Towards the design and implementation of a language dedicated to virtual machine introspection for security

Hemmerlé Lionel
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France

Hiet Guillaume
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France

Tronel Frédéric
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France

Wilke Pierre
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France

Prévotet Jean-Christophe
Univ Rennes, INSA Rennes, CNRS, IETR – UMR 6164, F-35000 Rennes
Rennes, France

***Abstract*—When using a Host-based Intrusion Detection System (HIDS), we need to protect it against attackers who manage to access a high privilege level. For that, we propose to use virtualization extensions: the protected system can be placed inside a Virtual Machine (VM) and the HIDS in the hypervisor. In this case, even if the VM containing the protected system is entirely compromised by an attacker, including the virtualized operating system, the IDS will still be functional. However, when the HIDS is located in the hypervisor, although it can access the VM memory and intercept all its communication with the hardware, it loses all the abstractions given by the virtualized operating system. To cross this semantic gap, we propose to create a new language that can be used by the VM to write and send programs to the hypervisor. Being produced by the virtual machine itself, we assume that these programs will have the necessary level of knowledge about the depths of the operating system used by the virtual machine. The hypervisor will process those programs and interpret them to detect intrusions.**

Besides, since those programs came from an untrusted source (the VM) and are executed in the hypervisor, we discuss some security constraints that must be enforced by our solution to ensure that it does not introduce new vulnerabilities in the hypervisor.

We already implemented two kernel rootkits that can hide a process from the userspace in the VM, and we created a detection mechanism that receives from a VM a list of memory areas that must be monitored by the hypervisor to detect the two rootkits that we have implemented.

1. Introduction

The first step for reacting to an attack is to detect it. An Intrusion Detection System (IDS) must be used for this. Such an IDS can be either located on some important nodes of a network or directly on the systems that must be protected. Network-based IDS are generally limited since they can only detect attacks leaving specific traces on the network, and are then unable to detect local attacks (for example a privilege escalation). Host-based IDS (also known as Endpoint Detection and Reaction or EDR), run directly on the system that must be protected, can detect a much broader range of attacks. Nevertheless, they can then be disabled by malware executed with sufficiently high privileges [1]. For example, an attacker that can compromise an operating system can kill processes related to the IDS or can send false information to them. To protect an IDS, we propose to use processor virtualization extensions.

A virtualization extension is a hardware extension allowing a processor to simulate multiple virtual processors. Those virtual processors are created and managed by a software component called a hypervisor, which can intercept every interaction between a virtual processor and the hardware to isolate the different virtual processors from each other. Each virtual processor can then execute its own operating system and constitutes a Virtual Machine (VM). Since the hypervisor is executed with a higher level of privilege than a VM, even if an attacker can entirely compromise a VM, it will not be able to attack the hypervisor or other VMs (assuming the hypervisor does not have a critical vulnerability). Consequently, the hypervisor can be used to protect a VM, for example, to protect the kernel code [2] or to enforce the integrity of some applications [3].

If we place an IDS inside the hypervisor, it has access to the full content of the VM, including the VM's memory, and its exchanges with hardware. However, the IDS loses the abstractions provided by the operating system running inside the VM. For example, to retrieve the list of processes running in a VM, the IDS needs to know how a process is represented in the VM memory and where the structures representing the running processes are stored. The IDS also needs access to specific events occurring in the VM, for example, the start of a new process. This problem is known as the semantic gap problem [4].

Multiple approaches have been proposed to cross the semantic gap. One such approach consists in patching the VM kernel code to attach hooks to specific kernel functions [5]. Those hooks can then detect some events in the VM and communicate useful information to the hypervisor, which can then potentially raise an alert. However, since this approach requires modifying the kernel code accessible in the VM, malware can then detect the modifications and can try to bypass those hooks.

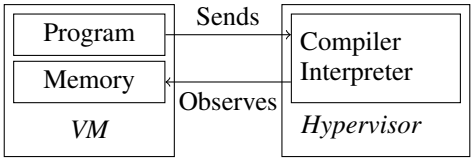

Figure 1. Representation of our detection system

Westphal et al. [6] developed a tool allowing users to indicate to the hypervisor, in a domain-specific language, which sequences of events occurring in a VM might be caused by an intruder. However, their approach is limited by the need for external configuration files for each OS used in the multiple VMs to cross the semantic gap.

To get rid of those files, another approach consists in using hyperupcalls [7]: a VM can send some programs to the hypervisor, which can then execute those programs to obtain information about the VM. This approach is interesting because it does not require external configuration files nor substantial changes inside the VM kernel. Nevertheless, it has only been used to optimize the usage of hardware resources by the hypervisor and its VM. Thus, it lacks protections against intruders who may be able to bypass an IDS made with those hyperupcalls.

In this work, our objective is to be able to detect intrusions by allowing VMs to send programs written in a specific language to the hypervisor. Section 2 gives a high-level overview of our approach. Section 3 introduces several malicious techniques that we aim at detecting with our solution. Section 4 presents the features of the domain-specific language we created, and the associated security concerns. The language is described with more details in appendix. Section 5 shows preliminary experiments in which our modifications to the Xvisor hypervisor, running on a Cortex-A53 ARM processor, allow to detect two of the rootkits described in Section 3. Finally, Section 7 concludes and gives perspectives for future works.

2. Threat model and approach

We suppose that a malicious agent can entirely compromise a VM, including the VM kernel, and we will focus on the detection of malware that may change internal data structures of the kernel (for example malware that takes the form of a malicious kernel module) since malware that runs only in userland can be detected without the need of a hypervisor (they can be detected by an IDS running inside the kernel). We will suppose that our hypervisor is safe, and one of our objectives is to ensure that our approach does not introduce new vulnerabilities to the hypervisor that can be used by an attacker to compromise the hypervisor, or another VM running on top of it.

To detect kernel rootkits, we want the VM to be able to describe, using a dedicated language, how intrusions should be detected in the VM. Programs written in this language will be communicated by the VM to the hypervisor during the early phases of the VM's life (we assume here that the VM has not yet been compromised during this early stage of its life) and should contain every piece of information to allow the hypervisor to cross the semantic gap. Those programs will be run as soon as the hypervisor detects specific events in the VM (Figure 1).

Safely executing programs from untrusted sources in a high-privilege context has already been done with eBPF [8], with proof-carrying code [9] or with Singularity [10], an OS which runs every process in ring 0 and relies on properties of its language to ensure the system safety.

To detect intrusions, the hypervisor should be able to attach programs sent by the VM to various events that may happen in the VM, including memory writes in specific structures, function calls, and register changes. The programs sent by the VM may then indicate to the hypervisor whether it should raise an alert or if new events should be monitored.

Our approach is analogous to the one used by eBPF inside the Linux kernel: eBPF is a language designed to trace events happening in the kernel from userspace. For that, processes running in userspace can send programs written in eBPF to the kernel, which can then compile those programs, execute them and send the results to the initial process. This approach is comparable to ours since it implies getting programs provided by an untrusted source (a process running in userspace) and executing them in a privileged environment (the kernel).

However, our approach differs from eBPF since the objective is not to provide the VM (the untrusted environment) insight into the state of the hypervisor (the trusted environment) but right the opposite: the hypervisor needs to get information about the state of the VM. Consequently, programs run by the hypervisor will only be able to access VM information, and their results will be exploited by the hypervisor. This means that our solution will have additional security constraints since an attacker may be able to manipulate our program's inputs (for example the VM memory).

Hence we need to introduce some constraints to prevent an attacker from compromising the hypervisor with our solution. As an example, we must restrict the sending of our programs during an initial phase in which we can consider that the VM has not been compromised yet (for example during the first boot). We must also guarantee that programs sent by the VM and executed by the hypervisor do not introduce vulnerabilities. For example, they should not be able to access all the hypervisor memory.

3. Rootkits

In this section, we introduce rootkits which aim at hiding their presence on the system. When users want to access the list of running processes, they can use the `ps` command, which parses the content of the `/proc` directory. The `/proc` filesystem contains a folder for every running process. To hide a process, a rootkit needs to delete the corresponding folder or hide it from userspace. In order to evaluate our ideas, we implemented two rootkits, which hide processes using two different methods. The first one hides a folder by changing the syscall made when a process wants to list the content of a directory, and the second one changes the structures used by the Virtual File System (VFS).

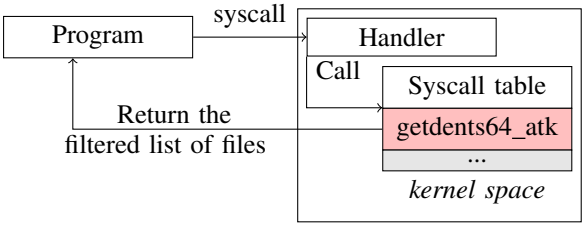

Figure 2. Manipulation of the syscall table

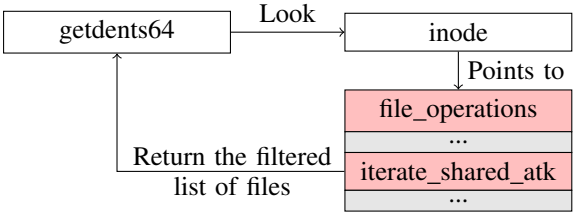

Figure 3. Manipulation of Virtual File System structures

3.1. Syscall table

System calls are a way for the kernel to expose some functionalities to running processes. For example, they can allow processes to interact with filesystems.

When a program running in userspace wants to list the content of a directory, it has to ask the kernel with the `getdents64` system call. When such a syscall is performed, a handler in the kernel looks for the corresponding function in the syscall table. This table consists of a structure containing a pointer to a specific function, one for each system call.

Consequently, to hide a folder from userspace, our rootkit can alter the syscall table by replacing the pointer to the function corresponding to the `getdents64` syscall with a pointer to a function controlled by the rootkit (Figure 2). This function can then change the results of the original `getdents64` syscall to hide some entries. In particular, it can hide folders located in the `/proc` filesystem to hide some processes.

Thus, if we want to detect this rootkit from the hypervisor, we need to be able to detect when a particular process located in the VM tries to write in the syscall table.

3.2. Virtual File System

The VFS is a component of the kernel providing a uniform interface for all the filesystem types supported by the kernel. It specifically exposes structures called `inodes` that represent files and folders, and structures called `file_operations` that contain various function pointers used to operate on inodes. Consequently, when a program in userspace uses the `getdents64` syscall to get the content of the `/proc` directory, the kernel accesses the corresponding `inode`, which contains a pointer to a structure `file_operations`. It then calls the function pointed by the `iterate_shared` field, which is used to read the content of a directory.

The second rootkit that we implemented can then replace the value of that `iterate_shared` field with a pointer to a function that can hide entries related to processes we want to hide (Figure 3).

If we want to detect this attack from the hypervisor, we need to detect writes to the structure `file_operations` used by the `inode` representing the directory `/proc`.

More generally, the hypervisor needs to detect some write operations in the VM memory. Note that, in the two examples we introduced so far, the addresses at which memory writes should be detected are constant, i.e. they do not evolve during the life of the VM (they could however be different at each boot, e.g. due to KASLR [11]).

3.3. More complex rootkits

The two rootkits described previously can be detected only by monitoring memory writes in some specific structures, but more complex attacks can also be performed.

For example, to hide a running process, a rootkit can alter the `task_struct` structures used to represent a process and the `pid` structures to ensure that the `/proc` filesystem does not see the hidden process.

Another approach is to create a new invisible process directly by manually creating a new `task_struct`, without using built-in functions. This approach requires to take steps to ensure that the newly created process is correctly scheduled by directly modifying the structures used by the scheduler. By doing so, this process will never have an entry created in the directory `/proc` and consequently, it will never be shown to userspace.

Those two attacks are harder to detect since they only need to modify structures that are evolving during the normal life of the VM since new processes are naturally created and destroyed. To detect them, we need to catch modifications of the scheduler's structure that do not go through the standard functions such as `fork` or `clone`.

4. Language

To detect complex attacks, we propose to create a dedicated language that a VM can use to communicate to the hypervisor when an alert must be raised. Programs written in this language will be executed by the hypervisor as a reaction to some events occurring inside the VM. We also need to ensure that the ability of the VM to send programs to the hypervisor does not introduce new vulnerabilities in the hypervisor. This can be done with verifications at compile and at run time.

Though the implementation of our language is still underway, we give an overview of its syntax and precise features in Appendix A. In this section, we describe the properties that our language must satisfy in light of the common techniques used by rootkits described in previous section.

4.1. Features

Since our objective is to detect and react to various events happening in a VM, our language needs to be designed using an event-driven approach (in the same vein as [12]). Some examples of events that our language needs to address are: (1) writes in some memory area; (2) changes of some system registers; (3) execution of instructions located at a given address.

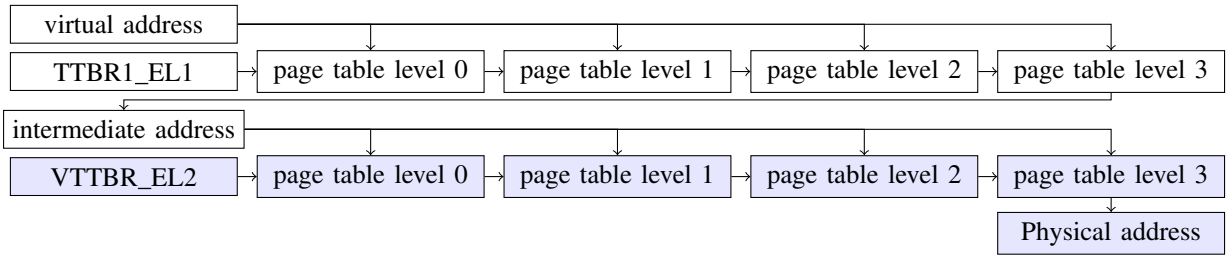

Figure 4. Address translation mechanism for a VM. Nodes in blues are controlled by the hypervisor while those in white are controlled by the VM.

4.1.1. Writes in memory. Detecting writes in some memory areas is necessary to detect most of the existing kernel rootkits (including the two rootkits we implemented), since they rely on the modifications of kernel data structures to hide information to userland. Those events are represented as `mem_access` in Figure 5, and can be detected by the hypervisor by modifying the structures used in the stage of the translation mechanism controlled by the hypervisor to forbid writes.

A process interacting with memory (e.g. through pointers) does not manipulate physical memory addresses, but virtual memory addresses. These virtual addresses are then translated by the processor using page tables whose physical addresses are stored in system registers called `TTBR0_EL1` (for userspace) and `TTBR1_EL1` (for kernel-space). Page tables are created for the kernel and for every running process.

With our configuration, the `TTBR0_EL1` registers are pointing to the level 0 page tables used by the address translation mechanism, which contain addresses of the level 1 page tables, which contain addresses of the level 2 page tables. In turn, these pages contain addresses of the level 3 page tables, which then contain physical addresses of 4kB contiguous regions of memory called pages. The physical address of any virtual address can then be obtained by combining the upper bits of the physical address of a page and the lower bits of the virtual address.

As shown in Figure 4, when the kernel is launched in a VM, the address translation mechanism described previously is used to get an intermediate address, which is then transformed into a physical address thanks to a similar mechanism controlled by the hypervisor.

The page tables controlled by the hypervisor contain flags indicating to the processor whether the VM can read, write or execute the content of each page. We can then forbid writes on a page containing protected data to ensure that the hypervisor is notified each time a write happens on a protected structure. When it happens, the hypervisor can then execute the corresponding program written in our language before resuming the VM execution.

4.1.2. System registers writes. System registers are used to configure the processor, and some of them can be accessed in the VM. For example, `TTBR1_EL1` and `TTBR0_EL1` which are used by the Memory Management Unit to translate virtual addresses into physical addresses can be changed by an attacker to replace some kernel data structures without writing directly into them. Writes in system registers can be detected by setting appropriate flags in the system register `HCR_EL2` which can only be accessed by the hypervisor. These events are represented as `reg_access` in Figure 5.

4.1.3. Execution of an instruction. Detecting the execution of an instruction located at a given address is a necessary step to detect function calls. By doing so, we can detect advanced attacks that modify structures that may be changed legitimately during the VM execution. These events are represented as `break` in Figure 5.

For example, if a rootkit tries to change the effective user id of a process to give it root privileges, we can catch it because this change can only happen legitimately under certain conditions, implying the use of a specific syscall (`setreuid` for example). Consequently, by monitoring the executions of the syscalls able to change the effective user id and the writes to the memory address at which the effective user id of a process is stored, we can detect if a change is benign or not.

Those can be detected by replacing the instruction we want to detect with an invalid instruction, and by configuring the hypervisor to handle the invalid instruction exception. Since this implies changing the code accessible from the VM, we must forbid writes on the page containing the modified instruction to avoid an attacker reverting our change, but since the kernel is not supposed to change its code, this should not impact untampered VMs. When the VM tries to execute the invalid instruction written by the hypervisor, an invalid instruction exception is raised by the hypervisor, which can then replace the invalid instruction with the real one, and execute exactly that instruction in the VM before replacing again that instruction with the invalid one (in a similar way than when the hypervisor detects writes in memory).

The hypervisor can then hide the instruction changed by the hypervisor by forbidding reads on a page until the VM effectively attempts to read it. In that case, the hypervisor can then replace the invalid instruction with the original one, and it can change the permission for that page to allow reads and forbid code execution until the VM tries to execute that code. In that case, the hypervisor places again the invalid instruction, reallows code execution, and forbids readings.

4.1.4. VM Memory accesses. When an event is detected in a VM, the corresponding program executed by the hypervisor may need to access the VM memory. For that, we provide the expression `VMmem` α (see Figure 5), which takes a virtual or intermediate address and return the value stored at that address in the memory of the monitored VM.

4.1.5. Dynamic event capture. The events that we want the hypervisor to monitor may change throughout the

execution of the VM: for example, if a new process is created, the hypervisor will have new structures to watch, like the `task_struct` structure that represents the new process, or the `creds` structure which contains the effective user id of the process. On the opposite, when a process is killed, the hypervisor should stop watching related events since the memory containing data structures representing the process can be reused for other purposes. Consequently, the language must allow dynamic registering and unregistering of events during the lifetime of the VM. This is the purpose of the `add_listener` and `remove_listener` instructions in Figure 5.

4.2. Security Constraints

Since the programs are sent by a VM and take as input events happening in the VM and its memory, some protections have to be implemented to ensure those programs cannot be used to compromise the hypervisor.

4.2.1. Emission of unwanted programs. First, we need to restrict the ability of a VM to send programs to the hypervisor to ensure that an attacker cannot send its own malicious programs. For that, we suppose that the VM is safe when booted for the first time, it can thus send its programs and then, can send a signal (`stop_trusting_me` in Figure 5) to the hypervisor indicating that the last program has been sent. Since this signal is emitted while the VM is still considered safe, an attacker cannot try to block it and cannot send its programs afterwards, since they will be ignored by the hypervisor (any attempt to send a new program may raise an alert when the VM is not trusted).

4.2.2. Language safety. Since the programs sent to the hypervisor react to events possibly triggered by an attacker, we must ensure that programs in our language cannot compromise the memory of the hypervisor, or leak data from another VM. The language described in Figure 5 does not allow programs to perform unrestricted memory accesses. The only memory accesses that are permitted are read accesses through the `VMmem` expression, and map manipulations. In the first case, the interpreter will ensure the requested address belongs to the VM and is valid; in the second case, the interpreter will check the validity of the map accesses.

The language only allows statically bounded loops, thus preventing infinite loops.

In order to defend against Denial-of-Service (DoS) attacks, we can bound the number of listeners registered for a single VM, and raise an alert when the threshold is reached.

4.2.3. Step by step mode. Some event detection needs to rely on the ability to execute only one instruction in the VM before re-enabling some protections. For that, we use the debug extension of ARM processors to trigger software-step exceptions each time an instruction is executed in the VM when we remove protection. This allows us to properly detect concerned events, but it introduces some security caveats: we need to change the value of the `MDSCR_EL1` register and the value of PSTATE which can be accessed by the VM. We also need to make sure that, in VM with multiple cores, protections are lifted only for the core in which the concerned event is detected.

To prevent changes to the `MDSCR_EL1` register and the PSTATE, we must then ensure that the instruction executed in step-by-step mode is not an `MSR` instruction (this instruction is used to write in system registers), and we need to guarantee that exceptions cannot be triggered in the VM, since such an exception may change the PSTATE, allowing the VM to execute multiple unwanted instructions.

5. Integration

We used XVisor [13], a minimalist hypervisor that can be executed on ARM processors, to run a VM containing a Linux kernel. We implemented XVisor on the Cortex-A53 processor of a Xilinx Zynq UltraScale+ ZCU104 board. XVisor and the VM were configured to use two physical UARTs: one connected to XVisor itself, and the other connected in pass-through to the VM.

We implemented the two rootkits described in Section 3 that can be executed in that kernel and which can be used to hide a process from userspace in the VM. Hidden processes cannot be seen inside the VM by running `ps` or by manually looking into the `/proc` filesystem). We then patched XVisor to detect those rootkits.

5.1. Rootkits

We implemented the two rootkits described in Section 3 as kernel modules for version 5.11 of the Linux kernel that implements the described behaviors.

To implement the rootkit that modifies the syscall table, we had to find the address of that table. Since it can change at each boot, we used the `kallsyms_lookup_name` function which takes any symbol's name as a parameter and returns its address in the kernel. Note that this function is not supposed to be accessible for kernel modules. To find the function's address, we used a kprobe [14]. Kprobes are a mechanism allowing to trap almost any function to debug the kernel. We can then register a kprobe targeting the `kallsyms_lookup_name` function to obtain a `kprobe struct` containing the function's address. It is then possible to call that function to obtain the syscall table address (since the syscall table is not a function, we cannot obtain its address directly with a kprobe).

For the aarch64 architecture, the entry we need to change to replace the `getdents64` syscall is located at index 61 of the syscall table. Since the syscall table is read-only, our kernel module cannot change it directly. It needs to bypass the write protection on it. For that, we decided to use the kernel function `aarch64_insn_patch_text` which maps the physical page containing the data that need to be written to a new virtual address on which writes can be performed.

To communicate the *pid* of the process we want to hide with this kernel, we decided to also change the function called when the syscall `kill` is made. When it is used to send a specific signal to a process, the syscall is intercepted by our rootkit, which then hides the targeted process from userspace.

To implement the rootkit modifying the VFS, we used the `filp_open` function that returns a pointer to a struct `file` containing itself a pointer to the inode of the opened file or folder. With that, we could obtain the address of the inode representing the `/proc` folder, allowing us to get the address of the corresponding `file_operations`. Then, we can change the field `iterate_shared` of that structure to perform the attack.

5.2. Rootkit detection

To detect the two rootkits we implemented, we patched XVisor to catch some writes in the VM memory. To do so, we changed the permission flag of the entries corresponding to the physical address of the structures we want to protect in the page table level 3 used by the translation mechanism (Figure 4) to reduce at the maximum the size of the memory area in which writes are forbidden by the hypervisor.

To increase its performance, XVisor uses mega pages when mapping the VM memory to the physical memory: It thus exploits the possibility for page table level 2 to point directly to physical memory instead of a new page table. This reduces the number of steps needed to translate a virtual address by one. Consequently, since the data we want to protect are mostly smaller than a mega page, we split those mega pages into 512 regular pages by creating the necessary level 3 page table to reduce the amount of data that may unnecessarily trigger exceptions in the hypervisor.

Since the structures in which we want to monitor writes are not aligned with page boundaries, some exceptions may be raised by legitimate writes on the same page of a protected structure, but not in those structures. Those writes should still be executed by the VM. Furthermore, some instructions that write on memory have side effects (they can change the value of some registers). Consequently, they cannot be skipped by the hypervisor without risking provoking a kernel panic in the VM, even if they try to write on protected structures. Thus, the hypervisor still needs to execute the writes and differentiate those on protected structures and those around them. For that, when a `data abort exception` (the exception received by the hypervisor when the VM tries to access an area of its memory protected by the hypervisor) happens, the hypervisor can allow writes on the concerned page and execute the VM in step-by-step mode to execute only the faulting instruction. Once the instruction is executed, the hypervisor can check if protected structures on the page have been modified. If a modification is detected, the hypervisor can then raise an alert and revert the write.

Since a page can contain up to 4 kilobytes of protected data, we can use the fact that an instruction will not write more than 8 consecutive bytes in memory and, when a `data abort exception` occurs, the `FAR_EL2` register contains the faulting virtual address. Consequently, the hypervisor only has to verify whether any of the 8 written bytes are protected.

To effectively protect a VM, the hypervisor needs to know where the structures we want to monitor are located in the VM memory. Their addresses should be sent by the VM using programs written in the language described in Section 4. Since this language is not implemented yet, we implemented a hypercall allowing a VM to send a list of memory areas, in which writes should be forbidden to the hypervisor.

With this system, XVisor is now able to detect writes on memory areas indicated by the VM. If that VM asks the hypervisor to forbid writes on the syscall table and the structure `file_operations` of the `/proc` folder, the hypervisor can now raise alerts if writes on those structures happen. This allows the hypervisor to detect the two rootkits we have implemented. Furthermore, since changes in the protected memory areas are reversed by the hypervisor, we can even protect the VM against those attacks.

6. Limitations

Our proposal is not fully implemented yet, hence we cannot provide a measure of the performance overhead induced by our approach. We expect that in order to prevent against advanced attacks, the number of events to monitor will grow significantly, and impact the performance negatively.

The current state of our implementation only allows to detect simple rootkits, that try to write on read-only data (Type-I malwares according to Rutkowska's classification [15]). More complex attacks (Type-II malware), that may change data which can be subject to legitimate changes, e.g. by modifying structures related to the scheduling process, or structures used to represent a given process, are currently out of reach of our implementation (although they could be tracked by dynamically adding new event listeners).

7. Conclusion and future work

We implemented two rootkits that can be used to hide a process in a VM by modifying data structures in the Linux kernel. We then implemented a way to detect those rootkits in XVisor, by monitoring writes in some areas in the VM memory, as well as an hypercall allowing the VM to communicate which areas of its memory may be written by a rootkit.

Our next step is to work on the implementation of the proposed language, in particular through the use of an interpreter in the hypervisor. We will evaluate the performance overhead on a set of benchmarks, and try to optimize our event monitoring. On the long-term, we could also explore using a Just-In-Time (JIT) compiler to improve performance while maintaining the same level of security.

References

[1] Wavestone Cybersecurity and Digital Trust practice. (2022) EDRSandBlast. [Online]. Available: https://github.com/wavestone-cdt/EDRSandblast

[2] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 335–350.

[3] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 143–158.

[4] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[5] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008, pp. 233–247.

[6] F. Westphal, S. Axelsson, C. Neuhaus, and A. Polze, "Vmi-pl: A monitoring language for virtual platforms using virtual machine introspection," *Digital Investigation*, vol. 11, pp. S85–S94, 2014, fourteenth Annual DFRWS Conference.

[7] M. Wei and N. Amit, "Leveraging hyperupcalls to bridge the semantic gap: An application perspective." *IEEE Data Eng. Bull.*, vol. 42, no. 1, pp. 22–35, 2019.

[8] BPF and XDP reference guide — cilium 1.10.6 documentation. [Online]. Available: https://docs.cilium.io/en/v1.10/bpf

[9] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *OSDI*, vol. 96, no. 16, 1996, pp. 229–243.

[10] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, 2007.

[11] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," ser. ASIA CCS '20, 2020, p. 481–493.

[12] Bpftrace github repository. [Online]. Available: https://github.com/iovisor/bpftrace

[13] Xvisor hypervisor. [Online]. Available: https://github.com/xvisor/xvisor

[14] Kernel probes (kprobes). [Online]. Available: https://docs.kernel.org/trace/kprobes.html

[15] J. Rutkowska, "Introducing stealth malware taxonomy," 2006.

A. Syntax

We present the syntax of our proposed language in Figure 5. Commands (**Cmd**) are the messages sent by the monitored OSes to the hypervisor in order to bridge the semantic gap. Commands can indicate the (intermediate) address of the kernel page table, create a map (*à la* eBPF, where maps are key-value stores), define a function, call a function, or indicate to the hypervisor that the OS should not be trusted after this point.

Functions are simply names associated with instructions (no parameter passing and no function calls from within functions). Instructions (**Instr**) can be `skip` (do nothing), a sequence $i_1; i_2$, an assignment $x := e$, a statically bounded loop `loop` n i (that runs i exactly n times), a conditional instruction `if` $-$ `then` $-$ `else`, map updates (`update` and `delete`), raising an alert and, most importantly, registering a new event handler with $x \leftarrow$ `add_listener` ev f and removing a listener with `remove_listener` x.

The events we can add listeners to are memory accesses (read or writes) to a specified range of addresses ι, system register accesses, and breakpoints at a given address. An address is given as a pair (k, e) where e is an expression evaluating to the concrete address as a `uint64` and k specifies the kind of address: intermediate (i.e. physical address for the virtual machine) or virtual (relative to the kernel page table). Similarly, a range of addresses ι is a 3-tuple (k, lo, hi) where k indicates the kind of addresses and lo and hi are the bounds of the range considered.

$$
\begin{array}{rcl}
x \in \mathbf{Var} & & \\
m \in \mathbf{MapId} & & \\
k \in \mathbf{Key} & & \\
u \in \mathbf{Uint64} & & \\
f \in \mathbf{FunName} & ::= & \texttt{string} \\
r \in \mathbf{Reg} & ::= & \texttt{pc} \mid \texttt{reg}(i) \\
e \in \mathbf{Expr} & ::= & \texttt{cst}\ u \mid \texttt{var}\ x \\
 & \mid & \texttt{binop}\ bop\ e_1\ e_2 \mid \texttt{unop}\ uop\ e \\
 & \mid & \texttt{lookup}\ m\ k \\
 & \mid & \texttt{VMreg}\ r \mid \texttt{VMmem}\ \alpha \\
 & \mid & \texttt{event} \\
 & & \\
k \in \mathbf{Kind} & ::= & \texttt{Intermediate} \mid \texttt{Virtual} \\
\alpha \in \mathbf{Addr} & ::= & \mathbf{Kind} \times \mathbf{Expr} \\
\iota \in \mathbf{Range} & ::= & \mathbf{Kind} \times \mathbf{Expr} \times \mathbf{Expr} \\
a \in \mathbf{Access} & ::= & R \mid W \\
ev \in \mathbf{Event} & ::= & \texttt{mem\_access}\ a\ \iota \\
 & \mid & \texttt{reg\_access}\ r \\
 & \mid & \texttt{break}\ \alpha \\
 & & \\
i \in \mathbf{Instr} & ::= & \texttt{skip} \mid i_1; i_2 \mid x := e \mid \texttt{loop}\ n\ i \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2 \\
 & \mid & \texttt{delete}\ m\ k \mid \texttt{update}\ m\ k\ e \\
 & \mid & \texttt{alert} \\
 & \mid & x \leftarrow \texttt{add\_listener}\ ev\ f \\
 & \mid & \texttt{remove\_listener}\ x \\
 & & \\
c \in \mathbf{Cmd} & ::= & \texttt{kernel\_pagetable}\ u \\
 & \mid & m := \texttt{create\_map} \\
 & \mid & \texttt{fundef}\ f\ i \\
 & \mid & \texttt{dofun}\ f \\
 & \mid & \texttt{stop\_trusting\_me}
\end{array}
$$

Figure 5. Syntax of the proposed language

Expressions (**Expr**) are either constants (`cst` u, where $u \in$ `uint64`), variables (`var` x), unary (`unop`) and binary (`binop`) operations on expressions, map lookups (`lookup`), guest OS register read (`VMreg`) and memory read (`VMmem`). A special expression `event` contains an encoding of the event we are currently responding to. Considering memory addresses do not use all the 64 bits available (restricting the address bus width to 56 bits for example), we can imagine the following encoding: bits 63-62 indicate the type of event (memory access, register access, breakpoint, no event), bit 61-60 indicates the access type (in the case of memory or register access, read or write or execution for breakpoints), and remaining bits 59-0 encode the memory address or the number of the register that was accessed.