KVMSandbox: Application-Level Sandboxing with x86 Hardware Virtualization and KVM

Andrew Ayer

1 Introduction

Security is one of the most important concerns in computing today. Despite this, the most common operating system security model has remained relatively unchanged over the last several decades. Unix-style file permissions, which are essentially a limited form of ACL-based authorization, are still the most pervasive model today and are used by modern operating systems like Linux and Mac OS X. More general ACL-based authorization, which allows permissions to be applied to any arbitrary user (instead of just classes like "owner," "group," and "other") is used on Windows.

However, this security model leaves much to be desired. For example, Unix permissions control access only to files, neglecting other resources like network access. Furthermore, Unix-style permissions dictate what a particular *user* can access, rather than what a *process* can access. This is often insufficient, especially when a user may be running programs that are either untrustworthy or insecure. Although an application may need access only to a subset of resources, it runs with the full privileges of the user executing the program. This is especially pertinent in the Internet era, when users regularly download and run potentially untrustworthy programs. Even trustworthy applications like web browsers and PDF readers constantly come into contact with untrusted data, and a single coding error can open the user to attack.

Various extensions to the model have been proposed over the years. SELinux [7] is probably the most featureful and well-known. SELinux adds hooks to resource-related functions in the kernel to enforce additional access control. It allows an administrator to set a centrally-controlled mandatory access control policy for all resources in the system. Unlike Unix permissions, SELinux policies can apply to individual processes as well as individual users. AppArmor [1] is an emerging alternative to SELinux. Like SELinux, it enforces centrally-administered mandatory access control via hooks added to the kernel.

Another approach is application-level sandboxing, in which a sandbox interposes a process' resource accesses, and applies alternative access control mechanisms. Application-level sandboxing is appealing because it doesn't require cumbersome central management like SELinux or AppArmor. Past attempts at sandboxing have adopted various approaches. Systrace [2] uses syscall tracing facilities, such as Linux's ptrace, to interpose system calls. Peng [9] modified Linux to add a "restricted exec" syscall and modified system calls like `open` to enforce security checks when called from a restricted process. Myers [8] developed a system in which sandboxed applications run under an operating system in VMWare and access files over NFS. A modified userspace NFS server on the host enforces filesystem security policy. Vx32 [6] uses application-level software virtualization to execute guests in a virtualized sandbox.

None of these solutions are completely satisfactory. Pure userspace solutions like Systrace and vx32 are appealing because they can be developed and deployed more quickly and easily than in-kernel systems like SELinux. However, Systrace is limited by the capabilities of ptrace and is vulnerable to a variety of race condition-based attacks. Meanwhile, vx32 uses software virtualization with dynamic code translation, which is complex, imposes overhead, and relies on legacy x86 features like segmentation.

An emerging alternative to software virtualization on x86 is hardware virtualization. While CPU support for trap-and-emulate virtualization was common in the pre-x86 era, it was absent in x86 until Intel released

the first CPU with "VT-x" in 2005. AMD followed suit in 2006 with "AMD-V." [5] Since then, increasingly more models of x86 CPUs have shipped with hardware virtualization support, and Intel and AMD have continued to improve the technology. Software has begun to take advantage of hardware virtualization. For example, Windows 7's XP compatibility mode requires hardware virtualization, as does VMWare to host 64-bit guests.

KVM is Linux's hardware-neutral interface to hardware virtualization. A userspace process with access to the KVM device `/dev/kvm` can open the device, and get and set the virtual processor state via various `ioctl` calls. Once the program begins executing the guest, the kernel does not return control to the process until there is an exit event, such as I/O, that requires emulation. KVM is most commonly used for full-system virtualization, in which a BIOS, boot loader, kernel, and userspace all run inside the virtual machine. The most notable example is the qemu emulator[3]. A system like qemu makes full use of CPU features in the guest, such as multiple rings and interrupts. However, KVM can also be used to virtualize a single process running in ring 3.

KVMSandbox uses KVM to offer the best-of-all-worlds for application sandboxing on Linux. KVM-Sandbox uses KVM to efficiently virtualize a single process running in ring 3. The process has its own virtual memory space and is completely isolated from the rest of the system. System calls are intercepted by KVMSandbox, security audited, and forwarded to the host kernel.

2 Sandbox Model

A process running inside KVMSandbox should run with the standard Unix permissions of the user running the sandbox, further constrained by a security policy specified by the user. The security policy should be able to provide fine-grained control over specific operations (like reading or writing) on individual file paths and network addresses. Any resource access made inside the sandbox, whether to a file or to a socket, should be checked against the security policy before being passed to the host kernel, where standard Unix permissions would apply. With this security model, the sandbox should be able to protect against an application that attempts to access files or make network connections that are not necessary for its desired functionality.

The sandboxed application should not be aware that it is running inside a sandbox. Apart from some operations returning access denied errors contrary to the standard semantics of Unix permissions, all other operations should perform identically as they would outside the sandbox. Any Linux program should be able to run inside the sandbox unmodified.

There should be no way for a process to escape out of the sandbox or gain access to resources which are denied by the security policy. Besides proper checking of syscall arguments, KVMSandbox must protect its virtual memory and file descriptors from tampering by the guest. The sandbox must also be resilient against shared memory attacks, where one guest thread manipulates the memory which KVMSandbox is auditing in another thread.

3 Implementation

3.1 Overview

KVMSandbox is implemented as a userspace command-line program written in C++11. On the command-line to KVMSandbox, the user specifies the path to the security policy file, the path to the program to execute, and the arguments to be passed to the program. The program to execute must be a 32-bit ELF executable. The program does not need to be specially compiled for KVMSandbox, and can be either statically or dynamically linked (though if it is dynamically linked, the security policy must allow the application to read the shared libraries).

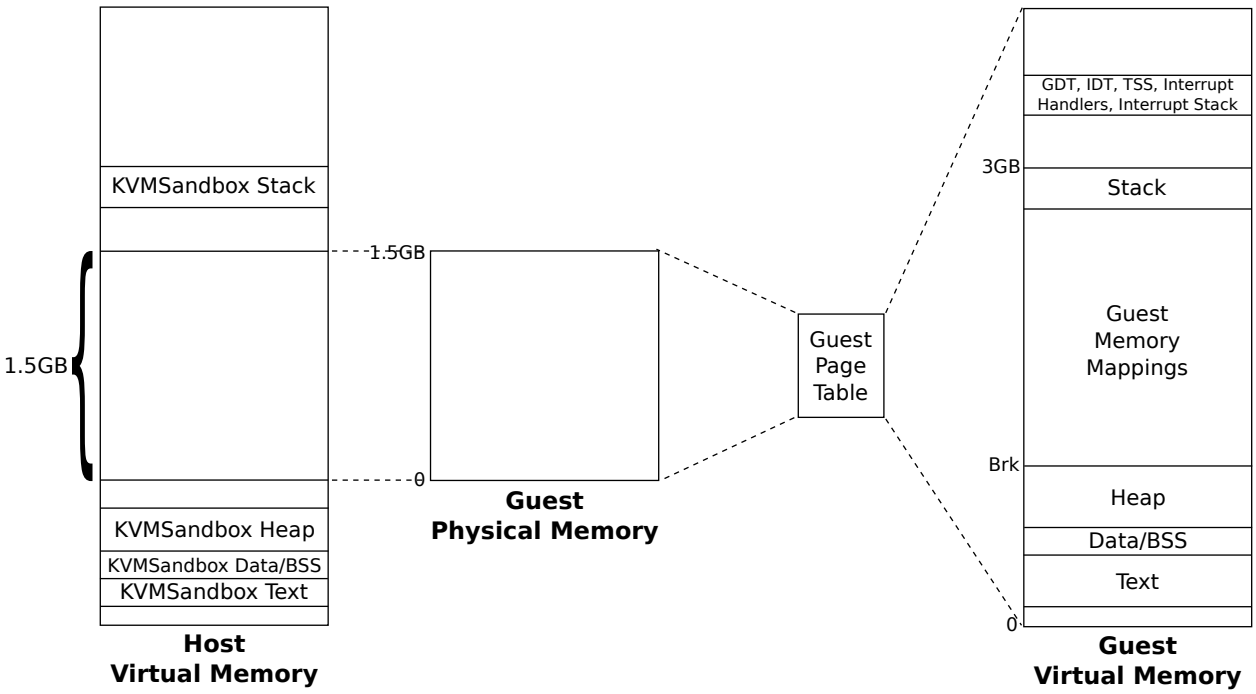

Figure 1: Memory layout in KVMSandbox.

After KVMSandbox has initialized the sandbox environment and loaded the guest program into the guest virtual address space, it instructs KVM to enter the guest. The program executes inside the KVM virtual machine, returning control to KVMSandbox only when a syscall is executed. The guest program has access to the file descriptors passed to KVMSandbox, allowing it to interact with the user over stdin and stdout, just like a natively-running program.

3.2 Virtual Memory

Address space is one of the most important resources to sandbox. Since the guest runs in the same host process as KVMSandbox, a breach of the sandbox would enable the guest program to access the host's address space, where it could alter the sandbox's internal state or inject arbitrary code that would be executed outside the sandbox. In addition, in order for guest programs to run unmodified, it is necessary to present to the guest what appears to be its own, complete 32-bit virtual address space.

KVM is used to give the guest program its own, isolated, address space. Since KVM is tailored for full-system x86 virtualization, there are in fact two address spaces within the guest: the arbitrarily-sized guest physical address space, and the 32-bit guest virtual address space. Translation between guest virtual addresses and guest physical addresses is handled by the CPU like a real x86 system: by using the page table pointed to by the guest's CR3 register. The guest physical memory resides in a contiguous region of KVMSandbox's virtual address space (hereinafter referred to as the host virtual address space). Translation between guest physical addresses and host virtual addresses can be accomplished by adding or subtracting the base address of the contiguous region. See Figure 1 for a pictorial representation of the various address spaces.

KVM provides an interface for mapping a contiguous region of guest physical addresses (at any length and offset) to a contiguous region of host virtual addresses. Upon initialization, KVMSandbox `mmap`s a writable, anonymous, and private region of roughly 1.5GB in the host virtual address space, and sets this as

the guest's physical address space, starting at guest physical address 0.

When the guest requests a virtual memory mapping, such as via the `mmap` system call, a region of the same size as the requested mapping is allocated from the guest physical memory. The allocation is done with a free list and a simple first-fit allocation strategy. The address of the allocated guest physical memory is converted to a host virtual address, and KVMSandbox executes a host `mmap` call to request that a fixed mapping of the requested resource be placed at this host virtual address.

At this point, the requested resource is visible in the guest's physical address space. To make it visible to the guest process, in its virtual address space, KVMSandbox adds appropriate entries to the guest's page table. Like a real operating system, if the guest program requests a fixed mapping, KVMSandbox places the mapping at the provided guest virtual address. Otherwise it finds an available region of the guest virtual address space and places the mapping there.

Page tables and page directories also must reside in the guest's physical memory. These are allocated as necessary by KVMSandbox using the physical memory free list.

When the guest asks to unmap a region of its virtual address space, KVMSandbox translates the guest virtual addresses to their corresponding guest physical addresses via the guest page table. It then translates the guest physical addresses to host virtual addresses and issues host `mmap` calls to map writable, anonymous, and private regions over the area. By the semantics of `mmap`, the host kernel will unmap the mappings that were previously occupying the region.

Care must be taken to ensure that KVMSandbox does not violate KVM's requirement that guest physical memory be backed by a writable and mapped region of host virtual address space. This poses a problem if the guest requests a read-only mapping. In this case, KVMSandbox asks the host to create a writable and private mapping, but unsets the write bit for the pages in the guest's page table to prevent the guest from actually writing to the region. The host mapping is made private in addition to writable since it is not possible to make a shared, writable mapping on a file descriptor not open for writing. Although the mapping is private, thanks to Linux's copy-on-write semantics, the region will behave exactly like a shared mapping until a write is made. Since the region is protected against writes by the guest, it will never be written absent a serious bug in KVMSandbox.

3.2.1 Internal Structures

KVMSandbox creates several internal structures in the guest's virtual address space, such as the GDT, IDT, TSS, and several interrupt handlers. Space for this data is allocated from the guest's physical memory, and mapped to high virtual addresses (above 3GB, the highest valid userspace address in 32-bit Linux) via the guest page table. To protect against modification by the guest program, the page table entries do not have the user flag set.

3.2.2 Guest Physical Memory Size

The size of the guest's physical memory, which defaults to 1.5GB, is configurable, but since it must reside in a contiguous region of host virtual memory, it must not be too large for the host's virtual address space, which also contains the stack, heap, text, and data of the KVMSandbox process. Since pages are always resident in the guest, the total size of the guest's virtual memory mappings is limited by the size of the guest's physical memory. Therefore, a sandboxed application will never be able to use as much virtual memory as a non-sandboxed application. However, in practice, few 32-bit applications need to use that much virtual memory. And on a 64-bit system (see Section 6.3), the virtual memory space would be vast enough that a large enough physical address space could be allocated for the guest.

Note that while guest pages are always resident in guest physical memory, since guest physical memory resides in host virtual memory, they may be paged in and out of host physical memory by the host kernel.

Also, thanks to demand paging, the large amount of anonymous memory that is mapped for the guest physical memory does not actually occupy host physical memory until it is used.

3.3 System Calls

Since system calls cannot be handled inside the sandbox, when a guest makes a syscall, it must exit to the host, where the arguments can be validated and forwarded to the real host syscall.

One method of executing system calls under Linux is for the userspace to generate software interrupt 0x80, which is handled by an interrupt handler in kernel land.[1] The syscall number is passed in register `%rax`, and up to six arguments are passed in registers. To force the guest to exit to the host, KVMSandbox writes a short interrupt handler for interrupt 0x80 into the guest address space and installs it in the guest's interrupt descriptor table. The two instruction, three byte interrupt handler executes a port I/O instruction and then returns from the interrupt. When executed, the port I/O instruction forces a VM exit, returning control to the host, where the syscall is handled by examining the register state of the guest.

Most system calls can be handled by calling the corresponding syscall in the host, with some minor pre- and post-processing which is described in detail below. Other system calls, such as `mmap`, alter primarily the state of the sandbox and thus are handled specially by KVMSandbox. `mmap` in particular is described in Section 3.2 above.

3.3.1 Argument Translation

The greatest challenge in executing system calls is validating and translating pointer arguments from guest virtual addresses to host virtual addresses before forwarding the arguments to the host syscall. For efficiency, copying of large buffers must be avoided. Furthermore, the sandbox must be robust and secure against another thread remapping the guest virtual address space or changing the contents of a string or data structure after it has been audited but before it has been used.

Any pointer argument whose contents need to be security audited is copied from guest virtual memory to temporary memory in host virtual memory. This is unavoidable if KVMSandbox is to be safe against time-of-check-time-of-use vulnerabilities. Fortunately, the arguments that require auditing are usually relatively small in size, such as null-terminated strings representing file paths.

Pointer arguments that do not need their contents audited (such as the `struct timeval* timeout` argument to `select`, or the buffer passed to `write`) still need to be checked to ensure that the entire range of guest virtual addresses is valid. For non-arrays, the size of the argument type is used to determine the address range. For arrays, the length of the array is usually specified by some other argument to the syscall. Null-terminated arrays are always copied out of guest virtual memory, as this is the only way to prevent accessing unchecked addresses if a guest thread removes the null terminator after the range has been checked but before it is used. The vast majority of null-terminated array arguments are file path strings, which need to be copied for auditing anyways.

Once the guest virtual address range is checked, the address must be translated to a host virtual address for passing to the host syscall. If the range does not cross a page boundary, the translation is accomplished by looking up the page in the guest page table and converting the resulting guest physical address to a host virtual address. However, if the range crosses a page boundary, it is possible that some guest virtual pages may map to non-contiguous guest physical pages (and by extension, non-contiguous host virtual pages). In this case, it is necessary to make a copy of the guest virtual memory in a contiguous range of temporary host virtual memory. This is potentially quite costly, especially for buffers. However, this would happen

[1] Newer versions of Linux use the new SYSENTER and SYSEXIT x86 instructions, if available, to avoid the overhead of a software interrupt. The current implementation of KVMSandbox uses the interrupt 0x80 method because it's simpler, though the design could be adapted to work with SYSENTER and SYSEXIT.

only if the memory crosses different virtual memory mappings in the guest, since KVMSandbox allocates contiguous guest physical memory for each mapping. This is probably unlikely in most programs, since, for example, the stack is a single mapping. If copying is necessary, KVMSandbox considers whether the argument is for input, output, or both, and only copies in the necessary direction(s).

3.3.2 Syscall Argument Metalanguage

Linux, as of version 3.3, has over 300 syscalls. Writing code by hand for each and every syscall, even if brief, would be extraordinarily cumbersome and hamper maintainability. Thus, KVMSandbox makes extensive use of C++ template metaprogramming to automatically generate syscall forwarders based on annotations of argument types.

For example, the code to create the syscall forwarder for the `read` system call is:

```
syscalls[3] = new Auto_syscall<fd_arg, out_arg<array_arg<char, 2>>, size_t>(3);
```

This line installs an automatically-generated system call forwarder with number 3 (`read`). The syscall takes three arguments: a file descriptor, a `char` array used for output whose length is specified by the syscall's 2nd argument (counting from 0), and a `size_t`. At compile-time, templates generate the appropriate code for checking the arguments, translating them, and calling the host syscall.

Each template parameter of the `Auto_syscall` class represents an argument to the syscall and can be any type. Additionally, the type can be wrapped in one or more of the following annotations to specify various properties of the argument:

- `array_arg<Type, index>` — the argument is an array of the given type whose length is specified by the integer argument with the given index
- `fixed_array_arg<Type, length>` — the argument is an array of the given type and length
- `nullterm_array_arg<Type>` — the argument is a null-terminated array of the given type
- `null_ok_arg<Type>` — the argument may be `NULL`, in which case `NULL` is forwarded to the host kernel instead of being considered an invalid guest virtual address
- `in_arg<Type>` — the argument is used for input to the syscall (this is the default)
- `out_arg<Type>` — the argument is used for output from the syscall
- `inout_arg<Type>` — the argument is used for both input and output

These annotations alter the generated code accordingly. For instance, the guest virtual address passed to an `out_arg` or `inout_arg` argument is checked for writability in the guest page table, whereas an `in_arg` argument is not.

Types requiring special processing can have their handling overridden specifically. For example, the `struct iovec*` arguments used with the `readv` and `writev` family of syscalls require deep translation of addresses.

Furthermore, there are several pseudo-types which indicate that special handling is required. For example, the type `fd_arg` indicates that the argument is an `int` representing a file descriptor that needs to be security audited. Likewise, `path_arg` indicates that the argument is a `char*` pointing to a filesystem path that needs to be audited. There is a standard interface for implementing pseudo-types, making it easy to add additional types.

This framework makes it easy to implement new syscalls. Once a common set of pseudo-type and type handlers have been implemented, adding a syscall forwarder is as simple as writing a single line of code describing the syscall's arguments.

3.3.3 File Descriptors

KVMSandbox opens several file descriptors for communicating with KVM in the kernel. It is important to prevent the guest from manipulating these file descriptors. On startup, KVMSandbox queries the kernel for the maximum file descriptor, and uses `dup2` to give its internal file descriptors high numbers near the top of the space. File descriptor arguments to system calls are checked to ensure they are not one of the internal descriptors. If so, the system call returns with errno `EBADF`, as if the file descriptor were simply not open. Furthermore, KVMSandbox intercepts `getrlimit` system calls and returns an artificially smaller value for the maximum file descriptor, so the guest should not expect to be able to use these file descriptors.

3.3.4 Processes

The `execve` and `fork` system calls require special handling.[2] `fork` is mostly straightforward: it calls the host `fork`, creating a copy of the KVMSandbox process and hence the entire state of the virtual machine, including guest memory. However, due to limitations in Linux, in-kernel KVM state is not duplicated by a fork. Thus, KVMSandbox saves the state of the virtual CPU (registers, etc.) before the fork, and once in the child process, closes, re-opens, and re-initializes KVM using the saved state, before returning from the syscall and resuming guest execution.

`execve` executes a program either in the sandbox or out of the sandbox according to the security policy, as detailed in Section 3.4. If the program is to be executed outside of the sandbox, KVMSandbox simply forwards the arguments to the host `execve`, and the new program completely replaces the KVMSandbox process. If the program is to be executed within the sandbox, KVMSandbox calls `execve` on the path to the KVMSandbox binary. The arguments to `execve` are passed as arguments to KVMSandbox, so the new KVMSandbox process executes the program with the given arguments inside a new sandbox.

3.4 Security Policy

3.4.1 Specifying the Security Policy

KVMSandbox controls access to the filesystem and network via a security policy file. This file, whose path is specified on the command line when launching the sandbox, contains a list of objects (file system paths or socket addresses) and, for each object, a list of capabilities either granted to, or revoked from, the sandboxed process.

Filesystem paths are specified in the framework as extended regular expressions, allowing rules to easily match multiple paths. Paths are used (as opposed to inode numbers) for ease-of-use and ease-of-implementation. The capabilities which apply to file paths are:

- `READ` — the file can be opened for reading
- `WRITE` — the file can be opened for writing, or truncated with `truncate`
- `CREATE` — (directories only) new files and directories can be created within this directory
- `REMOVE` — the file or directory can be removed
- `CHATTR` — attributes of the file or directory, such as times, ownership, and mode, can be changed
- `RENAME` — the file or directory can be renamed

[2] Note that modern Linux systems do not use the `fork` syscall. Rather, the library function `fork` calls the more general `clone` syscall with arguments that specify the behavior of `fork`. Thus, KVMSandbox actually handles the `clone` syscall. In this section, `fork` is used to mean `clone` with the appropriate arguments.

- `LINK` — a hard link can be created to this file
- `SYMLINK` — a symbolic link can be created to this path

Network access is controlled by specifying socket addresses and associated capabilities. Addresses can be remote addresses to which the process connects or sends data, or local addresses to which the process binds. In the current implementation, only INET family (i.e. IPv4) addresses are supported (attempts to access other types of sockets are denied), though support for additional socket families would be easy to add. INET addresses are specified as an address, netmask, and port number. A port number of 0 matches any port number, and a short netmask can be used to match ranges of addresses.

Capabilities which apply to socket addresses are:

- `BIND` — the process can bind to this address (with the `bind` syscall)
- `SEND` — the process can send to this address (with `sendto`)
- `CONNECT` — the process can connect to this address (with `connect`)

Any capability (for either files or socket addresses) can be prepended with a "`-`" character in the policy file to explicitly revoke the capability from the object. The special capability `ALL` represents all applicable capabilities.

Since it is possible that a file path or socket address could match multiple rules in the policy file, each rule is tried in the order specified until a rule is found that both matches the path/address and has either allowed or denied the capability needed to complete the operation. This way, it is easy to have broad, catch-all rules near the end of the policy, but grant or revoke specific capabilities on specific objects as needed with earlier rules.

If no rule matches, the default action is to deny.

An example security policy is presented in Section 4.3.

3.4.2 `execve` Permissions

In addition to specifying file and network capabilities, the policy can specify the behavior when the process executes another program with `execve`. Executable paths are specified with extended regular expressions, and can be given one of the following behaviors:

- `DENY` — the process is not allowed to execute the file
- `ALLOW` — the process is allowed to execute the file, and the file will be executed natively, outside of the sandbox
- `SANDBOX` — the process is allowed to execute the file, and the file will be executed within a sandbox using the same security policy as the current sandbox
- `SANDBOX` *policypath* — the process is allowed to execute the file, and the file will be executed within a sandbox using the specified security policy

See Section 3.3.4 for details on how `execve` is implemented.

3.4.3 Implementation Details

When executing a syscall, any argument of type `path_arg` is converted by KVMSandbox to an absolute, canonicalized path with no redundant slashes and no `.` or `..` components. This is the path name used to query the security policy, and is also the path name passed to the host system call. The type of syscall being invoked determines the capability which is sought. For example, `open` looks for `READ`, `WRITE`, or both depending on the flags argument, whereas `unlink` looks for the `REMOVE` capability. For some syscalls, such as `creat`, the parent directory path is also matched against the security policy for the `CREATE` capability.

Socket address arguments that need to be audited have type `sockaddr_arg`, and are checked in a similar fashion. Not all syscalls need their socket address arguments checked; for example `getsockname`'s `struct sockaddr*` argument is used for output, not input, and thus does not need to be checked.

3.4.4 Limitations

Since hard links essentially allow an existing file to be accessed with a different name, care must be taken when granting the `LINK` capability on a file. An attacker could hard link a file to a path for which the sandbox has additional capabilities, and access the file with these unintended capabilities. Similar care must be taken with `RENAME`.

Symbolic links pose particular hazards, since, although the path to the symbolic link itself is audited by KVMSandbox, when the path is opened, the kernel follows the symbolic link and opens the target, bypassing the KVMSandbox security policy. In the current implementation, it is not possible for KVMSandbox to securely audit the target of the symbolic link: it would need to check every single path component. This cannot be done without introducing a race condition in which an attacker swaps out a directory with a symbolic link after it has been audited but before the path is used. (But see Section 6.5 for possible solutions.)

Instead, the user of the sandbox must be aware of symlinks when granting capabilities. The user should avoid granting capabilities on symlinks with relative targets, as their actual destinations can change if the symlink, or one of its ascendant directories, is moved. Finally, the guest's ability to create symlinks must be severely restricted. For this reason, the target of the symlink is checked in the security policy for the `SYMLINK` capability. In addition, relative symlink targets are converted to absolute paths to ensure symlinks can't change meaning if moved. Strictly speaking, these are not correct semantics and could break a guest whose functionality depends on relative symlink targets.

There is no way, in the current implementation, to check the file descriptor argument to the `fchmod` and `fchown` syscalls to determine if the process has the `CHATTR` capability on the file referenced by the file descriptor. These syscalls are like `chmod` and `chown` except they operate on an open file descriptor rather than a path. Since the ability to obtain a file descriptor to a file is governed by the `READ` and `WRITE` capabilities, the guest can effectively chmod or chown any path to which it has a `READ` or `WRITE` capability. One possible solution would be for KVMSandbox to record the path used to open all file descriptors, and keep track of this information for the lifetime of the file descriptor. This is explored in depth in Section 6.4.

For similar reasons, the current implementation cannot securely support the `*at` family of syscalls (`openat`, `mkdirat`, etc.). These syscalls function like their non-at equivalents, except they also take an open file descriptor to a directory from which the file path is relative. Since KVMSandbox does not know the path which a file descriptor represents, it cannot construct an absolute, canonicalized path to look up in the security policy. For this reason, KVMSandbox does not support any of these syscalls. Since they are relatively new and Linux-specific, very few applications use them.

4 Evaluation

KVMSandbox has been tested with a variety of single-threaded programs on Linux. It works with the standard suite of command line programs, like grep, cat, and sed, as well as the gcc compiler and several graphical X programs, like xpdf (a PDF viewer), xv (an image viewer), and Dillo (a bare bones web browser). There is no noticeable performance degradation in any of these X programs, even as xpdf is used to browse complex PDFs, or xv is used to view large directories of high-resolution photos. xpdf is a compelling use case since PDFs are a common attack vector, and PDF interpreters are constantly found to contain security flaws. Similarly, web browsers are often the target of security exploits.

4.1 Benchmarks

KVMSandbox's performance has been evaluated with several benchmarks. Each benchmark was run both natively and within KVMSandbox to measure the slowdown caused by KVMSandbox. The benchmarks were run on a machine with a 2.66GHz Intel Core 2 Duo E6700 CPU and 4GB of RAM, running Linux kernel 2.6.33 and glibc 2.11.

The benchmarks are:

- primes — Calculates the first 300,000 primes. Does not make syscalls. Is heavily CPU- and memory-bound.
- factor — Factors the product of two large prime numbers. Does not make syscalls. Is heavily CPU-bound.
- bzip2 — Compresses a 1.7GB XML file with bzip2. Combines a CPU-intensive operation with many read and write system calls, which tests the efficiency of KVMSandbox's syscall forwarding mechanism.
- grep — Recursively greps for a string in the Linux 2.6 kernel source tree (approximately 39,000 files). Makes many filesystem syscalls as directories are scanned and files opened, which tests the efficiency of KVMSandbox's security policy code.
- mmapgrep — Recursively greps for a string in the Linux 2.6 kernel source tree, with the `--mmap` option to use mmap instead of read. Like the grep benchmark but makes many mmap syscalls. This tests the performance of KVMSandbox's VM system.
- syscall — Runs the close syscall in a loop 1,000,000 times. Tests performance of syscall handling and nothing else.

These benchmarks were chosen because they exercise a wide variety of different scenarios. For example, primes and factors are purely computational, whereas syscall makes only syscalls, and bzip2 and grep are in-between.

Results for the first five benchmarks are displayed in Figure 2. The overhead for the purely computational benchmarks (primes and factor) are negligible at less than 1%. This is expected, as purely computational code runs directly on the CPU in guest mode and never needs to exit to the host. The tiny overhead is likely due to the increased initialization cost of setting up the sandbox.

Overhead for the more I/O-intensive benchmarks (bzip2, grep, and mmapgrep) is modest at under 5%. Some overhead is to be expected due to the guest exit and the argument validation and translation that must occur upon every syscall. To help understand the runtime, the bzip2 benchmark was run again with some statistics collection to count the number of pointer arguments that were translated and copied. In

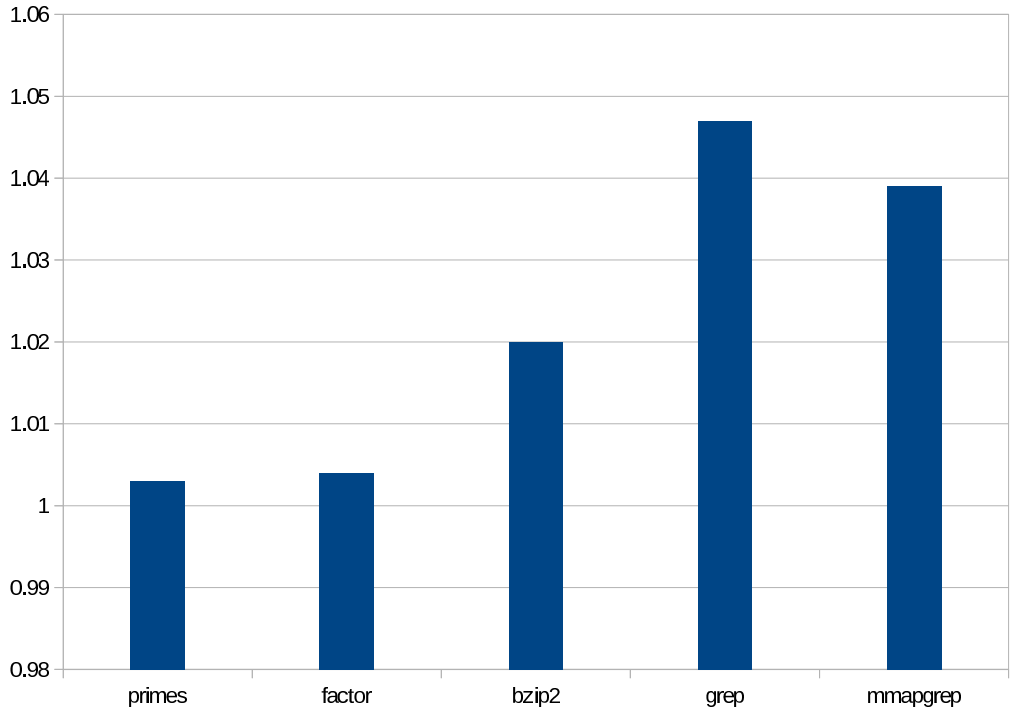

Figure 2: Normalized runtimes for benchmarks running under KVMSandbox. Each bar plots the runtime under KVMSandbox divided by the runtime for the same benchmark running natively (shorter bars mean faster KVMSandbox runs).

total, 548,950 pointer arguments passed through KVMSandbox. Of those, 3.7% needed to be copied via a temporary buffer because the guest virtual memory translated to non-contiguous host memory.

The syscall benchmark had vastly different results. Under KVMSandbox, it ran nearly 22 times slower than native speed. This indicates that there is significant overhead involved in handling syscalls. However, this benchmark represents an extraordinarily unlikely use case. A real program is comprised of far more than just syscalls, and the more realistic bzip2 and grep benchmarks show much lower overhead.

4.2 Comparison with Ptrace-based Sandboxes

KVMSandbox should be substantially more secure than ptrace-based sandboxes like Systrace. Ptrace is a Linux-specific mechanism that allows one process to intercept the system calls made by another. The monitor program can examine the syscall arguments and prevent the syscall or rewrite the arguments. Ptrace-based sandboxes use this technique to apply alternative security policies. However, ptrace-based sandboxes suffer from one crucial limitation: the host syscall must ultimately be made from the sandboxed process, meaning the arguments to the syscall point to addresses in a potentially malicious process' address space. This opens the door to race conditions in a threaded program: a thread could modify a syscall argument after it has been validated but before the syscall is made, bypassing the security checks. While not easy to exploit such a race condition, several practical techniques are described in [10].

KVMSandbox is immune to the problem because, when necessary to preserve security, it copies syscall arguments from guest virtual memory to host virtual memory, where they are safe from guest tampering. It is the copy that is audited and passed to the host syscall. Ptrace sandboxes cannot adopt a similar approach because a host syscall made from the monitor process would affect the state of the monitor process, not of the sandboxed process which made the syscall in the first place. KVMSandbox's key difference is that the

```
 1  file /etc/resolv.conf                          READ
 2  file /etc/services                             READ
 3  file /etc/hosts                                READ
 4  file /etc/host.conf                            READ
 5  file /etc/nsswitch.conf                        READ
 6  file /etc/fonts/fonts.conf                     READ
 7  file /var/cache/fontconfig.*                   READ
 8  file /lib/.*                                   READ
 9  file /usr/.*                                   READ
10
11  file /home/andrew/.dillo/.*                    ALL
12  file /home/andrew/.Xdefaults                   READ
13  file /home/andrew/.Xauthority                  READ
14
15  file /                                         READ
16  file /home                                     READ
17  file /home/andrew                              READ
18  file /home/andrew/Downloads                    READ
19  file /home/andrew/Downloads/.*                 CREATE WRITE
20
21  socket inet 0.0.0.0 0.0.0.0 53                 CONNECT SEND
22  socket inet 127.0.0.1 255.255.255.255 6023     CONNECT
23  socket inet 127.0.0.0 255.0.0.0 0              -ALL
24  socket inet 10.0.0.0 255.0.0.0 0               -ALL
25  socket inet 192.168.0.0 255.255.0.0 0          -ALL
26  socket inet 0.0.0.0 0.0.0.0 80                 CONNECT
27  socket inet 0.0.0.0 0.0.0.0 443                CONNECT
```

Figure 3: Example Security Policy for the Dillo Web Browser

monitor and the guest run in the same process, but there are parts of the address space that are isolated from the guest.

4.3 Security Policies

The utility of security policies was evaluated by writing a usable security policy for the Dillo web browser [4]. Dillo was used because it's single-threaded, unlike most other modern web browsers, and thus can run in the current implementation of KVMSandbox. Although it is much more minimalistic than browsers like Firefox, it should have similar resource access needs. For example, it uses sockets to connect to web servers, stores its preferences and data in a hidden directory in the user's home directory, and can download files to the filesystem. Thus, it is an adequate stand-in for studying browser security policies.

The final version of the security policy is shown in Figure 3. Lines 1–9 grant read access to various system locations containing configuration files, data directories, and libraries. Lines 11–13 grant access to configuration files in the user's home directory. Line 11 grants full access to a Dillo-specific directory, where cookies, cache, and configuration is stored. Lines 15–19 grant access to a Downloads directory, where Dillo can write (but not read) files. Lines 15–18 are not strictly necessary, but they allow the user to browse through the filesystem to the Downloads directory via Dillo's save file dialog box.

Lines 21–27 set the network policy. Line 21 allows Dillo to contact port 53 on any host so it can perform DNS queries. Line 22 allows Dillo to connect to the local X server. Lines 23–25 block access to localhost and systems with private IP addresses. Finally, lines 26–27 allow Dillo to connect to ports 80 and 443, the HTTP and HTTPS ports.

The policy was designed to minimize the damage if a security vulnerability in Dillo led to the execution of arbitrary code served by a malicious website. Such arbitrary code might attempt to read sensitive data and upload it to the attacker's server for nefarious purposes like identity theft. The security policy provides protection by only allowing reads to a few select locations, like system directories that contain no sensitive data. For convenience, the policy allows blanket access to `/lib` and `/usr`. On a modern Linux system, these directories should contain only files from software packages, and no user data.

Arbitrary code might also attempt to infect other parts of the system by writing to the user's configuration files or startup scripts. The security policy guards against this attack by allowing writes only to the user's `.dillo` and `Downloads` directories. By allowing writes to `.dillo`, an attacker could tamper with Dillo, but since Dillo runs within the sandbox, the attack still could not affect the rest of the system. The attacker could write arbitrary files to the `Downloads` directory, so a user must treat any file inside `Downloads` with suspicion, but this is already good practice when handling downloads from the Web.

An attacker might also use the host's network access as a backdoor into a firewalled network, or to turn the machine into a zombie in a botnet. The security policy protects against the former exploit by blocking access to IP address ranges that need protection. Of course, this comes with a trade-off, as it prevents the user from accessing web servers on the internal network. To protect against the botnet scenario, the security policy allows access to only the most common web server ports (80 and 443). This would prevent malicious code from sending spam or attacking non-web services. This also has a downside, as it blocks access to web servers running on alternative ports. A less paranoid security policy would block access only to frequently-attacked ports, like 25 (SMTP).

The one limitation of this security scheme is that while it does a good job protecting the system, it does nothing to protect Dillo itself. For example, by allowing blanket access to the `.dillo` directory, it enables malicious code to steal the user's browser history and cookies. This could be solved by running multiple instances of Dillo, each with its own configuration directory. (Or, for other browsers, by running multiple profiles; Firefox, for example, stores each profile in a distinct directory.)

A better solution would require changes to the web browser itself to ensure different websites are adequately segregated. This is beyond the scope of this work, but it is worth mentioning that a browser could use KVMSandbox internally to provide segregation. Google Chrome may be particularly adaptable to this, since it already runs each tab in a separate process. Running each tab's process inside KVMSandbox would provide added protection.

5 Conclusion

KVMSandbox is a userspace sandbox program that allows Linux binaries to run unmodified in a restricted environment. System calls are interposed by KVMSandbox and audited to ensure the guest is only allowed to access resources in accordance with a user-specified security policy. To provide isolation with minimal overhead, KVMSandbox uses hardware virtualization found in most new x86 processors via Linux's KVM interface.

KVMSandbox provides many advantages over existing solutions. It runs entirely in userspace with no need for elevated privileges or kernel modifications, allowing for easier development and deployment. While traditional userspace solutions have employed complicated and high-overhead dynamic code translation, KVMSandbox takes advantage of hardware virtualization, allowing for essentially zero overhead on pure computation. Though handling syscalls is expensive, KVMSandbox adds no more than 5% overhead in practical benchmarks.

6 Future Work

6.1 Trace and Interactive Modes

A simple but user-friendly enhancement would be to prompt the user interactively to allow or deny actions by the guest program that aren't covered by the security policy. This would save the user from having to anticipate every desired permission in advance. One advantage of KVMSandbox being implemented in userspace and not the kernel is that it is easy to implement features that interact with the user.

Another user-friendly feature would be a mode that allows all actions by the application but records a trace of them to a file. This file could be used as the basis for a security policy for the application.

6.2 Thread Support

6.2.1 Safety

The current implementation supports only single-threaded programs. Nevertheless, great care has been taken to make the design of KVMSandbox suitable for multi-threaded programs. Providing security in a multi-threaded environment is challenging because while one thread is executing a syscall whose arguments are being audited, another thread can modify the memory being audited. This is a race condition, and if exploited at the right time, can allow the application to bypass the sandbox's permissions checks. KVMSandbox is already invulnerable to this because it always copies arguments that need auditing to host-only address space. The copy, which is safe from guest meddling, is audited and passed to the host syscall.

For efficiency, KVMSandbox avoids copying syscall arguments whose contents need not be security-audited. Instead it passes to the host syscall an address that points directly into guest memory. During the execution of the host syscall, another thread could unmap the virtual guest page containing the argument. In turn, the guest physical page which backed the virtual page would be freed, and possibly re-allocated for a different purpose, such as to hold a guest page table. The host syscall would either read from or write to this re-purposed guest physical page. To prevent this, KVMSandbox needs a way to lock down guest physical pages being used during syscalls, to prevent those guest physical pages from being reused.

There are other edge cases to consider. For example, a guest could remap the virtual pages holding a syscall argument while another thread is in the host syscall. Since the address translation is done before executing the host syscall, the host syscall will read from or write to the old mapping. This cannot be used to compromise the sandbox (after all, the syscall might have taken place before the remapping occurred), though it might lead to unexpected results. However, this scenario represents undefined behavior, and no properly-written program would ever attempt it.

6.2.2 Implementation

First, any internal data structure in KVMSandbox that is accessed via syscalls would need to be protected with mutexes to prevent concurrent access from multiple threads.

Due to limitations in KVM, it is only possible to have one instance of KVM per address space. However, it is possible to have one virtual CPU per thread. Thus, KVMSandbox could support multi-threaded programs by creating one host thread for every guest thread, and running a virtual CPU in each. Unfortunately, KVM supports at most 256 virtual CPUs, limiting the number of threads per program to 256. In practice, this is likely sufficient for most programs.

6.3 64-bit Support

Although KVMSandbox has been implemented only for 32-bit x86 hosts and guests, a port to the x86-64 architecture should be straightforward. KVM supports x86-64, and no part of KVMSandbox's design or

approach wouldn't work in x86-64. Of course, x86-64 uses larger addresses and has a different page table structure, so KVMSandbox's virtual memory code would need to be modified accordingly. In addition, Linux uses a different set of syscalls on x86-64. These syscalls would need to be implemented for x86-64. This effort would be greatly facilitated by KVMSandbox's syscall framework (Section 3.3.2). The few syscalls that need manual implementations (like `mmap` and `execve`) would likely share much code with their 32-bit counterparts.

The possibility for 64-bit support gives KVMSandbox a huge advantage over software virtualization solutions like vx32, whose design depends heavily on segmentation, which does not exist in x86-64[6].

6.4 Virtual File Descriptors

Currently, KVMSandbox does not track guest file descriptors. Apart from the few KVM file descriptors that are isolated from the guest at the top of the file descriptor space, the guest and host share the same set of file descriptors. While simple, this has several disadvantages. First, once the guest has launched, KVMSandbox can no longer open file descriptors for internal use, as these file descriptors would become visible inside the guest. Second, in several places KVMSandbox would benefit from knowing additional details about a file descriptor, such as the type of object the file descriptor represents (a file, socket, terminal device, etc.).

KVMSandbox should virtualize the space of guest file descriptors and maintain a table of open file descriptors in the guest. In the table, KVMSandbox would store relevant information such as the corresponding host file descriptor. File descriptor syscall arguments and return values would need to be translated via the table, and any syscall that creates file descriptors would need to be specially handled in order to keep the table up-to-date.

Two features that would be enabled by such a system are described below.

6.5 Symbolic Link Security

As described in Section 3.4.4, symlinks pose a problem for security because KVMSandbox cannot safely check that no component of a file path is a symlink. If KVMSandbox could open file descriptors for internal use, it could validate paths by opening each component one-by-one using the `openat` syscall to avoid race conditions.

6.6 `ioctl` Support

`ioctl` is very difficult to handle because the type of its third argument depends not only on the ioctl number but also on the type of object represented by the file descriptor, since different drivers in Linux have overlapping ioctl numbers. Without information about the file descriptor, KVMSandbox doesn't know whether the third argument should be treated as a pointer or as an integer. If KVMSandbox tracked the types of file descriptors, it could use this information to properly handle `ioctl`.

References

[1] `http://wiki.apparmor.net/index.php/Main_Page`.

[2] `http://www.citi.umich.edu/u/provos/systrace/`.

[3] `http://www.qemu.org/`.

[4] `http://www.dillo.org`.

[5] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4):3–18, December 2010.

[6] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[7] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[8] Aaron Myers. Operating system protection domains, a new approach. Master's thesis, Brown University, 2008.

[9] Luke Peng. The sandbox: Improving file access security in the internet age. Master's thesis, Brown University, 2006.

[10] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies*, WOOT '07, pages 2:1–2:8, Berkeley, CA, USA, 2007. USENIX Association.