COMP9242: Analysis of Paper 1

Antikernel: A Decentralized Secure Hardware-Software Operating System Architecture

z5012384

November 9, 2017

Summary

The authors of this paper have set out to design and implement an operating system that is more secure and modular than that of traditional monalithic kernels - with differences from existing microkernels, exokernels and multikernels with the intention of respecting the principle of least required privilege.

Such a kernel has no concept of *ring 0*, and hence no privilege level to escalate to - given malicious remote code execution. This is a consequence of the fact that the operating system components have been isolated and separated into what the authors call *nodes* - which are simply individual hardware components such as the CPU, DMA controller and RAM, which communicate over a packet-switched network-on-chip (NoC).

The modularisation of the typical kernel into many independent hardware state machines has the effect of making formal verification easier, and what the authors believe to be an improvement in parallelism.

The threat-model that the kernel was designed around does not consider the impact that an attacker in physical locality to the machine would have - and hence only considers a remote actor. The authors of the paper list the following as actions that are not possible given a scenario in which an attacker has unprivileged code execution within a user application:

- download a backdoor payload and have it execute on boot persistently
- modify executable code in memory or persistent storage, and intercept / spoof / modify system calls or IPC of another process
- read or write private state of another process
- gain access to handles belonging to another process by any means

Within use of the antikernel, a userland application may perform what are commonly known as privileged functions, such as physical memory management, by sending RPC requests over the NoC to a node which handles this function. This is done by setting register values, similar to traditional C calling convention, and executing the *syscall* instruction. Since syscalls are no longer used to make a request to a privileged kernel - the syscall instruction has been re-purposed for use in making RPC calls.

Pros

The authors of this paper have implemented an actual system from a design in previous work. Ideas are taken from other kernel isolation designs such as exokernels, microkernels and multikernels and implemented in a different way to offer an interesting take on component isolation to bring about security.

Practical violations are touched on in several footnotes - and mentions of areas of potential further research are suggested. This shows that thought was put into the potential flaws of some aspects of the system as a whole without blind absolute trust in the system.

Although incomplete, the system has been open sourced and released to the public to provide feedback and modifications. However, the system is in a fairly embryonic state - labelled as *research grade* by the authors.

Cons

A main benefit of architecting such a system was defined to be easy-to-perform verification of each sub-system. However limited formal verification of the system was actually performed prior to the release of the paper - and where it was not done, testing suites of various tools such as *yosys* and others developed by Xilinx are substituted. In a kernel that is developed to be secure as a consequence of the aggregated security of all its modular sub components - ensured by formal verification - a huge trust is placed in the assumption that test suites will ensure similar security. It would have been beneficial to complete remaining formal verification of the system prior to the release of the paper.

It is disappointing that the authors did not provide any benchmarks of the system, as the idea of communicating with subsystems purely over a network protocol via on-chip networking introduces interesting performance differences, when compared with a traditional kernel.

In discussion of RPC function calls, it is outlined that the *call* field of a function call packet may describe one of 256 possible functions that the caller wishes the recipient to perform. Although the purpose of function calls is not explicitly outlined, it is left to the reader to assume a function call may include requests such as memory allocation. Given this assumption, it seems fairly limiting that the system has a hard limit of 256 potential recipient functions. More information on this RPC transaction would have been beneficial, especially in assisting to gauge whether such a limit is an issue.

When sending RPC packets, a transmitter will block until an ACK arrives from the next-hop router.

This could cause denial of service issues in the event that a packet was dropped, forged, or sent to a non-existent node. A malicious actor could potentially perform any of the aforementioned activities - without mention of how susceptible packet transport is, it is hard to say how much of risk this may be.

The ability for a thread of execution to be removed from the run queue and added to the free list without causing use-after-free problems is discussed. However what is not covered, and may cause potential process memory disclosure, is a scenario where a thread performs an RPC call to write to memory. Normally the system disallows any further memory-related RPC calls using this address due to the ongoing memory access, however if the thread were to be removed from the run queue and added to the free list prior to the memory RPC call returning, the node performing the memory access (e.g. write) would continue to write to physical pages that are no longer allocated to a process. It is mentioned that pages are zeroed as they are freed - and not as they are allocated. Such behaviour could result in another process receiving pages upon request for allocation that contain remnant data from the memory write request that was performed - potentially leaking important data.

Another potential vulnerability lies in the memory management system, that is unfortunately not sufficiently touched on to confirm. Processes have the ability to change ownership of pages to other processes in order to share data, as shared memory is intentionally not supported. Unfortunately the semantics of this is not touched on. Depending on implementation, a vulnerability may exist where-by Process-A has a page of memory containing function pointers (for example, in an object Vtable). Process-B could use the ability to change ownership of a crafted page of its own to be mapped over the page containing the function pointer in Process-A's address space. Thus changing the control flow of Process-A to wherever Process-B wanted to within Process-A's address space when the function pointer within the crafted page is called. The semantics of how transfer of page ownership is handled is not covered - so the aforementioned or a derivative of it may be possible.

Criticisms

The authors make multiple claims throughout the paper without any form of proof - which stems from the fact that no benchmarks were performed. It is mentioned throughout the paper that the system is in an early form with work to be done, however claims of the system's capability are made without any form of backing. Such examples include an apparent improvement in system parallelism and the reliable delivery of packets between any two endpoints. An explanation of how these benefits are met would have been useful.

The paper has been released at a state where the system has not been fully implemented - and many claims of security through hardware component isolation are made. The authors mention that the only components of the system that have undergone formal verification are the network related sub systems. Given that the paper does not provide anything ground breaking, it may have been a good idea to have waited until the system was more mature in development before releasing a paper.

A concept of a node-based *security policy* is referenced throughout the document, but the authors do not touch on nor do they explain its purpose or how it is managed. More detail on this would have been beneficial - as it seems it would play a core role in preventing malicious misuse of a node.

When comparing with related work, the authors fail to outline how their kernel design is superior,

outside the fact of implementation differences. This is evident in the discussion of the *zero-kernel operating system* which was built to meet the same ends, but did so using a slightly different communication protocol. The authors fail to explain how their design differs in meaningful and beneficial ways.

It is stated that physical access to the system is not prioritised in the threat model. However the authors mention that existing anti-tamper techniques can be used to produce a system with some degree of robustness against physical tampering. This is mentioned without any examples of how this could be done, especially in the unique context of a modular operating system that communicates through a on-chip networking interface. More could have been said about how these techniques could be applied to such a system.

References

[1] Andrew D. Zonenberg; Bulent Yener. Antikernel: A decentralized secure hardware-software operating system architecture. Cryptology ePrint Archive, Report 2016/550, 2016. `http://eprint.iacr.org/2016/550`.