Subverting Windows Debugging Mechanism to Hide Malware Programs

This document is written by Sina Karvandi, Sima Araasteh and Shahriar Eftekhari. We get the prize of (Worthy of appreciation) article in 8th festival of Cybersecurity and Defense Competition in the Sharif University of Technology. (January, 2018)

Abstract

Debugging is an essential aspect of all modern Operating Systems, which makes finding kernel bugs and troubleshooting system problems easier.
Windows implements a robust easy to use debugging interface with public symbols to enable driver programmers or security researchers to explore kernel in low-level parts of Windows.
In this paper, we will use Windows Debugging mechanism to hide known malware from different Anti Viruses or even execute code in Kernel-Mode despite modern protections like Driver Signature Enforcement or Kernel Patch Protection (PatchGuard).

In this paper we will introduce a method to circumvent Windows protections by using Debugging Mechanism to insert a Kernel-Mode rootkit or even control Windows kernel using a User-Mode code and finally show some ways in which it is made impossible for AV's to detect modification and lastly some suggestion which can help us avoid these kinds of attacks.

Windows Debugging Environment

With local debugging, you can examine the state of Windows, but not break into Kernel-Mode processes that would cause the OS to stop running or in the other words you can't put a breakpoint in local kernel debugging.

All versions of Windows have a tool called **bcdedit** which can enable debugging environment so you can pass **/debug on** to this tool and after a reboot, Windows makes everything ready for you and you're all set.

At this time you'll need other command-line tools which interact with Windows in order to make it possible to debug Windows easily.

There are several options in this step but in this paper, we use **kd** which comes from Windows SDK with other tools like **Windbg**.

Now we have full access to this kernel memory and this is because Windbg internally has a Signed Driver with lots of capacities which makes it easy for us to change kernel internal memory, by the way, we bypassed Windows Driver Signature Enforcement which is a sophisticated protection that was extremely effective in preventing kernel-mode rootkits from infecting windows kernel.

Another protection that was bypassed in this case is PatchGuard.

PatchGuard is a powerful protection which makes it hard for Third Party developers or Kernel-Mode rootkits to change windows kernel structures and CPU tables like IDT, GDT, SSDT and etc.

By the way, Microsoft says, PatchGuard will never start when Windows starts Debugging so we can easily change Kernel Structures without worrying about auditions by PatchGuard.

By now we have a full access to kernel memory without any protection so we need to analyze Windows in order to inject User-Mode code into other processes in order to run and patch another process memory in a hidden way or make access into kernel from malware's User-Mode code or we can make changes in kernel in order to have permanent access into victim's Windows kernel or change Windows kernel structures attributes and cause attacker specific behavior and finally run a Kernel-Mode rootkit and hide it among system modules.

Inject User-Mode code into another process

Injecting User-Mode code in another process have always been a good way to hide malware from Anti-Virus heuristic detection, It is possible to inject your assembly module into another running process.
In User-Mode you are limited to your virtual address so you can't have access to another process memory and you should use Windows APIs in order to access another

process’ virtual address.

There are several ways of injection using Windows API.
i) Using SetWindowsHookEx and LoadLibrary will cause an injection into the target process.
ii) Opening a process via OpenProcess and using VirtualAllocEx cause your shell code or module DLL to be written into a remote process then you can use WriteProcess-Memory to write your stuff and finally invoke a call to CreateRemoteThread that will cause to create a remote thread which will run in the context of the target process.
iii) You can also use CreateRemoteThread with LoadLibrary in order to achieve the same result.

There are also other ways of using a combination of above functions other functions (e.g process hollowing). and they will all have the same effect.

But the problem caused by this approach is, these kinds of functions are heavily monitored by AVs, so if a malware uses this kind of functions, AVs probably prevent it from running or at least it would have a very bad effect in malware’s heuristic analytics and most AVs will send this malware for further analyzes to their servers and if malware continues injecting or doing other suspicious actions, they will block the malware.

By using the debugging methods, you will have access to kernel physical and virtual memory, you can simply change machine memory from the kernel and have complete access to the process’ virtual memory.
Let me explain another way of changing memory and controling program’s flow without creating a new thread or invoking any Windows native API.

Windows (and almost all modern Operation Systems) use a combination of fs and gs registers (in Intel 80x86 or AMD64) in order to translate current EIP or RIP into physical addresses which finally causes read or modify of RAM with the desired value.

As a matter of fact, Windows is forced to save context switching or exception handling trap routines information into kernel structure and they're all present in _KThread.

In the above example if you create a User-Mode debugger and a kernel debugger then debug your program with the User-Mode debugger and then pause the kernel debugger. It causes you to be able to see the trapped registers in the ktrap_frame.
It is clear that you change these values even if this process is theoretically hard because CPU switches its context really fast and we could lose the new state to recover the program again, but by dealing with IRQL or temporarily disabling the execution of a thread by changing its kernel attributes directly, you could finally modify the physical address of where the current EIP or RIP points to.
As a result we completely compromise our target from kernel without raising any flags. The result of testing this method in context with some Anti-Viruses is worth studying! We examined over 5 top AVs and none of them got any suspicion to our malware sample.

Patch Another Process Memory

Another way in which you can gain access from another process' memory without using Windows Native API is changing another process memory and as you can imagine windbg has tons of commands in which you can easily change address physically or virtually.

The problem of changing in live debugging is that you can't set User-Mode addresses directly because you do not know how to translate your virtual address and make sure that you patch the right place.
Every process in Windows (or all modern Operating Systems) has its own register which helps CPU to calculate current registers (Specially Eip or Rip) to physical addresses as I mentioned previously but the problem is CPU changes fs and gs in context switching very quickly and you can't be sure about which process' virtual address you are currently editing so it might cause a wrong change which subsequently brings about catastrophic problems in the currently running process (The process that you patched by mistake. Not the process you wanted to patch originally.)

But Intel documents [ref] demonstrates how virtual address are translated to physical address and Intel's variant of converting Page Table Entry (PTE) addresses utilizing fs or gs depending on your system structure. There are also some differences between

converting addresses between x64 and x86 structure but the problem is that the attacker should analyze the target PE in advance and know about where he/she desires to edit which is almost possible in most of the cases.

In this scenario attacker should read the PE's PEB in order to gain information about module addresses which are compiled based on ASLR. In the end attackers will have a previously analyzed program and module addresses so they know where the program flow comes frequently and they can inject or patch the target address with their supplied assemblies physically or virtually and finally it causes to run the code in the context of that process, every time the program's flow passes from that location.

We have built this model and tested this scenario with a known shellcode (in our case a shikata_ga_nai encoded reverse shell injected it into a process memory).
Even if shikata_ga_nai is hard to detect theoretically but nowadays almost all AVs detect variants of this shell and block them but we see none of them detects this shellcode because AV doesn't expect to see previously loaded modules,
modified silently.

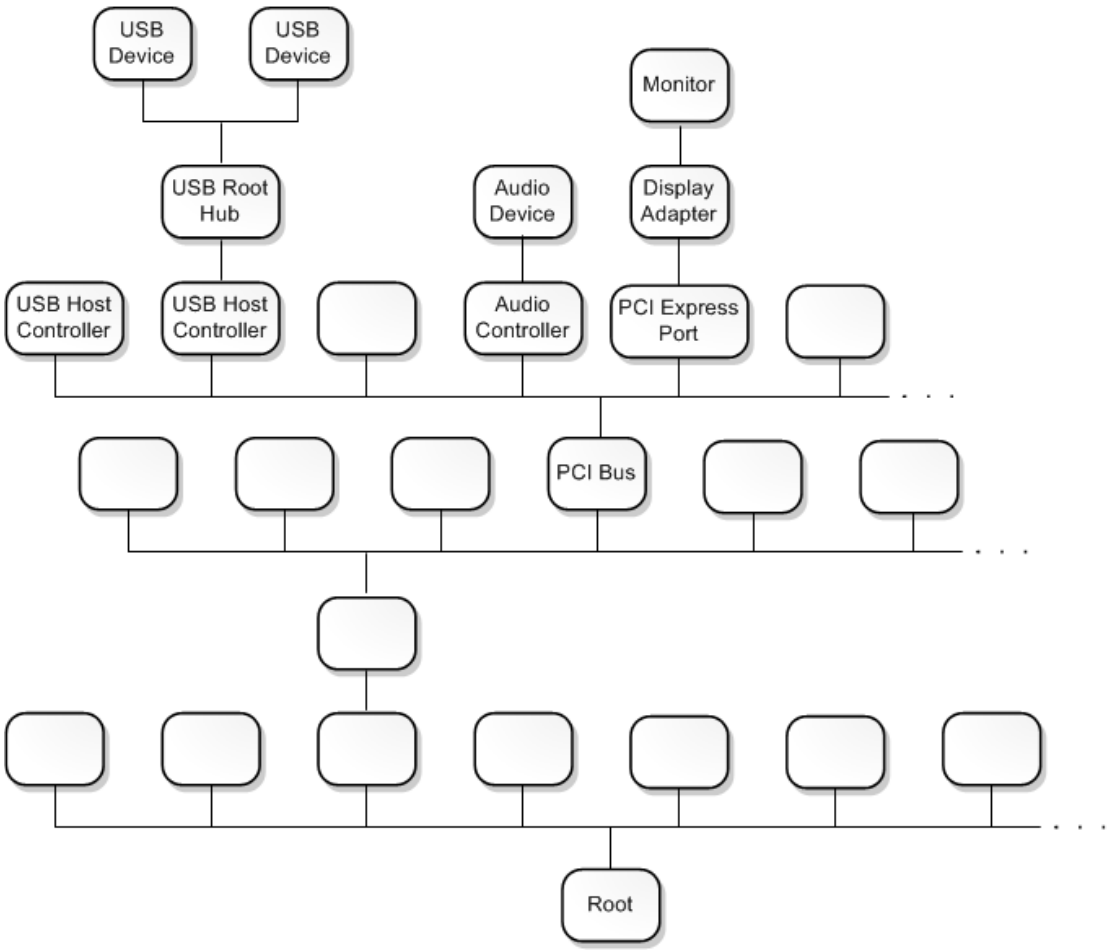
Create access into kernel in malware's user-mode code

In the previous section, I explained the impact of this attacks in User-Mode executable but the threat is definitely more than that. In this kind of attacks, we can change the Kernel-Mode structures and it gives us the power of changing almost everything that Windows Kernel is able to do on our computer.

The access to kernel in malware's User-Mode is definitely one of the most dangerous aspects of this kind of attack and the context of doing innovative patching is almost unlimited so this type of attacks are really hard to detect and in the writers opinion it is effectively impossible in the case the attacker makes some kind of obfuscation.

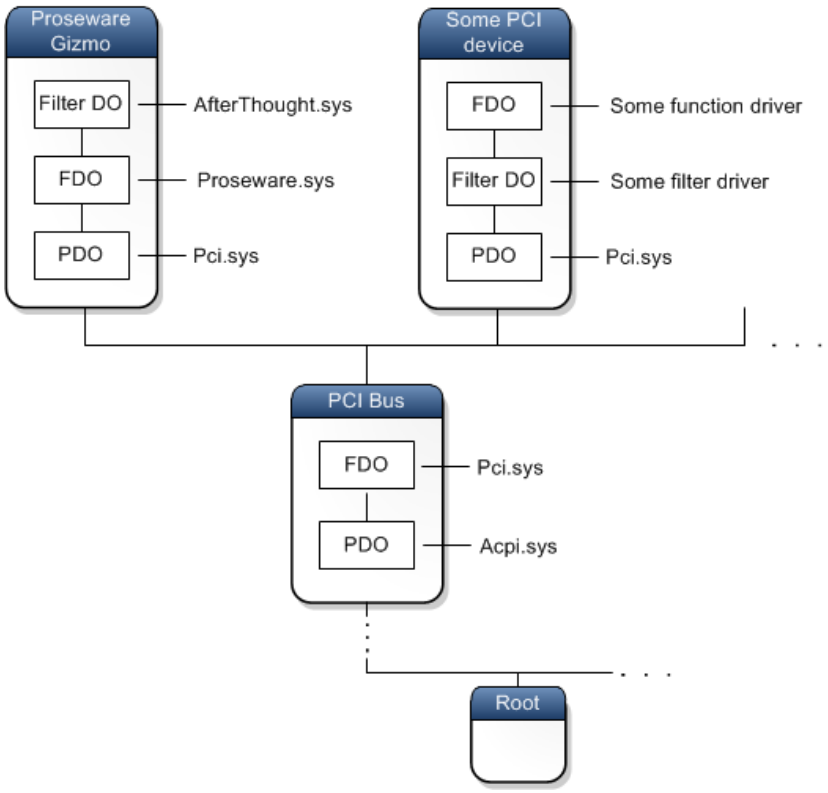
In the following example, I describe an innovative way which can be used by the attackers in order to control kernel memory and run arbitrary code in it from User-Mode. Hardware and Software drivers are components which need to be run in Kernel-Mode.

Microsoft have tried to build a framework to cover all the needs of third-party companies to ease the development of kernel drivers.



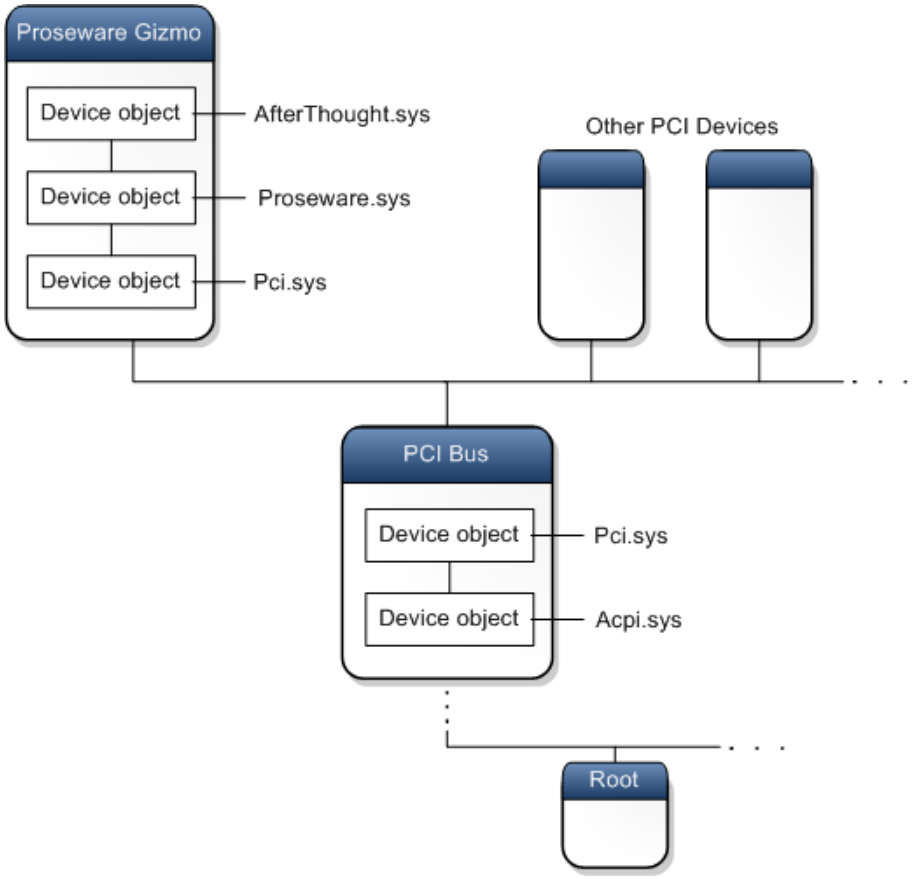
Every hardware company should exercise use of Windows Driver Kit (WDK) in order to build a Windows driver for their appliances.
On the other hand, Windows has a structure in which every driver should register to one of the layers of system and Windows will call that driver function when a new event happens or any signal that is specified to a special driver so hardware companies can use Windows Driver Model in order to get their goals.
From the above picture, you can see how Windows manages its driver through a tree

mode and it is good because it gives you an image of how often a driver function will be called.
Another stack-based model which drivers can get involved in Windows is as follows:



You can see that it consists of Filter DO drivers and FDO and PDO and in our case, all of the above drivers can be attacked in order to achieve the desired results.
All Windows drivers should implement a Driver Object model which should describe what is in the driver and export the methods which will be invoked by Windows or another Kernel-Mode driver.

Every time a driver needs to create a device in order to enable User-Mode application to talk with the driver, It registers a new device. all the devices have their own device object (which is also available in windows public symbols):




```
typedef struct _DEVICE_OBJECT {
  CSHORT                    Type;
  USHORT                    Size;
  LONG                      ReferenceCount;
  struct _DRIVER_OBJECT  *DriverObject;
  struct _DEVICE_OBJECT  *NextDevice;
  struct _DEVICE_OBJECT  *AttachedDevice;
  struct _IRP  *CurrentIrp;
  PIO_TIMER                 Timer;
  ULONG                     Flags;
  ULONG                     Characteristics;
  __volatile PVPB           Vpb;
  PVOID                     DeviceExtension;
  DEVICE_TYPE               DeviceType;
  CCHAR                     StackSize;
  union {
    LIST_ENTRY         ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
```

```
  } Queue;
  ULONG                           AlignmentRequirement;
  KDEVICE_QUEUE                   DeviceQueue;
  KDPC                            Dpc;
  ULONG                           ActiveThreadCount;
  PSECURITY_DESCRIPTOR            SecurityDescriptor;
  KEVENT                          DeviceLock;
  USHORT                          SectorSize;
  USHORT                          Spare1;
  struct _DEVOBJ_EXTENSION  *  DeviceObjectExtension;
  PVOID                           Reserved;
} DEVICE_OBJECT, *PDEVICE_OBJECT;
```

This model is also illustrated in the following picture :

As you can see, the above structure contains links to _driver_object which is also important for us because we need to edit the function of this structure :

```
typedef struct _DRIVER_OBJECT {
  PDEVICE_OBJECT     DeviceObject;
  PDRIVER_EXTENSION  DriverExtension;
  PUNICODE_STRING    HardwareDatabase;
  PFAST_IO_DISPATCH  FastIoDispatch;
  PDRIVER_INITIALIZE DriverInit;
  PDRIVER_STARTIO    DriverStartIo;
  PDRIVER_UNLOAD     DriverUnload;
  PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION+1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

So the MajorFunction is important for us because in our example we will modify one of these 28 predefined functions in order to create an environment where can be accessed easily from User-Mode.
We can generally expect a filter device object that is attached before a volume to receive the following types of I/O requests:

IRP_MJ_CLEANUP

IRP_MJ_CLOSE

IRP_MJ_CREATE

IRP_MJ_DEVICE_CONTROL

IRP_MJ_DIRECTORY_CONTROL

IRP_MJ_FILE_SYSTEM_CONTROL

IRP_MJ_FLUSH_BUFFERS

IRP_MJ_INTERNAL_DEVICE_CONTROL

IRP_MJ_LOCK_CONTROL

IRP_MJ_PNP

IRP_MJ_QUERY_EA

IRP_MJ_QUERY_INFORMATION

IRP_MJ_QUERY_QUOTA

IRP_MJ_QUERY_SECURITY

IRP_MJ_QUERY_VOLUME_INFORMATION

IRP_MJ_READ

IRP_MJ_SET_EA

IRP_MJ_SET_INFORMATION

IRP_MJ_SET_QUOTA

IRP_MJ_SET_SECURITY

IRP_MJ_SET_VOLUME_INFORMATION

IRP_MJ_SHUTDOWN

IRP_MJ_WRITE

There are also some other functions which we can use in order to achieve our goals but in the this example we will use the above functions (however in Windows, all of the following functions can also be used in the same manner):

FastIoCheckIfPossible
FastIoDetachDevice
FastIoDeviceControl
FastIoLock

FastIoQueryBasicInfo
FastIoQueryNetworkOpenInfo
FastIoQueryOpen
FastIoQueryStandardInfo
FastIoRead
FastIoReadCompressed
FastIoUnlockAll
FastIoUnlockAllByKey
FastIoUnlockSingle
FastIoWrite
FastIoWriteCompressed
MdlRead
MdlReadComplete
MdlReadCompleteCompressed
MdlWriteComplete
MdlWriteCompleteCompressed
PrepareMdlWrite

In our example we use IRP_MJ_DEVICE_CONTROL which makes it possible to give out an IOCT number that is implemented for arbitrary special use so we simply add another IOCTL to this function by patching it. This IOCTL will act as our own secret key by which we can notice the driver from our malware in User-Mode in order to execute a buffer in kernel.
Windows treats all of the drivers' devices in the same manner so we can use OpenFile (OpenFileA or OpenFileW) to get a handle from a specific device then we can pass our buffer to that device.
In the end windows passes an IRP to the desired function (Our buffer and IOCTL Code). After that we should use IODeviceControl function which is designed for invoking IRP_MJ_DEVICE_CONTROL and after finding the _device_object of our target device we should dereference _driver_object pointer to reach to the _driver_object and finally find PDRIVER_DISPATCH. Now we have two options:

1. 1) We can change the pointer entirely and build a completely different function in a different place so every time the patched function is invoked, our function will be called and because our User-Mode program changed the CPU state from ring0 to

ring3 by a syscall then our function will run under the ring0 state (same as the privilege of Windows kernel). Remember, we also passed arguments as a buffer to the function from the User-Mode application and it is available as an IRP to the invoked function.

2. 2) The second method which can be used is modifying the dispatched function to do the desired out of ordinary behavior. This is a better approach because some AVs or analyzers might audit this addresses and result in your modification to be revealed or due to the fact that functions addresses are almost always placed nearby each other, in the case of a simple human audit, the change has an increased risk of detection if you use the first method because the changed address would be far from others.

In the above example, we patched the Windows memory to get access from User-Mode. Our patched kernel now behaves normally in all cases but in the case of a special IOCTL, our patched function executes the buffer (Which is previously supplied in the User-Mode application). So our simple application patched the kernel by itself and now we successfully have a User-Mode program which can run its arbitrary instructions in Kernel-Mode.

Make changes in kernel in order to have a persistent access

There are many ways which you can obfuscate your access to a computer to make it hard for analyzers to detect your access.
In the below example, we use a technique in which it is made possible to make changes in another process which runs normally in a system and cannot affect any other process or any system objects because of lack of privilege.

In the above scenario an attacker could run a process with low-privileges and an elevated process which has an access to run a local debugger, so this two separate process don't have any relation with each other, even they could be run by totally different users (in the local system or Active Directory users), then after running local debugging the attacker will be able to find the PSActiveProcess head which in turn gives the current process _EProcess to him/her.

After finding the aforementioned address, we should now map this address to a _EProcess to see the contents in a meaningful way. After that we find ActiveProcessLinks (which is a field of _EProcess) and its datatype is _LIST_ENTRY which consist of a blink and a flink, this way you can easily traverse across all the processes and find two essential processes which will be discussed below.

First, we should find System process which runs under the, NT AUTHORITY\SYSTEM and second is the process which we need to elevate its privileges.
An attacker can find the desired process by just traversing and looking for the ImageFileName property which is a part of _EProcess (or statically; process with PID 4 is always System process). After finding these two processes we should next find the Token property of the System process form _EProcess that is a field of data type EX_FAST_REF which is used by Windows to find the privilege of a process and then gives the appropriate access to all objects based on this privilege.

So as long as System process has the privilege to access all Windows objects then we should dereference the Token to see where the windows saves its access privilege of the System process, after finding this address then we should change the Token of the target process in _EProcess. Now by having changed this value, subsequently the low-privilege process, is run as NT AUTHORITY\SYSTEM which is the highest available privilege in Windows.

In this manner, you can observe how a minimally privileged process can easily run with the highest available privilege.
In order to keep this privilege then the elevated process can register itself in one of the many places in which are run with the highest privilege by Windows; like Task Scheduler.

There are also other impacts of this method, an attacker could also change the normal privilege behavior of Windows or sometimes change a specific process privilege to a lower one.

In this kind of privilege substitution, neither of Windows security measures denies us from changing privileges, nor any Anti-Virus detects it.

An innovative way which can be used by the attacker, is lowering the privilege of specific process of an Anti-Virus and in the case the AV process no longer have any access to specific files and during our test some AVs no longer can scan the files which they don't have access to (even if some AVs have a totally different manner in scanning files with directly reading them by a Kernel-Mode driver but this method needs a special reversing to find where they decide to scan the file and finally it could definitely be bypassed by this method).

Change Windows kernel structures attributes

The described method could also be used to change Windows kernel attributes or protections and cause anomalies in windows behavior.
This could be lots of ways an attacker can change Windows behavior (which include all parts of windows) regardless of the fact that most of them are not possible even with the highest available privilege. (e.g an attacker can easily disable all of the system security controls or normal reporting behavior of system and cause massive system software issues or even physical problem without any detection from any AVs or other security measures being in place). The impact of this kind of attacks can be tremendous in industries and factories in which sensitive data of current status of devices is constantly needed, so an attacker who accesses such systems can modify the kernel to present everything is normal in a device and on the other hand destroys the devices by destructive commands and the destruction will be unavoidable in these cases.

Imagine we have a driver which checks for every IRQ request; in the case of every IRQ, Windows creates an IRQL to send to a special device, so it avoids concurrency between two requests (which could cause destructive results), then an attacker could easily patch the checks and send two or more requests to the device at the same time, as the result it will cause the considered device to be stopped and destroyed. It's just a simple example, on another imaginary scenario the attacker could also do something to forge input results from a special device, this way the attacker will be able to analyze the functionality of Windows calculation of CPU

usage percentage and finally show the fake final result, less than the actual result and make lots of problem for the CPU.
During our tests, we use Windows Resource Monitor to show how destructive this attack could be. The target process is Perfmon.exe but changing User-Mode instructions in order to reach to our goal will be eventually pointless, because almost all of Anti-Viruses audit such modifications, therefore we should see where these results come from (even if they calculated in a user-mode program). after reversing and obtaining the needed info, we successfully patch the network result entrance directly from its Kernel-Mode driver, so its normal behavior (after patch) would be continuing without any usage that comes from network driver, so the User-Mode code continues without any modification because we didn't change anything in Perfmon.exe but the result is completely false. The Kernel-Mode driver just doesn't give anything to the User-Mode driver because when user-mode calls KeWaitForSingleObject, it doesn't return correct results so everything seems legitimate even if we reached our target and faked perfmon readings successfully.
This kind of attack can be leveraged in dozens of unpredictable situations and in most of the cases, it is a full-scale compromise of all sensitive devices.

Run a Kernel-Mode Rootkit and hide it among system modules

For many years, Rootkits and Bootkits were two major dangers in all computer systems that have caused enormous problems and in most of the situations the final solution was to change the operating system completely to get rid of these types of malware.
In 2005 and by introducing the new X64 version of Windows XP, Microsoft introduced PatchGuard and after that, they introduced Driver Signature Enforcement in the newer version of Windows to defeat this kind of malware.
It was quite an effective way which changed the world of malware programming completely, so the rate of new rootkits appearing, became gradually much lower and reached a point that today we can hardly see new rootkits, because of this two sophisticated security features.
In our example, we want to show how we could bypass these protections (Because we have access to the kernel) and then run our rootkit instructions in the latest version of Windows which has enabled PatchGuard and Driver Signature Enforcement.
As it was mentioned before, we bypassed DSE (Driver Signature Enforcement) by using Windows drivers for debugging. It is crucial to know that Windows doesn't allow

PatchGuard to be started in Debug-Mode, as a result one of the effective ways of implementing such rootkits which are directly commanded from User-Mode is following the previously described procedure ("access from kernel in malware's user-mode code"). in this fashion you can have the same effect of running Kernel-Mode code and control it from attacker's program but successfully running a rootkit is a completely different scenario.
By analyzing the osr driver loader which is an application that lunches driver .sys files directly we could find where the Windows checks for Driver Signature Enforcement and by this way, we could patch this mechanism completely. It can be an effective method which will need some reverse engineering work.
Another way by which we can achieve the same result is using a mapped driver from disk or memory. Windbg driver has a method which gives us the location where all the modules (Kernel-Mode or User-Mode) are loaded. Then by using these kinds of addresses we could find where the target driver is loaded (Remember by using this way we don't have to deal with ASLR because if we know the address of where our function is located, then we could add this distance to the address which is obtained from windbg and as result know the exact address which needs to be patched. Image Rebasing protection does not put any obstacles in our way here because we know the start address of all the modules) and this is because there is no PatchGuard to watch out for driver modifications.
Seems like Windows has made everything ready for an attacker to get kernel access.

Suggestions

In our opinion, letting Windows to be debugged in such a way is definitely wrong. Microsoft can introduce a separate edition per each normal Windows edition with the ability to be debugged.
In this way, almost all of Windows-running computers will be safe from such attacks.
Another way which can be leveraged by Microsoft is letting Debuggers to be launched on a separate desktop and not be abled to be controlled by User-Mode programs. To cite an example, Microsoft has already implemented such mechanism for UAC (User Account Control) and as you know any User-Mode application can't access the window which asks for UAC permission, thus eliminating any possibility of attackers inter-

acting with Windbg window causing any possibility of damage from these types of attacks to be minimized in the wild.

References

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-local-kernel-debugging-of-a-single-computer-manually