Secure I/O Path from User Space to Network Driver

Christian Martinez

Advisors: Lin Zhong and Ph.D student Caihua Li

Submitted to the faculty of the Department of Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Science

Department of Computer Science
Yale University

Dec. 14, 2023

Abstract

Modern commodity Operating Systems (OSes) are large and not secure. As such, an OS with vulnerabilities may be compromised and used maliciously against users. This paper addresses the security concerns associated with the Linux OS's unfettered access to user data during system calls, specifically, the networking send() system call. A malicious Linux OS could intercept the user's data before it reaches the wire, and this gives the attacker the ability to modify or reroute the data without any indication to the calling user that this happened.

We propose an extension to the MSG_ZEROCOPY flag for the send() system call to enhance security for the calling user. Our primary focus is on prohibiting unauthorized access by the kernel to user data during the execution of the kernel network stack. We intend to show that it is possible to complete the send() networking call without the OS ever needing access to the user's data. The approach involves techniques such as disabling page table walks and invalidating Page Table Entries (PTEs) in the kernel's page tables. These measures are implemented on a Raspberry Pi 4-B running on ARM architecture, using Linux version 5.15.92. The results show success in enhancing the security of the send() system call, providing a more secure zero-copy implementation while minimizing the impact on performance, and lines of code footprint. While a more generic zero-copy implementation of the system call is possible, more work is necessary to make it universal for all calls to send(), not just those that use the MSG_ZEROCOPY flag.

Table of contents

Front matter

Abstract i
Table of contents ii
Introduction 1
Trust Model 2
Protecting the User 3
Development Environment 4
Approach 4
MSG_ZEROCOPY 4
Page Table Invalidation 5
Blocking Capabilities 10
Results 10
Testing 11
Conclusion 11

Introduction

Operating Systems (OSes) are responsible for managing a computer's resources. From managing hardware input and output (I/O), system memory, to task scheduling, OSes are essential for a functional and efficient system.

However, modern OSes are large and complex; their large scale and complexity makes having errors in them almost a guarantee. Depending on the significance of these errors, or bugs, an OS may be susceptible to attacks of varying impact. A sufficiently dangerous attack could compromise an OS.

This is especially true with commodity OSes, like the Linux Operating System, one of the most widely used open-source OS. With over 30 million lines of code in its kernel (across the "core" and drivers), vulnerabilities are common in Linux's large code surface area [1]. In Linux, processes are isolated from each other and given their own share of resources. Further division occurs across the OS boundary, however. User processes only have access to memory that belongs to their process and lives within the user-space defined regions of memory. Kernel processes on the other hand, have access not just to their own process's memory region, but the entire system's memory. As such, applications are written with complete trust of the OS, while any distrust of other applications should be handled by the memory isolation model of the Linux OS.

While this model may protect user-space processes from each other, it does nothing to protect them from the OS because of the assumption that the kernel should always be trusted. As mentioned earlier, however, the OS is vulnerable to attacks, and if compromised, could be used against the user.

User-space processes regularly rely on the OS to complete their own tasks, like when making system calls to read from or write to a buffer, up to sending a message over a network. As such, in regular execution, context switches between user-space and kernel-space privileges are very common. This gives a malicious OS plenty of opportunities to inject itself into a process's workflow. This said, there exist ways to protect user-space processes from unauthorized accesses by the kernel into their memory, but current implementations are are unable to handle all use cases, or they carry significant overhead.

In my work, we explore ways in which we can provide greater security to the `send()` system call in the Linux OS, a system call that allows the calling process to send data to a network through a socket. We aim to provide this security by expanding upon the existing zero-copy capabilities of the send system call to ensure that a malicious OS is unable to access the data sent by the caller. We will narrow the scope to `send()` with TCP, however,

due to limited kernel support for necessary features.

Trust Model

In our trust model, we define three major components to a system.

The user-space refers to the set of privileges and memory region given to a process that is not part of the OS kernel. User-space processes have minimal privileges and should only be able to access memory given to them by the OS. That is, it is isolated from other user-space processes.

The ”core” kernel, or simply the kernel from here on out, refers to the set of privileges and memory region given to the OS. On Linux, the kernel has complete control over a system. Simply put, the kernel can do almost anything with complete access over all memory space, even memory and data that belongs to other processes. These capabilities are essential for the Linux OS as it simplifies the design of the OS and makes the implementation of necessary functionality possible.

Kernel drivers, or simply drivers, are loadable pieces of software that extend the capabilities of the kernel. These often serve as the interface between the kernel and hardware devices. As the name suggests, these drivers live within the kernel space, and as such, gain the privileges and capabilities of the kernel. Drivers need these capabilities as they often require access to memory across the entire system to function correctly.

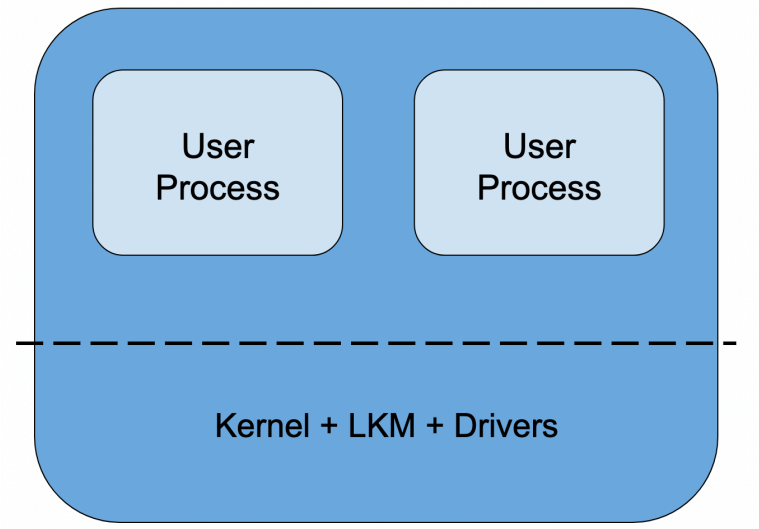

Figure 1: Representation of isolation levels in Linux

We note that the kernel, kernel modules, and drivers, all live within the same privilege space, while user processes ”own” a subset of the available resources and privileges.

We then make the assumption that the kernel is not trusted due to the possibility of it being compromised and having total access to the system. User processes are assumed to be trusted as these are typically written by end-users, and the user should have complete control over what their process is doing. Drivers, although living in the kernel space, are assumed to be trusted as they are typically much smaller in code, easier to verify, and provide the necessary interface between software and hardware devices.

The kernel then exists as a connection between two trusted parts, the user-space and the system drivers. Any attempt by a user-space process to interface with a driver must invoke a context switch where the kernel does any setup that may be necessary for this communication to happen.

Protecting the User

Given that the kernel cannot be trusted, we need ways to protect the user and their data from it. Taking our case, the `send()` system call, before the data the user wants to send can leave the system, the kernel gets access to it before it is handed off to the driver. A malicious kernel could therefore modify the data or send it elsewhere without a clear indication to the calling user that this happened.

There are a few ways we can protect the user in a more general case from unauthorized access by the kernel. Software systems like MProtect [2], developed by Yale's Efficient Computing Lab, can provide isolation from the kernel for user-space processes. With features like page table swaps, interrupt vector overwrites, and encryption, MProtect is able to prevent protect a user-space process's memory from unauthorized access by other processes, kernel processes included.

Other systems, like ARM's TrustZone or Intel's SGX, are extensions to the hardware to provide a trusted environment for sensitive processes [3; 4] to perform secure operations.

Both these types of security extensions, and many more, exist to solve the issue of untrusted systems by providing some type of isolation to the calling processes. However, security is not free, and the protections provided by these systems carry overhead. For encryption capabilities, for example, expensive computations are necessary, and it may not be viable in all use cases. Furthermore, the hardware-based security systems rely on hardware features that aren't available on all systems.

Therefore, we explore a different way to provide security for the user in a more generic setting. To narrow our scope, we look to show how a system call like `send()` can be made more secure against unauthorized accesses. We do this by using the framework provided by the MSG_ZEROCOPY flag to the system call that attempts to complete the call by deferring

data access and copy to the driver. If we can prove that when successfully completing a call with this flag enabled, the kernel does not access the user's data, then we have provided the minimum security necessary to conform to the trust model we described earlier.

Development Environment

We will develop on a RaspberryPi 4-B (RPI) running on ARM architecture, using Linux version 5.15.92. We decided to develop on the RPI due to the hardware capabilities of the platform and familiarity with ARM architecture. However, the techniques described in future sections can be modified to support different platforms and architectures.

Approach

We intend to extend the capabilities of the MSG_ZEROCOPY flag for the `send()` system call to create a more generic zero-copy implementation of the call. However, for this paper, we are limiting ourselves to TCP, due to greatly differing implementations of UDP and TCP in Linux, and the large code surface area for both.

MSG_ZEROCOPY

MSG_ZEROCOPY is a flag that can be passed to the `send()` system call that instructs the OS to attempt to complete the call while deferring the data copy to the driver. Without this deferral, the kernel copies the user's data into the kernel space to create a complete TCP packet that can be built throughout the OS's different networking layers.

However, there are a few scenarios where this flag may be ignored. The most common is when the hardware on the system lacks Scatter-Gather (SG) I/O capabilities. This is because the Linux implementation of zero-copy send relies on having the driver collect the user's data directly from memory, and since data is not guaranteed to exist in contiguous physical memory, the driver may be required to simultaneously access various parts of system memory to build the packet before it is sent out. Therefore, if a `send()` call is made with the MSG_ZEROCOPY call but SG is not available on the hardware, the flag is ignored.

There are a few other shortcomings with the current zero-copy implementation. Aside from the "optional" nature of it, the current zero-copy implementation leverages the error queue for the socket as a termination notification queue [6]. This is necessary because TCP may require re-transmission, therefore the sending user must not modify their data before transmission is complete, else they hinder their own communication. The benefit to this is

that the calling user is not blocked when they make the system call, with the downside being that additional overhead is needed to continuously check the notification channel to know when a buffer is safe to modify.

We note that these "shortcomings" exist only because the zero-copy implementation was not done with security in mind, but rather performance, with the goal of MSG_ZERO_COPY being minimizing the amount of data that must be read/written from/to memory.

Nonetheless, a working zero-copy implementation serves as a great starting point for our purposes, and we aim to address the issues mentioned above with our modifications.

Page Table Invalidation

Page tables are an important part of Linux's memory management framework, where Page Table Entries (PTEs) store information about physical memory. By invalidating PTEs, we disallow standard reads/writes from/to memory that the PTEs point to. We intend to demonstrate that this allows for a true zero-copy `send()` implementation by disallowing access to the regions of memory that store the user data. If the call can be completed while access to these pages is not allowed, then we can claim `send()` is zero-copy.

The control flow of `send()` covers a large surface area of code, with hundreds of function calls specific to networking, and hundreds more pertaining to standard kernel mode operations.

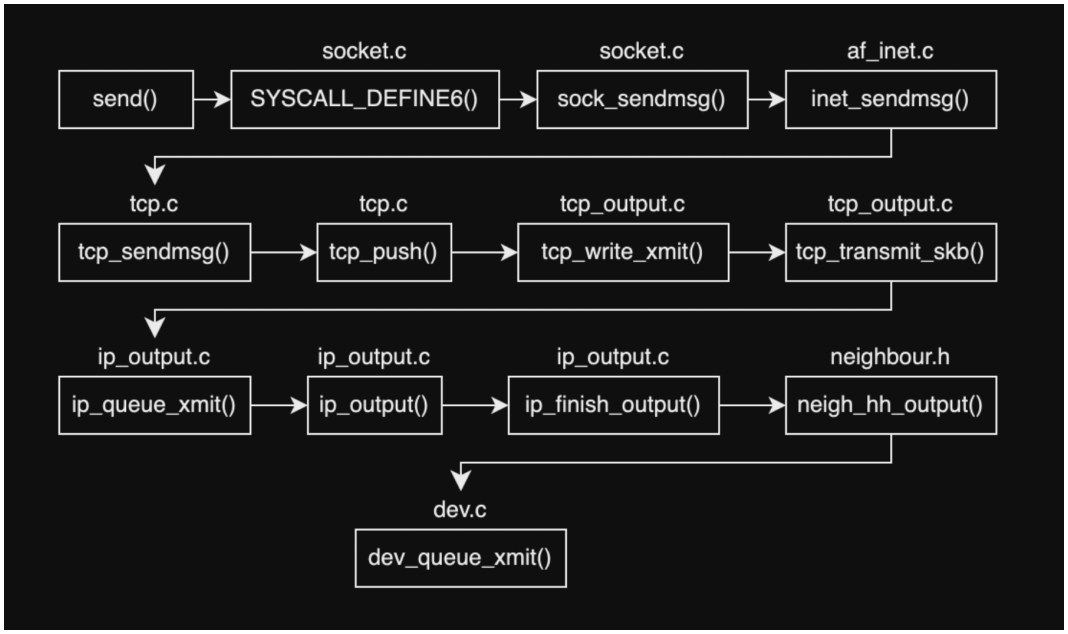

Figure 2: Representation of control flow of `send()`

The figure above attempts to capture a high-level view the control flow of a `send()` call in Linux, from user call to driver hand-off. This visualization does not include function calls made within each of these functions, and it omits many intermediary functions that perform necessary set-up between these function calls. The final function on this list `dev_queue_xmit()` is one of the final steps before the built packet is passed off to the driver for transmission.

Our goal is therefore to show that in the thousands of lines of code that build a TCP packet, no access to the user data is necessary. We plan to disable access to the user's data immediately after the `send()` call is made, and re-enabling it immediately before control is given to the driver. This way, throughout the kernel's execution, we protect the user's data, and according to our trust model, we have a secure `send()` call.

The primary techniques we will use to disable memory access and protect user data are disabling translation table walks and page table invalidations.

TCR_EL1

The first of these techniques relies on the Translation Control Register (TCR_EL1) of the ARM processor of our Raspberry PI (RPI). The TCR_EL1 stores the address of the first level page table for user-space processes. By modifying specific bits on the register, however, we are able to completely disable page table walks using the user process's page table [5]. By enabling the EPD0 bit on TCR_EL1, a Translation Lookaside Buffer (TLB) miss generates a fault, and no translation is made [5]. Thus, we can disable page table walks using the user process's page table and then wipe the TLB. By doing this, we ensure that using a user's page table to find the physical addresses of memory that contain user data is not possible, and we also ensure that caching isn't used to find the physical address.

TCR_EL1 Modification Pseudocode

```
asm volatile ("mrs %0, tcr_el1" : "=r" (getter))
setter = getter | (1 << 7)
asm volatile("msr tcr_el1, %0" :: "r" (setter))
flush_tlb_all()
```

The code above does the required modifications: it pulls in the value on the register TCR_EL1, sets the 7th bit (EPD0), and then saves this value back onto the register. We then do the opposite before handing control off to the driver to re-enable page table walks.

However, this isn't enough to guarantee that no access is made to the user data by the kernel. This is because the kernel maintains a direct mapping to the entire system's physical

memory [7], called a logical/linear mapping. This means that even without using the user's page table, the kernel would be able to find the physical page that contains the user's data.

Therefore, to protect against this type of access, we must do another set of invalidations, this time, on the kernel's page table.

Kernel PTE Invalidation

We must invalidate the Page Table Entries (PTEs) in the kernel's page tables that correspond to the logical mapping of the user data's physical memory. Otherwise, it would be possible to get a logical mapping from the user's address, then use the kernel's page table to find the physical address of the data.

To invalidate the necessary PTEs, we must get a logical mapping of the user's data ourselves. First, we perform a page table walk using the calling process's page table. We can access the first level of the user page table with the kernel variable `current->mm->pgd`. `current` refers to the struct representation of the current running user process. With the first level page table available, we are then able to perform a page table walk using the virtual address provided by the user to find the physical address of their data.

The page table walk is done with the assumption of four levels of translation [7] and no huge pages (pages with sizes $>$ 4096B); however, in reality, only a few levels of translation are needed and huge pages aren't common due to the limited amount of memory on the RPI platform. Additionally, the page table walk code can be modified to handle larger page sizes if necessary. So while the page table walk code is written assuming all layers of translation are used, a few of the translations are essentially no-ops that handle type casting. After completing all levels of translation, we get the physical address of a page of the user's data.

Page Table Walk Pseudocode

```
pgd = pgd_offset_pgd(current->mm->pgd, virtual_address)

p4d = p4d_offset(pgd, virtual_address)
if(p4d_none(*p4d) || p4d_bad(*p4d)) return NULL

pud = pud_offset(p4d, virtual_address)
if(pud_none(*pud) || pud_bad(*pud)) return NULL

pmd = pmd_offset(pud, virtual_address)
if(pmd_none(*pmd) || pmd_bad(*pmd)) return NULL

ptep = pte_offset_kernel(pmd, virtual_address)

page_addr = (pte_pfn(*ptep)) << PAGE_SHIFT
page_offset = start & ~PAGE_MASK
paddr = page_addr | page_offset
logical_addr = page_addr + PAGE_OFFSET
```

While this pseudocode handles the translation for a single page, in my implementation we iterate over the entire memory space used by the user's data.

The translation shown above will convert the virtual address provided by the user into a physical address. Once we have the physical address, we can add the value `PAGE_OFFSET` to index into the logical mapping of the physical address. Once we have the logical mapping, we perform one more set of page table translations, this time using the kernel's page table, as this stores the last way the kernel could access the user's data.

Once we have the corresponding PTE for a given logical address, we disable that PTE in the following way:

```
kernel_pte = logical_to_pte(logical_addr)
kernel_pte->pte &= ~(1)
```

This sets the PTE for this physical address as not present, so attempting to access this page through the kernel's page table would trigger a fault.

Attempting to run this code as is does trigger a fault; to fix this, we just need to re-enable the page table entries before we hand off control to the driver, similarly to what we did with

the TCR_EL1 register. However, re-enabling the PTE requires more care, as by the time we reach the driver hand-off point, we no longer have a pointer to the user's buffer. Instead, within the `struct skbuff` that represents our packet, we have an array of "fragments" that store pointers to the pages that make up the user's buffer.

Therefore, for each fragment in our frag list, we convert the `struct page` to its page frame number, into a physical address, before finally converting it to a logical address. Once we have the logical address for a page of user memory, we do one last set of page table walks using the kernel's page table to re-enable the PTE for that page. With the PTE set to valid once more, the `send()` system call is able to complete without issues.

Fragment Handling Pseudocode

```
frag = &skb_shinfo(skb)->frags[frag_counter]
pfn = page_to_pfn(frag->bv_page)
paddr = pfn << PAGE_SHIFT
paddr += frag->bv_offset
logical_addr = paddr + PAGE_OFFSET
logical_address = logical_addr
{Page Table Walk}
lptep = pte_offset_kernel(pmd, logical_address)
ptep->pte = lptep->pte | 1
```

Both, disallowing page table walks and invalidating PTEs serve us in ensuring the system call is truly zero-copy. Therefore as long as there exists hardware support for SG I/O, a secure `send()` is possible. Even if SG is not supported on a platform, these two techniques allow us to create a pseudo-secure implementation of zero-copy `send()`.

SG by the Kernel

For drivers that expect data to exist contiguously when sending their packets, i.e. drivers that don't support SG, we can place this responsibility back on the kernel. However, we follow similar processes as discussed above: we invalidate access to user memory for the duration of the kernel's execution. Only before control is given to the driver and after access to the user's data is reinstated, we have the kernel insert the user's data into the packet. Since access to the data is still not allowed for the rest of the execution, we can consider this to be pseudo-secure, as the kernel only has access to the user's data in a narrow part of the execution. A major caveat here is that this would require that the copy mechanism is itself safe. Such an implementation would also require additional kernel computation for

checksums, and could therefore require larger code modifications. Regardless, if done, this would remove the hardware requirements for "zero-copy" `send()`. Zero-copy in this scenario only has value through the scope of security, any performance benefits exhibited by the existing implementation are void since we perform an extra copy by the kernel. However, this implementation was not tested for this paper.

Blocking Capabilities

The last major step necessary to make generic `send()` calls zero-copy is to add blocking capabilities. In non-zero-copy `send()`, the system call returns after all data has been copied into the kernel's memory. As such, it is always safe to modify the data in the user's buffer after the system call. Due to the nature of zero-copy `send()`, the calling user is not granted the same freedom. Instead, the user must add additional code that constantly checks for notifications of when it is safe to modify the buffer. To make zero-copy more generic then, we need a way to remove this notification system. One possible approach could be to offload the notification parsing to the kernel.

A possible strategy for this could use a busy-wait loop at the end of the system call, before it returns, that checks for the completion notification of the zero-copy system call. As such, the system call would only return control to the user once the data in buffer was no longer necessary. With this, the user would be able to interface with a zero-copy `send()` the same way they would otherwise. This would still require handling special cases, as with non-blocking sockets, but it serves as a basic framework for our use case.

Nonetheless, implementing this strategy was troublesome. After much experimentation, it began to seem that a naive implementation of this blocks the kernel indefinitely, even if the data is received on the other end of the network. This is due to the way the termination notification is queued upon completion. Therefore, this feature is not complete at the time, but alternative strategies are being explored.

Results

Our implementation expands upon the existing `MSG_ZEROCOPY` flag for the `send()` system call by making it more secure. With around 200 new lines of code, we are able to do extend the security of zero-copy send without much effect to performance (200 lines of execution vs thousands). The current prototype for blocking capabilities on the other hand is around 100 lines of code. Maintaining a low code footprint was a secondary goal for this project, as bloated code is what can lead to security issues in the first place.

Testing

Throughout development, I tested my changes with a sample user program that sets up a basic server that communicates with a remote server. The biggest challenge was during work with PTE invalidation, as errors here would result in faults and require system restarts. In our current implementation, our server is able to successfully communicate with the remote client with messages of varying size, ranging from 1 to `PAGE_SIZE` $* 10^5$ (or about 500 MB), and appears to only be limited by system capabilities.

To test the security capabilities of our implementation, we allowed the kernel to attempt copying from the user's data buffer's while our protections were enabled. However, whenever the kernel would attempt these copies, it would generate a fault and no data would be copied.

```
Dec 14 12:29:41 raspberrypi kernel: At start of tcp_sendmsg_locked using msr/mrs
Dec 14 12:29:41 raspberrypi kernel: Reading partial user data before protections
Dec 14 12:29:41 raspberrypi kernel: Read this from user (no protections): bbbb
```

Figure 3: Read from user memory with no protections

```
Dec 14 12:29:41 raspberrypi kernel: Unable to handle kernel paging request at virtual address ffffff8107468180
```

Figure 4: Read attempt with protections enabled causes fault

The original goal of making the `send()` system call more secure is therefore fulfilled. The only methods of accessing the user's data, whether through walking the user's page table with their virtual address, or doing a direct lookup on the kernel's page table with a logical address, are disabled in our implementation. As such, any ordinary attempt to read this data by the kernel will result in a fault that prevents the data from being accessed.

Conclusion

In this project, we explored ways to improve system security in the Linux OS. More specifically, we enforced greater security requirements on the zero-copy implementation of the `send()` system call. Using techniques like prohibiting page table walks and PTE invalidation, we ensured that the OS kernel is unable to perform unauthorized accesses to a user's data.

While this serves to fulfil our original goal of greater protections against the OS kernel, it still lacks certain functionality that would allow it to work for all use cases. This lacking functionality is the blocking capabilities on the system call until completion. If it is possible

to implement this, further experimentation could include setting the `MSG_ZEROCOPY` flag as the default behavior of the system call. Then, all `send()` calls would be zero-copy.

For future work beyond just the `send()` system call, we can look to mimic similar behavior for other system calls that require access to user data. Nonetheless, this could prove difficult as each system call behaves differently and may interact with the user's data in different ways.

Bibliography

[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] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, Sarah Marti. TrustZone Explained: Architectural Features and Use Cases

[4] Victor Costan, Srinivas Devadas. Intel SGX Explained

[5] ARM Cortex-A53 MPCore Processor: Technical Reference Manual

[6] Willem de Bruijn. sendmsg copy avoidance with MSG_ZEROCOPY

[7] The Linux Kernel. https://www.kernel.org/doc/html/v5.8/arm64/memory.html