# “CoWSpot”: real-time module-stomping detection

## Recap

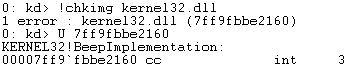
This is the final post in a three-part series about module stomping. While the first two parts were focused on understanding the attack and its limits, this final part will focus on detection – culminating in the promised real-time scanner.

As before, the code that accompanies this post (and the previous two in the series) is available in the Countercept GitHub account - <https://github.com/countercept/ModuleStomping>.

## Existing detection methods

The most obvious way to detect such an attack is to simply compare the contents of modules in memory with their original contents on disk. Any mismatch in the code sections indicates that the code has been modified after loading. However, this approach is (predictably) very slow, since it requires that a script reads all the loaded modules from disk. Additionally, if the hunter is working from a crash dump, the original modules may not be easily accessible without an image of the hard drive of the affected system.

A slightly better – though still not ideal – method is to acquire the original module from the vendor, via their published debug symbol server. One underutilized feature of the venerable WinDbg automatically does this for any module which has debug information available (for example, almost all of Windows) - the [ChkImg](https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-chkimg) (https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-chkimg) command. Running this command will use information from the PE header of loaded modules to download original files from the symbol server, and will then proceed to verify them. This is shown in the following screenshot, in which I have patched kernel32!Beep, inserting an INT 3 instruction (0xCC):



While this is a useful thing to know during hunts, it again suffers from performance issues; PDBs must be downloaded and processed in addition to being compared with data on the target system. It is also of no use against images for which no symbols are available, and finally, it is intended as a system-stability checking tool and not as a forensic utility, so its resilience to malicious memory is unknown – perhaps an attacker can modify the in-memory PE header of the target module such that symbol lookup will fail.

Wouldn’t it be nicer if there was a solution so fast that it could be used in real-time, and which wasn’t dependent on any user-mode data structures?

## Plans for a new detection method

Given this state of affairs, I set out to determine if Windows itself maintains information on which memory ranges have been modified after load. If this information is available, it would enable us to very quickly locate cases of module stomping, and also lend itself to detecting other attacks which involve patches to memory.

Those familiar with virtual memory and modern operating system internals will be aware that the OS usually ‘deduplicates’ modules in memory; if two processes both load the same module, only one copy will be kept in RAM, and changes made by one process will not be visible to the other. Surely Windows keeps track of memory write in order to provide this functionality – and maybe this information can be abused to expose the information we want! Time for us to dig into the kernel.

## Making it happen: Detecting modifications

A quick read of “Windows Internals” reveals that modules are indeed deduplicated in physical memory space. As is normal in x86, this is achieved using a part of the CPU known as the MMU, or “Memory Management Unit”. This is the component that allows us to separate memory ranges and enforce permissions – for example, to prevent user-mode applications from touching kernel-space memory, or from interfering with each other.

This deduplication is achieved by configuring the MMU to map an address range from each process into the same range in physical RAM. Applications all think they are accessing their own memory ranges – the address translation performed by the MMU is transparent to them.

Here’s a diagram to show what I mean:

Notepad.exe

Explorer.exe

Taskmgr.exe

Cmd.exe

MMU

Physical memory

Kernel32.dll

Here, we have four processes running, each of which have loaded kernel32.dll into their memory space. The OS has configured the MMU to map each of these address ranges into the same physical memory space, where a copy of kernel32.dll has been loaded from disk by the OS.

However, eagle-eyed readers may have already spotted a flaw in my explanation! Each application is, naturally, able to write to its own version of kernel32.dll during normal system operation – but not to modify the version of kernel32.dll which other processes see. How does the deduplication mechanism deal with this? The answer lies in a technique known as “copy-on write”.

Initially, the kernel maps modules as read-only when they are loaded, even though the userspace application has requested for the page to be read/write. This means that any write to these modules will generate an access violation.

Once a write occurs, the kernel will catch the associated access violation. It will proceed to suspend the user-space process, and copy the read-only module into a different memory area, which is dedicated to this process only. The kernel will then change the memory mapping such that any subsequent access by that process goes to this dedicated memory range, and the process is resumed. The process continues as normal, using the same memory ranges – since the mapping is transparent, the process has no idea what’s gone on underneath it. Here’s our diagram showing what happens after notepad.exe modifies kernel32.dll:

Notepad.exe

Explorer.exe

Taskmgr.exe

Cmd.exe

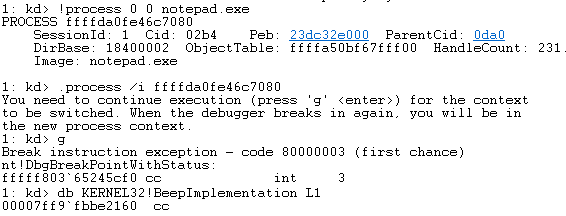
MMU

Physical memory

Kernel32.dll

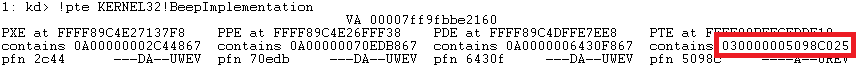
Kernel32.dll

We can verify this with WinDbg, by using the “!pte” command, which displays information about the MMU configuration for a given virtual address. We will be using a 64bit Windows 10 system, running notepad.exe, into which we have modified a single byte in kernel32.dll to simulate a module stomping attack. We have changed the byte at kernel32!BeepImplementation to 0xCC:



If you’re following along on you own system, note the use of the “/I” flag when switching process context. This is required to ensure that WinDbg uses the correct process context to decode physical addresses.

We can use the “!pte” command to get some information about the virtual address we are interested in:

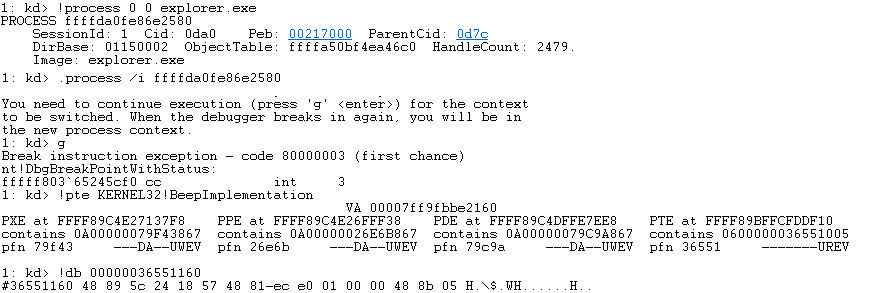


I’ve highlighted the data structure we’re interested in, known as the PTE or “page table entry”. The smallest amount of memory that the MMU can make is known as a ‘page’, and on most x86/amd64 systems (though not all), this is 0x1000 bytes. The PTE describes the page containing this VA, which (in this example) corresponds to the range 0x7ff9fbbe2000 to 0x7ff9fbbe2fff.

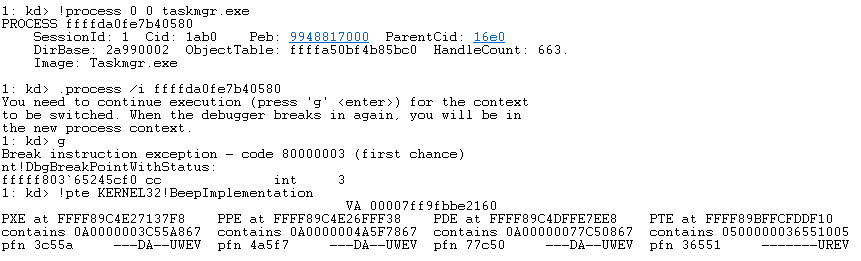
The PTE’s structure is documented in the Intel reference manuals. Reading this, we discover that if we mask off the first byte, and the last 24 bytes (ie, three hex digits, neatly enough) we get the physical address the page maps to. Then, we can add the offset into this page – 0x160 in our example – to get the physical address of the memory itself. We can use WinDbg’s “!db” command, which shows physical memory, to verify this:



Here we can see the 0xCC byte we injected for our example. If we repeat the procedure with another process, however, we will observe that it is mapped to a different physical address:



Notice the new PTE value – 0x0600000036551005 – which tells us that the physical address for this page is, indeed, different. Observing it reveals the unmodified code that explorer.exe is using. We can see the effect of the deduplication by switching to yet another process:



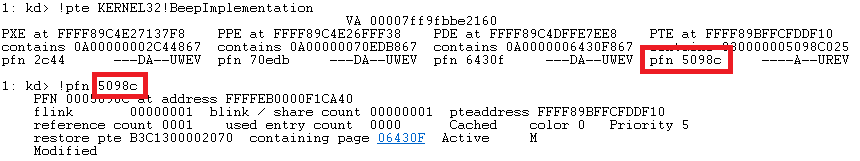
Note the identical PTE value! This reveals that both processes are using an identical in-memory version of the module.

This is the artefact we have been looking for! If we can detect which processes have their own private copy of a module in-memory, then we will know which have been modified since load.

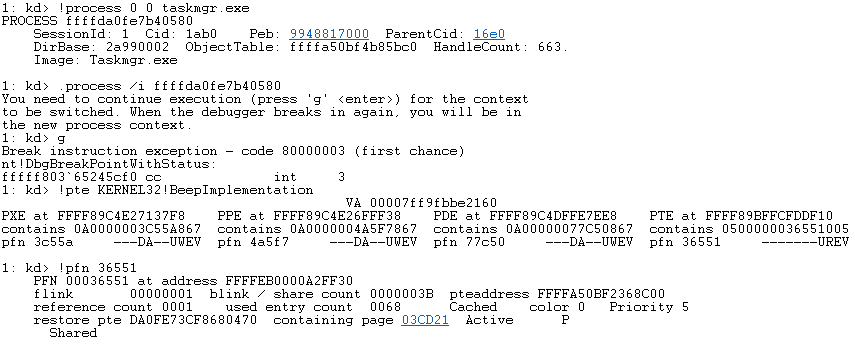
Fortunately for us, the Windows kernel keeps track of physical pages which have their own dedicated copies of memory. It does this by setting a flag in what is called the “PFN database”.

The “PFN database” is an internal, undocumented kernel data structure. It is simply a large array of \_MMPFN structures. Each of these, known as a “PFN” (“Page Frame Number”), describes a single page of physical memory, containing such details as its permissions and current status – for example, if that page is currently swapped out to disk in the pagefile, if it is in use, or if it is shared between processes. The PFN database is simply indexed by the physical address of the page of memory in question. If we wanted to query memory for the physical address 0x12345678, we would look at entry number 0x12345000 in the PFN database – remember that 0x12345678 references an offset of 0x678 into the memory page at 0x12345000.

As you may have noticed, the output of the “!pte” command presented us with a number marked “pfn”. This is the index into the PFN database, which we can observe with the “!pfn” command. Here’s what notepad.exe!BeepImplementation looks like. I’ve re-run the previous “!pte” command to illustrate:



If, for contrast, we observe the same range from the perspective of a different process, we see the following:



Interestingly, the memory which we have patched has lost its ‘shared’ status, and gained a ‘modified’ flag. Also, the ‘share count’ has been set to zero. This aligns with our expectations – because the memory is no longer the same as it is on disk, it cannot be shared with other processes. With this ability to query the PFN database, we can now very quickly determine if a page of memory has been modified or not.

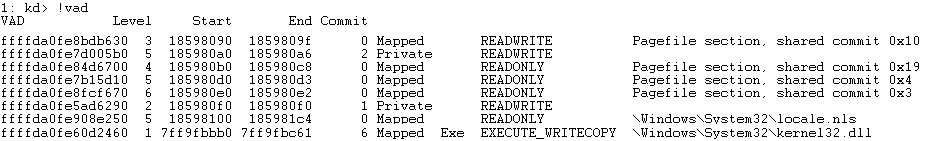
## Making it happen: Reliably finding what to scan

The last piece of the puzzle is to locate modules in memory in order to scan them. A simple way would be to enumerate modules via the usual \_PEB\_LDR\_DATA structure, and then parse the PE header and locate executable sections for scanning. However, this has some drawbacks. Firstly, the extra work of parsing the PE headers limits performance slightly. Secondly, since this memory may be paged out to the swapfile, it may not be present in the crashdump we examine. And, finally, the \_PEB\_LDR\_DATA is located in userspace and may be tampered with by a malicious user as we mentioned before. We’d much prefer to assume the entire process is hostile and cannot be trusted.

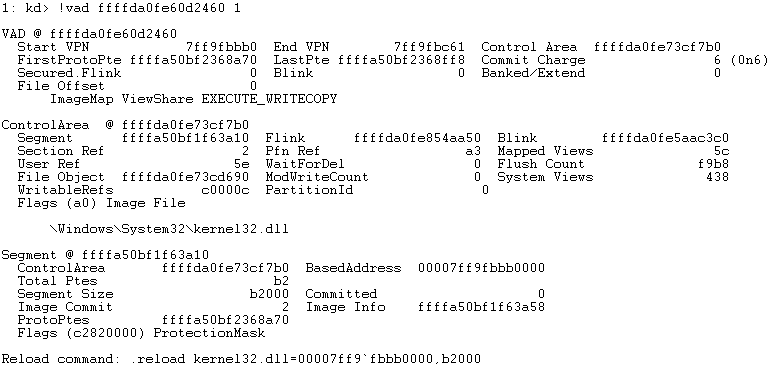
Fortunately, there’s a better way.

In addition to the PFN database, which describes physical memory, Windows also maintains other data structures to describe a process’s address space. One such structure is the “VAD tree”, which is a tree of VAD (or “Virtual Address Descriptor”) structures, each describing a contiguous range of virtual memory (in contrast to the PFN, which requires one entry for each page, a single VAD entry can represent an arbitrary range of virtual memory).

Each process has its own VAD tree, which describes every virtual address range accessible to it. A VAD structure stores a variety of information, including page permissions. Here is an example (truncated for brevity), showing some of notepad’s address space via WinDbg’s “!vad” command:

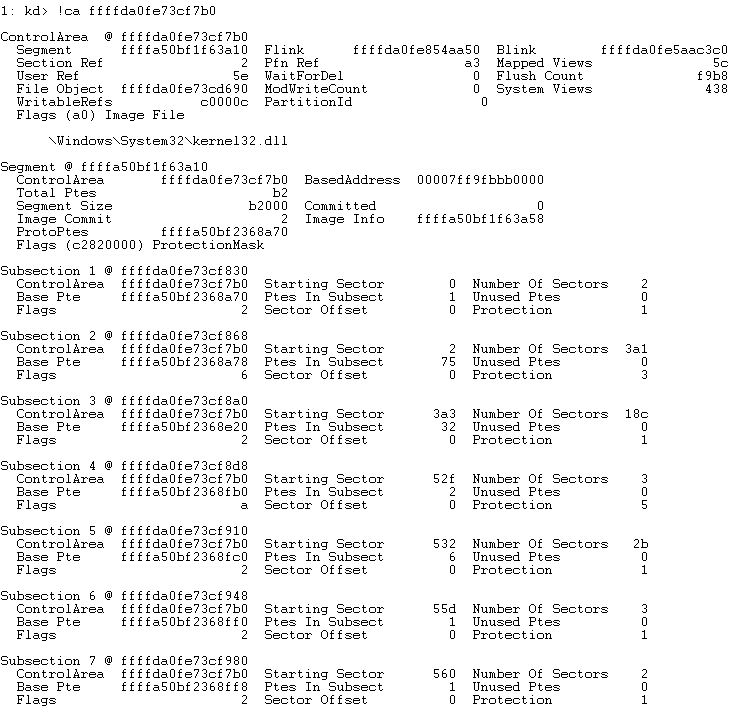


We can see that there’s a read-write section of memory mapped at memory location 0x18598090 to 0x1859809F, backed by the pagefile. Beneath it, we can see a private memory range, and the final two entries represent two mapped files. The first is a normal mapped file, most likely mapped with CreateFileMapping, and the final is an executable image – exactly what we’re looking for! Let’s take a closer look at that VAD entry, by passing the address of the VAD directly:



That’s a lot of information! We even get the filename that the module was loaded from, and we learn the base address and size of the image. However, we want to scan only the executable sections, and ideally we don’t want to process the PE header to locate them.

Fortunately, the “ControlArea” structure holds even more information than is presented here. If we invoke WinDbg’s “!ca” command, we can get information about the ‘subsections’ present in this section:

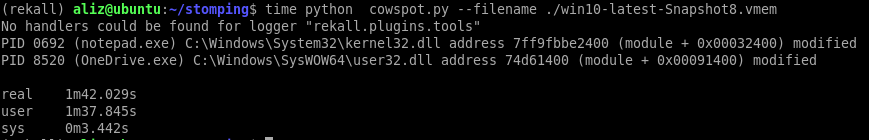


That’s exactly what we’re looking for! Each subsection represents a contiguous range of memory with the same permissions and flags. We can simply filter for any which are executable, and scan those via the PFN database!

Since we’ve got all the information we need, it’s a good idea to write a script to do our processing for us. I reached for the ever-faithful Volatility to do the heavy lifting work for me, but I quickly ran into one of Volatility’s shortcomings – it does not expose constants from the debug symbols, such as the address of the PFN database, which is vital here. I took a quick look for a tool that would expose this data and settled on [Rekall](http://www.rekall-forensic.com/). Rekall, for those unfamiliar, is a Google-backed fork of Volatility, which brings new approaches to some workflows.

Writing a Rekall script is subtly different from writing a Volatility plugin, but is still fairly straightforward. Instead of writing a ‘plugin’, we start Rekall from our Python script and do processing there. I did run into performance issues with the way that Rekall parses data from the target, but they were easy to work around by parsing the performance-critical \_MMPFN structure manually (at the cost of some portability to future windows versions). The included Rekall script provides options to locate executable regions to scan by parsing the PE, or from the VAD (the default). Note that PE parsing will fail to read many areas, since the memory is lazy-initialized from disk.

Here’s the output from the script. Note the false positive from the OneDrive process – the occasional false positive is to be expected.



## Live detection

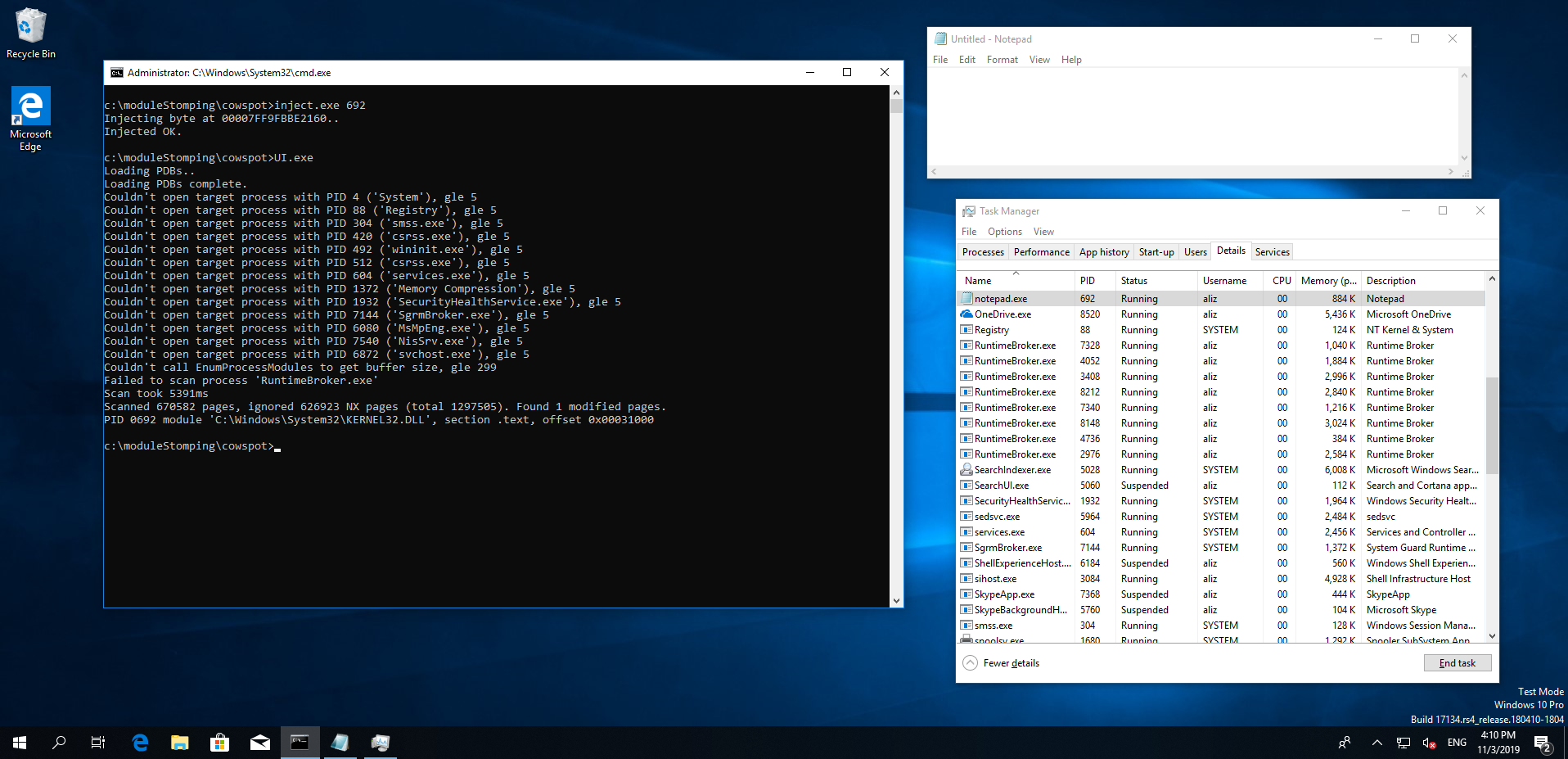
Finally, since our approach performs well enough, we would love to run this ‘live’ in the background of our systems. Obviously, doing this sort of thing is often very difficult on a live system, mostly because we depend on undocumented quirks of the kernel, which are vulnerable to change by Microsoft at any time and which can lead to our scanning application crashing. However, there is a neat way to do almost all of what we want to do via documented calls. The only undocumented thing we depend on is the structure and location of the PFN database itself. We can acquire these via the Windows debug symbols, in much the same way that WinDbg or Rekall would do.

To access the PFN database (and some other things) we require ring-0 access, so we fire up Visual Studio and make a new driver. We also make a user-space component with which to interact with the driver, and a simple test application which will patch kernel32 in a process of our choice, allowing us to test and exercise our solution.

The driver itself is straightforward. It requires that the user-space component tell it the location of the PFN database (more on this later!), after which it allows the status of pages of memory to be queried. To do this, we first allocate an MDL (via IoAllocateMdl), which is a data structure used by drivers to describe memory ranges. We ensure that this range of memory is resident and not swapped out (MmProbeAndLockPages), and then ask the kernel which PFN numbers the memory is described by (MmGetMdlPfnArray). We then use the MmCopyMemory function to read the PFN database, rather than reading it directly. This means that the use of a bad pointer (such as may happen if Microsoft alters the structure of the PFN database) won’t BSoD the machine. Finally, we parse the PFN structure and return the contents of the ‘Modified’ bit to userspace, where our scanner application can present the results to the user.

Unfortunately, this does require the address of the PFN database and knowledge of the PFN data structure, both which are undocumented and thus subject to the whims of Microsoft. In order to locate the table, however, we don’t need to resort to heuristics and fuzzy guesswork – as alluded to earlier, we can simply use the somewhat-underrated Microsoft DIA SDK to obtain debug symbols for the currently-running kernel, and pass this information to our driver.

The final result is an almost-production-ready solution to the problem of module stomping. Our threat hunters can get an alert almost as soon as the hiding technique is used, making it a dead giveaway. Our only wish is that we didn’t need to page in memory in order to scan it, as this can cause performance issues.

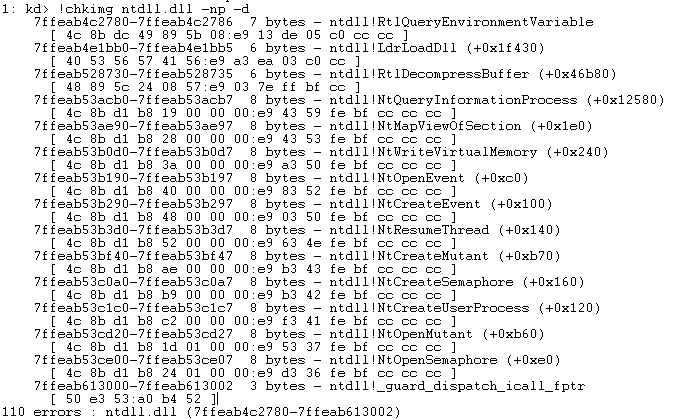


Note that the user-mode component has failed to open various system processes, due to Windows 10’s process protection. Evading this feature is a separate topic.

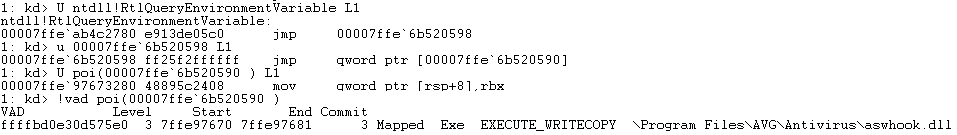
The whole scan took slightly over five seconds on this clean VM, hosted on my (somewhat over-loaded) laptop. In this time it scanned 670852 pages, representing 2.5GB of executable memory. It ignored about 2.4GB of non-executable virtual memory, and generated no false positives in this example.

## Effect of anti-virus

One complication is the tendency of anti-virus applications to insert hooks into system code, which are then detected by our searches. For example, after installing “AVG Free”, a scan found some 280 modified pages! WinDbg’s ChkImg has the same issue, and reveals what’s going on:



In the first result here, we see that the byte sequence at the start of the function ntdll!RtlQueryEnvironmentVariable is changed from 0x4c8bdx49895b08 to the nefarious-looking 0xe913de05c0cccc. Those who’ve stared at x86 long enough will recognise the 0xe9 opcode as a JMP (JMP rel32 to be precise) – some kind of hook. Indeed, if we look closer, we can see that it goes to aswhook.dll:



Unfortunately, our approach provides no real way of discarding such false positives. The only way to deal with them is to read the modified memory, find the new code, disassemble it, and whitelist based on the changes. While this may seem like a major problem, in practice it is not – the 280 modified pages that AVG generates, for example, amounts to around 1.1 megabytes of data, which is small enough to be practical to compare on the host CPU.

## Final words

I hope this series of articles has been informative! I’ve certainly found it an interesting area of research. We have shown how parts of Windows memory management operate, and created a workable tool to detect module stomping. While this tool is imperfect, it enables a previously unattainable level of analysis. Also convenient is the ability to operate on a live system without doing a lot of shady things – unfortunately we do require access to the PFN database, and so we can’t operate 100% within the abstraction of Windows, but we’re pretty close (and careful coding using MmCopyMemory makes the operation pretty safe). Hopefully the folks at Redmond aren’t too angry.

## Further reading

If anyone is looking for more information on the subject, I’d suggest reading the following:

* Rekall themselves have a blogpost (<http://blog.rekall-forensic.com/2016/05/rekall-and-windows-pfn-database.html>) in which they use a similar method to detect the modifications that a Zeus infection has made to the running OS.
* Also interesting are two blogposts: <https://rayanfam.com/topics/inside-windows-page-frame-number-part1/> and <https://rayanfam.com/topics/inside-windows-page-frame-number-part2/>, which discuss PFNs and memory management in Windows.
* For anything relating to Windows, the “Windows Internals” book is my go-to. Chapter 5, “Memory Management”, was a real help understanding how prototype PTEs are used by the system.
* To get a solid understanding of page translation, study parts of Intel’s Software Developer’s Manual (<https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4>). It’s a long, hard read, but contains some absolute gems. Start at section 3.1, “Memory Management and Paging”.