Booting Linux Within lvisor

Nicholas Lunceford and Xinyu Sui
luncenic@uw.edu, suix2@uw.edu
University of Washington

ABSTRACT

This paper outlines the implementation of a Linux bootloader inside the lvisor hypervisor. Although this is not a novel task, it does give the chance to inspect how exactly Linux boots, as well as how to perform the handoff from the hypervisor into a production kernel. Additionally, since the existing documentation of Linux's long mode boot protocol is difficult to both find and understand, we will give a breakdown of the exact process necessary to boot Linux in 64-bit mode.

1. INTRODUCTION

lvisor was provided as a minimal hypervisor, capable of running a virtual machine with a single core, but restricted to a few toy operating systems. While the minimalism does make reading and comprehending the source code significantly easier than digging through a more fully featured hypervisor, it means that the practical usefulness of lvisor is almost nonexistent. As a result, we decided to add a Linux bootloader to lvisor capable of running Buildroot Linux kernels.

Although Linux has been successfully virtualized many times before, creating a bootloader, even one based on existing code, is not easy. As a result, a significant amount of this paper will be dedicated to describing the steps we found necessary in creating our bootloader, as well as steps we chose to omit for various reasons.

Aside from the Linux boot protocol itself, we will also go over the specific transition from lvisor to Linux using our bootloader. While this is less applicable to other implementations, whether they be based in other hypervisors or as a standalone bootloader, it does give an reference for how the specific platform affects a bootloader's implementation.

The reset of this paper is organized as follows. Section 2 discusses background and previous work and resources about booting Linux kernel in long mode. Section 3 gives an overview on lvisor. Section 4, as a concise interpretation of the long mode Linux boot protocol, goes through requirements and eliminates unnecessities for booting Linux. Section 5 presents our implementation in detail. Section 6 discusses the result of our implementation and some limitations we are still facing. Section 7 discusses some possible exploration in the future. Section 8 concludes.

2. RELATED WORK

2.1 Linux Boot Process

The official Linux boot documentation that we found focuses primarily on the real mode boot protocol. Although the 32- and 64-bit protocols are mentioned, they are not described in such detail [1]. However, much of the real mode documentation also applies to the 64-bit boot protocol, most notably the setup header described in section 4.2. While we will briefly describe the most important elements in this header, as well as the basics of booting Linux, the official documentation should be used as a reference or for additional detail.

2.2 bzImage Kernel

Linux kernel image is provided as a bzImage file. Since the Linux kernel itself is large compared to the real mode address space, the main kernel itself is compressed within the bzImage file [2]. Thus, the bzImage file contains the setup headers for booting Linux (described in more detail in section 4.2), real mode code, protected mode code to decompress the kernel proper, and then the compressed kernel itself [2].

2.3 Existing Hypervisors

Since Linux has been booted in a hypervisor before, one of our strongest reference points was another hypervisor. lguest64. While it is a 64-bit port of the lguest hypervisor, the bootloader it uses is still a protected-mode bootloader. However, it did provide a skeleton of code for us to work with in creating our bootloader, and it proved to be the foundation for our final efforts in creating the Linux bootloader.

3. LVISOR OVERVIEW

lvisor is run through QEMU on a Linux platform. After booting, it first initializes itself as the hypervisor, leveraging Intel's VT-x implementation. Once it is fully initialized, it instantiates and transfers control to its guest, which initially runs lvisor's own firmware code. As lvisor's initialization mentioned above is not the focus of this project, we will not go into more detail about it; instead, we will focus on lvisor's firmware, and the firmware's handoff to the actual guest OS. However, the host lvisor is responsible for creating the *guest_params* struct for use in

the firmware. This struct contains the critical information needed to boot the guest OS, including the location of the guest OS in memory, the e820 table that informs the guest of valid regions of memory, and the command line with which to initialize the guest OS.

3.1 Firmware Initialization

Since the firmware is the first code that runs as the guest, its first duty is to setup the virtualized processor and transition it from the initial real mode to the 64-bit long mode. This is done in assembly, and is located in firmware/head.S. Once the transition is done, the main function in firmware/main.c is executed. This function simply outputs some informational text, and calls the bootloaders implemented in lvisor.

3.2 Multiboot Bootloader

Although the multiboot loader plays no part in booting Linux itself, it served as a starting example for our efforts in creating a Linux bootloader. The multiboot bootloader's primary responsibility is to serve as the bridge between the lvisor host and the multiboot guest. While it naturally must actually hand execution off to the guest, it also must provide the information from lvisor to the guest by populating the multiboot headers. Finally, since the multiboot OSes for lvisor are 32-bit, the multiboot bootloader steps back into assembly to transition the processor back to 32-bit mode before finally jumping into the guest kernel. Once this is done, the guest kernel runs normally.

4. LINUX BOOTLOADER OVERVIEW

The Linux kernel contains a designated entry point for an x86 system already in 64-bit mode, but does have some requirements for the bootloader to fulfill before entering the kernel. The largest of these is the boot parameters structure, but the kernel also requires a valid Global Descriptor Table (GDT), as well as a few other miscellaneous mode requirements.

4.1 Boot Parameters

Linux defines a boot_params structure, also known as the "zero page", that must be utilized by the bootloader to boot Linux successfully. The default boot_params structure is located at the very beginning of the kernel image.

Some of the fields in this structure, such as the size of the setup code in the setup header, are intended to be read for information about the kernel image. And others, such as the commandline's location, are settings that can be specified by the bootloader. The fields we are interested in this project for booting Linux kernel are *ext_ramdisk_image*, *ext_ramdisk_size*, *ext_cmd_line_ptr*, *e820_entries*, *e820_table*, and the setup header (*hdr*). *ext_ramdisk_image*, *ext_ramdisk_size*, and *ext_cmd_line_ptr* are the high 32 bits for *ramdisk_image*, *ramdisk_size*, and *cmd_line_ptr* in the setup header (discussed in 4.2), for compatibility with 64-bit mode and such that all components needed to boot the kernel can be loaded in memory at an address above 4G. *e820_entries* and *e820_table*, let the bootloader to pass e820 mapping information to the OS.

4.2 Setup Header

Linux boot protocol [2] discussed the setup header in detail. Linux setup header was originally intended for real mode and then expanded to be compatible with protected mode. Therefore, there are some unnecessary or incomplete fields when the kernel is booted directly in long mode. Examples of fields that are unnecessary in the 64-bit protocol are *heap_end_ptr* (offset 0x224) and *loadflags* (offset 0x211). While not all fields are explored for their necessity, we will mostly discuss fields we consider minimally required to boot the kernel, and leave full exploration as a future work. As discussed in previous section, although required, *ramdisk_image* (offset 0x218), *ramdisk_size* (offset 0x21c), and *cmd_line_ptr* (offset 0x228) in the setup header are incomplete and only contains the low 32 bits when the kernel is booted in long mode. *ramdisk_image* and *ramdisk_size*, along with their extensions, contain the address and size respectively of the initial ramdisk, and must be filled out if a ramdisk is to be provided. Similarly, *cmd_line_ptr* and its extension contain a pointer to the string command line for the kernel. Even if a command line is not provided, this variable should point to either a null terminator, or, preferably, "auto".

Several fields in the header provide necessary information about the kernel image back to the bootloader. In particular, *header* (offset 0x202) contains magic number "HdrS", which can be used to identify that the file is a Linux kernel. Similarly, *version* (offset 0x206) identifies the Linux boot protocol version. Finally, *setup_sects* (offset 0x1f1) is the size of the setup code in 512-byte sectors, after which the protected mode code starts.

4.3 Other Requirements

Linux also requires a valid GDT to be loaded. Specifically, the code and data segments must be at least 4GB in size, with the code segment loaded at 0x10 in the GDT, and the data segment at 0x18 in the GDT. Additionally, the code segment must be read/execute, and the data segment must be read/write. Next, the CS register must be set to the code segment value, and the DS, ES, and SS registers must be set to the data segment value. Finally, interrupts must be disabled, and rsi must point to the prepared boot_params struct. Once this is done, the bootloader can jump to the 64-bit entry point, which is the beginning of the protected

mode code, calculable from the *setup_sects* value in the setup header, adding 0x200.

5. IMPLEMENTATION

The majority of development time was spent researching and looking through documentation related to the Linux booting process. The actual code written for this project was only a few tens of lines, and is wholly contained within the linux.c file within lvisor. This code is invoked when the multiboot loader already in lvisor fails to load a kernel, and the Linux bootloader is specifically designed to load bzImage files.

Bootloader works as follows. First, the boot parameters structure is allocated, and the existing headers within the bzImage are read into it. Then the magical number "HdrS" is checked to make sure the image file is valid for this bootloader. Next, the required boot parameters are filled in as necessary, including e820 mapping, ramdisk information, and command line pointer. Finally, the bootloader uses assembly to disable interrupts and jump to the kernel's entry point. Because lvisor's firmware automatically sets up a valid GDT, we do not need to set up our own GDT. Additionally, since the firmware code is in 64-bit mode, it can be written in C rather than assembly. While it would certainly be possible to initialize everything necessary in assembly, it would be significantly more difficult. For this reason, if one is writing a bootloader starting from real mode, they would need to decide whether they would rather perform the transition to 64-bit mode and boot Linux as described above, or instead follow the real mode boot protocol not included in this paper.

6. RESULTS

The bootloader successfully starts and runs Buildroot Linux kernels within lvisor. We have found a few limitations, however. First, while Linux will successfully execute commands from installed packages, we are unable to get it to execute arbitrary programs provided from other sources, such as simply located within the file system. Second, Linux is unable to use any networking. Adding these two capabilities would most likely require reconfiguring the Linux kernel. Finally, the Linux kernel has no persistent disk - while the initial file system can be loaded and functions correctly, any modifications to the file system will be lost if lvisor is restarted.

7. FUTURE WORK

For the bootloader, as discussed in previous sections, there are a few areas to be explored in the future, such as fully identifying the usage of each field in the setup header. Additionally, backwards compatibility would be an important addition to the bootloader, as it currently only support modern Linux boot protocols in 64-bit mode.

lvisor itself can be expanded in the future to provide more functionality to Linux. This may include changes in the configurations used to build Linux kernel. One function we explored is to add multiprocessor support to lvisor. We modified lvisor to let Linux kernel run natively on physical CPUs. We changed the limit on supported number of CPUs, NR_CORE, from 1 to 64 to be compatible with our Linux kernel configuration, and let all writes to the MSR pass through lvisor to physical CPUs, so that the guest system can wake CPUs up. Although this method allows the guest to use multiple processors, it presents a security risk as lvisor does not manage these cores. Modifying lvisor with multicore support would make it much more secure.

Finally, the issues outlined in section 6 could be remedied.

8. CONCLUSION

In this paper, we discussed necessary setup and procedure for a bootloader to boot Linux in long mode. While our implementation was based in lvisor, the same protocol with the same steps are required for even a standard bootloader. By collecting and condensing the information necessary to create a Linux bootloader, we hope that we can make future research on booting Linux less difficult.

9. REFERENCES

[1] 0xAX, "Linux Inside", *GitBook*, 2018. [Online]. Available: https://0xax.gitbooks.io/linux-insides/content/. [Accessed: 11- Mar- 2018].

[2] L. Torvalds, "Linux Repository", *GitHub*, 2018. [Online]. Available: https://github.com/torvalds/linux. [Accessed: 11- Mar- 2018].

[3] psomas, "lguest64 Repository", *GitHub*, 2018. [Online]. Available: https://github.com/psomas/lguest64. [Accessed: 11- Mar- 2018].