Secure I/O Path from User Space to Network Driver

Christian Martinez, Department of Computer Science, Professor Lin Zhong, Department of Computer Science, Yale University

Motivation

Commodity Operating Systems are large and vulnerable to attacks. A compromised OS has unfettered access to the entire system and could be used against the user. Therefore, we want a way to protect the user's data from an attack by the OS.

Taking a look at the Linux OS, the risk of these attacks are most apparent during system calls, where the calling process yields control to the OS. Looking specifically at the *send()* sys call, we intend to protect the user's data transmission from a malicious OS.

Our goal is to demonstrate that the *send()* sys call can be completed in a secure, zero-copy, manner.

KEY FINDINGS

> Can completely remove kernel's access to user data in send()

> Zero-Copy send() may be hardware dependent because of non-contiguous memory

> Certain security techniques may be locked to hardware (ARM's *TCR_** Registers)

> Primary technique (page table entry invalidation) is independent of architecture with Linux

> There is an existing framework in Linux for zero-copy send() that can be expanded on for security purposes

Approach

There are a few ways to provide this security, but we intend to leverage Linux's memory management framework combined with the Raspberry PI 4-B's architecture to protect the user's data.

Idea is to restrict OS access to user data while in kernel mode (i.e. while sys call is being completed).

Using the RPI's architectural support, we can disable page table walks using the calling process's page table.

Using Linux's memory management framework, we can invalidate page table entries that store the user's data

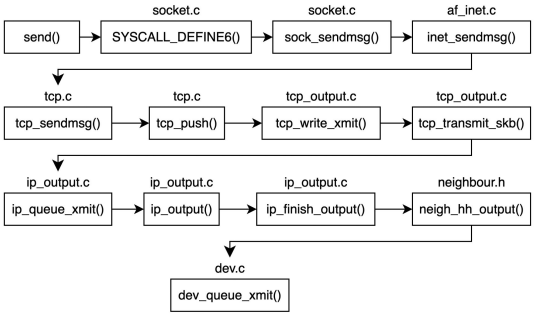
Insight

Linux TCP network stack is *huge.*

Thousands of lines of code to build the packet before it is sent to the network.

We could restrict access to data to the beginning and end of sys call.

Existing zerocopy framework exists in the code to leverage for our implementation; current implementation acts as more of a "suggestion," we need to make it stricter



Control Flow through Kernel

Implementation

At the start of TCP handling, modify the *TCR_EL1* to disallow page table walks and flush the TLB cache. This prevents the system from using the user's page table for page table walks.

At the start of TCP handling, perform a page table walk to find the physical address of the user's data.

Using the physical address, perform an additional page table walk on the kernel's page table to prevent direct access to the physical memory.

Before packet transmission, re-enable page table walks and re-enable all page table entries for the user's data.
Non-contiguous memory makes this more challenging than earlier steps: data is stored in "fragments" on the packet.

```
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;

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

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

PTE Invalidation and Page Table Walk Disabling Pseudocode

Conclusion

We managed to show that it is possible to make the Linux *send()* system call more secure by removing kernel access to the data.

While we managed to implement the security features we aimed for initially, there are other considerations that are currently being explored.

The biggest alternative goal is making the zero-copy implementation be blocking until transmission completion, this way, we could experiment with making all send() calls zero-copy.

With this proof of concept, we can look to creating a framework to make other similar system calls zero-copy using the techniques here.