PIkit: A New Kernel-Independent Processor-Interconnect Rootkit

Wonjun Song, Hyunwoo Choi, Junhong Kim, Eunsoo Kim, Yongdae Kim, and John Kim,
Korea Advanced Institute of Science and Technology (KAIST)

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/song

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

PIkit : A New Kernel-Independent Processor-Interconnect Rootkit

Wonjun Song, Hyunwoo Choi, Junhong Kim, Eunsoo Kim, Yongdae Kim, John Kim
KAIST
Daejeon, Korea
{*iamwonjunsong, zemisolsol, jh15, hahah, yongdaek, jjk12*}*@kaist.ac.kr*

Abstract

The goal of rootkit is often to hide malicious software running on a compromised machine. While there has been significant amount of research done on different rootkits, we describe a new type of rootkit that is *kernel-independent* – i.e., no aspect of the kernel is modified and no code is added to the kernel address space to install the rootkit. In this work, we present *PIkit* – Processor-Interconnect rootkit that exploits the vulnerable hardware features within multi-socket servers that are commonly used in datacenters and high-performance computing. In particular, PIkit exploits the DRAM address mapping table structure that determines the destination node of a memory request packet in the processor-interconnect. By modifying this mapping table appropriately, PIkit enables access to victim's memory address region without proper permission. Once PIkit is installed, only user-level code or payload is needed to carry out malicious activities. The malicious payload mostly consists of memory read and/or write instructions that appear like "normal" user-space memory accesses and it becomes very difficult to detect such malicious payload. We describe the design and implementation of PIkit on both an AMD and an Intel x86 multi-socket servers that are commonly used. We discuss different malicious activities possible with PIkit and limitations of PIkit, as well as possible software and hardware solutions to PIkit.

1 Introduction

Rootkits are used by attackers for malicious activities on compromised machines by running software without being detected [47]. Different types of rootkits can be installed at the application-level, kernel-level, boot loader level, or hypervisor level. There has been significant amount of research done on different types of rootkits [57, 28, 25, 24] as well as different rootkit detections [21, 31, 11]. Recently, there have been other types of rootkits [20, 50] that exploit vulnerable hardware features such as de-synchronized TLB structures and unused interrupt vector. Prior work on rootkit can be classified based on whether the payload consists of user-level or kernel-level code and whether rootkit is installed in the software or with the support of the hardware (Table 1). In this work, we propose a new type of rootkit that modifies hardware state but enables malicious activities with simple user-level code that consists of read/write memory accesses to user-level memory space. While prior work on user-level software rootkit often modified existing files [35, 48, 23], this work does not modify the kernel or any existing files. Since this work does not require any code modification or code injection to the kernel, traditional approaches to detect software rootkit, such as kernel integrity monitoring [42, 31, 53] or code signature-based detection [1, 2] can not be used for detection.

		Malicious Payload	
		User-Level	Kernel-Level
Vulnerabilities	Software	t0rn [35], lrk5 [48], dica [23], etc.	ROR [25], DKOM [15], knark [17], etc.
	Hardware	This work (PIkit)	Cloaker [20] Shadow Walker [50]

Table 1: Classification of different rootkit attacks.

In this work, we present PIkit, processor-interconnect rootkit, that exploits hardware vulnerability in x86 multi-socket servers. x86 is the most dominant server processor in datacenters and high-performance computing [36] and a recent survey found over 80% of the x86 servers are *multi-socket servers*. The multi-socket servers contain a processor-interconnect that connects the sockets together (e.g., Intel QPI [37], AMD Hypertransport [10]) and we exploit the processor-interconnect to implement PIkit. Once PIkit is installed, the payload or the malicious code to carry out an attack exists in user space and appears like a "normal" user program – i.e., all of the memory accesses from the payload are legal memory accesses and it becomes very difficult to identify such user code as "malicious" code. As a result, PIkit is a seri-

ous threat to multi-socket servers that is difficult to detect with currently available rootkit detection mechanisms.

PIkit that we propose is implemented on x86 servers from both AMD and Intel to demonstrate how PIkit enables an attacker to continuously access the victim's memory region without proper permission. [1] In particular, we exploit the configurability in the DRAM mapping table that enables a memory request packet to be routed to a different node by modifying a packet's destination node. We also exploit the extra entries available in the DRAM mapping table to define an *attack memory region* when installing PIkit. As a result, user-level memory read or write requests to the attack memory region get re-routed to another memory region or the *victim's memory region*. To the best of our knowledge, this represents the first rootkit where with the support of hardware state modification, user-level code or payload is sufficient to carry out malicious activities.

Most rootkits often modify some components of the OS while other rootkits add malicious payload to the kernel without modifying the OS to carry out malicious activities. However, such approaches can be exposed by signature-based detection and integrity checking. In comparison, PIkit only requires user-level payloads with the support of hardware state modifications as no malicious payloads to the kernel space are added or modified (Figure 1). In addition, any signature scan of the memory that contains the user-level payload can not identify the user code as "malicious" since the memory accesses appear to be legal accesses as the malicious access is only achieved through the support of the hardware modifications. As a result, PIkit demonstrates how a very stealthy rootkit can be achieved compared to previously proposed rootkits.

The proposed PIkit is a non-persistent rootkit [20, 45, 41] and does not remain after the server is restarted. However, servers are rarely rebooted to minimize the impact on availability – for example, one study measured the average time between reboot in the server room to be 481 days [22]. Thus, PIkit poses a serious threat to servers while powered on. Prior work on non-persistent rootkit [20] has argued that non-persistence can also significantly reduce the detectability of rootkits.

In particular, the contributions of this work include the followings.

- We show that the DRAM address mapping table structure in the processor-interconnect of multi-socket servers has security vulnerability that can be exploited maliciously in both an AMD and an Intel-based x86 server.

[1] This work can also be viewed as a "backdoor" since once PIkit is installed, it provides a covert mechanism for the attacker to gain privileged access to the system.

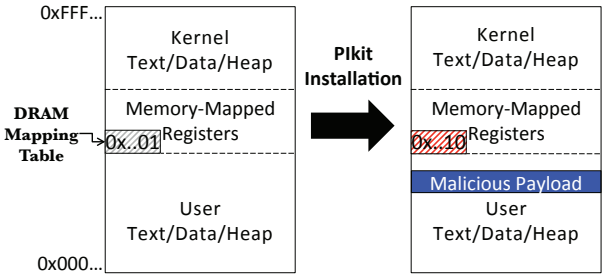

Figure 1: High-level overview of PIkit showing DRAM address mapping table modification and user-level malicious payload added for malicious activities.

- We describe a new type of rootkit that is *kernel-independent* that requires only hardware state modification with user-level payload, no modification or addition to the kernel is necessary. In particular, we present PIkit, Processor-Interconnect rootkit, that exploits the mapping table vulnerability to enable the malicious attacker privileged access (both read and write) to a victim memory address region with only user-level access.
- Once PIkit is installed, we demonstrate how different malicious activities can be carried out including bash shell credential object attack, shared library attack, and keyboard buffer attack with only user-level memory accesses.
- We describe alternative solutions, including a software-based, short-term solution to detect PIkit as well as hardware-based, long-term solutions to prevent PIkit.

We responsibly disclosed this vulnerability to CERT before publishing this paper. The rest of the paper is organized as follows. We first describe our threat model in Section 2 and background into processor-interconnect as well as related work. The DRAM address mapping table structure is described and analyzed in Section 3. The design and implementation of PIkit that modifies the mapping table structure is described in Section 4 and we illustrate different malicious activities in Section 5. We provide some discussion on different solutions as well as limitations of PIkit in Section 6 and we conclude in Section 7.

2 Background

2.1 Threat Model

In this work, we assume an attacker and a victim share the same multi-socket server that is commonly used in cloud servers and high-performance computing. We assume an attacker has no physical access to the hardware and also assume the same threat model as prior work on rootkit attack – the attacker, through some vulnerabilities (e.g., vulnerabilities in commodity OSes [39, 16, 6] or perhaps through an administrator (or an insider) who

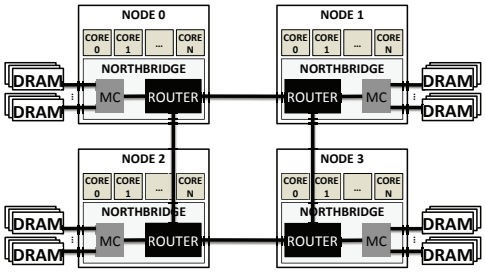

Figure 2: Block diagram of a processor-interconnect in a 4-socket server. (MC: memory controller)

maliciously provides one-time root access [54]) or social engineering, is assumed to have privileged access for rootkit installation. Once this is achieved, the next goal is to avoid detection of intrusion while carrying out malicious activity, similar to other rootkits. After PIkit is installed, it becomes very difficult to detect or determine the source of the attack as there are no changes to the kernel in the target system.

2.2 Processor-Interconnect Overview

A high-level block diagram of a processor-interconnect is shown in Figure 2. The processor-interconnect provides connectivity between multiple nodes (or sockets) in a NUMA (non-uniform memory access) server with each node containing multiple cores. The router within the Northbridge is used to connect to other nodes and it is also used to access local memory. For simplicity, the example in Figure 2 shows a ring topology to interconnect the 4 nodes together but for a small-scale system, all of the nodes can be fully connected as well. Given the processor-interconnect and its topology, the routing algorithm determines the path taken by a packet to its destination [19]. To provide flexibility, a routing table is commonly used in the processor-interconnect to implement the routing algorithm. Based on the destination of the packet, a routing table look-up is done to determine the appropriate router output port that the packet needs to be routed through. However, the routing table is only used to determine which router output port should be used (and the routing path); the routing table is not responsible for determining the packet destination.

The packet destination information is determined by the packet header. The format of messages (or packets[2]) in the processor-interconnect is similar to other interconnection networks [19] as shown in Figure 3. A packet is the high-level message that is sent between the nodes, which can include memory requests, cache line replies, coherence message, etc. A packet is often partitioned into one or more *flits* or flow control units within the interconnection networks – thus, a packet can be partitioned into a head flit, one or more body flits, and a tail flit. The head flit contains additional "header" information, which can include the packet type and both the source and destination node information of the packet. The body/tail flits do not contain destination information but only the payload and simply follows the head flit from the source to its destination.

[2]A message can consist of multiple packets but within the processor-interconnect, messages are often single packet.

2.3 Related Work

To the best of our knowledge, very few prior research have investigated security vulnerabilities within the hardware of the processor-interconnect in multi-socket servers. Song et al. [49] demonstrated the security vulnerability of the routing table in a multi-socket server This vulnerability enabled performance attacks by sending packets through longer routes and degrading both interconnect latency and bandwidth. In addition, it also enabled system attacks by creating a livelock in the network to crash the system. However, the routing table did not modify the *destination* of a packet and thus, the scope of the attack was limited. In this work, we show how the *destination* of a packet can be changed by modifying the DRAM mapping table to enable a rootkit attack.

Rootkit Attacks: User-level, software rootkits (upper left box in Table 1) often modify existing system utilities to enable malicious codes. Lrk5 [48], T0rn [35] and Dica [23] replace the system binaries (e.g., ls, ps and netstat) with modified versions to hide files, processes or network connections. SAdoor [40] is a non-listening daemon that grabs packets directly from the NIC and watches for special key and command packets before executing a pre-defined command (e.g., /bin/sh). However, it has been shown that these rootkit are often easily detected by integrity checking for the system binaries.

Traditional kernel-level, software rootkits (upper right box in Table 1) exploit the control hijacking and interception, modifying static kernel data structures (e.g., system call table) to jump to malicious codes indirectly. DKOM [15] introduced a more advanced kernel-level rootkit approach which exploits dynamic (non-control) kernel data structures (e.g., processor descriptors) to install the rootkit. Hofmann et al. [24] introduced a rootkit which allows for malicious control flows by replacing pointer variables. Hund et al. [25] introduced a return-oriented rootkit based on Return-Oriented Programming [14] to bypass integrity checking for the kernel code. However, these kernel-level, software rootkits require modifications to the kernel and can be detected with protecting return addresses on the stack and critical data structures from the modification.

In addition to software rootkits, hardware-supported rootkits have been proposed (lower right box in Table 1). ShadowWalker [50] hid the trace of the rootkit by hooking the page tables while Cloaker [20] exploited ARM-specific architectural feature to conceal the rootkit with-

Figure 3: Packet format in interconnection networks with packet consisting of multiple flits.

out altering existing kernel code. However, these work require adding malicious payload to the kernel which can be prevented by guaranteeing the execution of only verified kernel code or detected by checking the flow of the hijacked code. In comparison, while PIkit also leverages a hardware-vulnerability, PIkit enable malicious activity with only user-level code.

The rootkits that we categorized in Table 1 and described earlier focus mostly on the software or hardware (CPU)-related rootkits. There have also been other device-specific rootkits such as network interface card [51], hard-drives [57], USB mouse [33], and printers [18]. In addition, rootkits involving BIOS [56] have also been proposed. However, these rootkits also involve modifying existing firmware to carry out the attack. Subvirt [28] presented a more stealthy rootkit by using virtual machine monitor (VMM) but Subvirt can also be detected with physical memory signature scans. Run-DMA [44] is a DMA (direct memory access) rootkit attack that enables a "malicious computation" where attacker modifies data inputs to induce arbitrary computation, in comparison to a more traditional "malicious code" model attack. The PIkit that we present in this work is not necessarily limited to either malicious computation or malicious code model since any region in the memory can be modified as long as the memory mapping can be determined by the attacker.

Rootkit Detection: The rootkit detection can be largely divided into two types, checking the integrity of kernel codes and data structures and detecting malicious control flows (e.g., hooking system call table and interrupt vector table). Copilot [42] detects the modification of kernel and jump tables with the separated PCI card monitor. However, such approaches do not guarantee hardware register integrity, such as the register modified in Cloaker [20] or the DRAM address mapping table modified in this work. KI-mon [31] introduced a hardware-assisted monitor, which snoops all bus traffic to verify updates of the kernel objects with the address filter while Shark [53] proposed an architectural supported rootkit monitor. However, since PIkit does not access the kernel objects directly, such approaches cannot detect PIkit.

3 Analysis of Processor-Interconnect in Multi-Socket Servers

In this section, we describe the *DRAM address mapping table* structure within the Northbridge (or the uncore) that we analyze in detail and then describe how it can be exploited to enable a hardware vulnerability-based rootkit attack through the processor-interconnect.

	Base Address	Limit Address	Destination ID
0	0x0000000000	0x041F000000	0
1	0x0420000000	0x081F000000	1
2	0x0820000000	0x0C1F000000	2
3	0x0C20000000	0x101F000000	3
4	RESERVED	RESERVED	RESERVED
5	RESERVED	RESERVED	RESERVED
6	RESERVED	RESERVED	RESERVED
7	RESERVED	RESERVED	RESERVED

Figure 4: An example of DRAM address mapping table for a 4-node system.

3.1 DRAM Address Mapping Table

One of the critical information in packet's header is the destination information; this information is often based on the destination memory address in modern multi-socket servers. The destination node is determined by a memory address mapping table structure between the core and the processor-interconnect router – we refer to this as the *DRAM Address Mapping Table*. [3] Based on the destination address of the packet, the DRAM address mapping table determines the destination. As a result, regardless of the address, the packet is simply forwarded to the destination based on the packet header information within the processor-interconnect.

A separate copy of DRAM address mapping table structure exists within each node of a multi-socket system, between the core (or the last level cache) and the router. Each entry in the mapping table contains a DRAM physical memory address range, often including the start (or the base) address and the limit address. Each entry also contains the destination node information – thus, if an address falls within the address range, the destination node information is appended to the packet. An example of a DRAM address mapping table is shown in Figure 4.

The number of entries in the DRAM address mapping table should be equal to or greater than the maximum number of nodes in the system. In the AMD system that we evaluate (AMD Opteron 6128), the system contains 4 nodes but the system is scalable up to 8 nodes. Thus, the DRAM mapping table contains 8 entries with only 4 of the entries used and the remaining 4 entries not used (or shown as RESERVED in Figure 4).

The DRAM address mapping table is initialized by the BIOS at boot time. Since the table entries are memory-mapped registers, the BIOS uses memory operations to initialize the memory mapping table. The contents of

[3] In Intel-based NUMA systems, this structure is referred to as the DRAM address decoder [5] while in AMD- multi-socket systems, similar structures are referred to as DRAM address map register [13].

the address mapping table entries are dependent on the DRAM capacity installed on each node. To determine the address range for each entry (and each node), the BIOS calculates the current memory capacity by obtaining DRAM information such as the number of rows, banks and ranks from the SPD (Serial Presence Detect) [4] on the DRAM.

3.2 Vulnerable Hardware Features

To implement PIkit, we exploit the following three aspects of the DRAM address mapping table in multi-socket servers.

Configurability: The memory mapping table needs to be configurable since the memory capacity per system (and per node) is flexible and determined by the system user.

Extra entries: Since the system needs to be designed for scalability, the number of entries in the DRAM address mapping table needs to equal to or greater than the largest system configuration. For most multi-socket servers today, the maximum number of nodes in the system is often 8; however, the most dominant NUMA servers on the market are often 2 or 4 nodes [26] and thus, there are memory mapping table entries that are unused.

Discrepancy: The DRAM mapping table content values can be modified after the initialization such that the values are not consistent with the original values. This discrepancy may or may not cause a problem, depending on how the table is modified. [4]

Thus, given these three hardware vulnerabilities, ***the destination node information of packets in the processor-interconnect can be modified such that the packets are sent to a different node (and its corresponding victim's memory address) to allow an attacker to access unauthorized memory space without proper permission.*** In the following section, we describe the challenges in the design and implementation of PIkit.

4 PIkit Design & Implementation

PIkit installation procedure that includes modifying the DRAM address mapping table is described in this section. We first provide an overview and describe how the attack address region needs to be prepared by the attacker, and then modify the DRAM mapping table based on the attack address region. Our initial design and implementation are shown for an AMD-based server but we also discuss how PIkit can be implemented on Intel-based servers as well.

[4] Steps to ensure that the discrepancy does not cause a problem is discussed in Section 4.3 (for AMD) and Section 4.5 (for Intel).

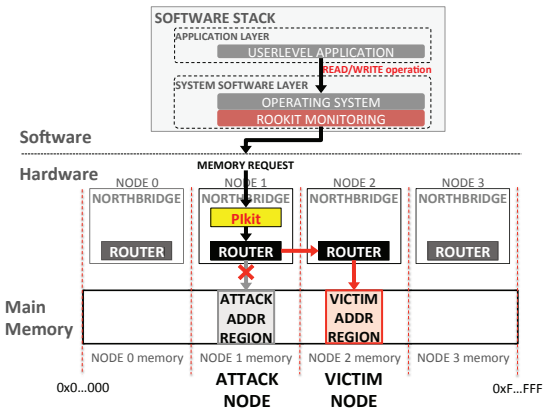

Figure 5: High-level description of the proposed PIkit on a 4-node multi-socket server.

4.1 Overview

In this work, we define the *attack* node as the node in the multi-socket system where the DRAM address mapping table is modified to implement PIkit, and the *victim* node is the node where its memory is maliciously accessed, through user-level read or write operations without proper permission. The address region that is modified in the memory mapping table in the attack node is defined as the *attack address region* while the corresponding address in the victim node is the *victim address region*, as shown in Figure 5. The PIkit and the modification in the DRAM address mapping table result in read/write memory requests to the attack address region being routed to the victim address region. The high-level diagram in Figure 5 also shows how the PIkit relates to the entire system. Most rootkit monitoring mechanisms (or solutions) exist at the software-level but the PIkit that we propose in this work is at the hardware-level (within the processor-interconnect) and exploiting vulnerability in the mapping table structure.

After the core injects a memory request into the Northbridge and before the packet is actually routed through the processor-interconnect by the router, a packet header is created based on the physical address of the memory request as shown in Figure 6. The processor-interconnect does not observe the memory address as that is included in the packet payload [5] and is only observed at the destination (i.e., memory controller). The processor-interconnect only observes the destination node information that is appended at the interface between the core and the router. The PIkit that we propose in this work exploits this vulnerability of modifying the *destination* of a packet – in particular, the DRAM address mapping table structure to modify the packet's destination.

[5] Packet payload refers to the non-header or the data portion of the packet while the payload terminology used in the rest of this paper refers to the malicious code used after rootkit is installed.

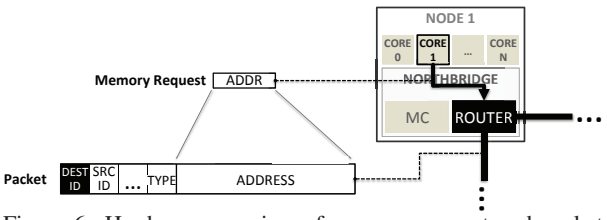

Figure 6: Hardware overview of memory request and packet header.

4.2 Defining Attack Address Region

Before the PIkit installation, the attacker first needs to prepare the attack address region such that only the attacker has access to that particular memory region. This step is critical to ensure that PIkit and the DRAM address mapping table modification do not cause any unknown system behavior including crashing the system. For example, if another program (or user) attempts to access the attack memory region after the PIkit is installed, the memory access will be routed to the victim node and unexpected system behavior will occur if a memory write is being done on an unintended memory address – i.e., it can cause critical data in kernel space to be overwritten.

The memory address range of the attack address region needs to be equal to granularity or the *resolution* of the memory mapping table. Although each entry in the DRAM address mapping table specifies both the base and the limit addresses, the full width of the address (i.e., 48 bits) is not stored as some of the lower bits are not specified in the DRAM address mapping table. For example, in the AMD Opteron 6128 system that we evaluate, the granularity of the memory mapping table is 16 MBs as only 24 most-significant bits are stored – thus, the attacker needs to obtain at least 16 MBs of physically contiguous memory region. To achieve this, we take advantage of huge pages that are commonly available. In the system that we evaluate, we used 1 GB huge page. After successful `malloc` for a huge page, an address range within the contiguous memory region allocated can be used as the attack address region, as long as the process that received the memory allocation is continuously running.

4.3 Modifying the DRAM Address Mapping Table

The DRAM address mapping table consists of physical addresses while the attack address region obtained in the previous section are virtual addresses. Thus, an attacker needs to obtain the translated physical address of the attack region before modifying the DRAM address mapping table. The translation of the virtual to physical address can be determined by using */proc/(pid)/pagemap* interface from the Linux kernel. Based on this translation information, the DRAM address mapping table can be modified using the corresponding physical address of the attack region.

	Base Address	Limit Address	Destination ID
0	0x0000000000	0x041F000000	0
1	RESERVED	RESERVED	RESERVED
2	0x0820000000	0x0C1F000000	2
3	0x0C20000000	0x101F000000	3
4	0x0420000000	0x07BF000000	1
5	**0x07C0000000**	**0x07C1000000**	**2**
6	0x07C2000000	0x081F000000	1
7	RESERVED	RESERVED	RESERVED

Figure 7: A modified DRAM address mapping table where entry 5 (highlighted) is used as the attack address region.

An example of a modified mapping table is shown in Figure 7, based on the original DRAM address mapping table shown earlier in Figure 4. We assume the attack node is node 1 and the victim node is node 2, with the attack address region defined as the address between `0x07C0000000` and `0x07C1000000` in node 1. In Figure 7, the entry 1 of the table which originally identified node 1 memory region has been removed. Instead, the same address range has been partitioned across entries 4, 5, and 6 of the modified DRAM mapping table (Figure 7). The key difference compared with the original mapping table is that for entry 5, the destination node ID has been modified such that it is no longer node 1 but modified to node 2 – thus, entry 5 represents the *attack address region*. Any address requests between `0x07C0000000` and `0x07C1000000` from node 1 have a destination of node 2 added to the packet header, instead of the original destination of node 1. When this particular packet arrives at node 2, the DRAM memory controller within the node 2 will receive this packet and convert the address within the payload of the packet into the actual *victim address region*. For example, in the AMD system that we evaluate, address `0x07C0000000` from node 1 ends up being mapped to address `0x0840000000` in node 2. As a result of the mapping table modification, the physical memory connected to node 1 that originally corresponded to the address range between `0x07C0000000` and `0x07C1000000` can no longer be physically accessed from node 1.

Since the table entries are memory-mapped registers, the entries can be modified through system read/write commands (e.g., `setpci` utility). However, to properly modify the DRAM address mapping table entries, the following caution must be taken.

1. The new entries must be written before the old entry is removed (e.g., in Figure 7, entries 4, 5, 6 must be written before entry 1 is cleared).
2. For the new entries added, the base address register must be written before the limit address register.
3. For the existing old entry that needs to be removed, the limit address register must be cleared first, before the base address register.

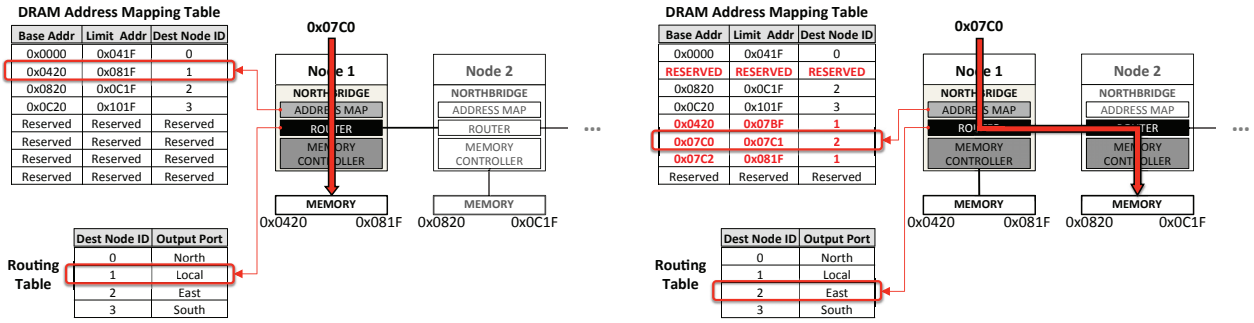

Figure 8: Example of PIkit (a) before and (b) after PIkit is installed on an AMD Opteron 6128 server.

Since memory accesses continuously occur in the system, randomly changing the mapping table in any order can result in a memory request that is not able to match an entry in the mapping table and result in a system crash.

4.4 Example

A complete example of PIkit is shown in Figure 8 for a 4-node system with only 2 nodes shown in the figure for simplicity. The same DRAM address mapping tables shown earlier in Figure 4 and Figure 7 are used in the example.[6] Assume a read access to address 0x07C0000000 is made by a core in node 1. With an unmodified DRAM address mapping table, the address 0x07C0000000 finds a match in entry 1 and determines that the destination should be node 1. Based on this information, a routing table look-up is done for the node 1 to determine the output port. Since the current node is the local node, the output port determined by the routing table is the "Local" port and the memory request is routed appropriately to the local node's memory.

However, with the modified DRAM address mapping table (Figure 8(b)), the same request to address 0x07C0000000 finds a match to entry 5 where the destination is now node 2. Based on this new destination node ID information, the routing table look-up is done within the router and the output port returned is the "East" port – thus, the packet is routed to node 2. Within the node 2, the packet is simply treated as a packet that was destined for node 2 (or the "Local" output port) and will be routed to the memory controller. Since the processor-interconnect only looks at the destination node information to determine where to send the packet, the PIkit shown in Figure 8(b) results in packets accessing a memory region where it does not have proper permission. If this packet was a read request, the packet would read data from the corresponding victim memory address while if this packet was a write request, the packet would modify or overwrite existing data in the victim memory address.

[6]For simplicity, the lower 3 bytes of the address are not shown in Figure 8.

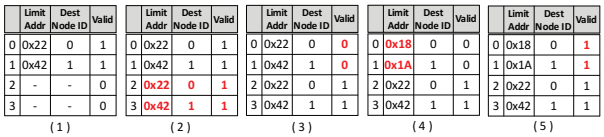

Figure 9: Example of how the Source Address Decoder (SAD) can be modified on the attack node to implement PIkit on an Intel Sandybridge architecture.

4.5 Extending PIkit to Intel Architecture

In the previous sections, we described how PIkit is implemented on an AMD multi-socket server and in this section, we discuss how PIkit can be extended to an Intel-based server. A structure similar to the memory mapping table exists within Intel x86 server architecture and is referred to as the *Source Address Decoder (SAD)* [5]. A key difference with the AMD architecture in the memory mapping table is that instead of specifying both the base and the limit memory address for each entry, only the limit address is specified. In addition, a valid bit per entry exists in the SAD which specifies if the entry is enabled or not. As a result, the "base" address is implied from the previous entry limit address and PIkit design needs to properly add/modify entries of SAD to ensure proper behavior for memory accesses. An example of how the SAD table can be modified is shown in Figure 9. The initial entries are first duplicated in the table (step (2)) and then, the initial entries are invalidated (step (3)) before the addresses are modified (step (4)) and then, the modified addresses are made valid (step (5)) to create an attack address region with entry 1 of the SAD table.

Another key difference in the Intel architecture compared with the AMD system is the *Target Address Decoder* (TAD) which is accessed before the address is sent to the memory controller at the destination node. [7] TAD is an additional table that is responsible for mapping discontinuous address regions [5] and includes both a limit address and an offset. While the purpose of the "offset" within the TAD is to relocate the memory location as necessary, it enables PIkit to be implemented by *defining*

[7]While a similar structure existed in the AMD system that we evaluated, it only had a single entry and could not be exploited for PIkit.

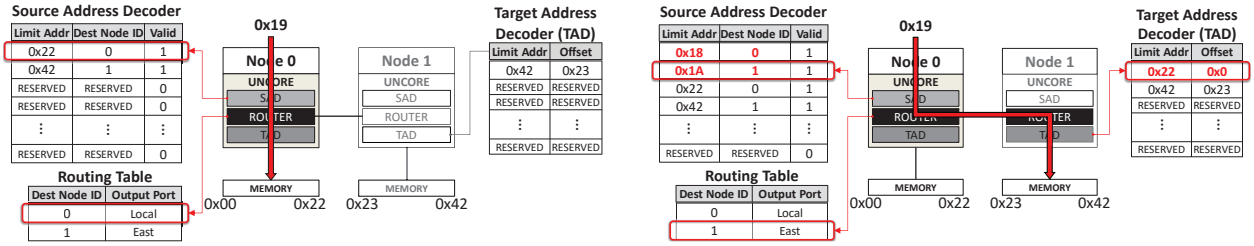

Figure 10: PIkit example on an Intel Sandybridge-based server (a) before and (b) after PIkit is implemented. For simplicity, the TAD on Node 0 and SAD 0 on Node 1 are not shown.

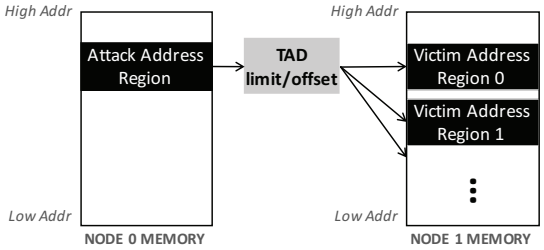

Figure 11: Impact of the Target Address Decoder (TAD) on the mapping of the attack memory region to the victim memory region.

the victim memory address region based on the attack address region. The impact of TAD offset is shown in Figure 11. Since the offset is subtracted from the address, by varying offset entry, the attack memory address region can be mapped to different victim memory region based on the offset value and enables a more "controlled" attack by providing control over the memory mapping (i.e., victim's address range). One constraint is that since the offset is being subtracted, the victim memory address region can only be equal or smaller than the attack memory address region.

A PIkit example for an Intel-based architecture is shown in Figure 10 based on the SAD modification shown in Figure 9. The PIkit consists of both the SAD modification in the attack node and the TAD modification in the victim node. Thus, the same vulnerability that was exploited for PIkit on the AMD-based system is available in the Intel-based servers – the table structures are memory-mapped registers that are configurable and extra number of entries are available. Based on documentations [5], the number of entries for SAD is 20 and thus, it is more than sufficient for a 2-node multi-socket system.

5 Malicious User-level Payloads

After PIkit is installed, PIkit enables access to the victim address region regardless of the privilege level. In this section, we describe different malicious activities with user-level payloads that can exploit PIkit. While many different attacks (and payloads) have been proposed as part of rootkit attacks, previous attacks often require leveraging (or modifying) some OS capability or creating additional payload to mimic the OS. In comparison, the ***malicious payload for PIkit is fundamentally different as the payload is relatively simple with the source code mostly consisting of memory read and write commands.*** [8] The main challenge with PIkit payload is determining the attack (or the corresponding victim) address region to carry out the malicious activity.

Description	Value
System	AMD Opteron 6128
# of Sockets (Nodes)	2 (2 per socket)
# of Cores	4 per node
Interconnect	6.4 GT/s HT 3.0
# of HT Links	4 per node
OS version	Linux Kernel 3.6.0

Table 2: Dell PE R815 server used in our evaluation.

5.1 Bash Shell Credential Object Attack

In the operating system, a process is represented by a process control block (PCB) data structure in the privileged memory space. The process control block has critical information such as memory information, open-file lists, process contexts and priorities, etc. In particular, we exploit the credential kernel data structure which is contained within the PCB and is responsible for access controls of a process in the Linux kernel. If the attacker locates the credential data structure in the victim address region, the attacker can modify any value within the credential data structure with PIkit. In this work, we modify the UID or the EUID of a bash shell process to achieve root privilege escalation. An overview of the malicious activity is shown in Figure 12. We demonstrate this attack on a 4-node AMD server described in Table 2.

5.1.1 Scanning the Fingerprint

In this attack, we assume the attacker uses a common user-level application, *Bash Shell*, to obtain root privilege. After the PIkit is installed on the attack node, the attacker starts the bash shell on the *victim* node and attempt to modify the credential data structure for privi-

[8] Pseudo-code for the malicious payload is shown in Appendix for the three different malicious activities described in this section.

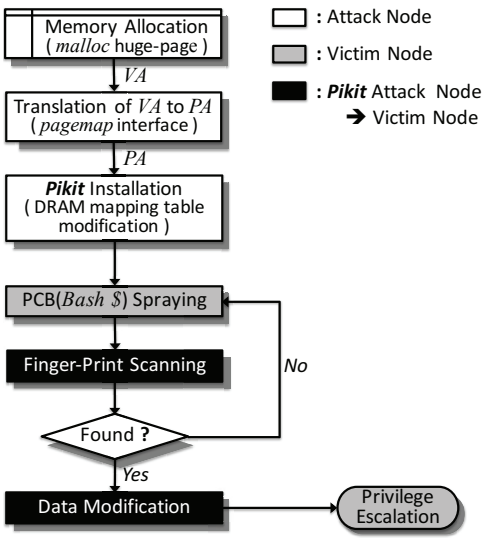

Figure 12: High-level overview of the attack with PIkit for privilege escalation.

lege escalation. One challenge before obtaining privilege escalation is determining the actual address of the PCB (or in particular, the credential data structure) of the bash shell on the victim node. In order to determine the memory location, we use the fingerprint of the credential data structure to identify the proper starting address. As shown in Figure 13, the credential data structure consists of multiple integer variables and pointers which contain 64 bits addresses. Since the bash shell user-level process is owned by the attacker, the attacker knows the user/-group ID of the process and can use the consecutive user and group IDs (e.g., UID, GUID, ..., FSGID), as shown in Figure 13 ① as these variables often have the same values.

To increase the accuracy of the fingerprint, additional pointer information within the credential data structures can also be used. Since most of the x86-64 Linux systems use the specific virtual address ranges for the kernel objects based on the kernel virtual address map [29] (e.g., 0xffff880000000000 – 0xffffc7ffffffffff for the direct mapping), the pointers shown in Figure 13 ② should have virtual addresses that are within this range. This approach is similar to what was used in prior work [46] that used virtual address characteristic to find the process control block used by Window operating system in dumped memory. In addition, some addresses of different variables used in kernel space are publicly available even in the user-level space, including *the Symbol Lookup Table ('/boot/System.map')* [32]. In particular, the virtual address of *'user_ns'*, shown in Figure 13 ③, can be found in the *Symbol Lookup Table* which is determined at kernel compile time and can be used as part of the fingerprint.

Based on the three types of information described above, a fingerprint for the credential kernel data structure can be used to determine the location of the credential data structure. The attacker from the attack node can issue read operations for the attack address region – determined by the allocated memory region and the modified DRAM address mapping table on the attack node as described in Section 4 – and begin the fingerprint scanning. The read requests will be routed to the victim node and the data in memory will be returned to the attack node where it will be compared against the fingerprint. If there is a match, the starting address of the credential data structure is found and the attacker can modify the data. If no match is found, the credential data structure of the bash shell is not found on the victim address region – thus, the attacker needs to stop the current bash shell process and restart the bash shell on the victim node, and repeat the fingerprint scanning.

```
struct cred {
  ...
  uid_t       uid;          /* real UID of the task */
  gid_t       gid;          /* real GID of the task */
  uid_t       suid;         /* saved UID of the task */
  gid_t       sgid;         /* saved GID of the task */
  uid_t       euid;         /* effective UID of the task */
  gid_t       egid;         /* effective GID of the task */
  uid_t       fsuid;        /* UID for VFS ops */
  gid_t       fsgid;        /* GID for VFS ops */
  ...
  struct key   *thread_keyring; /* keyring private to this thread */
  struct key   *request_key_auth; /* assumed request_key authority */
  struct thread_group_cred *tgcred; /* thread-group shared credentials */
  ...
  struct user_namespace *user_ns; /* cached user->user_ns */
  ...
};
```

Figure 13: The credential kernel data structure in the Linux kernel 3.6.0 and the fingerprint that we exploit in this work, with the fingerprint highlighted with a rectangle.

Note that the attacker does not need to know the starting *virtual* address of the kernel data structure on the victim node. In fact, the appropriate *physical* address within the victim address region does not need to be known as well. Only the *corresponding* address on the attack node needs to be determined from the fingerprint scanning and the attack (or the modification of the data) is done based on the physical address of the attack node and the corresponding virtual address within the attack node is used in the actual data modification.

5.1.2 Modifying the Data

Once the corresponding address of the credential data structure is determined from the scanning, the offset within the data structure can be easily determined based on the credential data structure definition (e.g., Figure 13) and the different variables within the data structures can be easily modified with PIkit. Thus, the root privilege can be obtained by modifying either the *euid* (Effective User ID) or the *uid* (User ID) field within the credential data structure to a value of 0 (instead of the original value) as the *uid* of 0 specifies the root user. In order to overwrite this variable, a memory write instruction in assembly language can be used by the attacker to obtain the root privilege – e.g.,

`movnti $0, (Virtual Address)`

where the virtual address is the address determined from

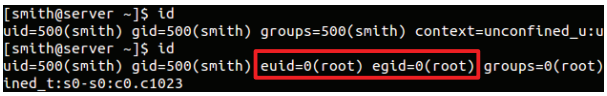

Figure 14: Screen capture from `id` command of privilege escalation, before the attack and after the attack.

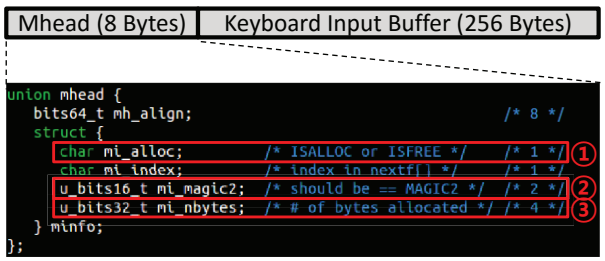

Figure 15: Keyboard buffer data structure in the bash 4.3 and the fingerprint that we exploit in this work, with the fingerprint highlighted with a rectangle.

scanning and PIkit routes this write instruction to the victim node. However, we used a non-temporal SSE instruction in our evaluation in order to bypass the cache within the processor. If the write occurs to the cache within the attack node, the effect of the root privilege escalation can be delayed until write-back to the memory occurs.

The result from the attack is shown in Figure 14, consisting of the ID information before and after that attack using the *id* command from of the bash shell on the victim node. The EUID of *Bash Shell* in the victim node is modified to root user and thus, root privilege escalation is achieved.

5.1.3 Spraying the Process Control Block

As described earlier, the attack address region is mapped to some victim address region on the victim node through PIkit. As a result, all memory accesses to the attack address region on the attack node are constrained to some victim address region based on the DRAM physical mapping which is not known. As a result, if a user-level application (i.e., bash shell) executing on the victim node is not placed in the victim address region, an attacker can not access the kernel objects of the process. To increase the probability that the credential data structure can be found, the PCB can be sprayed across the victim node by executing multiple bash shells on the victim node. This increases the probability that one of the processes (and the corresponding PCB) is placed within the victim address region and reduces the amount of time it takes to achieve privilege escalation.

5.2 Bash Keyboard Buffer Attack

In this section, we describe how PIkit can be exploited to carry out an information leakage attack on another user's bash shell and perform a *bash keyboard buffer monitoring* attack. Since no data modification is required, this attack can be classified as a read-only attack. When a

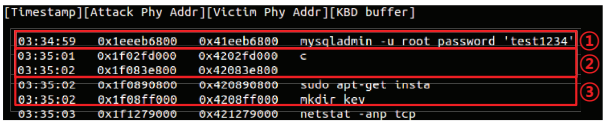

Figure 16: Snapshot of bash keyboard buffer monitoring.

Description	Value
System	Intel Xeon E5-2650
# of Sockets (Nodes)	2 (1 per socket)
# of Cores	8 per node
Memory Capacity	8 GB per node
Interconnect	6.4 GT/s QPI
OS version	Linux Kernel 3.6.0

Table 3: Dell PE R620 server using in our evaluation.

user types any word on their own shell prompt, all characters are stored in a bash keyboard buffer in the memory unencrypted.

In the Bash shell (v4.3), the bash keyboard buffer is represented by a data structure referred to as `mhead`, as shown in Figure 15. Similar to the bash shell credential object attack described earlier, a fingerprint is necessary to detect this data structure in memory. For the fingerprint of the bash keyboard buffer, we use three unique values as the fingerprint based on the `mhead` data structure. The character variable (Figure 15①) has a unique 8 bits value (e.g., either 0x7F when allocated or 0x54 when freed). In addition, the 16-bit variable (Figure 15②) is always a predefined magic number (0x5555) and the 32-bit variable (Figure 15③) is always 0x100 that refers to the size of the buffer.

Based on the fingerprint, after PIkit is installed, an attacker can search for the fingerprint to gather information from victim users' shell prompt, including potentially password information since data in the keyboard buffer is unencrypted. To evaluate this attack, we use the system described in Table 3 and assume a SSH server where multiple users use bash shell prompts from remote connections. Multiple remote SSH connections are made on the victim node and for each shell, different prompt inputs are used to evaluate the bash keyboard buffer attack.

By scanning for the fingerprint on the victim address region, we were able to monitor the bash keyboard buffer of other users. Different examples are shown in Figure 16 – Figure 16① shows a user typing in their password while Figure 16③ show other commands being typed by another user. In comparison, Figure 16② shows a keyboard buffer for another user that does not contain any content. Thus, with the buffer monitoring with PIkit installed, the dynamic information from other users' keyboard input can be leaked.

5.3 Shared Library Attack

The attack described earlier in this section required both heap spraying and fingerprint scanning to obtain privi-

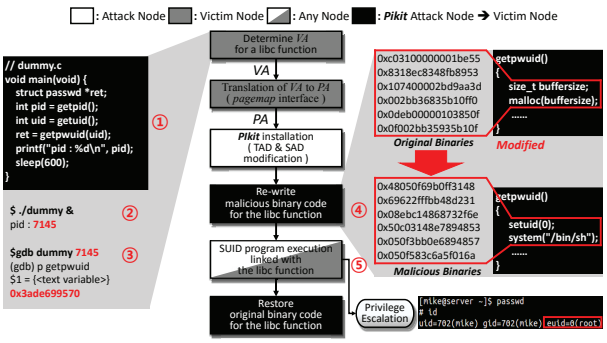

Figure 17: Shared library (`libc`) Attack with PIkit.

lege escalation. While this approach was successful, the attack can be time consuming because of the amount of time to scan the memory and attack reliability can become an issue since if the heap spraying does not fall within the victim memory region as the attack process needs to be repeated. To increase the attack reliability with PIkit, we take advantage of the target address decoder (TAD) structure available in the Intel Sandybridge architecture that provides the ability to "fix" the victim memory address region based on the attack memory address region. This approach is useful if the victim address region contains information that can be maliciously modified at a fixed physical memory location and does not change over time.

In this section, we take advantage of shared libraries that are often loaded into user memory space at single physical location and shared by different users. Once the shared library is initially loaded, the physical address of the library will likely not change. A commonly used shared library is `libc` and in this section, we exploit PIkit to overwrite existing functions in the `libc` with a malicious code to obtain privilege escalation. No fingerprint scanning or heap spraying is required and results in a highly reliable attack.

An overview of the `libc` attack is shown in Figure 17. To first obtain the virtual address of the `libc` shared library, an attacker can write a simple attack program (Figure 17①) which links `libc` dynamically. By executing a function within `libc` (e.g., `getpwuid()`) in the attack program, the `libc` library is dynamically linked. After obtaining the process ID for the attack program that loaded the shared library (Figure 17②), we use a debugger (e.g., `gdb`) on the running process to determine the virtual address of the `getpwuid()` function in the shared library (Figure 17③). The physical address corresponding to the virtual address of the `libc` library can be determined through the */proc/pid/pagemap* interface and PIkit can be installed as described earlier in Section 4.5. Since the target victim address (i.e., `libc` function code location) is known, we modify the TAD structure offset accordingly, based on the attack address region obtained with PIkit.

After PIkit is installed, the attacker can re-write the runtime code loaded in the memory with user-level memory operation (Figure 17④). In our example attack, the first 48 bytes of *getpwuid()* function is re-written with malicious binary code that executes shell with root privilege. Even if the shared library is located in non-writable memory regions, PIkit bypasses any OS permission check ($W \oplus X$[9] based implementations [3, 52, 34]) and write the malicious code to the physical DRAM directly. The modified code executes `setuid(0)` to obtain root access but this does not work with user-level privilege and requires root access. However, the `passwd` Linux program has SUID (Set owner User ID) permission and is linked with *getpwuid()* function at runtime, the execution enables the malicious binary code to escalate the user privilege to the root (Figure 17⑤). Once privilege escalation is obtained, the attacker can restore the original binary of *getpwuid()* function to prevent the execution of the malicious code for other users that executes `getpwuid()`. While `getpwuid()` function for the `libc` was used to demonstrate this attack, other seldomly used library functions can be used or the malicious code can be modified to enable root shell only for specific user ID.

6 Discussion

In this section, we provide discussion on how PIkit can be potentially exploited for other types of attack, possible solutions both in software and hardware, as well as some limitations of PIkit.

6.1 VM Escape Attack

VM (virtual machine) escape is defined as enabling a VM to interact with the hypervisor directly and/or access other VMs running on the same host [30, 55]. If PIkit is installed on a multi-socket server that supports VMs, we expect that VM escape attack can be carried out. However, there are two challenges to implement VM escape with PIkit – huge pages and virtual address translation. PIkit exploited the availability of huge pages in modern OS to define the attack address region. Modern virtualization hardware technologies such as Intel VT-d support 1 GB huge pages [43] and hypervisors such as KVM, Xen and VMware also support 1GB huge page for guest VMs. We evaluated the VM escape attack possibility with Xen 4.4 but the hypervisor underneath the guest OS implemented the huge pages as a collection of smaller pages (e.g., 2 MB pages), likely because of implementation complexity. However, there is no fundamental reason why the hypervisor cannot support 1 GB

[9] A memory page must never be writable and executable at the same time.

Algorithm 1: PIkit monitor to detect modification to the DRAM address mapping table.

```
Input : monitoring
begin
    while monitoring do
        – Get DRAM information from SPD
        – Calculate Valid_address_ranges of installed
          DRAM from information
        A ← Valid_address_ranges
        – Get Current_address_ranges from DRAM
          address mapping table
        B ← Current_address_ranges
        if A != B then
            PIkit detected
return
```

huge pages. [10] Another challenge is that address translation needs to be done twice to properly install PIkit – from the guest virtual address (gVA) to the guest physical address (gPA) within the VM and then, another translation to the machine physical address. While */proc/pid/pagemap* interface can be used for the translation from a gVA to a gPA for the VM local OS system, similar to the PIkit implementation described earlier, the hypervisor is responsible for another translation from gPA to machine PA. The hypervisor likely maintains a separate table/data structure for this translation and this needs to be reverse engineered to implement VM escape.

6.2 Possible Solutions

Possible solutions to PIkit can be classified as either a software-based or a hardware-based solution. The actual solution to PIkit is highly dependent upon the manufacturer of the hardware (e.g., Intel, AMD) as well as the system software used.

6.2.1 Exploit Existing Features

While evaluating PIkit on different systems, some of the recent AMD systems were not vulnerable to PIkit. To the best of our knowledge, the vulnerability was not removed for security reasons but removed for power saving implementation. C6 state is an ACPI defined CPU power saving state where the CPU is put in sleep mode and all CPU contexts are saved. In some AMD implementations, when the CPU enters the C6 state, the processor context is saved into a pre-defined region of the main memory. To avoid any possible corruption of the processor contexts that are saved, the AMD systems implement `LockDramCfg` option where some memory system related configurations cannot be modified, including the DRAM address mapping table [7]. However, the BIOS can disable the C6 state for some of these systems – which would enable PIkit to be installed. A simple solution for PIkit on such AMD systems is to always enable `LockDramCfg` to prevent PIkit from being installed. For the Intel systems, C6 power state is supported but the processor contexts are saved to the last level cache (and not the memory) to provide faster context switch – thus, to the best of our knowledge, similar `LockDramCfg` feature is not readily available in Intel x86 CPUs.

[10] A very recent version of Xen (v 4.6) actually has support for 1 GB huge pages.

Description	Ratio (%)
SPD access period (I/O bound)	99.975 %
DRAM size calc period (CPU bound)	0.003 %
DRAM table access period (I/O bound)	0.019 %
Table Comparison period (CPU bound)	0.003 %

Table 4: Breakdown of CPU cycles ratio for the PIkit monitor.

6.2.2 Software-based Solutions

Prior rootkit monitors can be extended to detect the presence of PIkit. The monitor continuously compares the value of the current DRAM address mapping table with the "correct" DRAM mapping table value – where the correct value is determined similar to how the DRAM mapping table is initialized by the firmware at boot time. Thus, if we assume the software monitor is protected with a secure platform [8, 9], the solution to detect PIkit can also be protected.

We implemented the PIkit monitor as a Linux kernel thread and we evaluate its performance overhead. High-level description of PIkit monitor is shown in Algorithm 1. We used the PARSEC 3.0 [12] and evaluated workloads with varying MPKI (misses per kilo-instructions), using the system described earlier in Table 2. To measure overall system performance, we run each workload with the number of threads equal to the number of physical cores in the system. Based on Linux kernel 3.6.0, we implemented the rootkit monitor as a loadable kernel module to avoid kernel code modification and re-compilation. The execution time with PIkit monitor is normalized to the baseline without the monitor and the performance overhead from the software is negligible as there is less than 2% impact on overall performance (Figure 18).

The analysis of the PIkit monitor overhead is shown in Table 4. The two CPU computation periods (e.g., DRAM size calculation and comparison) take only 0.006% of the total PIkit monitor execution time. In comparison, the other two I/O bound periods (e.g., SPD and PCI address access) occupy 99% of the monitor execution time since these accesses have long latency – resulting in the kernel thread waiting on the I/O and mostly experience uninterruptible Sleep state. Thus, the PIkit monitor and software solution has minimal impact on performance.

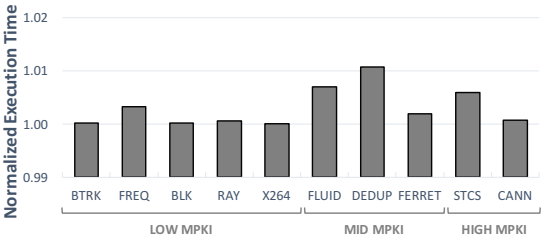

Figure 18: Performance overhead for PIkit monitor

6.2.3 Hardware-based Solutions

PIkit can be prevented with minimal hardware modifications. One hardware solution is to restrict the usage of the DRAM address mapping table entries used in multi-socket servers, based on the number of nodes in the system. If the hardware restricts the number of entries used to equal the number of nodes in the system, PIkit can be minimized. This approach does not completely remove the possibility of the PIkit since the attacker could use the entire local node's memory as attack address region. However, unless the attacker is the only user on the local node, this can cause non-deterministic behavior. If the number of entries is restricted, the attacker can also modify the DRAM address mapping where only small region is specified as the attack address region but the remaining address range of the local node would not be mapped in the table. As a result, if any memory access occurs to an unmapped address region, the system will likely crash.

Another possible hardware solution is to design the DRAM address mapping table entries as write-once memory-mapped registers such that the DRAM mapping table can not be modified after it is initially written. A block diagram of such write-once register is shown in Figure 19 – after a write is done, the write enable (WE) to the register will be disabled and no further writes can be done to the registers unless system reset is asserted. This approach avoids any possibility of the PIkit attack since the DRAM address mapping table cannot be modified after it is initialized; however, this removes any flexibility in the system if CPU hotplug [38] or memory hotplug [27] is supported. If the system supports hotplug where the DRAM (or CPU) can be added or removed while the system is running, the DRAM address mapping table needs to be modified after it is initialized to reflect the change in the memory capacity. This would require a minor change to the hardware (i.e., OR'ing the RESET signal with another signal that detects a hotplug event). However, this can also open up other attack opportunities if the attacker has physical access to the memory modules – for example, doing a hotplug creates an opportunity to modify the DRAM address mapping table and install the PIkit.

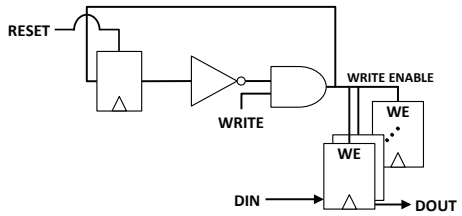

Figure 19: A PIkit hardware solution by creating a write-once register for the DRAM address mapping table.

6.3 Limitations

Memory mapping granularity: As described earlier in Section 4.2, there is a granularity (or the resolution) of the attack memory address region that can be specified with the DRAM address mapping table entries. In the AMD Opteron 6128 system, the smallest amount of memory address range that can be used as the attack address region is 16 MB. For the Intel Xeon E5-2650 (Sandybridge), the lower 26 bits are not specified and the granularity is 64 MBs. Thus, the granularity specifies the lower limit on the smallest attack address region that can be specified. This can be problematic if only normal page size is supported – i.e., with memory page size of 2 MBs, the attacker would require obtaining 8 *physical* pages to cover the 16 MBs attack region. If the entire attack region is not obtained by the attacker, other users (and programs) will access the victim address region and can cause unknown behaviors. However, huge pages that are available in modern OS can overcome this limitation.

DRAM coverage: One challenge of PIkit is the unknown DRAM physical mapping when an attack address is sent to the victim node. The details of how the DRAM physical address is mapped to the DRAM (i.e., row, column, channel, etc.) is vendor-specific. Although these details are often described in the specifications, the PIkit results in an unintended address arriving at the victim node memory system and it is not clear how the physical address bits are interpreted by the DRAM. For example, in the AMD Opteron 6128 system, the memory controller (or MCT) consists of two DRAM controllers (DCT0 and DCT1) where each DCT is responsible for half of the DRAM main memory connected to that particular node. When the attack node ID was *smaller* than the ID of the victim node (i.e., node 1 attacking node 2), only DCT0 address range could be accessed. However, by attacking node 2 from node 3, we discovered that DCT1 range can be accessed. Another limitation was that for some memory accesses, multiple attack addresses can map to the *same* victim address but this is a fundamental limitation of our proposed PIkit on the AMD system. In comparison, for the Intel Xeon E5-2650 (Sandybridge) server, the DRAM coverage issue was significantly minimized as there was an 1:1 memory mapping between the attack and the victim memory address region through the TAD structure. However, it is

not clear if this can be generalized to other Intel-based servers.

7 Conclusion

In this work, we described a new type of rootkit where the vulnerable hardware feature enables malicious activities to be carried out with only user-level code or payload. In particular, we presented PIkit – a processor-interconnect rootkit that enables an attacker to modify a packet's destination and access victim's memory region without proper permission. We described the design and challenges in implementing PIkit across both AMD and Intel x86 multi-socket servers. Once PIkit is installed, user-level codes used for malicious activities become very difficult to detect since memory accesses within the attack code appears as "normal" memory accesses to user-allocated memory. We demonstrated different malicious activities with PIkit, including bash shell credential object attack, keyboard buffer attack, and shared library attack.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by the Mid-career Researcher Program (NRF-2013R1A2A2A01069132), in part by IITP grant funded by MSIP (No.10041313, UX-oriented Mobile SW Platform), in part by the MSIP under the ITRC support program (IITP-2015-H8501-15-1005), and in part by Next-Generation Information Computing Development Program through NRF funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065647) and (No. NRF-2014M3C4A7030648).

References

[1] chkrootkit. http://www.chkrootkit.org/.

[2] Rootkit Hunter. https://rootkit.nl/projects/rootkit_hunter.html.

[3] OpenBSD 3.3 release notes. http://www.openbsd.org/33.html, May 2003.

[4] JEDEC Standard, SPD General Standard, 2008.

[5] Intel® Xeon® Processor 7500 Series.

[6] Linux Kernel Vulnerabilities Over Time. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2015. [Online; accessed 19-Aug-2015].

[7] ADVANCED MICRO DEVICES. BIOS and Kernel Developer Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2012.

[8] ALVES, T., AND FELTON, D. TrustZone: Integrated Hardware and Software Security. *ARM white paper 3*, 4 (2004), 18–24.

[9] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), vol. 13.

[10] ANDERSON, D., AND TRODDEN, J. *Hypertransport System Architecture*. Addison-Wesley Professional, 2003.

[11] AZAB, A. M., NING, P., AND ZHANG, X. SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 375–388.

[12] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] BIOS, A. kernel developers guide for amd family 10h processors, 2008.

[14] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 27–38.

[15] BUTLER, J. Direct Kernel Object Manipulation (DKOM). *Black Hat USA* (2004).

[16] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), ACM, p. 5.

[17] CREED. Information about the Knark Rootkit. http://ossec-docs.readthedocs.org/en/latest/rootcheck/rootcheck-knark.html, 1999.

[18] CUI, A., COSTELLO, M., AND STOLFO, S. J. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *NDSS* (2013).

[19] DALLY, W. J., AND TOWLES, B. P. *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[20] DAVID, F. M., CHAN, E. M., CARLYLE, J. C., AND CAMPBELL, R. H. Cloaker: Hardware Supported Rootkit Concealment. In *2008 IEEE Symposium on Security and Privacy (S&P'08)* (2008), IEEE, pp. 296–310.

[21] HEASMAN, J. Implementing and Detecting an ACPI BIOS Rootkit. *Black Hat Federal 368* (2006).

[22] HEATH, T., MARTIN, R. P., AND NGUYEN, T. D. Improving Cluster Availability Using Workstation Validation. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 217–227.

[23] HOCK, R. Dica rootkit. https://packetstormsecurity.com/files/26243/dica.tgz.html, 2002.

[24] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring Operating System Kernel Integrity with OSck. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 279–290.

[25] HUND, R., HOLZ, T., AND FREILING, F. C. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium* (2009), pp. 383–398.

[26] INTEL. Intel® Xeon® Processor E7-8800/4800/2800 v2 Product Family.

[27] ISHIMATSU, Y. Memory Hotplug. *LinuxCon Japan* (2013).

[28] KING, S. T., AND CHEN, P. M. SubVirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)* (2006), IEEE, pp. 14–pp.

[29] KLEEN, A. Virtual Memory Map with 4 level page tables. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2004.

[30] KORTCHINSKY, K. CLOUDBURST: A VMware Guest to Host Escape Story. *Black Hat USA* (2009).

[31] LEE, H., MOON, H., JANG, D., KIM, K., LEE, J., PAEK, Y., AND KANG, B. B. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security* (2013), pp. 511–526.

[32] LOVE, R. *Linux Kernel Development*. Pearson Education, 2010.

[33] MASKIEWICZ, J., ELLIS, B., MOURADIAN, J., AND SHACHAM, H. Mouse Trap: Exploiting Firmware Updates in USB Peripherals. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014).

[34] MICROSOFT. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2. http://support.microsoft.com/kb/875352, 2008.

[35] MILLER, T. Analysis of the T0rn rootkit. *SANS Institute* (2000).

[36] MORGAN, T. P. X86 Servers Dominate The Datacenter-For Now. http://www.nextplatform.com/2015/06/04/x86-servers-dominate-the-datacenter-for-now/, 2015.

[37] MUTNURY, B., PAGLIA, F., MOBLEY, J., SINGH, G. K., AND BELLOMIO, R. QuickPath Interconnect (QPI) Design and Analysis in High Speed Servers. In *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems* (2010), IEEE, pp. 265–268.

[38] MWAIKAMBO, Z., RAJ, A., RUSSELL, R., SCHOPP, J., AND VADDAGIRI, S. Linux Kernel Hotplug CPU Support. In *Linux Symposium* (2004), vol. 2.

[39] NIU, S., MO, J., ZHANG, Z., AND LV, Z. Overview of Linux Vulnerabilities. In *2nd International Conference on Soft Computing in Information Communication Technology* (2014), Atlantis Press.

[40] NYBERG, C. M. SAdoor - A non listening remote shell and execution server. http://krutibrko.sk/school/dp/samples/SAdoor/sadoor.pdf, 2002.

[41] OP, F. The FU rootkit. https://www.soldierx.com/tools/FU-Rootkit, 2008.

[42] PETRONI JR, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium* (2004), San Diego, USA, pp. 179–194.

[43] RIGHINI, M. Enabling Intel® Virtualization Technology Features and Benefits. *Intel White Paper. Retrieved January 15* (2010), 2012.

[44] RUSHANAN, M., AND CHECKOWAY, S. Run-DMA. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (2015).

[45] RUTKOWSKA, J. Subverting the Vista Kernel For Fun And Profit. *SyScan* (2006).

[46] SCHUSTER, A. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *digital investigation 3* (2006), 10–16.

[47] SHIELDS, T. Survey of Rootkit Technologies and Their Impact on Digital Forensics. http://www.donkeyonawaffle.org/misc/txs-rootkits_and_digital_forensics.pdf, 2008.

[48] SOMER, L. Linux Roootkit 5. https://packetstormsecurity.com/files/10533/lrk5.src.tar.gz.html, 2000.

[49] SONG, W., KIM, J., LEE, J.-W., AND ABTS, D. Security Vulnerability in Processor-Interconnect Router Design. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 358–368.

[50] SPARKS, S., AND BUTLER, J. Shadow Walker: Raising The Bar For Windows Rootkit Detection. *Black Hat Japan 11*, 63 (2005), 504–533.

[51] SPARKS, S., EMBLETON, S., AND ZOU, C. C. A Chipset Level Network Backdoor: Bypassing Host-Based Firewall & IDS. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (2009), ACM, pp. 125–134.

[52] TEAM, P. Documentation for the PaX project - overall description. http://pax.grsecurity.net/docs/pax.txt, 2008.

[53] VASISHT, V. R., AND LEE, H.-H. S. SHARK: Architectural Support for Autonomic Protection Against Stealth by Rootkit Exploits. In *2008 41st IEEE/ACM International Symposium on Microarchitecture* (2008), IEEE, pp. 106–116.

[54] WANG, G., ESTRADA, Z. J., PHAM, C., KALBARCZYK, Z., AND IYER, R. K. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In *9th USENIX workshop on offensive technologies (WOOT 15)* (2015).

[55] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Xen 0wning Trilogy. *Invisible Things Lab* (2008).

[56] WOJTCZUK, R., AND TERESHKIN, A. Attacking Intel BIOS. *BlackHat, Las Vegas, USA* (2009).

[57] ZADDACH, J., KURMUS, A., BALZAROTTI, D., BLASS, E.-O., FRANCILLON, A., GOODSPEED, T., GUPTA, M., AND KOLTSIDAS, I. Implementation and Implications of a Stealth Hard-Drive Backdoor. In *Proceedings of the 29th annual computer security applications conference* (2013), ACM, pp. 279–288.