VB2020 localhost

30 September - 2 October, 2020 / vblocalhost.com

A NEW OPEN-SOURCE HYPERVISOR-LEVEL MALWARE MONITORING AND EXTRACTION SYSTEM – CURRENT STATE AND FURTHER CHALLENGES

Michał Leszczyński
CERT Polska

Krzysztof Stopczański
CERT Polska (former)

michal.leszczynski@cert.pl
krzysztof@stopczanski.pl

ABSTRACT

During this talk, we will present DRAKVUF, an open-source blackbox binary analysis system. This project leverages virtual machine introspection and *Xen*'s altp2m in order to serve its purpose in a very stealthy manner. We will describe our recent contributions to the project, including *Windows* API tracing and heuristic malware unpacking. Moreover, we will present how this approach can be used to extract configuration from malware samples. In addition, we will present some unique challenges that can be encountered when developing hypervisor-level monitors.

Virtual machine introspection is a technique used to determine the runtime state of a virtual machine (VM). It can be employed on the host side, without any explicit interaction that would be visible to the inspected guest system.

Xen's altp2m is a memory-management layer that leverages Intel Extended Page Tables in such a way that a single virtual machine can have multiple machine-to-physical page tables that can quickly be swapped at runtime. This feature makes it possible to create an efficient, external monitoring system.

The proposed malware monitor works in clean, unmodified environments, without any changes needed to the target guest system. This distinguishes it from typical user (or kernel) level monitors that need to run their code inside the monitored VM. On the other hand, a hypervisor-level monitor can easily play with kernel mode, but not with higher abstraction layers. This is due to the semantic gap that is not easy to overcome. During the presentation, we will show a few tricks that may be leveraged in order to understand what is happening in user mode.

INTRODUCTION

DRAKVUF is a novel black-box binary analysis tool that is capable of tracing an unmodified operating system. Using the technique called virtual machine introspection, it can intercept and interpret various events happening inside the VM without requiring an agent program to be installed on the inspected system. In this paper, we will provide a basic overview of DRAKVUF's inner workings. We will also thoroughly describe our contributions that involved WinAPI tracing and memory dumping features.

DRAKVUF is a program that can attach to a given virtual machine (particularly meaning *Xen*'s DomU) and trace many kinds of events that are occurring within the VM's kernel and the processes running inside of that VM. On a high level, it could be thought as a '*strace* for virtual machines'. It also contains a plug-in system and an internal framework (libdrakvuf), which provides a base for further extensions. The most representative plug-in is a system call tracer:

```
# drakvuf -d <vm-name> -r <volatility-profile> -a syscalls
1591449887.690727 DRAKVUF v0.7-git20200604002607+141a541-1 Copyright (C) 2014-2020 Tamas K
Lengyel
[SYSCALL] TIME:1591449888.114670 VCPU:1 CR3:0x2c459000,"\Device\HarddiskVolume2\Windows\System32\
svchost.exe" PID:984 PPID:400 TID:3048 SessionID:0 21:nt!NtAllocateVirtualMemory Arguments: 6
       IN HANDLE ProcessHandle: 0xffffffffffffffff
       INOUT PVOID *BaseAddress: 0xfffff880022de928
       IN ULONG_PTR ZeroBits: 0x0
       INOUT PSIZE_T RegionSize: 0xfffff880022de918
       IN ULONG AllocationType: 0xfffff88000002000
       IN ULONG Protect: 0xfffffa8000000004
[SYSRET] TIME:1591449888.115130 VCPU:1 CR3:0x2c459000,"\Device\HarddiskVolume2\Windows\
System32\svchost.exe" PID:984 PPID:400 TID:3048 SessionID:0 21:nt!NtAllocateVirtualMemory Ret:0
Info:STATUS_SUCCESS
...
```

MALWARE ANALYSIS ENHANCEMENTS

Considering the process of malware analysis, one may leverage different tools and techniques depending on the assumed extents of the analysis, which are dependent on the question that the analyst wants to answer:

- Is sample X malicious or not?
- What kind of malware does X represent? (banker, remote access trojan, ransomware, etc.)
- What family of malware does X belong to? (e.g. Emotet, GandCrab, etc.)
    - What is the static configuration of sample X? (e.g. hard-coded domain names, encryption keys, etc.)

One of the most basic tools used by malware analysts trying to find answers to the above questions are WinAPI trace behavioural logs. Such kind of trace is a common artifact produced by malware analysis systems. During our research, we have upstreamed patches implementing user-mode hooking and WinAPI call tracing to the DRAKVUF system [1]. The details related to this implementation will be described in a later section.

In addition, considering the very last of the above-mentioned questions, the dynamic analysis may aid the process of static configuration extraction by providing memory dumps of heuristically chosen regions of the malware machine code. Such memory dumps could then be used to identify the particular malware family and to obtain hard-coded configuration values like URLs, encryption keys, etc. Patches regarding this part were also upstreamed to the DRAKVUF system [2] and will be described in the 'Memory dumps' section.

INTERNALS OF DRAKVUF

Xen's altp2m

One of the preconditions of DRAKVUF is that the system that we want to trace must run as a fully virtualized *Xen* HVM (hardware virtual machine) guest. In this mode of operation, the hypervisor is virtualizing the physical address space of the virtual machine. As a result, there are two physical memory spaces:

- Guest physical address space – 'what the VM thinks is physical RAM'
- Machine physical address space – the actual physical RAM

The part of the *Xen* hypervisor responsible for providing mappings between guest physical addresses and machine physical addresses is called p2m ('physical to machine'). For modern *Intel* processors, this mechanism may be backed with the Intel Extended Page Tables (EPT) hardware feature. Usually, the hypervisor creates a single EPT for each virtual machine.

However, *Xen* also contains an extension to p2m mechanism called altp2m, which allows multiple EPTs to be created and to switch between them at runtime. This makes it possible not only to stealthily trace memory accesses, but also to implement breakpoints [3] that trap directly into the hypervisor and are not detectable from the perspective of the monitored virtual machine [4].

LibVMI

Since DRAKVUF is an agentless system, special provisions are required in order to access the state of the observed virtual machine. Such an interface is provided by LibVMI ('Virtual Machine Introspection Library'), which is capable of inspecting the VM's CPU state and memory. It also has the capability to parse the VM's page tables in order to map virtual addresses to guest physical addresses. In addition to all these static access features, LibVMI can subscribe to various *Xen* events that occur within a VM.

INITIAL ASSUMPTIONS

In order to better describe our recent contributions to the DRAKVUF project, let's start with the features that were already implemented before our first commit[1]. Our changes will be then presented as new features built on top of the existing ones.

The first important abstraction that is already available to us are *stealth breakpoints*. Using existing DRAKVUF functions, it is possible to place a breakpoint instruction 0xCC on an arbitrary physical address. Such a breakpoint is covered with altp2m machinery in a way that the guest VM cannot detect it by reading the code from memory, i.e. the guest would read the original code without breakpoints.

Another existing feature that will be leveraged in our examples is *kernel structure traversal*. There is already a framework provided by DRAKVUF which bridges the semantic gap between the malware monitor and the guest kernel. The offsets are calculated using pre-computed Rekall/Volatility kernel profiles.

MEMORY DUMPS

It is known that malware authors often leverage many different packers in various configurations in order to protect malware samples against static analysis techniques. In order to counter this, one can launch the malware sample inside a sandboxed environment and dump the unpacked malware core directly from memory.

Memory dumping routine

First of all, some core mechanism is required in order to record a specified region of memory. Let's assume we want to design function `__dump_memory(pid, ptr, size, reason)`, which saves the logical region pointed out by virtual address 'ptr' from process identified by 'pid' to a file.

For that purpose, we might leverage the standard `vmi_read` function offered by LibVMI. However, this is a general-purpose function which involves memory copying through intermediate buffers. Because of that, we have decided to implement a special call which will allow mapping of foreign DomU memory directly on the host side. This can be accomplished using the new `vmi_mmap_guest` call, which accepts an arbitrary guest physical/virtual address and returns a pointer which is valid on Dom0 side and allows us to directly read this memory.

[1] Before July 2019. The status is described for commit `6ef602c`.

Hooking memory-related functions

For these considerations, let's assume that we are able to build on top of existing DRAKVUF features, i.e. it is possible to stealthily hook an arbitrary system call and script the behaviour around it. Given such possibility, the following heuristic may be proposed:

```
// this hook is invoked before first instruction of NtFreeVirtualMemory is executed
static event_response_t free_virtual_memory_hook_cb(drakvuf_t drakvuf, drakvuf_trap_info_t* info)
{
    // HANDLE ProcessHandle
    uint64_t process_handle = drakvuf_get_function_argument(drakvuf, info, 1);
    // OUT PVOID *BaseAddress
    addr_t mem_base_address_ptr = drakvuf_get_function_argument(drakvuf, info, 2);
    addr_t mem_base_address = __dereference_ptr(mem_base_address_ptr);

    if (process_handle == ~0ULL) {
         char buf[2];
         __read_vm_memory(mem_base, buf, 2);
         if (buf[0] == 'M' && buf[1] == 'Z') {
             __dump_memory(mem_base_address, "possible binary detected");
        }
    }
}
```

Note: functions starting with '__' are a pseudocode used for simplicity.

In case this check succeeds, there is a probability that the buffer being freed is holding the unpacked portable executable (PE) and this could be considered a potentially useful memory dump.

An example – dumping Ramnit from a packed sample

Detonating a malware sample prone to such heuristics[2] inside an environment monitored by DRAKVUF, one may retrieve a result like this:

```
{
  "Plugin": "memdump",
  "TimeStamp": "1592183338.948034",
  "ProcessName": "\\Device\\HarddiskVolume2\\Users\\Admin\\Desktop\\MALWARE.EXE",
  "UserName": "SessionID",
  "UserId": 1,
  "PID": 1260,
  "PPID": 1116,
  "Method": "NtFreeVirtualMemory",
  "DumpReason": "Possible binary detected",
  "DumpPID": 1260,
  "DumpAddr": "0x280000",
  "DumpSize": "0x17000",
  "DumpFilename": "280000_7918dc6ce8c8598b"
}
```

Let's test it:

```
$ upx -d 280000_7918dc6ce8c8598b
$ yara ramnit.yar 280000_7918dc6ce8c8598b
ramnit_general 280000_7918dc6ce8c8598b
ramnit_dll 280000_7918dc6ce8c8598b
ramnit_injector 280000_7918dc6ce8c8598b
```

The memory dump was successfully identified as Ramnit core, using the rules provided in CERT.PL's analysis blog post [5, 6].

[2] 084cf8aeb40dda3bf733eb263ea227f8a838ece199b48460910e9ff2a28d8f96

VAD inspection

We have a single pointer that belongs to some interesting memory region (e.g. a pointer passed to `NtFreeVirtualMemory`), but we still don't know the exact boundaries of this region, i.e. the starting and ending virtual address. Given a single pointer which exists in the context of some process, one might find the corresponding Virtual Address Descriptor (VAD) by traversing the descriptor tree provided in the `_EPROCESS->VadRoot` kernel structure.

Hooking other functions

Another potentially interesting moment at which an interesting memory dump might be created is a process termination request sent with the `NtTerminateProcess` system call. Contrary to the previous case, we don't have any pointer to start with, so it is not clear what code region should be dumped in this case.

In order to find out where the termination request originated from, we have to reconstruct the call stack. This can be achieved by reading the user-mode part of the stack. Note that in kernel mode, the user stack can be accessed in the following way:

64 bit: `_KTHREAD->TrapFrame->Rsp`

32 bit: `(WOW_CONTEXT*)(_KTHREAD->Teb->TlsSlots[1] + 4)->Esp/Ebp`

The proposed reconstruction method is to perform a linear scan over the stack contents, trying to pick candidate pointers. Although this method is prone to various errors, it is compatible with both x86 and x64 modes of operation and is, in fact, the simplest method in terms of implementation.

```
for (int i = 0; i < 500; i++) {
   addr_t ptr = *(rsp+i);
   if (__has_mmvad(ptr) && !__is_dll(ptr) && __is_executable_page(ptr))
     __add_stack_entry(ptr);
}
```

This procedure could be used to find candidate pointers belonging to regions within the malware code.

An example – dumping Emotet from a packed sample

Suppose that we detonate a sample[3] inside an environment monitored in DRAKVUF. After the malware process executes successfully, one may get following result:

```
{
   "Plugin": "memdump",
   "TimeStamp": "1592183343.868269",
   "ProcessName": "\\Device\\HarddiskVolume2\\Users\\Admin\\Desktop\\MALWARE.EXE",
   "UserName": "SessionID",
   "UserId": 1,
   "PID": 2404,
   "PPID": 2296,
   "Method": "NtTerminateProcess",
   "DumpReason": "Stack heuristic",
   "DumpPID": 2404,
   "DumpAddr": "0x400000",
   "DumpSize": "0x1a000",
   "DumpFilename": "400000_9e31a34c94d6eb89"
}
```

Let's test it:

```
$ yara emotet.yar 400000_9e31a34c94d6eb89
Emotet 400000_9e31a34c94d6eb89
```

This buffer was successfully identified as Emotet payload using a YARA rule provided by kevoreilly [7].

WINAPI CALL TRACING

Educated guesses from system calls

Considering the existing DRAKVUF features related to system call tracing, it is already possible to implicitly deduce execution of some particular user-mode calls. The first group of such syscalls are ones that have one-to-one correspondence

[3] 09160f0aae57d08465220b38564145642c38e99ba27174356eae3229922ed187

for particular user-mode calls. For instance, WinAPI's `VirtualFree` function is simply a wrapper over the `NtFreeVirtualMemory` system call, so in a case where such a system call is observed, one might assume that there is a high probability that this call originated from `VirtualFree`.

For the WinAPI functions that don't have a direct mapping to the corresponding system calls, some calls can still easily be deduced if their side effects involve some characteristic registry access patterns. An example could be a `WSAStartup` API call which issues an `NtQueryValueKey` for `WinSock_Registry_Version`. Observing such registry access at kernel level means – with high probability – that `WSAStartup` has been called.

Contrary to that, by definition there are still lots of user-mode functions that render system call patterns unpredictable or that don't issue any system calls at all, so it's impossible to make an 'educated guess' in this case. Moreover, this set also contains many functions related to the Crypto API and the observation of these is critical for malware analysis, especially when the malware uses them to generate keys for file encryption (ransomware) or to communicate with C&C servers.

Actual WinAPI tracing

As our need was to provide clear and user-readable logs of execution for analytics, we have decided to implement a complete user-mode hooking framework.

A high-level method of tracking user-mode functions may be described as follows:

1. Set a breakpoint on the function loading a DLL to the newly created process.
2. After the DLL is placed in memory, find the addresses of the API methods we want to observe and place hooks on them.
3. In a callback function used for API hooking, read a value from the top of the stack (function return address) and place a hook on it – it will create a so-called return hook, allowing us to find the result of the function (both the return value and the buffers it modified, if any).
4. Place a hook on the process termination to avoid hanging hooks in random places of memory.

Setting the hook on the process startup

It's not obvious how to find the right syscall that is being called during DLL loading. After analysing logs from a couple of executions it turned out that the closest match are the `NtMapViewOfSection` and `NtProtectVirtualMemory` syscalls that are being called while loading 32- and 64-bit DLLs respectively.

The first problem we face here is lazy loading of DLLs by *Windows*. If the library hasn't been loaded in the system before, the physical pages for it are not allocated until first use, making it impossible to create a hook (no method in physical memory at all) or even to find the method's virtual address (no export table in physical memory). In order to solve the issue, we need somehow to trigger the page fault manually, which will make physical pages populate. There are several ways to do this, and we'll present all of the approaches we tried.

Approach 1

In our first approach we didn't trigger the page fault ourselves. Instead, we waited for it to be triggered 'naturally' (it always happens before the first call to a hooked function) and placed a hook right after it. In order to do that we used to trace writes to the process's page table – set a breakpoint on write to it, and checked if the pages containing an interesting virtual address had just been loaded into physical memory. This approach had no stability issues, but the code was complicated due to the four-level page table in *Windows*. Another problem with this solution is that it caused a massive loss of performance because the hypervisor needed to stop the VM and run its callback on every page fault happening in the system.

Approach 2

The next approach was to inject simple code that will trigger the page fault for us:

1. Save several bytes of program code starting from the current RIP and current state of registers to temporal variables.
2. Write a simple piece of code causing a page fault, e.g. `mov eax, DWORD [vaddr]` to the address pointed to by the current RIP.
3. Set a breakpoint on the next instruction.
4. Continue execution – it will only execute our injected instruction, causing a page fault and stop right after it.
5. In callback of breakpoint from point 3, restore the state of the machine saved in point 1 and continue execution.

That method was much better in terms of performance, but turned out to be extremely unstable, probably due to problems with multithreading, and often ended up with a BSoD.

Moreover, it's highly intrusive and might be used by attackers to detect the presence of the monitor, e.g. by trying to detect changes in the DLL's code from a different thread.

Approach 3

During our research the function for injecting page faults was added to LibVMI with commit `34ec2e5`, providing a method for fast, stable, clear and transparent population of physical memory on the hypervisor's level. That's the approach we successfully used in production.

The whole flow of making a hook using this techniques presents as follows:

1. Set a breakpoint on the `NtMapViewOfSection` and `NtProtectVirtualMemory` syscalls.
2. Get the DLL's name from MMVAD and compare it with a list of functions we want to observe.
3. If it matches, find the virtual address of its export table.
4. If the export table hasn't been loaded to physical memory yet, trigger a page fault on it.
5. Get the virtual address of the API method we want to hook from the export table.
6. If it's unavailable, again trigger a page fault.

Now the method is loaded into the physical memory and we can set a hook on it.

Trap – wrong virtual address

Let's imagine that the malware's author intentionally modifies a system DLL and puts there an incorrect virtual address that is not present in the page table. In normal conditions a read from the address like this would crash the application with a segmentation fault. However, when triggered manually from the hypervisor, such an injected page fault would crash the whole VM with BSoD. In order to prevent this, we make a hook on the exception handler in the *Windows* kernel – `KiSystemServiceHandler` – and silently ignore it if it was caused by our manually injected page fault.

```
static event_response_t system_service_handler_hook_cb(drakvuf_t drakvuf, drakvuf_trap_info_t*
info) {
  bool our_fault = /* ... */; // check if the exception was caused by injected
    // page fault
  if (!our_fault)
      return VMI_EVENT_RESPONSE_NONE;

  // emulate `ret` instruction
  addr_t saved_rip;
  __read_vm_memory(info->regs->rsp, &saved_rip, sizeof(addr_t));

  constexpr int EXCEPTION_CONTINUE_EXECUTION = 0;
  info->regs->rip = saved_rip;
  info->regs->rsp += sizeof(addr_t);
  info->regs->rax = EXCEPTION_CONTINUE_EXECUTION;
  return VMI_EVENT_RESPONSE_SET_REGISTERS;
}
```

Copy on write

Another problem occurs when the malware tries to modify the code of hooked functions. This might happen, for example, to avoid being run in a debugger or in traditional user-mode monitor. The malware may then try to take down the write protection using `VirtualProtect` and write its own code there in order to overwrite the `int3 (0xcc)` instruction. There are several more reasons why the application might be modifying its DLLs and there are known examples even of non-malicious software working like that – for example, *Internet Explorer* modifies a couple of API functions with its own versions of them for compatibility reasons[4].

However, dynamic libraries in *Windows* are shared among processes – if two applications are using the same system DLL, they share the same physical page. Those pages have their protect options set to 'copy on write'. That means that on the first write on that DLL a new physical page is allocated, the content of the old one is copied there, and the entry in the page table is changed to the new one, which now belongs only to the process that requested the write and is safe to be modified.

Unfortunately, during this process we lose our breakpoint, which is tied to physical address. In order to preserve the breakpoint during the CoW (copy on write) process we decided to track all the copies with a hook to the `MiCopyOnWrite` syscall and re-insert hooks on newly allocated pages.

[4] This feature is called Internet Explorer Compatibility Shims and is not well documented.

Example WinAPI trace

```
#include <iostream>
#include <windows.h>

int main()
{
  FILE* fp;
  uint8_t* ptr = (uint8_t*)GetFileType;

  if (*ptr != 0xE9 && *ptr != 0xCC)
    fp = fopen("ok.txt", "w");
  else
    fp = fopen("detected.txt", "w");

  if (fp)
    fclose(fp);

  return 0;
}
```

An example is the Visual C++ 2019 program which allows to differentiate *Cuckoo Sandbox* (creates a file `detected.txt`) and DRAKVUF (creates `ok.txt`). Note that in the case of DRAKVUF there is a stealth breakpoint on `GetFileType` and it cannot be detected this way.

FURTHER CHALLENGES

Portability

Currently DRAKVUF strongly relies on *Xen*'s altp2m feature, which under the hood uses the Intel Extended Page Tables feature. Thus, DRAKVUF can run if, and only if, the underlying machine is running a modern *Intel* family processor. Moreover, we are not aware of any virtualization technologies which would be able to efficiently emulate EPT on non-EPT capable hardware.

Leveraging additional hardware features

Due to the fact that the described malware monitoring system is working on a hypervisor level, there is still a possibility to leverage many specific processor features like Last Branch Record or Intel Processor Trace. This could provide additional layers of stealthy monitoring.

Better support for HLL

There is a rising popularity of malware written in languages which provide a higher level of abstraction than C/C++, in particular .NET technology can be highlighted as an example. Proper behavioural analysis and memory dumping of these would require additional dedicated logic in DRAKVUF.

Popularization

We put efforts into increasing the availability of DRAKVUF for users who do not have experience with virtual machine introspection technology. In order to accomplish such a goal, CERT Polska has provided a user-friendly wrapper around the monitoring engine called *DRAKVUF Sandbox*, which is available for free on *GitHub* [8].

SUMMARY

Within this paper, we have described our contributions to the DRAKVUF project related to WinAPI call tracing and memory dumping. Particular sections have described plenty of unique problems that can be encountered with the development of a hypervisor-level malware monitor.

As it was previously mentioned, all changes described within this paper are already upstreamed to the DRAKVUF [9] project. We do also encourage readers to try out the *DRAKVUF Sandbox* project [8], which additionally provides a user-friendly installer and graphical interface.

ACKNOWLEDGEMENTS

Kudos to the following:

- Tamas K Lengyel (maintainer of DRAKVUF, LibVMI and Xen) – for an enormous amount of help with design, development and bug hunting.

- chivay, BonusPlay (CERT.PL developers, co-authors of *DRAKVUF Sandbox*) – for programming advice and collaboration on the related projects.
- psrok1, nazywam, msm (CERT.PL malware researchers) – for malware research advice and collaboration on the related projects.
- Maciej 'mak' Kotowicz (a former CERT.PL member).

This research was co-financed by Action 2018-PL-IA-0168, as part of the European Union's Connecting Europe Facility programme.

REFERENCES

[1] https://github.com/search?q=apimon+is%3Apr+is%3Amerged+repo%3Alibvmi%2Flibvmi+repo%3Atklengyel%2Fdrakvuf+author%3Aicedevml+author%3Asasza8+author%3Achivay+author%3ABonusPlay&type=Issues.

[2] https://github.com/search?q=memdump+is%3Apr+is%3Amerged+repo%3Alibvmi%2Flibvmi+repo%3Atklengyel%2Fdrakvuf+author%3Aicedevml+author%3Asasza8+author%3Achivay+author%3ABonusPlay&type=Issues.

[3] Improving the stealthiness of virtual machine introspection on Xen. Xen Project. June 2018. https://xenproject.org/2018/06/21/improving-the-stealthiness-of-virtual-machine-introspection-on-xen/.

[4] Lengyel, T. K. Stealthy monitoring with Xen altp2m. Xen Project. April 2016. https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/.

[5] Praszmo, M. Ramnit – in-depth analysis. CERT.PL. September 2017. https://www.cert.pl/en/news/single/ramnit-in-depth-analysis/.

[6] https://github.com/mikesxrs/Open-Source-YARA-rules/blob/1953f96ae0b1b8431f0b14e04aef34b9e4494ae4/PL%20CERT/ramnit.yar.

[7] https://github.com/ctxis/CAPE/blob/bb60ac1278e3cbc11ca669b7a926e59513d06544/data/yara/CAPE/Emotet.yar.

[8] https://github.com/CERT-Polska/drakvuf-sandbox.

[9] https://github.com/tklengyel/drakvuf.