Secure I/O Path from User Space to Network Driver

Christian Martinez

Advisor: Lin Zhong and Ph.D student Caihua Li

September 2023

Background

Operating systems (OSes) are responsible for managing a computer's resources. This can include managing software and hardware resources like memory, I/O, and scheduling. However, modern OSes are large and complex; such a large surface area of code creates opportunities for vulnerabilities to exist. Because of these possible vulnerabilities, an OS may become compromised. Once compromised, an OS could be used maliciously against the user.

This is especially true with the Linux OS, with over 30 million lines of code in its kernel (across the "core" and its drivers) as of 2021 [1]. In its kernel mode, the Linux OS is capable of free access to just about anything it wants. There is no hardware or software that could prevent the Linux kernel from touching memory that doesn't belong to it. This means that a malicious Linux OS could access secure user data from applications.

Applications regularly interact with the Linux kernel in a way that could put users at risk if the OS was malicious. For example, if a user application must perform some form of networking, the application may perform some system calls that invokes the kernel to perform some work for it, often in the form of manipulating or copying the data being transmitted. System calls like this provide a point of entry for a malicious kernel to intercept the data before it reaches its destination, potentially putting the user at risk.

However, there are some approaches to protecting the user from a potentially malicious OS. Systems like MProtect [2] developed by the Efficient Computing Lab (ECL) led by Lin Zhong, work to isolate memory spaces to processes. With capabilities like cloaking page tables, setting immutable interrupt vector tables, and encryption, MProtect is capable of protecting user data from a potentially malicious kernel. This means that if a system that

implements MProtect is loaded onto a machine running Linux, a malicious kernel would not be able to access user data unless specific access is granted by the user application.

Nonetheless, security isn't free, and a system like MProtect incurs some overhead. For example, in MProtect, page table updates are trapped into a "Guardian" process that oversees that the correct pages are being accessed, and this additional context switch and checks occupy some cycles. Other functionalities supported by MProtect, like encryption, are simply computationally expensive, and although effective at protecting data, they are not efficient.

While such trade-offs may be permissible in certain workflows, other use cases cannot benefit from the services provided by MProtect. For the purposes of this research, we want to focus on Linux kernel drivers, which can have very specific limitations and interfaces.

Kernel drivers (or simply drivers) are loadable modules that attach themselves to the kernel tasked with controlling a specific piece of hardware [3]. Because they must interact directly with hardware and resources on a system, they also operate with similar privileges as the "core" kernel.

Drivers are significantly smaller in code, and as such, are easier create and/or verify as secure. The issue is that communication between a user space application and a driver requires invoking the core kernel, as the driver lives in the kernel space. Going back to our networking example, a network driver serves to connect the system with the device that performs data transmission. However, for the user data to reach the driver, a system call must be made that invokes the core kernel to perform work so that the driver can accept the data from the user.

In the model we plan to explore, we assume that the "core" kernel (and some drivers) may be malicious, while a target driver we want to interact with is secure. The goal then is to protect this target driver from untrusted components. As such, in the example above, we are in need of a way to securely move the data from the user space to our target driver while preventing untrusted parts of the kernel to access the data.

Approach

Our goal is to demonstrate that secure I/O is possible between a user application and a kernel level driver. As used here, we consider I/O to be secure if we are able to prevent the core kernel from ever needing cleartext access to the data being communicated.

To do this, we look to the network stack of the Linux kernel as our example. The goal of this project is to take the system call `send()`, which is used to communicate through a socket, and modify the control flow as necessary to ensure the core kernel does not perform a copy/read of the data being sent. This will require familiarization with the network stack as written in the Linux kernel, and later modifications to the source code to refactor any raw data access.

As a starting point, a flag for `send()` exists that performs something similar to our use case: `MSG_ZEROCOPY` [4]. However, this flag has a very narrow use case and requires extra work from the calling application and the receiving driver. Our goal is to build a more generic zero-copy `send()` implementation that works for as many use cases as possible. The optimal result is an implementation of `send()` that can defer data copy responsibilities to the network driver with minimal overhead for the calling application and receiving driver.

However, there is much uncertainty whether such an implementation is possible, but we will aim to exhaust all possibilities for a comprehensive view of the problem and possible solutions. We aim to provide a deeper understanding of the costs of secure I/O (e.g. compatibility with devices, lines of code, performance).

For ease of development and testing, work will be done on Linux v5.15.92 mainly for compatibility with an MProtect prototype, MGuard, maintained by the ECL. Being compatible with MGuard will allow for easier testing and experimentation through MGuard's capabilities.

Deliverables and Timeline

Below is a brief outline of deliverables and a rough timeline. Please note that specific dates and tasks are subject to change based on gained knowledge and guidance.

- (9/16) Complete background readings: 1 week
- (9/30) Read through Linux source code (as necessary for `send()`): 1-2 weeks
- (10/7) Set up development environment: $<$ 1 week
- (10/7) Integrate/create benchmarks: 1 week
- (10/21) Design zero-copy strategies: 2 weeks
- (11/25) Iterate on zero-copy strategies, evaluate different approaches based on performance and overhead: 4-5 weeks
- (12/01) Complete draft of final report for advisors: 1 week
- (12/14) Complete and submit final report: 1-2 weeks

References

[1] Linux kernel. Wikipedia https://en.wikipedia.org/wiki/Linux_kernel

[2] Caihua Li, Seung-seob Lee, Min Hong Yun, Lin Zhong. MProtect: Operating System Memory Management without Access

[3] TLDP: Device Drivers. https://tldp.org/LDP/tlk/dd/drivers.html

[4] Linux Documentation. https://www.kernel.org/doc/html/v4.18/networking/msg_zerocopy.html