Chapter 7: What's Next?

Introduction

And now, after writing a simple operating system kernel and learning the basics of creating kernels, the question is "What's Next?". Obviously, there is a lot to do after creating 539kernel and the most straightforward answers for our question are the basic well-known answers, such as: enabling user-space environment in your kernel, implementing virtual memory, providing graphical user interface or porting the kernel to another architecture (e.g. ARM architecture). This list is a short list of what you can do next with your kernel.

Previously, I've introduced the term *kernelist* [1] in which I mean the person who works on designing an operating system kernels with modern innovative solutions to solve real-world problem. You can continue with your hobby kernel and implementing the well-known concepts of traditional operating systems that we have just mentioned a little of them, but if you want to create something that can be more useful and special than a traditional kernel, then I think you should consider playing the role of a kernelist. If you take a quick look on current hobby or even production operating system kernels through GitHub for example, you will find most of them are traditional, that is, they focus on implementing the traditional ideas that are well-known in operating systems world, some of those kernels go further and try to emulate another previous operating system, for example, many of them are Unix-like kernel, that is, they try to emulate Unix. Another examples are ReactOS[2] which tries to emulate Microsoft Windows and Haiku[3] which tries to emulate BeOS which is a discontinued proprietary operating system. Trying to emulate another operating systems is good and has advantages of course, but what I'm trying to say that there are a lot of projects that focus on this line of operating systems development, that is, the traditionalists line and I think the line of kernelists needs to be focused on in order to produce more innovate operating systems.

I've already said that the kernelist doesn't need to propose her own solutions for the problems that she would like to solve. Instead of using the old well-known solutions, a kernelist searches for other better solutions for the given problem and designs an operating system kernel that uses these solutions. Scientific papers (papers for short) are the best place to find novel and innovative ideas that solve real-world problem, most probably, these ideas haven't been implemented or adopted by others yet[4].

In this chapter, I've chosen a bunch of scientific papers that propose new solutions for real-world problem and I'll show you a high-level overview of these solutions

[1] In chapter where the distinction between a kernelist and implementer has been established.
[2] https://reactos.org/
[3] https://www.haiku-os.org/
[4] Scientific papers can be searched for through a dedicated search engine, for example, Google Scholar.

and my goal is to encourage the interested people to start looking to the scientific papers and implement their solutions to be used in the real-world. Also, I would like to show how the researches on operating systems field[5] innovate clever solutions and get over the challenges, this could help an interested person in learning how to overcome his own challenges and propose innovate solutions for the problem that he faces. Of course, the ideas on the papers that we are going to discuss (or even the other operating system's papers) may need more than a simple kernel such as 539kernel to be implemented. For example, some ideas may need a networking stack being available in the kernel, which is not available in 539kernel, so, there will be two options in this case, either you implement the networking stack in your kernel or you can simply focuses on the problem and solution that the paper present and use an already exist operating system kernel which has the required feature and developing the solution upon this chosen kernel, of course, there are many open source options and one of them is HelenOS[6] microkernel[7].

However, before getting started in discussing the chosen papers, the first section of this chapter discusses general concepts that are related to operating systems, we haven't discussed these concepts previously and they will be needed to make the papers that we are going to present easier to grasp. A small note should be mentioned, this chapter only shows an overview of each paper which means if you are really interesting on the problem and the solution the a given paper represents, then it's better to read it[8].

In-Process Isolation

In current operating systems, any part of a process can read from and write to any place of the same process' memory. Consider a web browser which is an application like any other application consists of a number of different modules [9] and each one of them handle different functionality, rendering engine is one example of web browser's module which is responsible for parsing HTML and drawing the components of the page in front of the user. When an application is represented as a process, there will be no such distinction in the kernel's perspective, all application's modules are considered as one code that each part of it has the permission to do anything that any other code of the same process can do. For example, in web browser, the module that stores the list of web pages that you are visiting now is able to access the data that is stored by the module which handles your credit card number when you issue an online payment. As you can see, the first module is much less critical than the second one and unfortunately if an attacker can somehow hack the first module through an exploitable security bug,

[5] Or simply the kernelists!

[6] http://www.helenos.org/

[7] The concept of *microkernel* will be explained in this chapter.

[8] It is easy to get a copy of any mentioned paper in this chapter, you just need to search for its title in Google Scholar (https://scholar.google.com/) and a link to a PDF will show for you.

[9] In the perspective of programmers.

she will be able to read the data of the second module, that is, your credit card information and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 nothing is going to stop her. This happens due to the lack of *in-process isolation* in the current operating systems, that is, both sensitive and insensitive data of the same process are stored in the same address space and any part of the process code is permitted to access all these data, so, there is no difference in your web browser's process between the memory region which stores that titles of the pages and the region which stores you credit card information. A severe security bug known as *HeartBleed vulnerability* showed up due to the lack of in-process isolation, next, HeartBleed will be explained to show you how this real-world problem may impact our systems and then one of the solutions that has been proposed by kernelists will be discussed.

Lord of x86 Rings

A paper named "Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86" [10] proposes an architecture (named LOTRx86 for short) which provides an in-process isolation [11]. LOTRx86 doesn't use the new features of the modern processors to implement the in-process isolation, Intel's Software Guard Extensions (SGX) is an example of these features. The reason of not using such modern feature in LOTRx86 is portability, while SGX is supported in Intel's processors, it is not in AMD's processors[12] which means that employing this feature will make our kernel only works on Intel's processor and not AMD's. Beside that, SGX is a relatively new technology[13] which means even older Intel's processors don't support it and that makes our kernel less portable and can work on only modern Intel's processors. So, if we would like to provide in-process isolation in our kernel, but at the same time, we want it to work on both Intel's and AMD's processors, that is, portable [14], what should we do? According to LOTRx86, we use privilege levels to do that.

Throughout this book, we have encountered x86 privilege levels and we know from our previous discussions that modern operating systems only use the most privileged level `0` as kernel-mode and the least privileged level `3` as user-mode. In LOTRx86 a new area in each process called *PrivUser* is introduced, this area keeps the sensitive data of the process and it's only accessible through special code that runs on the privilege level `2`, so, in a kernel which employs LOTRx86 a process may run in privilege level `3` (user-mode), as in modern operating systems, and may run in privilege level `2` (PrivUser). Most of the

[10] Authored by Hojoon Lee, Chihyun Song and Brent Byunghoon Kang. Published on 2018.
[11] The paper uses the term *user-mode privilege separation* which has the same meaning.
[12] Beside Intel, also AMD provides processors that use x86 architecture.
[13] Intel's SGX is deprecated in Intel Core but still available on Intel Xeon.
[14] In LOTRx86 when the term *portable* is used to describe something it means that this thing is able to work on any modern x86 processor. The same term has another boarder meaning, for example, in the if we use the boarder meaning to say "Linux kernel is *portable*" we mean that it works on multiple processors architecture such as x86, ARM and a lot more and not only on Intel's or AMD's x86.

normal work of a process will be done in level 3, but once the code is related to sensitive data, such as storing, accessing or processing them, the process will run on level 2. Of course, the sensitive data cannot be accessed by process' normal code since the latter runs on level 3 and the former needs a code that runs on privilege level 2 to be accessed. If an attacker exploit a vulnerability that allows him to read the memory of the process, he will not be able to read the secret data if this vulnerability is on the normal code of the process. A kernel with LOTRx86 should provide a way for the programmers to use the feature provided by LOTRx86, so, the authors of the paper propose a programming interface named *privcall* which works like Linux kernel's system calls. Through this interface an application programmer can write functions (routines) that process the secret data, these functions will run on privilege level 2 and will be stored in PrivUser, we will call these functions as *secret functions* in our coming discussion. When the normal code of the process need to do something with some secret data that is stored in PrivUser a specific secret function can be called through `privcall` interface, once this call is issued, the current privilege level will be changed from 3 (user-mode) to 2 (PrivUser[15]) by using x86 call gates that we have discussed earlier in this book . Note that this solution **mitigates** vulnerabilities like HeartBleed but doesn't **prevent** them necessarily.

To implement this architecture, two requirements should be satisfied in order to reach the goal. The first requirement is called `M-SR1` in the paper and it states that the PrivUser area should be protected from the normal user mode which most of the application's code run on. The second requirement is called `M-SR2` in the paper and it states that the kernel should be protected from PrivUser code. To satisfy the first requirement, the pages of PrivUser are marked as privileged pages in their page entry [16], that is, the code that run on privilege level 3 cannot access them while the code that runs on levels 0, 1 and 2 can. To satisfy the second requirement, the authors propose to use segmentation, `LDT` table is employed to divided each process into segments and a special segment for the secret functions and data, that is, PrivUser is defined and the definition of this segment indicates that the secret functions can only access the secret data under privilege level 2 in order to protect the kernel's data which reside in privilege level 0. This is the high-level description of LOTRx86 solution, there are some challenges that have been faced by the authors and the details of them and how they overcame them can be found in the paper, so, if you are interested on implementing LOTRx86 in your kernel, I encourage you to read the original paper which also discusses how the authors managed to implement their solution in Linux kernel as kernel modules, also, the paper shows the performance evaluation of their implementation. There is something to note, the authors assume that the solution is implemented in `64-bit` environment instead of `32-bit` and due to that the faced some challenges that the may not face in `32-bit` environment.

[15] In the paper, the name PrivUser means two things, the execution mode and the secret memory area.

[16] We have discussed this bit in a page entry in Chapter .

Of course LOTRx86 is not the only proposed solution for our problem, there are a bunch more and some of them are mentioned on the same paper that we are discussing. What makes LOTRx86 differs from them is the focus on a solution that has a better performance and portable as we have examined in the beginning of this sub-section . As you saw in this solution how the authors played the role of a kernelist, they proposed a solution for real-world problem, they used some hardware feature that is usually used in a different way in the traditional operating systems (privilege level 2) and they proposed a different and useful idea for operating system kernels.