design has several goals in common with Qubes, but we would like one that works without requiring users to choose a specialized operating system.

Summarizing the above, we have a small list of desiderata in our sandboxing system:

- capable of securing all interaction with the system
- capable of working with unmodified binary applications
- reliably secure to actively malicious attackers
- requiring as few changes to core kernel code as possible
- operating as much in userspace as possible
- compatible with existing operating systems

Our Design

Taking Advantage of Virtual Machines

Virtual machine technology has matured and improved much in the recent past, most notably including the advent of hardware virtualization support for the x86 family by both Intel and AMD which makes the architecture classically virtualizable [29] [24] and the availability of various stable virtual machine monitors using these interfaces. We will focus our efforts on Linux running on x86, using the KVM hypervisor [39]. KVM is structured to run as a userspace Linux application, based on the QEMU emulator, using a kernel interface (/dev/kvm) to the hardware virtualization extensions. This structure allows us to integrate our work with existing Linux operating systems, unlike Qubes, which is its own OS based on the separate Xen hypervisor. Our design will focus on isolating individual less-trustworthy applications, as opposed to categorizing and confining every element of the system like Qubes or even SELinux.

We have a couple of advantages on our side in taking the virtual-machine strategy as a primitive in our design. While we have seen that there is some resistance to introducing new kernel mechanisms or wide-ranging changes for the sake of a direct interface for sandboxing, there is an immense amount of commercial interest in virtualization and its efficiency and flexibility, and so we can expect an in-kernel virtualization interface to only be better-maintained, more secure, and more efficient over time. Similarly, if we want to launch and shut down virtual machines quickly, many people in the Linux community have been working on optimizing Linux's boot time [31] [32]. By setting up our efforts in synergy with these goals, we can hope to have a practical, usable solution for years to come, instead of one that remains usable only on older systems (Janus, for instance, is only compatible with Linux 2.2, long obsolete).

Our basic approach is simply to take the application we wish to sandbox and run it inside a KVM guest, running a Linux kernel solely for the sake of supporting this process. This allows us to run unmodified applications by providing a full Linux kernel – instead of some custom-written interface to fake responses to system calls, or a restriction on which system calls are permissible – but by carefully controlling what virtualized hardware and other resources we provided to the KVM process, we can determine exactly what sort of access the sandboxed processes has to our host system. For instance, reads and writes can use any of the various system calls for IO, whether `read` and `write` themselves or fancier interfaces like `vmsplice` or `writev`, but the only interface to the host file system is whatever virtualized disk or file system we provide, which can use a much tighter interface. Assuming the security of the hypervisor, which as we have already noted can be expected to increase over time, the only point of attack here is whatever small interfaces we provide to the KVM process.

It is worth noting explicitly that, if we only have one application per KVM virtual machine, such a setup is in fact secure not only to arbitrary userspace code but also to arbitrary *kernelspace* code: in other words, we are effectively immune to kernel vulnerabilities, and only need worry about hypervisor vulnerabilities. Everything running within the guest, whether userspace or kernelspace, root or not, has exactly the same level of privilege with respect to the outside system. This is a powerful and extremely useful barrier.

Userspace Helpers

To sandbox a program, we trap calls to that program, for instance by replacing the binary with a shell script wrapper, and pass the arguments off to the sandboxing core, vwrap. This program will identify the program and its immediate dependencies (such as shared libraries) and start up a new virtual machine instance, running a lightweight Linux kernel whose init process is a shell script to run the sandboxed program with the appropriate arguments. vwrapwill also arrange for resources such as limited file system access, open file descriptors, etc. to be made available to the VM, such that the process can run as normal.

Our current approach for filesystem access is to run an NFS server as a separate process in the host, and use this as the root filesystem in the guest. The NFS server does not need to be privileged (in particular this is not the standard kernel NFS server); it does not even need to bind privileged ports, because we can specify the port numbers as a mount option. This means that the server does not affect the security of other processes running on the system, and can run with, at most, the credentials of the user invoking vwrap. As compared to the guest, though, this is a trusted process and mediates all external file access available to the guest. This is an important part of our ability to allow root and the kernel of the guest to be untrusted: if we provided full disk access to the guest and relied on, say, a union file system to prevent permanent writes to the host system, anyone with root inside the guest could simply unmount the unioned file system and get direct access to the host's file system, or read files on the block device belonging to other users, or the like. Our NFS server is written in Python, which provides two advantages to us: the code is easier to audit and easier to change. By writing in a high-level bytecode-based language, we avoid the risk of buffer overflows, null pointer dereferences, and other similar attacks common to OS kernels writtein in a low-level language (assuming the security of the Python core runtime, but this is much more static than a constantly-evolving kernel. We also have the ability to implement as complex or as simple access-control logic as we desire, and are not limited by the desire to avoid introducing complex parsers or interpreters into the kernel.

The userspace NFS server provides a limited copy of the host's filesystem to the guest. This allows applications to be installed only once on the host, instead of requiring several separate installations, and more importantly patched for security updates only once. We cannot, after all, prevent applications from destroying data they are legitimately authorized to work with. Since this runs in userspace, we can provide any files known to the OS's packaging system, which can be assumed public, without exposing private configuration on this system, for instance, the passwd file. We can also allow modifying files as much as the application wants without risking allowing the application to maliciously affect the host system, such as by installing rootkits or winning race conditions in /tmp. However, we have the flexibility to make these writes appear to the guest to succeed, instead of simply refusing them, as SELinux and other LSM-based systems must do. This design choice also allows us to not have to worry about the overhead of creating persistent installations for our virtual machine instances, as for instance is the case with the Tahoma web browser, which also uses virtual machines for isolation [28].

We can provide dynamically-generated faked configuration files, such as the passwd file, which really needs only contain information about a single user. We also provide a separate home directory, except for the files that the program should have legitimate access to, which eliminates several types of attacks involving stealing private files or writing to various configuration files (.bashrc, .wgetrc, etc.) that can be leveraged to attack other programs. Again, though, we allow sandboxed processes to write their own configuration files within the sandbox, which eliminates a large class of annoyances in trying to confine programs with a system like SELinux. By running in userspace, we have significantly more flexibility about making these sorts of access control decisions: in particular, we can even have the trusted process bring up UI to interact with the user before permitting access to the guest, similar to the file-opening API in OLPC's Bitfrost security model [33].

Challenges and Future Direction

The major challenge in this design will be to provide flexible, performant, and secure access to non-file resources. The most important is the display server: obvious options include the X SECURITY extension [34], but the policies it enforces are fairly inflexible and also require denying requests for things like keyboard grabs instead of returning success to the client. Much like with the filesystem case, we would like to be able to let the client speak to a proxy X server interface for increased compatibility with unmodified applications that expect X requests to succeed. Other

options include a design using X compositing and running another X server inside the guest, similar to the approach taken by Qubes or the SELinux-based `sandbox -X` do, or a lighter-weight approach involving something else that speaks the X client protocol. Other resources that we will need to provide limited access to include audio and networking. For the latter, QEMU's user-mode network stack gives us a fair amount of flexibility.

Another resource that we will need to provide is access to file-like resources that are not regular files: pipes, sockets, terminals, and the like. For a large amount of this, we can make do by providing guest serial devices connected to the appropriate inputs and outputs on the host side, but this has several drawbacks. The file descriptors will appear unconditionally as serial terminals, but we could install wrappers in the guest to work around this. More problematic is that we have no way to make file-type-specific calls, whether setting terminal attributes or sending datagrams or accepting connections from a listening socket. We will need to design some interface for providing this functionality to the guest, but the interface needs to be simple enough that we can be assured of its security, or at least have reason to believe that the attack surface is significantly smaller than the kernel's native interfaces for the system calls that we will be replicating.

While the design may seem at first to be inescapably slow because of the cost of emulating and virtualizing every interface with the outside world, we have the option of providing so-called paravirtualized drivers, which interact directly through the privilege boundary with host code, instead of going through a layer of hardware emulation. KVM and other Linux-based hypervisors support the virtio interface [30], which allows the hypervisor to provide functions to guest drivers. We currently use the stock virtio network and console drivers to avoid the overhead of serializing all network traffic, including filesystem requests, and console traffic through an emulated hardware device just to deserialize it on the other end. We may end up developing additional virtio drivers and controllers to avoid specific hardware overheads or just address measured inefficiencies in our setup.

Current Results

We have prototyped our system with a straightforward NFS server in 1200 lines of Python with no native code dependencies. The file server currently enforces no access control other than to check that the file is world-readable and to refuse all write requests, but this can be changed for the actual system. We have a small C program to serve as init and execute xpdf as a child process with an appropriate minimal environment, and run KVM with this program as init for a minimally-configured Linux kernel (recompiled, but with no source changes) optimized for a bare KVM environment. The kernel boots in half a second, configures its networking within 2 seconds, mounts and executes the initial process within 0.1 second, and successfully launches xpdf within about ten seconds. The resulting PDF viewer is responsive for interactive use and feels comparable with xpdf run on the host. So this system is already eminently usable for limited use cases.

Planned Work

To make the system practical for general use requires significant further development, which can generally be split into two categories, adding secure functionality and improving performance. Specifically, we propose the following work:

- Extend the file system with reasonable policies for applications that write to files.
- Allow the file system to provide persistent per-application home directories.
- Design a scheme for handling sockets, terminal devices, and the like.
- Design a performant, secure proxy for guest application access to the window server.
- Design an appropriate interface for specifying the policy by which guests are permitted to access the host.
- Reduce the boot-up time of the virtual machine, that is, the time for the application to launch.
- Ensure performance for running multiple VMs, likely by using memory ballooning and host memory deduplication.
- Find a way to optimize performance of the userspace file server without compromising security.
- Write unit and system tests for the various components, and integration tests involving exploits.

- Investigate the security of our hypervisor.

While xpdf is useful, it is admittedly a fairly simple application, never writing to the filesystem. We would like to support larger applications such as web browsers and office suites, which will require a fair amount of engineering work and performance analysis to properly run under this system. However, we are confident that this approach will work, given our success with xpdf and given the design that lets us export any resources to the guest. Additional possibilities for this technology include isolating each tab within a browser, in the manner of Chromium, Gazelle [27], or Tahoma [28], or encapsulating web applications.

A few possibly-related goals are specifically out of scope for our design. First, we intend to confine unmodified applications, and leave intra-application security to the application. For instance, the UserFS authors demonstrate modifying a web application, DokuWiki, to assign separate local users to each remote user [35], thereby reducing the risk of leaks between private content as compared to running the web application as a single local user. We do not intend to pursue this route, although we note in passing that UserFS can be combined with our system, thereby eliminating our previous objection to UserFS requiring a kernel module since this now only requires modifying the guest kernel. We also do not design our system for the case in which sandboxed applications are trusted, but a sandbox rerouting layer is more straightforward than modifying the applications; if one can assume that the application inside is trusted, one can as well use a much simpler and weaker protection layer. Finally, although it would be possible to use our setup to run non-native programs, such as Linux applications on a non-Linux system or ARM applications on an x86 host using QEMU as a strict emulator, our efforts will focus on optimization assuming we can use hardware virtualization and so that we can assume that certain host components, such as copies of system libraries, can be passed to the guest.

Our final system should be reasonably usable on existing Linux desktops and servers as a practical but secure sandboxing technique. We have proposed a new approach for sandboxing, running individual programs inside their own virtualized kernel, which has only become practical recently with well-matured full system virtualization technology, but which eliminates a number of difficulties and problems with previous sandboxing designs. It is worth noting again that our system works entirely unprivileged: as long as /dev/kvm is accessible to all users, kvm itself runs unprivileged, and our file server and any other servers also need no special privileges from the perspective of the host and no configuration from the system administrator other than installing kvm. We have demonstrated the viability of this approach in a fairly simple prototype and argued for the validity of our approach, and we propose additional design and engineering work to make a system usable in practice.

References

[1] F. J. Corbató and V. A. Vyssotsky. "Introduction and Overview of the Multics System." AFIPS 1965 Fall Joint Computing Conference.

[2] Simon Burr. "How to Break Out of a `chroot()` Jail." `http://www.bpfh.net/simes/computing/chroot-break.html`

[3] Thamer Al-Herbish, ed. "Secure UNIX Programming FAQ." comp.security.unix. May 16, 2009. `http://www.faqs.org/faqs/unix-faq/programmer/secure-programming/`

[4] Bill Cheswick. "An Evening With Berferd: In Which a Cracker is Lured, Endured, and Studied." 1992 Winter USENIX Conference.

[5] Alan Cox. "Re: Chroot bug." Linux kernel mailing list. `http://kerneltrap.org/Linux/Abusing_chroot`

[6] grsecurity feature list. `http://grsecurity.net/features.php`

[7] Eelis. "geordi - C++ eval bot." `http://www.xs4all.nl/~weegen/eelis/geordi/`

[8] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)." 1996 USENIX UNIX Security Symposium.

[9] Michael Kerrisk. Manual page for `prctl`(2). Linux man-pages project release 3.05.

[10] Daniel J. Bernstein. "Selected research activities." January 7, 2005. `http://cr.yp.to/cv/activities-20050107.pdf`

[11] Jonathan Corbet. "vmsplice(): the making of a local root exploit." Linux Weekly News. Feburary 12, 2008. `http://lwn.net/Articles/268783/`

[12] qaaz and Morten Hustveit. http://www.ping.uio.no/~mortehu/disable-vmsplice-if-exploitable.c

[13] Corbet, Jonathan. "Seccomp and sandboxing." Linux Weekly News. May 13, 2009. http://lwn.net/Articles/332974/

[14] Adam Langley. "Chromium's seccomp Sandbox." August 26, 2009. http://www.imperialviolet.org/2009/08/26/seccomp.html

[15] Tal Garfinkel. "Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools." February 2003. Proc. Network and Distributed Systems Security Symposium.

[16] Robert N. M. Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers." 2007 USENIX Workshop on Offensive Technologies. http://www.watson.org/~robert/2007woot/

[17] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. "Linux Security Modules: General Security Support for the Linux Kernel." 2002 USENIX Security Symposium.

[18] Joshua Brindle. "Software not working? Disable SELinux." May 21, 2006. http://securityblog.org/brindle/2006/05/21/software-not-working-disable-selinux/

[19] CentOS mailing list discussion thread starting at http://lists.centos.org/pipermail/centos/2010-November/101840.html

[20] Ulrich Drepper. "Text Relocations." http://www.akkadia.org/drepper/textrelocs.html

[21] Dan Walsh. "Cool things with SELinux... Introducing sandbox -X." September 16, 2009. http://danwalsh.livejournal.com/31146.html

[22] Joanna Rutkowska and Rafal Wojtczuk. "Qubes OS Architecture." January 2010. http://qubes-os.org/files/doc/arch-spec-0.3.pdf

[23] Stuart E. Madnick and John J. Donovan. "Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation." 1973 Workshop on Virtual Computer Systems.

[24] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third-Generation Architectures." July 1974. Communications of the ACM, volume 17, issue 7.

[25] Tom's Hardware. "A Strategy For Protection". From interview with Joanna Rutkowksa, "Going Three Levels Beyond Kernel Rootkits." http://www.tomshardware.com/reviews/joanna-rutkowska-rootkit,2356-6.html

[26] James Morris. "A brief note on the 2.6.30 kernel null pointer vulnerability." July 18, 2009. http://blog.namei.org/2009/07/18/a-brief-note-on-the-2630-kernel-null-pointer-vulnerability/

[27] The Multi-OS Construction of the Gazelle Web Browser

[28] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. "A Safety-Oriented Platform for Web Applications." 2006 IEEE Symposium on Security and Privacy.

[29] Keith Adams and Ole Agesen. "A Comparison of Software and Hardware Techniques for x86 Virtualization." 2006 ACM Conference on Architectural Support for Programming Languages and Operating Systems. http://www.vmware.com/pdf/asplos235_adams.pdf

[30] Rusty Russell. "virtio: Towards a De-Facto Standard for Virtual I/O Devices." July 2008. ACM SIGOPS Operating Systems Review, volume 42, issue 5.

[31] Don Marti. "LPC: Booting Linux in five seconds." Linux Weekly News. September 22, 2008. http://lwn.net/Articles/299483/

[32] Robbie Williamson and Scott James Remnant. "Boot Performance." Ubuntu wiki. https://wiki.ubuntu.com/FoundationsTeam/BootPerformance

[33] Ivan Krstić and Simson Garfinkel. "Bitfrost: The One Laptop per Child Security Model." http://wiki.laptop.org/go/OLPC_Bitfrost

[34] David P. Wiggins. "Security Extension Specification." November 15, 1996. http://www.xfree86.org/current/security.pdf

[35] Taesoo Kim and Nickolai Zeldovich. "Making Linux Protection Mechanisms Egalitarian with UserFS." 2010 USENIX Security Symposium.

[36] Poul-Henning Kamp and Robert. N. M. Watson. "Jails: Confining the omnipotent root." 2000 System Administration and Network Engineering Conference. http://phk.freebsd.dk/pubs/sane2000-jail.pdf

[37] Frederick Peters. "xpdf code security, removal of pdftohtml." debian-release mailing list. January 5, 2008. http://lists.debian.org/debian-release/2008/01/msg00057.html

[38] Peter A. Loscocco and Stephen A. Smalley. "Meeting Critical Security Objectives with Security-Enhanced Linux." 2001 Ottowa Linux Symposium. http://www.nsa.gov/research/_files/selinux/papers/ottawa01.pdf

[39] Avi Kivity et al. "kvm: the Linux Virtual Machine Monitor." 2007 Ottowa Linux Symposium. http://www.linux-kvm.com/sites/default/files/kivity-Reprint.pdf