Analysis of Monolithic and Microkernel Architectures: Towards Secure Hypervisor Design

Dr. Jordan Shropshire
Associate Professor
Department of Information Technology
Georgia Southern University
jshropshire@georgiasouthern.edu

Abstract
This research focuses on hypervisor security from a holistic perspective. It centers on hypervisor architecture—the organization of the various subsystems which collectively compromise a virtualization platform. It holds that the path to a secure hypervisor begins with a big-picture focus on architecture. Unfortunately, little research has been conducted with this perspective. This study investigates the impact of monolithic and microkernel hypervisor architectures on the size and scope of the attack surface. Six architectural features are compared: management API, monitoring interface, hypercalls, interrupts, networking, and I/O. These subsystems are core hypervisor components which could be used as attack vectors. Specific examples and three leading hypervisor platforms are referenced (ESXi for monolithic architecture; Xen and Hyper-V for micro architecture). The results describe the relative strengths and vulnerabilities of both types of architectures. It is concluded that neither design is more secure, since both incorporate security tradeoffs in core processes.

1. Introduction
To achieve marked improvements in hypervisor security, it may be necessary to return to the drawing board. The majority of studies in this research space have so far provided incremental improvements in security. Reconsidering the relationship between architecture and system risk could lead to significant reductions in risk. The present study takes an initial step toward this goal by examining the vulnerabilities and security tradeoffs in hypervisor architectures. Specifically, it compares monolithic and microkernel architectures. Monolithic hypervisors are completely contained within a single entity. Features such as hardware drivers, application program interfaces, and the virtualization stack are integrated into a thin layer of software which supports virtualization. This platform exists wholly within security ring 0. By contrast, microkernel architectures consist of a relatively small hypervisor and a separate management partition. In this design, the hypervisor contains only the kernel, while the associated management partition contains drivers, virtualization managers, and administrative interfaces. Although both architectures perform the same risky processes, this analysis will indicate that they have different weaknesses and points of resilience [1]. An appreciation of the security tradeoffs in each design is necessary in order to make informed acquisition decisions. By considering architectural weaknesses prior to developing new platforms, developers can integrate better controls and security features into their designs.

To understand the vulnerabilities in monolithic and microkernel architectures, this research focuses on six hypervisor subsystems. The subsystems serve as passages through the hardened perimeter. They have even been described as liabilities in separate studies. If compromised, they may be used as potential attack vectors [2, 3]. They are: management interfaces, monitoring systems, hypercalls, interrupts, I/O and networking. These interfaces serve as avenues into and out of the hypervisor and represent major attack vectors [2]. To gauge the scope of their attack surfaces, the architectures are compared in terms of these subsystems.

The remainder of this paper is organized as follows: The following section provides background information on monolithic and microkernel architectures. It introduces examples of both architectures and discusses security issues related to security rings, trusted code base. After the background, the third section describes the threat model. The fourth section analyzes the security of monolithic and microkernel architectures. It considers the impact of each subsystem on the scope of the attack surface. Finally, the last section provides concluding remarks and hints toward future research.
2. Background

Prior to conducting the analysis, some background information is necessary. First, this section describes the basic components of hypervisors. Next, it addresses two relevant design issues: ring configuration and trusted code base. These factors influence the hypervisor attack surface. Finally, this section provides basic information on monolithic and microkernel architectures. The security implications of these architectures will be addressed in the analysis section.

2.1 Hypervisors

All Type 1, bare-metal hypervisors perform the same basic job – they act as an intermediary between virtual machines and underlying hardware. In order to share resources with hosted virtual machines, they abstract system hardware and broker usage requests. Virtual machines may send these requests in the form of hypercalls. A hypercall is a paravirtualization concept in which the virtual machine kernel is altered. Instead of sending system calls which are intercepted by the hypervisor, they send deliberate system calls to specific subsystems. Hypercalls are similar to system calls in that they bridge user-mode applications with kernel functions. Portions of the original virtual machine kernel are replaced with software that is engineered to interact with the hypervisor directly. In response, hypervisors use interrupts to communicate with virtual machines. An interrupt is a signal sent to a processor indicating that an event needs immediate attention. The hypervisor receives these signals and determines which virtual machine should receive the interrupt. It also must handle traps and exceptions resulting from one virtual machine while not impacting the hypervisor or other virtual machines.

To share processing capabilities among virtual machines, a scheduler is used. It abstracts the CPU into one or more virtual CPUs (vCPUs) for each virtual machine. The vCPU appears as a normal CPU to virtual machines. To take advantage of multiprocessing, a host can be configured with multiple vCPUs. A vCPU in the hypervisor is akin to a process in a traditional operating system. Although a number of different scheduling algorithms exist, they all perform the same basic task. Resources such as networking and storage are also abstracted. They are usually represented by emulated devices, with a separate instance for each virtual machine. Even though their architectures differ, all major hypervisors use the same basic techniques to support virtual machine activity.

2.2 Security Rings

A defining feature of hypervisor architectures is the arrangement of security rings [4]. A security ring is a hierarchical level of privilege within a computer system [5]. It acts as a gated community. Security rings are arranged from most privileged to least privileged. In most modern systems, they are numbered 0 through 3, where 0 is the innermost ring and 3 is the outermost. Ring 0 is usually reserved for the OS kernel. It interacts with hardware such as CPU and memory. If rings 1 and 2 are used, they correspond with privileged mode. This may include device drivers. Ring 3 is for user mode. It consists of untrusted applications. For the purposes of this research, ring 0 corresponds with the hypervisor, ring 1 corresponds with virtual machine kernel space, and ring 3 corresponds with virtual machine user space. This arrangement mirrors descriptions from earlier studies.

Security rings limit the capabilities of executing programs by regulating access to shared memory. They provide a security buffer against accidental or intentional corruption. A fault or exception at an outer ring has less impact on system stability than an attack in an inner ring. Minimizing software access to kernel mode reduces security risks. With less trusted software there is a reduced likelihood of system destabilization. A system of flags is used to associate permission level with specific memory segments. A gate feature enforces these permissions. In order to cross between rings, a context switch occurs. A common design goal is to minimize the number of context switches a subsystem must make in order to perform a common task [6]. Correctly gating access prevents programs with reduced privileges from misusing resources that belong to trusted rings. Security rings correspond with CPU modes in some systems. This provides added hardware protection. It is noted that some hypervisors make use of advanced CPU features to run in special hardware-enabled security modes.

2.3 Trusted Code Base

The Trusted Code Base (TCB) includes components which support core system processes. For bare-metal hypervisors, this includes the hardware, drivers, operating system, and software for abstraction and resource brokering. Its main responsibility is to ensure the system’s basic functionality and security [7]. As with many earlier studies of hypervisor security, this research uses the definition of TCB that is taken from the “Orange Book” [8]. Also known as
the Trusted Computer System Evaluation Criteria (TSEC), it describes TCB as “all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based.” The bounds of the TCB equate to the security perimeter referenced in some computer security literature. TSEC also states that the TCB should be as small as practically possible, considering the processes it must execute. It must be implemented in such a way that non-essential elements are excluded. Non-essential elements can be defined as components which do not need to be trusted in order to maintain normal operations [9]. To date, there has been little effort to define the components that hypervisors must consider to be part of their trusted code base.

2.4 Monolithic and Microkernel Architectures

All hypervisors perform the same basic tasks: starting and maintaining virtual machines, abstracting system resources, and sharing hardware and assets. Despite having a common purpose, their designs differ in order to meet specific performance expectations, embody desired characteristics, emphasize certain roles, or limit inherent weaknesses [1]. Their differences can be summarized in terms of two common hypervisor designs: monolithic and microkernel architectures (see Figure 1).

Figure 1: Microkernel (left) and Monolithic (right) Architectures

Microkernel architectures are designed to minimize code in the hypervisor space. Microkernel architectures retain few components within the actual hypervisor, making use a separate partition to handle networking, storage, hypercalls, and other support functions [10]. The management partition contains an independent operating system. It hosts the components which were shuffled out of the hypervisor. The management partition is considered part of the TCB. This means that part of the trusted code base is retained outside of ring 0 in user space. It is given direct access to the kernel and system resources. It acts as an intermediary between virtual machines and the hypervisor kernel. Because the microkernel architecture makes use of a separate partition, the system is divided between the hypervisor and the management plane.

To support virtual machines, the microkernel hypervisor traps hardware requests and transfers them to the management partition. The partition processes the requests, funneling them through a virtualization stack before calling hardware drivers. Two widely-used hypervisors can be classified as having microkernel architectures. The first is Xen, an open-source platform sponsored by Citrix. Citrix sells cloud solutions which integrate modified versions of Xen. The second platform is Hyper-V by Microsoft. Although it is a relatively new entrant, it has a growing user base. Although Xen and Hyper-V utilize microkernel designs, their architectures differ in some respects. In order to accurately compare hypervisor architectures, both platforms are included in this analysis.

In contrast to microkernel designs, monolithic architectures condense all the subsystems within a single entity. This includes the kernel and all of its subsystems, interfaces, and drivers. Compared to microkernel hypervisors, the monolithic architecture has a larger footprint. The entire hypervisor resides in ring 0, within kernel space. This means that the sum of its trusted code base lives in the most secure area. Unlike the microkernel architecture, this design does not incorporate a management partition. Virtual machines are located directly above the hypervisor, where a virtual machine monitor interacts with each guest. The virtual machine monitor receives hardware requests and from within the hypervisor. Like microkernel architectures, monolithic architectures also support hypercall communication. However, the monolithic architecture does not integrate a separate partition. For administration, external entities connect directly to the hypervisor via an application program interface. An example of a hypervisor with a monolithic architecture is the ESXi platform by VMware. ESXi is a bare-metal hypervisor offered as part of a larger cloud solution. Its major components are included the layer between the virtual machines and the hardware.

3. Threat Model

It expected that aggressors will have logical access to all hypervisor interfaces, but no physical access. Hackers are expected to pursue any weaknesses in the hypervisor attack surface. Using
classical information security theory, the attack surface is defined as any part of the software that can be manipulated or is a potential vulnerability. An attack might stem from an entity with a virtual machine which is legitimately hosted on the hypervisor or it may be stimulated by an outsider. In the latter case, an attack might begin with a port scan of all virtual machines on the network. Finding a host with an open port, an attacker could compromise the associated application in order to gain entry into the virtual machine. Such attacks have been documented in the National Vulnerabilities Database [11]. Once access is secured, the next step in the attack is to elevate privileges. A number of approaches have surfaced for achieving root level permissions[12]. The purpose of the attack may be to destabilize the hypervisor, compromise other virtual machines, or disrupt hosted cloud services [13]. The attacker could focus on any number of vulnerable hypervisor subsystems [14].

4. Analysis

This section describes the analysis of six features of hypervisors. For all of the elements, comparisons between monolithic and microkernel architecture are made. The analysis focuses on differences in subsystem footprint, distribution of trusted code, and impact of security ring configuration.

4.1 Management Interface

This section notes the vulnerabilities and security tradeoffs in the management interface. The management interface allows remote agents to connect with a hypervisor and perform managerial duties such as initiating virtual machine migrations, provisioning hardware, and reconfiguring resource pools. An external agent may be automated or manually controlled. The management interface makes a prime target for attack. Once access is gained, attackers can manipulate software objects and classes for deviant purposes [15]. For the purpose of this research, the attack surface is interpreted as the breadth of classes and modifiable objects available to the application program interface [16].

Microkernel architectures tend to require more configuration options. They need the flexibility to configure the operating system which controls the management partition. Hyper-V was found to support 14 main groups of classes and many thousands of objects. The majority of the objects are not associated with virtualization, but other functions built into the server in the management partition. This interface offers a high degree of flexibility in customizing the hypervisor, virtual machines, networking, and I/O. To cope with the risk of an extensive API, an access control platform is included. The Xen management interface includes 45 classes of objects. Access to all of these classes of objects creates many opportunities for malfeasance. Xen also includes fine-grained controls for limiting system access and enforcing permissions. The management interfaces for the microkernel hypervisors have relatively large attack surfaces. This is because they also support the administration of an operating system in the management partitions.

Unlike microkernel designs, monolithic architectures do not need to manage operating systems in separate partitions. Therefore, the management interface for the monolithic architecture requires fewer classes than the microkernel architecture. For instance, the ESXi management interface consists of 21 device profiles which can be modified to various degrees. To restrict user activity and improve security, ESXi makes use of a stripped down interface called BusyBox. This executable allows for basic file system interaction while restricting other usage. Based on number of objects in the management API, it appears that the monolithic architecture has a smaller attack surface. However, this does not mean that the microkernel architecture is better. Although the constrained management interface reduces the attack surface, it limits system customization because flexibility and supporting utilities are not available.

4.2 Monitoring

The monitoring interface allows external devices to query a hypervisor, receive metrics of system state, prepare virtual machine performance reports, or poll hardware. Attackers would be interested in tapping into the monitoring system and changing reported metrics [4]. They could underreport system load in order to hide illicit activities. To do this, they would exploit a seam in the monitoring framework [17]. Larger footprints provide more opportunities for exploitation [18]. Therefore, this study considers the size of the monitoring system’s footprint in terms of the number of steps taken to receive a request, retrieve the monitoring data, and respond to the request. These metrics correspond with those used in an earlier study of hypervisor architecture [18, 19].

Hypervisors patterned in the microkernel architecture were considered first. Xen was found to be the longest at 14 steps. It was top heavy; 9 steps were conducted in ring 3 (64.29%), 2 steps were conducted in ring 1 (14.29%), and 4 steps were completed in ring 0 (21.49%). Hyper-V was shorter than Xen at 13 steps. 5 of these steps occurred in ring 3 (38.46%), 5 were in ring 1 (38.46%), and 3 were in ring 0 (23.8%).
For purpose of illustration, the Hyper-V monitoring process is depicted in Figure 2 and the 13 steps are described here: (1) the virtual machine management service (VMMS) receives a request for monitoring data. (2) The VMMS transfers to the WMI-provider (windows management instrumentation). (3) The WMI reads the request and triggers the virtual interface driver (VID) in ring 1 to gather the performance data. (4) VID passes the request down the virtstack. (5) The virtstack references the WinHV library in order to formulate the request. (6) The root partition issues a hypercall to update the counter. (7) Hardware performance is observed (8) Collected statistics are moved to a shared memory page for retrieval by root partition. (9) The hypervisor interrupts the root partition with the results. (10) The performance data is moved up the virtstack to the VID. (11) The virtual machine kernel interrupts the WMI with the results. Context switches from ring 1 to ring 3. (12) When the metric is prepared, the operation is passed to the VMMS. (13) The interface finishes the request by sending one or more datagrams across the network.

The ESXi hypervisor, an example of monolithic architecture, required 10 steps. All steps were completed in ring 0. The difference in process length may be attributed to the inclusion of the management plane. Monolithic architectures are more compact. While they leave a smaller attack surface they force all software into ring 0, risking the trusted code base.

The analysis shows that Xen and Hyper-V support 100 and 113 hypercalls, respectively, while the ESXi 3.5 VMI module uses 82 hypercalls. It appears that microkernel architectures require more hypercalls than monolithic architectures. This is because microkernel hypervisors must provide hypercalls for supporting virtualization processes and for interaction with the OS in the management partition. Although microkernel architectures have a larger attack surface, they are not necessarily more vulnerable than monolithic architectures.

Because a simple count of hypercalls provides one-dimensional perspective, a deeper inspection is necessary. To characterize the potential damage a hypercall library could invoke, this study projected the impact that each hypercall could cause if it was used maliciously. Hypercalls for the ESXi, Xen, and Hyper-V platforms were sorted into one of four groups based on the degree of potential damage. The first group of hypercalls is characterized as non-critical threats. The second group consists of hypercalls which could be used for passive activities such as system blueprinting or data gathering. The third group consists of hypercalls which could cause temporary system damage. The fourth group consists of hypercalls which could cause long-term damage. This method of assessment is based on an earlier study of Xen hypercalls [17].

The results of the categorization indicate that the monolithic architecture supports more dangerous calls. Over 75% of ESXi hypercalls fell into the third or fourth categories, indicating that they can cause short term or long term damage. This is in contrast the microkernel architectures, which supported more benign hypercalls (see Table 1). A key architectural feature which explains this difference is the inclusion of a management partition. The management partition serves as a buffer, processing hypercalls while minimizing access to the hypervisor kernel. The monolithic architecture lacks such a buffer and processes hypercalls within the hypervisor. The entire process is performed in ring 0, potentially risking the TCB. To recap, even though the monolithic architecture requires fewer hypercalls, those which are included present greater security risk.

4.3 Hypercalls

This section analyzes weaknesses inherent in the hypercall subsystem. This interface is susceptible to attack by rooted virtual machines [20]. Virtual machines use hypercalls to communicate with hypervisors. Each hypervisor platform supports a library of hypercalls. In the wrong hands, these expedited hardware requests can be used for a number of attacks [15]. They can be used to conduct DoS attacks against other VMs or aimed at the hypervisor itself. With respect to the attack surface, it is assumed that larger hypercall libraries increase risk [21]. They provide more tools with which to attack the hypervisor or deprive other virtual machines of hardware resources.
Table 1: Ranking of Hypercalls for Microkernel (left and middle) and Monolithic (right) Architectures

<table>
<thead>
<tr>
<th>Hypercall Category</th>
<th># Calls</th>
<th>Hypercall Category</th>
<th># Calls</th>
<th>Hypercall Category</th>
<th># Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class 1:</strong> Non Critical Threat</td>
<td>2</td>
<td><strong>Class 1:</strong> Non Critical Threat</td>
<td>0</td>
<td><strong>Class 1:</strong> Non Critical Threat</td>
<td>3</td>
</tr>
<tr>
<td><strong>Class 2:</strong> Prospective Information Gathering for Illicit Purposes</td>
<td>19</td>
<td><strong>Class 2:</strong> Prospective Information Gathering for Illicit Purposes</td>
<td>7</td>
<td><strong>Class 2:</strong> Prospective Information Gathering for Illicit Purposes</td>
<td>16</td>
</tr>
<tr>
<td><strong>Class 3:</strong> Causation of Temporary or Reversible Damage</td>
<td>20</td>
<td><strong>Class 3:</strong> Causation of Temporary or Reversible Damage</td>
<td>20</td>
<td><strong>Class 3:</strong> Causation of Temporary or Reversible Damage</td>
<td>40</td>
</tr>
<tr>
<td><strong>Class 4:</strong> Causation of Permanent or Irreversible Damage</td>
<td>13</td>
<td><strong>Class 4:</strong> Causation of Permanent or Irreversible Damage</td>
<td>11</td>
<td><strong>Class 4:</strong> Causation of Permanent or Irreversible Damage</td>
<td>19</td>
</tr>
</tbody>
</table>

4.4 Interrupts

The other aspect of virtual machine-hypervisor communication is the interrupt subsystem. Interrupts are used to communicate with virtual machines. Hypervisors emulate the system hardware which is ordinarily used to generate interrupts. Though seemingly harmless, a rapid flow of interrupts will overwhelm a virtual machine, impeding its ability to support the cloud [11]. To do this, attackers would need to gain access to the interrupt system and load the conduit with trivial interrupts.

The location of the emulator and its interrupt descriptor table has implications for information security. Emulators which are located in user space are more susceptible to tampering. An attacker could trigger interrupts aimed at specific virtual machines, causing denial of service. In contrast, emulators which are located in privileged zones are harder to reach. However, if they are used as an attack conduit they would give access to the hypervisor kernel. The microkernel architecture features an interrupt emulator in the management partition. The emulator exists in ring 3. They operate with minimal permissions but exist in a zone which has more attack vectors. The monolithic architecture retains its interrupt emulator in the hypervisor. The emulator operates with more permissions than necessary, but exists in a more secure location.

4.5 Networking

This section describes the security risks and tradeoffs affecting the networking component. The network subsystem provides each virtual machine with its own emulated network adapter. Traffic leaving a virtual machine is routed through the virtual network, passed down the virtualization stack, and through the physical NIC before entering the network. The network subsystem encompasses a number of different elements, including hardware, drivers, shared storage, and virtualization software. The network subsystem is a primary target for attack [17]. If a virtual machine can be starved of traffic then the cloud will be disrupted. Alternatively attackers may seek to read or modify network packets by tapping into the network stack. It is necessary to minimize the size of this attack surface and expose as little as possible. For the purposes of this assessment, the attack surface is based on the distribution of trusted code and the number of steps in a basic network operation. This approach is based on an earlier study of the risks in backend interfaces [18].

Regarding code distribution, microkernel architectures split the network process between the management partition and the hypervisor. The network subsystem is divided between kernel space in the hypervisor and user space in the partition. However, the impact of an attack will be limited to the management partition. In contrast to the microkernel
architecture, the entire network process occurs in ring 0 for monolithic architectures.

To compare the length of the network subsystems, we counted the number of steps in the process of retrieving data from virtual machines, passing it through the virtualization stack, and sending it out network adapters. A combination of manual code inspection and stack trace was used in this analysis. The results indicate that the distributed subsystem in the microkernel architecture has a longer path. The network process took 31 steps for Hyper-V, the longest of all hypervisors. 14 of these steps occurred in ring 3 (45.16%), 12 occurred in ring 1 (38.71%), and 5 steps in ring 0 (16.13%). The network process took 25 steps for the Xen hypervisor. 9 of the steps occurred in ring 3 (36%), 10 occurred in ring 1 (40%) and 6 occurred in ring 0 (24%).

Compared with the microkernel design, the network process in the monolithic architecture is more condensed. ESXi required just 20 steps to collect data from virtual machines and send it across the network. Only 1 of the 20 steps occurred in ring 3 (5%), 3 steps occurred in ring 1 (15%), and the remaining 16 occurred in ring 0 (80%).

For purposes of illustration, the ESXi network process is depicted in Figure 3 and the 20 steps are described here: (1) Application issues a system call to initiate an asynchronous network operation. (2) The guest OS starts the network operation. (3) The hypervisor traps the instruction and copies the packet into a temporary data store in ring 0. (4) The hypervisor returns to the guest OS in ring 1. (5) The guest returns to the appropriate application in ring 3. (6) The emulated NIC (vlance instance for strict virtualization) device polls the temporary store and retrieves the packet. (7) The emulator queries VMFS for the MAC address with the packet. (8) The associated MAC address is returned for pairing with the datagram. (9) The formed packet is retrieved by the virtual switch for routing. (10) The packet exits the virtual switch process and enters the network stack. (11) The network stack finalizes the frame header and passes it to the drivers. (12) As the packet is being sent, a message is passed up the stack to inform the appropriate virtual NIC. (13) The device driver transmits the packet onto the network. (14) Sometime later, the hypervisor receives a response packet and stores it in a temporary buffer. (15) The packet is passed up the network stack. (16) The received packet elements enter the virtual switch for processing. (17) The emulated NIC accepts the packet and reformats the header. (18) The packet is loaded into shared storage using the VMbus. (19) An interrupt is sent to the virtual machine kernel, alerting to the received packet. (20) The guest kernel interrupts the guest application.

It appears that the network subsystem is more distributed in the microkernel architecture. This minimizes the size of the hypervisor and improves its stability. This comes with the tradeoff of forcing trusted code to execute outside of ring 0 [22]. In contrast with the microkernel design, the monolithic architecture features a more condensed network subsystem. The majority of the network process occurs in kernel space. From a design standpoint, this makes the network subsystem’s vulnerable as well as increases risk to the hypervisor kernel. With more lines of code in ring 0, the potential attack surface increases.

4.6 Storage

The I/O subsystem is another backend interface. It abstracts a physical device and provides a unique software emulation for each virtual machine. Although all hypervisors perform the same basic storage process, their I/O subsystems differ in several respects [6]. For the monolithic architecture, the I/O process is condensed into hypervisor ring 0. This simplifies the architecture but increases kernel vulnerability. As more code is executed in trusted mode, the risk to the kernel increases and it becomes harder to distinguish between trusted and standard code.

The microkernel architecture stretches the I/O process across several rings – from ring 3 in the
management partition to ring 0 in the hypervisor. Because the I/O subsystem is part of the trusted code base, it should be retained in restricted areas. However, the microkernel architecture locates part of the subsystem in less secure areas. Although this increases the risk that the I/O process will be compromised, it reduces access to the hypervisor kernel.

For each of the architectures, the length of the I/O path was measured in terms of steps. This approach to assessing I/O process length was used in a previous study of hypervisor security [18, 23]. This begins with a virtual machine’s request to write data to storage, winds through the hypervisor virtualization stack, and ends with a completed write cycle. The storage process in the microkernel architecture was found to be longer. It passes through a management partition before entering the hypervisor.

Hyper-v was found to consist of 25 steps. 4 steps occurred at ring 3 (16%), 15 steps occurred at ring 1 (60%), and 6 occurred in ring 0 (24%). At 23 steps, Xen is nearly as long. Its ring 3 contains 6 steps (26.09%), ring 1 consists of 12 steps (52.17%), and ring 0 includes 5 steps (21.74%).

To illustrate the storage process, the Xen I/O architecture is depicted in Figure 4 and the 23 steps are described here: (1) The application issues a system call to initiate an asynchronous I/O operation. (2) The guest OS starts the I/O operation by passing data through the file structure. (3) The instruction that starts the I/O is trapped by the hypervisor. (4) The hypervisor returns to the guest OS given that this is an asynchronous I/O request. (5) The guest returns to the appropriate application. (6) The hypervisor passes the I/O operation to Dom0 via the XenBus. (7) The OS in the root partition passes the request to the IDE Emulator in the virtualization stack. (8) The IDE emulator passes the request to the Xenstore agent. (9) The trapped data is sent from Xenstore to the virtual file system. (10) The request is sent to the I/O stack in the kernel ring. (11) The I/O stack sends the data to the device drivers residing in Dom0. (12) The root partition uses the IOMMU to start the write process. (13) The OS returns to the virtual ring 3 software given that this is asynchronous. (14) After the write operation is complete, an interrupt is sent to the hypervisor. (15) The hypervisor interrupts Dom0 to let it know the I/O operation finished. (16) The device driver passes this message to the I/O stack. (17) The root partition updates the virtual file system. (18) The virtual hard disk informs the Xenstore that the I/O operation has completed. (19) Xenstore updates the IDE emulator. (20) The IDE emulator completes the I/O operation and issues a system call to wake up the guest. (21) The root partition issues a hypercall to the hypervisor to update the guest. (22) An I/O complete interrupt is delivered to the guest via the Xenbus. (23) The guest OS sends a corresponding interrupt to the application to alert it that the I/O operation completed.

Based on these results, it appears that the bulk of the storage process occurs in ring 1 for microkernel hypervisors. Ring 1 contains the virtualization stack and the device drivers. Compared with microkernel hypervisors, the storage process in the monolithic hypervisor is relatively short. It includes 16 steps. 1 of these steps occurs in ring 3 (6.25%), 3 occur in ring 1 (18.75%), and 12 occur in ring 0 (75%). The only parts of the I/O process which occur outside of the hypervisor involve virtual machine interaction. The implication is that more privileges are given to the I/O subsystem than is necessary, putting the hypervisor kernel in jeopardy.

5. Conclusion

This manuscript explored the vulnerabilities and security tradeoffs in monolithic and microkernel architectures. It assessed the impact of six architectural features on the scope of the attack surface. A number of architectural differences had implications on the attack surface. One of the primary differences between architectures is the distribution of trusted code into a separate management partition. This improved hypervisor kernel isolation but increased the vulnerability of subsystems which had been migrated out of the hypervisor. Because both architectures have inherent weaknesses it is not possible to conclude that
one is more secure than the other. Regardless of the size of the footprint or the type of architecture, well-written and tested kernel code will provide better security. At best, it could be said that they meet different, competing objectives. Microkernel designs have minimal kernels and offer highly customizable management planes. They allow for tailored solutions and simplify the development process. Monolithic architectures are preconfigured with necessary drivers and software in the trusted zone. They require less domain knowledge to set up and provide fewer options for tampering.

This analysis will prove valuable to both IT professionals and software engineers. Enterprise architects and information security professionals are often forced to make acquisition decisions will less than complete information. This research identified a number of factors to consider before implementing new technologies. However, the present study should not be considered comprehensive. Other design factors should be elicited in future research. This study should also be of interest to software engineers. Often, security is considered late in the software development cycle. By analyzing security implications at the architectural level, this research attempts to incorporate security concerns earlier in the design process. Future studies should focus on approaches to compensating for security weaknesses in each type of architecture.

6. References


