File Injection for Virtual Machine Boot Mechanisms

Till Müller, Johannes Naab*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: till.mueller@tum.de, naab@net.in.tum.de

***Abstract*—Virtual machines are widely used in today's computing infrastructure. They provide an easy way to isolate untrusted software and users from each other and the hosts they run on. This isolation, however, makes it difficult for administrators to exercise control over the VMs without compromising security.**

We provide a system that allows the admin to inject files into untrusted VMs in a secure manner. It uses a customized Linux kernel which is booted using QEMU direct kernel boot. To inject files safely, they are passed to the virtual machine via the `initramfs`, a read-only archive normally used to provide drivers to the kernel during boot. The file injector then uses `kexec` to load a kernel from the guest's filesystem, thereby assuming the functionality of a bootloader like GRUB to minimize user impact.

***Index Terms*—virtual machines, qemu, kvm, file injection, Linux kernel boot**

1. Motivation

The machines used in this project are untrusted virtual machines with potentially malicious actors having access as `root`. Nevertheless, these VMs need to be administered (i.e. to grant access or run updates), a non-trivial task to accomplish while not compromising security since most administration tools assume they have direct access to the machine that is being administrated. To protect the VMs' host, the file injector should not need to run any code originating from the VMs and overall enforce strict isolation.

This project has been motivated by the lecture "Grundlagen Rechnernetze und verteilte Systeme" [1] which requires such a tool to provide virtual machines for students taking the lecture.

For this lecture, every student is given access to a virtual machine to use for developing and debugging of homework assignments. The students log in as `root` on their respective machines, giving them full control over their VM. It is therefore paramount to keep these VMs isolated from the host system. All requirements and assumptions are based on this scenario.

This paper is structured as follows: In Section 2 we analyze the background of the technologies we used, as well as existing boot processes. The 3. Section evaluates different possible solutions to the issues described here. In Section 4 we present our bootloader and file injector and in Section 5 the performance of some boot methods is highlighted. Section 6 concludes the paper.

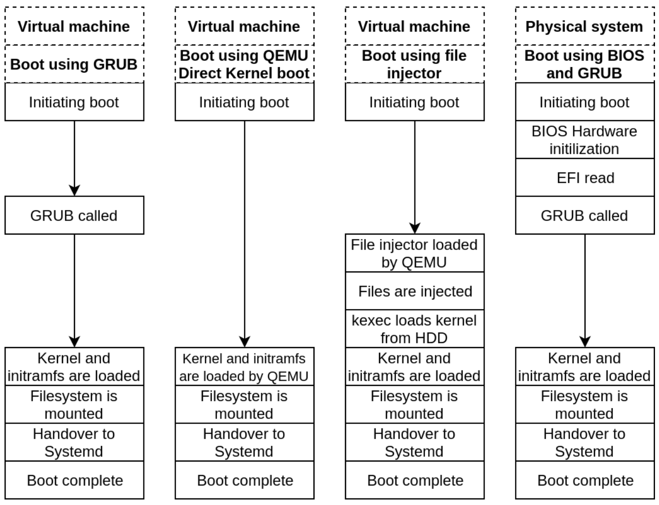

Figure 1: Flowchart of boot processes

1.1. Requirements and limitations

For our boot loader and file injection system we propose the following requirements:

- External data has to be written to the VMs' filesystem
- The VMs have to be persistent between reboots
- Boot times have to stay reasonable
- The system has to be able to scale to hundreds of machines
- Users should be able to update their VMs' kernel independently from the file injector
- The setup must not require a network connection to inject the files or boot the VMs
- In case the boot process fails, manual debugging should be possible

1.2. Assumptions

To make the implementation easier, the file injector is based on these assumptions:

- Virtual machines are hosted using QEMU/KVM with libvirt
- The VMs run Debian and are all set up in a similar way
- Only small configuration files need to be copied to the guests' filesystem

2. Background

In this section we will take a closer look at how a virtual machine boots using different methods. During this, we will also evaluate and explain how these technologies might be useful as part of the file injection system.

2.1. GRUB

A conventional boot loader like GRUB [2] or LILO gets invoked by the BIOS/EFI at the beginning of the machine's boot process. It then lets the user choose which operating system to boot. For this, GRUB reads the partition table, which contains information about what partitions are bootable and where on the disk they are located. Afterwards, GRUB loads the kernel from the chosen partition and hands over to it. The kernel then starts its first process, on Linux systems usually init or Systemd.

2.2. QEMU direct kernel boot

A different way of booting a QEMU-hosted virtual machine is the so-called direct kernel boot. In normal use, this enables a VM to skip the bootloader by using a kernel file from the host system. The virtual machine then jumps directly to the init or Systemd invocation, thereby not requiring a kernel to be present on the VMs HDD and speeding up boot-times. In the case of injecting files however, the direct kernel boot feature is used to boot up the file injector without the need to use the VMs' filesystem as intermediary storage, making the system less error-prone and resulting in no unwanted side-effects during this first boot stage (Fig. 1). This is possible because no part of a direct kernel boot requires the HDD, which is only mounted at the end of the init process.

2.3. initramfs

When booting a Linux kernel of almost any kind, two files are involved: The first is the Linux kernel itself, which contains the kernel code, the second is the *initial ram filesystem*. The `initramfs` contains all drivers and software needed to mount the filesystem from the HDD, e.g. RAID or decryption tools. It is a `cpio` archive which the kernel uncompresses into a temporary filesystem (TMPFS) during boot. This filesystem then resides entirely in the machine's memory, enabling the kernel to load modules required to continue booting.

The old format for this file was `initrd`, which behaved similar to a block device using a filesystem like `ext2`. Today, this format is rarely used, although the name regularly appears in commands, e.g. for QEMU direct kernel boot, even though an `initramfs` is used.

2.4. kexec

`kexec` is a program that emulates the function of a bootloader from within a running system. When running `kexec`, it initiates a normal shutdown, but immediately restarts the system using the chosen kernel. Therefore, no shutdown signal is sent to the motherboard and BIOS; hardware initialization and the bootloader are skipped. `kexec` is therefore similar to QEMU direct kernel boot: Both start the system with the kernel immediately available. In our implementation, `kexec` is used to load a user-provided kernel after the files have been injected.

3. Alternative approaches

Before the final decision on the implementation was made, other possible approaches to the problem had to be evaluated as well. We will, therefore, take a look at using existing systems, such as Virtio's `virt-install` or PXE.

3.1. Using direct kernel boot

The original setup we based this work on was similar to the one that was eventually chosen, with one major difference: Instead of using `kexec` to find and boot a kernel from the machine's filesystem, the injector would continue booting normally and therefore be the active kernel while the machine was running. This enabled a quick boot process (see benchmarks in Section 5).

The downside to this approach was that updating the kernel came with a lot of issues. The main one was that the kernel version installed inside the virtual machine and the one the machine was booted with had to be kept in sync.

This was required because the kernel loads version-specific modules from the machine's filesystem after it has been mounted. To load these modules, the kernel expects a folder named after the kernel version in `/lib/modules`. If this folder does not exist, the modules are not loaded. As a result, updating the kernel the machines were booted with was not an option since it would have led to all machines losing the functionality these modules provided (e.g. ACPI support).

Updating the kernel within the virtual machine did not have any effect due to the kernel being loaded during boot still being the same one originating from the host system. This could lead to user frustration, especially when building kernel modules, and while the manual usage of `kexec` could circumvent this limitation, a system is preferred that does not require such a workaround from the users.

3.2. virt-install

Virtio's `virt-install` is a tool to set up new virtual machines using a predefined image. When given a kickstart file, `virt-install` can make configuration changes and copy data to the machine.

While `virt-install` can import existing images, we were unable to find a way to for `virt-install` to alter the contents of the VMs disk image during this process. `virt-install` can edit a virtual machine's `virsh` configuration file, but this only allows it to change, for example, the connected devices or allocated resources to the VM. It was therefore ruled out after some initial testing in favor of the other ideas described here.

3.3. Mounting the guest's filesystem on the host

This approach was deemed too slow and insecure to use for untrusted VMs. While there should be no way for a mounted filesystem to execute any code, keeping the filesystems separated is the best way of ensuring isolation. Modern filesystem like `ext4` trust the images they mount, so a filesystem image crafted with malicious intentions could cause major security issues [3]. One solution here is using `libguestfs` [4], which mounts the guest's filesystem inside a minimal VM and therefore enables the host to securely alter the guest's HDD contents. This method, however, is unsuitable for our purpose, since the process would increase boot times significantly. Additionally, mounting and unmounting a filesystem with this method every time a VM boots can put additional load on the host, especially if multiple virtual machines are booted at the same time.

3.4. Network boot / PXE

The Preboot Execution Environment (PXE) is a standardized process of booting a physical or virtual machine over the network using a TFTP (trivial file transfer protocol) server. The PXE implementation by Syslinux [5] can transfer a kernel and `initramfs` file to the booting machine. Unfortunately, PXE requires a network connection and DHCP to connect to a server. Additionally, the `pxelinux` implementation does not provide tools for running scripts on boot.

After considering these issues, PXE was ruled out as a solution as well. While it might be possible to achieve the desired outcome using it, the infrastructure defined in the requirements does not provide DHCP, making PXE impossible to use.

3.5. Ansible

Another tool often used for applying configurations on multiple machines is Ansible [6]. It can automatically provision and set up physical and virtual machines, all configured using YAML files as so-called Playbooks. Like PXE however, Ansible needs a connection to its server to download these Playbooks, which makes it unsuitable for the requirements described in Section 1. Ansible also runs as an agent on the machines it manages, which would enable users to disable it, rendering it useless.

4. Architecture

All approaches listed in the previous section either do not fulfill some of the requirements or are not compatible with the limitations. The system described below was built upon some of them to achieve all desired results within the limits set in Section 1.

In summary, the file injector runs on the virtual machine before its real kernel is booted from the HDD, enabling full access while being controlled by the administrator.

4.1. Replacing the bootloader

Since the code to inject files needs to run before handing over control to the untrusted guest system, the injection process takes the place of the bootloader. QEMU direct kernel boot loads the injector and runs it inside the virtual machine. This behavior is not possible when using GRUB, since the injector and the files to inject are not located on the VM, but on the host's filesystem.

4.2. Injecting files

The file injector works in two stages:

1) Mount the machine's filesystem and inject the files
2) Find the correct kernel to switch to and execute it using `kexec`

During the first stage, the injector needs to find the `root` filesystem and mount it. Like a kernel during normal boot procedures, `mount` needs to know the label, UUID or path to the block device containing that filesystem. The type of the filesystem used is also required, although `mount` is sometimes able to get it from `blkid` or the `volume_id` library. Normally, `mount` would also try to read `/etc/filesystems` and `/proc/filesystems`, but these are not available in the `initramfs` environment [7]. The default values for both the block device's location and its filesystem type are set in a way that makes them compatible to most virtual machine setups. Additionally, it is important here that the guest's filesystem is mounted with read/write options. While `initramfs` normally mounts the filesystem as read-only, this would not be sufficient for file injection.

To enable injection, `initramfs` delivers the files the injector needs to copy to the guest's filesystem. While this limits the size of the files the system can inject, using `initramfs` is a fast, reliable and secure way to move files from the host's filesystem to the guest's.

4.3. Booting the correct kernel

After the first stage, the injector needs to load the right kernel to properly boot the system. During the second stage, the injector looks for a suitable kernel/`initramfs` combination to switch to. There are two possible locations for these files: They are either linked from `/vmlinuz` and `/initrd.img` or are located in `/boot`. The administrator can also set both manually using the boot parameters, as shown in Figure 2.

After the injector has found the kernel and `initrd.img`, it loads them using `kexec`, unmounts the filesystem and `kexec` switches to that kernel which continues the boot process.

If the system is unable to find a working kernel/`initrd.img` combination or `kexec` fails, it drops to a shell and automatically tries to start an SSH server so the issue can be debugged and resolved manually.

4.4. Implementation

Implementing the system as described above has produced several challenges that needed to be solved. Some of them are described below, along with possible solutions.

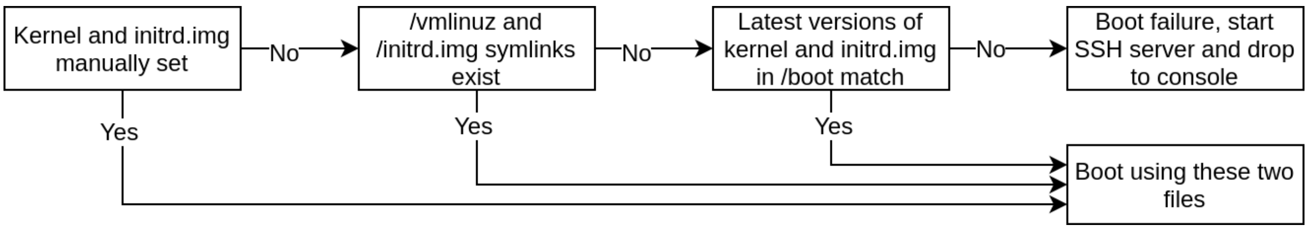
Kernel and initrd.img manually set
No
/vmlinuz and /initrd.img symlinks exist
No
Latest versions of kernel and initrd.img in /boot match
No
Boot failure, start SSH server and drop to console
Yes
Yes
Yes
Boot using these two files

Figure 2: Flowchart of how the kernel / initrd.img are chosen

4.4.1. Passing files to the injector. The main issue resulted from a combination of the requirements and limitations imposed by QEMU and its direct kernel boot-feature. To pass files to the injector and therefore the guest without a network connection or shared filesystem, they must be included in the `initramfs`. Creating a `cpio` archive and setting it as the `initramfs` for the QEMU direct kernel boot, however, is not sufficient. This is caused by the issue mentioned before (see Section 2.3) that the kernel itself does not contain all drivers to boot the system and requires an `initramfs` with additional modules.

This means that a customized `cpio` archive needs to be available for each virtual machine during the boot process. `cpio` archives can be concatenated, so simply combining the archives containing the files and drivers would result in one which contains all required files. This however creates another issue: The newly created archive would not only contain the files the injector needs to inject, but also about 27MB of static data required by the kernel.

Generating these archives on every boot for hundreds of machines is wasting time and space, so we focused on finding a way to circumvent this issue.

4.4.2. Including the static initramfs in the kernel file. Since most of the `initramfs` file is static, a possible solution is to include it with the other static file used during boot, namely the kernel. This solution proved to be the best one because it enables the `initramfs` given to QEMU to just include the files the system injects. When compiling the Linux kernel, one available option is `CONFIG_INITRAMFS_SOURCE`. The kernels shipped by Debian do not use this option, resulting in the kernel containing an empty `cpio` archive. Being able to include the static `initramfs` in the kernel though allows the injector to be one self-contained kernel file which includes everything needed to boot the system. The only downside to this approach is the added complexity from having to recompile the kernel to make changes to the file injector.

4.4.3. Adding the initramfs to a precompiled kernel. In theory, it is also possible to achieve the results from the previous section without having to recompile the kernel by editing the Linux kernel file (`vmlinuz`) to include a different `initramfs` from the one it was compiled with. However, making this work requires changes to multiple values within the kernel file [8] [9]. This means that not only the data has to be in the right place, but offsets and some pointers have to be updated as well. Therefore, this process was deemed too fragile to achieve the desired result.

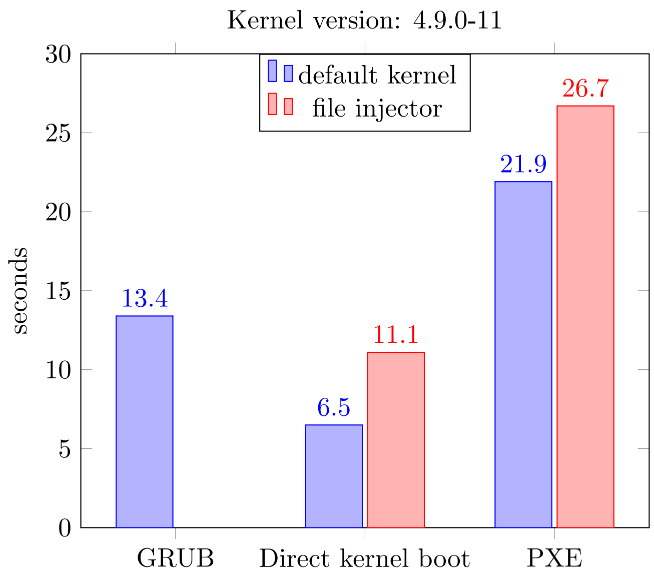

Figure 3: Boot times using an unaltered Debian kernel and the file injector

4.5. Executing the injector script

The `initramfs` that includes the file injector is based on a Debian `initramfs`, therefore requiring changes to adapt it for the injector. For example, some effort was needed to find the best way of getting the `init` process to execute the file injector script at the right time. Since `init` normally executes all scripts in the `/scripts/init-{premount,top,bottom}` folders as defined in the `ORDER` files placed in them, the init process was still executing the mounting mechanism normally required during boot. This could break the mount attempted by the injector, so the behavior had to be changed.

To accomplish this and to have more control over the `init` process, the `init` script was altered in the filesystem overlay to switch to the injector as soon as possible, effectively stopping at that point and making sure that it does not interfere with the file injection.

5. Benchmarks

A quick overall boot process was one of the goals while implementing this system. Therefore, we now take a look at how much longer booting takes while using the file injector. These tests were conducted using the same VM and the times were measured from sending the start command via `virsh` until the login screen was displayed. The results are shown in Figure 3.

Having GRUB boot an external file is outside its scope, so only the unchanged Debian kernel from the filesystem image was tested. Furthermore, the GRUB configuration was not altered, which resulted in the default five seconds delay during which the user can choose another operating system. Since this delay would exist on newly installed VMs as well, it was left in.

Even though the Preboot Execution environment was already ruled out in Section 3, we have included it here for comparison. It is immediately obvious that booting with `pxelinux` takes longer due to the need for a network connection. In this case, the DHCP- and PXE servers were located on the same host, so network latency or bandwidth limits are unlikely to have influenced these results.

The file injector increases boot times by about five seconds. The tests have been performed without any files to inject, so if the injector needs to copy large files, this would increase the boot time further. Copying large files, however, is not the intended use case for this implementation, we mainly focused on small configuration files. Booting a kernel directly is the fastest, but with the file injector using direct kernel boot being similar in boot time to using GRUB without file injection, the impact on users is negligible.

6. Conclusion

We developed a `kexec`-based boot loader for virtual machines. It is based on a modified Debian `initramfs`, which is directly embedded into a rebuilt kernel image. It allows to manage VMs by injecting configuration files and does not require mounting the guest's filesystem on the host. The file injector works reliably and will enable administrators to easily configure the untrusted systems hosted on their infrastructure.

Future work includes making the system more compatible to non-Debian distributions and adding functionality like deleting files in addition to injecting them, as well as passing arbitrary parameters to the user-kernel's command line.

References

[1] "Lecture 'Grundlagen Rechnernetze und verteilte Systeme'," https://www.net.in.tum.de/teaching/ss19/grnvs.html, [Online, accessed 19-September-2019].

[2] "GNU GRUB," https://www.gnu.org/software/grub/, [Online, accessed 26-September-2019].

[3] "On-disk format robustness requirements for new filesystems," https://lwn.net/Articles/796687/, [Online, accessed 25-September-2019].

[4] "libguestfs," http://libguestfs.org/, [Online, accessed 25-September-2019].

[5] "PXELINUX, official site," https://wiki.syslinux.org/wiki/index.php?title=PXELINUX, [Online, accessed 19-September-2019].

[6] "Ansible," https://www.ansible.com/, [Online, accessed 27-September-2019].

[7] "mount(8) - Linux man page," https://linux.die.net/man/8/mount, [Online, accessed 19-September-2019].

[8] "Stackexchange: Replacing section inside elf file," https://reverseengineering.stackexchange.com/questions/14607/replace-section-inside-elf-file, [Online, accessed 19-September-2019].

[9] "Gentoo forums: Linking existing kernel with new initramfs," https://forums.gentoo.org/viewtopic-t-1087792-start-0.html, [Online, accessed 19-September-2019].