Defeating VMProtect’s Latest Tricks

cyber.wtf/2023/02/09/defeating-vmprotects-latest-tricks/

February 9, 2023

A colleague of mine recently came across a SystemBC sample that is protected with VMProtect 3.6 or higher. VMProtect is a commercial packer that comes with advanced anti-debugging and VM detection capabilities. It also employs code virtualization – a technique where normal machine code is translated into a proprietary bytecode language that is interpreted at runtime – which makes it very hard to determine the exact logic implemented by the code. ScyllaHide, our anti-anti-debug tool of choice, was not up to the task of hiding the debugger from the packer, so we dove into the unexpectedly deep rabbit hole of figuring out what is going on.

Kernel mode tooling such as TitanHide/HyperHide would probably have been up to the task of defeating most of the checks, but we prefer user mode tooling, since it is much less complicated to use and easier to debug.

Debugger checks

On the face of it, VMProtect’s debugger checks don’t use any exotic techniques. We have seen the following checks, all of which Scylla has long since had support for:

- PEB.BeingDebugged
- ProcessDebugPort
- ProcessDebugObjectHandle
- NtSetInformationThread ThreadHideFromDebugger
- CloseHandle with invalid handle value
- Non-zero debug registers in CONTEXT when catching exceptions

If you’d like to get a deeper understanding of these techniques, please refer to Check Point Research’s excellent Anti-Debug Tricks page.

The first problem we encountered is that when debugging a 32-bit executable on a 64-bit system, it is possible for the executable to hide code from the debugger by doing a far jump into the 64-bit segment 0x33 (also dubbed “Heaven’s Gate“). In particular, most of the anti-debug checks will be executed in 64-bit mode. x32dbg/x64dbg can only debug their respective bitness, but not both at once. The only debugger we know of that can achieve such a feat is WinDbg, but its interface is rather unpleasant to deal with. So we switched to a native 32-bit system for this sample.

The next problem is that VMProtect has built-in syscall tables for the most common Windows builds. If it finds that it runs on a known system version, it uses the `sysenter` instruction to directly call into kernel mode for its checks, bypassing any user mode hooks (there is `ProcessInstrumentationCallback`, but it has limits – the callback is triggered after a syscall has executed but before it returns, which means we cannot prevent changes such as setting the `ThreadHideFromDebugger` flag). The obvious approach to deal with this is to change the build number to a fake number everywhere, so that the packer has no choice but to call the APIs the regular way.

Enter the myriad ways of obtaining the Windows build number these days.

ScyllaHide already supports patching the OsBuildNumber field in the PEB (Process Environment Block), which is also what APIs such as `GetVersion`/`GetVersionEx` read:

```
if (flags & PEB_PATCH_OsBuildNumber)
{
    peb->OSBuildNumber++;
}
```

Doing a simple increment comes with a little caveat. What if you happen to be on a recent Windows 10 build such as 19043 or 19044? 19044/19045 are both valid builds. Oops.

In a ScyllaHide issue, Mattiwatti, who is one of the maintainers of ScyllaHide, already outlined the next technique that VMProtect uses on modern Windows versions. `KUSER_SHARED_DATA` is a read-only page that is mapped into every Windows process. It allows quick access to many commonly needed values, such as versions, current tick count, current time, and much more. APIs such as `GetTickCount` access this page in order to avoid a context switch to the kernel. Since Windows 10 the structure also contains the build number. This is unfortunate because ScyllaHide cannot modify this value. But it is possible to set an access hardware breakpoint and change the value in the register after VMProtect has read it. Alternatively, Windows 8.1 or older can be used.

If these ways have proven unfruitful for obtaining a known build number, VMProtect will suspect that we're up to no good and starts inspecting system library versions. It will parse NTDLL's version resource, which contains plenty occurrences of the build number. ScyllaHide patches one of them (the FileVersion string), which apparently was sufficient at some point in the past. Not anymore. Nowadays, VMProtect inspects *all* four build numbers (two in binary form, two in strings). So we adjusted ScyllaHide to set all of them to a fake version.

Memory breakpoints on other libraries' resource sections have not been hit, so that should suffice for fooling the packer, right? Right?! Nope. You can imagine that at this point we were quite confused how it *still* managed to find the correct version.

I NEED THE BUILD NUMBER
YOU WOULDN'T MAP DLLS FROM DISK, RIGHT?
YOU WOULDN'T MAP DLLS FROM DISK.

After some more tracing, we found the call sequence `NtQueryVirtualMemory` (to get NTDLL's full path on disk) → `NtOpenFile` → `NtCreateSection` → `NtMapViewOfSection`. This maps a fresh copy of the NTDLL image into memory. Oh, well. They call APIs, we hook APIs. One somewhat mean detail is that `NtCreateSection` is called with the flag `SEC_IMAGE_NO_EXECUTE`. This prevents image load notify routines and debugger events from being raised when the image is loaded, however the flag is only supported since Windows 8. As a result, anything packed with this VMProtect version will not run on Windows beta builds from before Windows 8, and incidentally this also comes to bite us when faking the version on a Windows 7 system – VMProtect knows the usual build numbers of Windows 7 (7600/7601) and it would ordinarily never take this code path. Since we're hooking anyway, we change this value to `SEC_IMAGE` when detecting an older OS and everyone is happy.

If you thought we're finally rid of the direct syscalls now, think again. VMProtect has one final trick up its sleeve: it tries to extract syscall numbers from the library code. We expected this all along, but it makes sense that it only happens on the fresh mapping from disk. This way, the packer can avoid any hooks and other code patches placed on the regular NTDLL image in memory. That is, until we come in and deliberately destroy some API entrypoints in the mapping (smile). The packer expects the first instruction to be `mov eax, CallNumber`, and if it cannot find that, it finally gives up and calls the regular NTDLL API export.

VM Checks

Overview of VM checks:

- cpuid hypervisor bit & hypervisor vendor
- Trap Flag tricks in combination with forced VM exit via rdtsc/cpuid

- NtQuerySystemInformation with SystemFirmwareTableInformation, TableIDs FIRM and RSMB
- (presence of `sbiedll.dll` in process, for Sandboxie detection)

For further reading about these checks, please refer to Check Point Research's Evasion techniques page (particularly, the "CPU" and "Firmware tables" sections).

The first is relatively easy to mitigate by disabling paravirtualization, which will remove any hypervisor information from cpuid.

The second trick is somewhat mean and took us a while to figure out. Consider the following code block:

```
<prepare flags value with TF bit (0x100) on stack>
popfd    ; apply flag change
cpuid    ; force VM exit
nop      ; filler for EIP check
push ebx ; next regular instruction
```

The Trap Flag provides single stepping functionality for debuggers. If you set it, the processor will raise an interrupt *after* executing the following instruction. So we expect the instruction pointer in the exception that the OS gives us to be at the `nop`. As it turns out, older VirtualBox versions will rat you out, because they have a bug that causes EIP to be at the `push` instead. This is fixed in version 7.0.4, which was pretty recent at the time of writing.

Finally, VMProtect will inspect some firmware bits. The RSMB provider is used to obtain raw SMBIOS values such as BIOS vendor, BIOS version, system family, system UUID, etc. VirtualBox also has custom OEM fields for "VBoxRev" and "VBoxVer". It is possible to change all of these through VM configuration changes (`VBoxManage setextradata`). The FIRM provider is a different story. It allows reading 128K at physical addresses 0xC0000 and 0xE0000, respectively. These ranges contain BIOS Option ROM code, which may contain some strings that give away that virtualization is in use. This cannot be helped without modifying and recompiling the VM software.

What we can do is hook `NtQuerySystemInformation` and return empty data. Since ScyllaHide hooks that API anyway, we simply integrated a code path for the `SystemFirmwareTableInformation` class.

Conclusion

With the aforementioned counter-measures (or rather, counter-measures for the counter-measures) in place, we can finally debug and unpack the sample. Fun fact: The sample is packed twice, so after VMProtect has finished initialization, another layer of packer code runs and unpacks the final SystemBC malware. This is not very smart, because this nullifies the effects of VMProtect’s import protection, making it trivial to obtain a fully functional dump. Malware packers often lack the sophistication of commercial grade packers.

We’ve committed all ScyllaHide code changes to GitHub and are currently waiting for them to be accepted upstream.

IoCs

`e21f50a1794acd0a585c86a157e8f70b044adcc860d6d0648d874deccd7ba653` (SystemBC sample)