Detection of execution in a virtualized environment

Malware class 2020

ShellCode33, Hippwn

March 4, 2020

Contents

Abstract

This paper has been written during an *Introduction to Malware* class in a French engineering school. We will be focusing on the runtime detection of virtualized environment. Most malware today uses complex techniques to detect sandboxes and prevent their own execution, thus making their analysis more complex. Through the following pages, we will come back on the differences the different virtualisation techniques and then study the state of the art of today's virtual machines detection.

The virtualisation

The concept of virtualisation itself is hard to define as it is subject to many different interpretations – or at least different levels of application. In fact, even the way the different malware-auditing sandboxes work is often distinctive from one another. Moreover, the term of virtualisation sometimes hides something else (isolation, containerization, emulation…) and can be misleading. If we will be focused on the virtualisation of processes in this paper, it must be acknowledged that everything can be virtualized, from the storage to the networks.

Isolation

This is an old concept on Linux-based systems which has more recently appeared on Windows 10 (1803). It is not really a virtualisation but more of a way of running a process in an independent environment that we call *context*. The isolated process access and system calls are filtered so that it is not aware of the host he's running on. This is basically the way containers work (LXC, Docker), excepted on Windows which also provides a per-process virtualisation based on Hyper-V, its own Type-1 virtualisation system (see below).

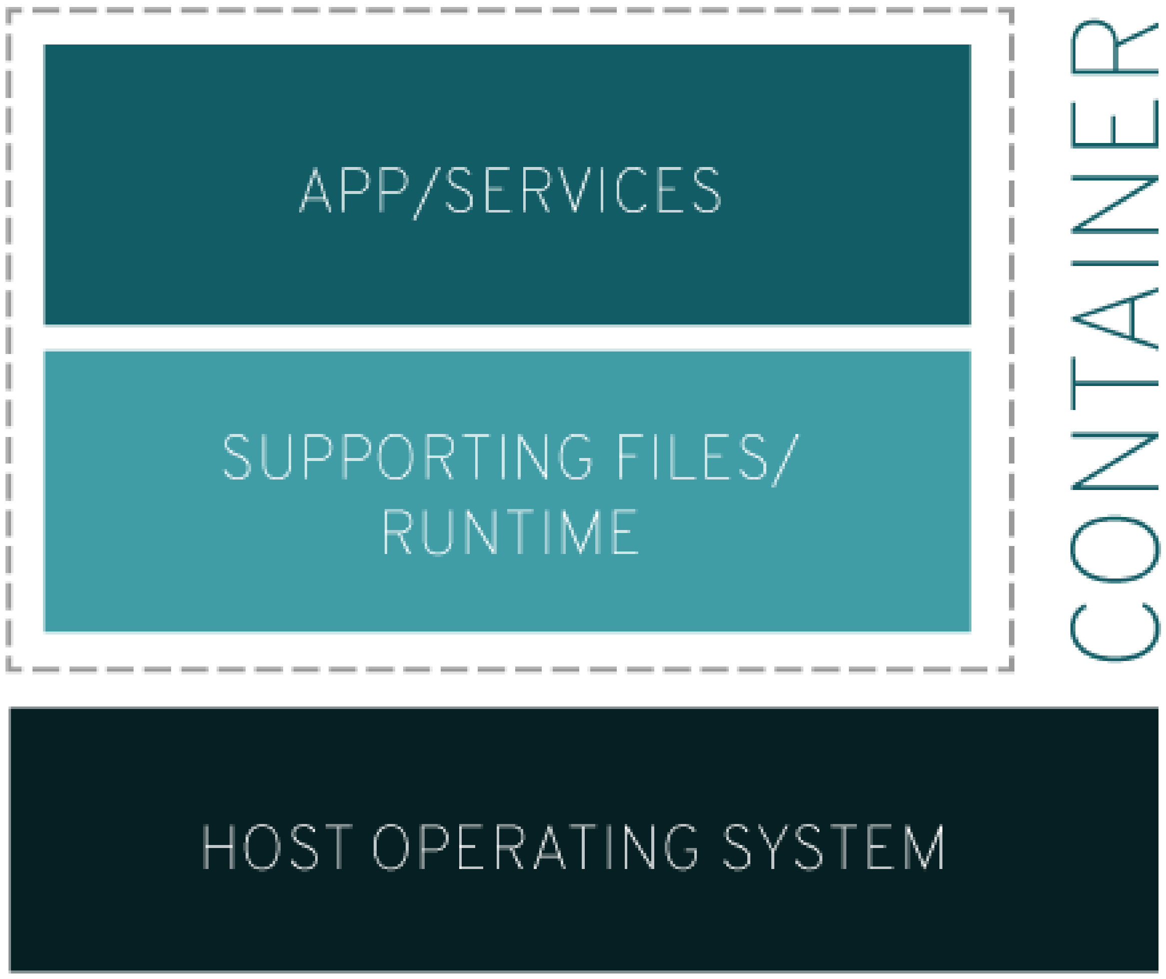

Figure 1: Container architecture by RedHat

In more technical terms, the isolation of processes rests on kernel-level features like *cgroups* (isolation of material resources: RAM, CPU…), *chroot* (change the *root* – or **/** – directory), *namespaces* (partitioning of kernel resources) on Linux. Such functionalities also exist on Windows environment but with other names.

Kernel in user-space

Just like the isolation, it is hard to call this virtualisation as it does not virtualize the hardware. This is mostly used in kernel development, allowing one to run the kernel above its own operating system like any other program. It is not really used in other contexts so we will not elaborate.

Type-2 hypervisor

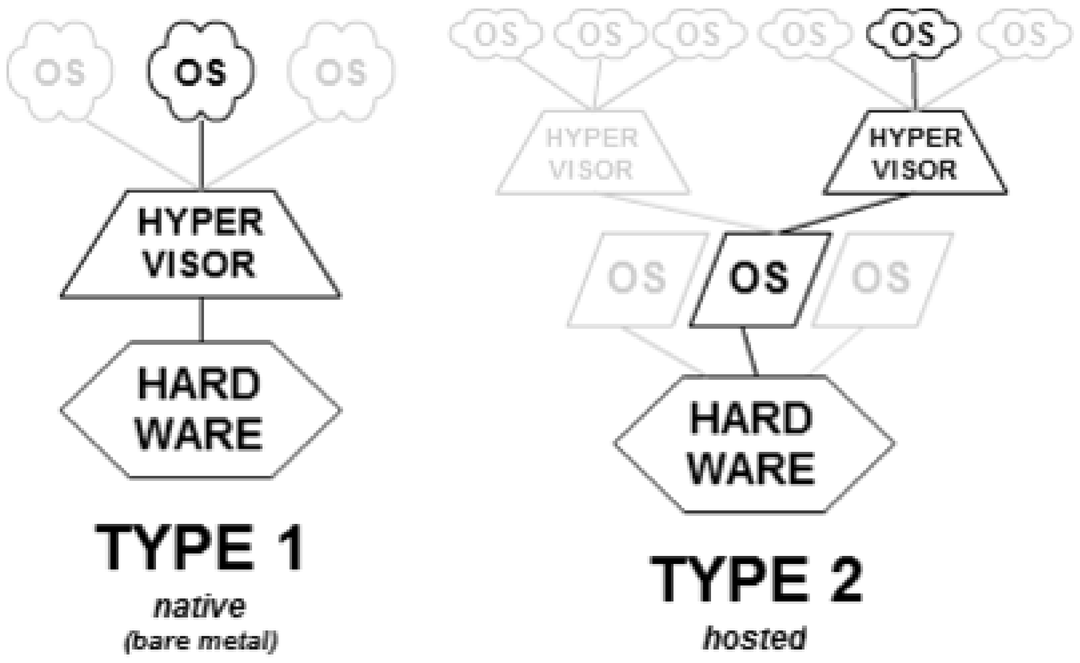

Figure 2: The difference between hypervisors

An hypervisor is a platform that creates and runs virtual machines. We talk about type-2 or hosted hypervisors when the software runs above an operating system (*VirtualBox*, *VMWare Workstation*, *Parallel Desktop*, *QEMU*…). This is often the solution chosen by the average users as it allows them to run different operating systems on their day-to-day computer. But in this category we have to make the distinction between the hardware virtualisation and the software that we will call emulation.

In the first case, the virtualisation is supported directly by the hardware, thanks to specific CPU instructions. Those technologies (*Intel VT* for the blues and AMD-V for the reds) allow the hypervisor to delegate the memory and CPU management to the hardware itself, thus simplifying the software virtualisation. But this is only possible when you virtualize a machine with the same architecture than the host: an x86_64 guest on a x86_64 host for instance.

The emulation, on the other hand, is required when the CPU does not support virtualisation assistance or when you run another architecture (like ARM when virtualizing an Android device). This time, the host has to simulate the whole hardware on which the guest is supposed to be running and thus translate each system call. The type-2 is already the slower way of virtualizing a system, but it is even truer with emulation as this is an inefficient process by definition.

Type-1 hypervisor

Type-1 or bare-metal hypervisors are the closest to the hardware. In this situation, the hypervisor runs directly on the machine and serves as a lightweight operating system (*VMWare vSphere*, *Citrix Xen Server*, *Microsoft Hyper-V*…). This is way more efficient than the type-2 because you do not have an host OS that would consume resources. This kind of infrastructure is mostly used in data centers to simplify the deployment of virtual machines and their performance.

Note that this is not incompatible with the fact of running an operating system on the host machine: *Hyper-V* can be run from Windows 10 Pro for instance, allowing good virtualisation performance and still having a user-friendly OS.

State of the art

In this part, we will dive into the concept of evasion. Malware authors are ahead of the analyst and it is a lead they have to maintain. To this end, they deploy more and more techniques to make harder the analysis and comprehension of their code. Against static analysis they use obfuscation, encryption and such, as for dynamic ones they use evasion.

The concept of evasion refers to all the techniques used by a malware to hide its behaviour according to its environment. For instance, if a malware detects a sandbox (from an analyst or an antivirus), it will make low profile to so as not to arouse suspicion. One of the most known example of this is the Red Pill demonstration (reference to the legendary film *The Matrix*) presented by Joanna Rutkowska in 2004 – just two years before she presents the Blue Pill attack which is a type of *hyperjacking*.

Red Pill is a small piece of code written in C that checks the address of the *Interrupt Descriptor Table* (IDT). The address of this table has to be modified by the hypervisor to avoid memory conflicts. Therefore, there is a correlation between the address being superior to `0xD0` and the fact of being executed in a virtual machines. This technique is however less efficient on today's systems as those filter the access to certain zones of the memory, such as the DTI. It still works on *QEMU* though.

```c
int swallow_redpill ()
{
  unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
  *((unsigned*)&rpill[3]) = (unsigned)m;
  ((void(*)())&rpill)();
  return (m[5]>0xd0) ? 1 : 0;
}
```

Cross-platform solutions

When developing a malware, people usually target an operating system, as it can be pretty difficult to build something that works as expected on each environment. Moreover, releasing variants of the same binary (compiled for the different environments) can facilitate the work of malware analysts. Despite these considerations, we aimed our researches toward cross-platform solutions in order to mutualize efforts.

Networking

Network adapters usually come with a MAC address (*MAC* stands for Media Access Control, referring to the lowest part of the OSI model) which can be used to identify its vendor. The first half of the address (the first 3 bytes) are booked by constructors with the IEEE (Institute of Electrical and Electronics Engineers, an international organisation dedicated to the writing of standards for new technologies) to make the OUI, an unique vendor identifier.

Most hypervisors have an OUI so that it makes the network adapter easily recognizable for the guest system. So if the a system sees such an OUI on its network adapter, it is highly likely that it is a virtualized guest.

Using CPUID

THe `CPUID` instruction has been introduced with Intel's *x86* architecture to allow CPU discovery by the operating system. This way, the system can adapt its behaviour to the characteristics of the processor. The use of this instruction has been extended in 2008 to allow the hypervisor to "interact" with the guest and thus optimizing its performance. By watching specific values of certain registers – mostly EBX, ECX, EDX – we can deduce the hypervisor if any.

Measuring resources availability

Finally, low resources may be an indication that the operating system is running inside a sandbox or virtual machine. It surely cannot be used as the only clue but it can lead you to investigate: most sandboxes are ran on the laptop of the analyst, who often will give the fewest resources they can. That is why we look for resources below 3 vCPUs or 3 GB of RAM.

Linux techniques

The DMI table

DMI stands for *Desktop Management Interface*. It is a standard developed in the 90' with de goal of uniforming the tracking of the components in a computer and abstracting them from the softwares supposed to run them. Parsing this table can reveal practical information on the hardware used by the operating system and possibly detect the presence of names specific to virtualized environment, such as *vbox*, *virtualbox*, *oracle*, *qemu*, *kvm* and so on.

Linux kernel's hypervisor detection

Linux's kernel comes with an hypervisor detection feature that can be used to identify a potential hypervisor below the operating system. Based on this, we easily can listen for the kernel event to see if an hypervisor has been detected by the kernel:

```c
static inline const struct hypervisor_x86 * __init
detect_hypervisor_vendor(void)
{
    const struct hypervisor_x86 *h = NULL, * const *p;
    uint32_t pri, max_pri = 0;

    for (p = hypervisors; p < hypervisors + ARRAY_SIZE(hypervisors); p
        ++) {
        if (unlikely(nopv) && !(*p)->ignore_nopv)
            continue;

        pri = (*p)->detect();
        if (pri > max_pri) {
            max_pri = pri;
            h = *p;
        }
    }

    if (h)
        // this line prints the hypervisor in the `/dev/kmsg` file
        pr_info("Hypervisor detected: %s\n", h->name);

    return h;
}
```

Checking Linux's pseudo-filesystems

Linux provides a lot of information via a certain type of files (mostly in `/proc`) that are generated at boot and modified during runtime. A lot of binaries use this directory like `ps`, `uname`, `lspci` and so on. These information are really helpful when trying to identify wether or not you are in a virtualized environment, like UML for instance. UML refers to the aforementioned way of executing a Linux kernel in user-space. This can easily be verified by looking for the string “User Mode Linux” in the file `/proc/cpuinfo` which describes the CPU of the machine.

In the same way, a lot of these virtual *files* can provide information on the environment, including – but not limited to – `/proc/sysinfo` (in which some distribution expose data about virtual machines), `/proc/device-tree` (that lists the devices on the machine), `/proc/xen` (a file created by the *Xen Server*) or `/proc/modules` (that contains information about the loaded kernel modules, modules that are used by hypervisors to optimize the guests).

Like *procfs* (mounted in `/proc`), *sysfs* can be useful. Its role is to provide to the user an access to the devices and their drivers. The file `/sys/hypervisor/type`, for instance, is sometimes used to store information about the hypervisor Linux is running on.

Windows

On Windows, most configuration can be done through the *Registry Hive* – some kind of database that contains every configuration option about either the operating system itself, or any software that would like to store information in it. A lot of indicators of hypervisors can be stored there, especially if the *guest addons* (small pieces of software that are installed on the guest to allow interoperability between the guest and the host, permitting shared clipboard, *drag'n'drop* and so on) are installed.

Most keys will be installed inside the `HKEY_LOCAL_MACHINE` register which mostly contains information about hardware, security and such. Parsing its content looking for particular patterns is efficient enough and quite a good indicator of the presence of an hypervisor if any. Here is an example of keys that we are looking for:

```
virtualBoxKeys := []string{
    `HKLM\SYSTEM\CurrentControlSet\Enum\PCI\VEN_80EE*`,
    `HKLM\HARDWARE\ACPI\DSDT\VBOX__`,
    `HKLM\HARDWARE\ACPI\FADT\VBOX__`,
    `HKLM\HARDWARE\ACPI\RSDT\VBOX__`,
    `HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions`,
    `HKLM\SYSTEM\ControlSet001\Services\VBoxGuest`,
    `HKLM\SYSTEM\ControlSet001\Services\VBoxMouse`,
    `HKLM\SYSTEM\ControlSet001\Services\VBoxService`,
    `HKLM\SYSTEM\ControlSet001\Services\VBoxSF`,
    `HKLM\SYSTEM\ControlSet001\Services\VBoxVideo`,
}
```

Sources

- https://docs.microsoft.com/fr-fr/virtualization/windowscontainers/manage-containers/hyperv-container
- https://poweruser.blog/lightweight-windows-containers-using-docker-process-isolation-in-windows-10-62519be76c8c
- https://fr.wikipedia.org/wiki/Virtualisation#Diff%C3%A9rentes_techniques
- https://en.wikipedia.org/wiki/Hypervisor
- https://fr.wikipedia.org/wiki/X64
- https://www.docker.com/resources/what-container
- https://securiteam.com/securityreviews/6z00h20bqs/
- https://arxiv.org/pdf/1811.01190.pdf
- https://daks2k3a4ib2z.cloudfront.net/5757fcb8825e8dbc6c852e3c/59ad6c357ba794000108098c_Minerva_Int
- https://en.wikipedia.org/wiki/Desktop_Management_Interface
- https://github.com/torvalds/linux/blob/31cc088a4f5d83481c6f5041bd6eb06115b974af/arch/x86/kernel/cpu/hy
- https://www.ibm.com/support/knowledgecenter/en/linuxonibm/com.ibm.linux.z.lhdd/lhdd_t_sysinfo.html