Fast and Furious: Outrunning Windows Kernel Notification Routines from User-Mode

Pierre Ciholas[1(✉)], Jose Miguel Such[2(✉)], Angelos K. Marnerides[1(✉)], Benjamin Green[1(✉)], Jiajie Zhang[1(✉)], and Utz Roedig[3(✉)]

[1] Lancaster University, Lancaster, UK
{p.ciholas,angelos.marnerides,j.zhang41}@lancaster.ac.uk
[2] King's College London, London, UK
jose.such@kcl.ac.uk
[3] University College Cork, Cork, Ireland
u.roedig@cs.ucc.ie

Abstract. Modern Operating Systems (OSs) enable user processes to obtain full access control over other processes initiated by the same user. In scenarios of sensitive security processes (e.g., antivirus software), protection schemes are enforced at the kernel level such as to confront arbitrary user processes overtaking with malicious intent. Within the Windows family of OSs, the kernel driver is notified via dedicated routines for user-mode processes that require protection. In such cases the kernel driver establishes a callback mechanism triggered whenever a handle request for the original user-mode process is initiated by a different user process. Subsequently, the kernel driver performs a selective permission removal process (e.g., read access to the process memory) prior to passing a handle to the requesting process. In this paper we are the first to demonstrate a fundamental user-mode process access control vulnerability, existing in Windows 7 up to the most recent Windows 10 OSs. We show that a user-mode process can indeed obtain a fully privileged access handle *before* the kernel driver is notified, thus prior to the callback mechanism establishment. Our study shows that this flaw can be exploited by a method to (i) disable the anti-malware suite Symantec Endpoint Protection; (ii) overtake VirtualBox protected processes; (iii) circumvent two major video game anti-cheat protection solutions, BattlEye and EasyAntiCheat. Finally we provide recommendations on how to address the discovered vulnerability.

1 Introduction

Process isolation acts as a core OS security function, prohibiting user interaction with processes that do not belong to them. Hence, the OS prevents access to process memory, and does not allow interference with process execution. Nonetheless, interaction is possible if two processes are owned by the same user. Whilst both processes have the same owner, interaction is not considered as a security

© Springer Nature Switzerland AG 2020
C. Maurice et al. (Eds.): DIMVA 2020, LNCS 12223, pp. 67–88, 2020.
https://doi.org/10.1007/978-3-030-52683-2_4

risk as in reality many processes require such interaction to fulfill their tasks. For instance, a debugger must be able to attach itself to another process to control it and access its process memory. In fact, OSs provide Application Program Interface (API) functions to support such interactions. Namely, Linux provides the *ptrace* system call to observe and control another process whereas Windows provides the *OpenProcess* system call to obtain process handles that can then be used to interact with other API functions (e.g., *ReadProcessMemory*).

Regardless of the usefulness derived from the interaction between various user-mode processes, there exist situations where such functionality needs to be controlled. For example, on a desktop computer all User Interface (UI)-dependent processes initiated by the user have access to each other. Unavoidably, if the user accidentally executes a piece of malware, the resulted spawned process is able to access and control all other processes belonging to the same user. Consequently, the malware can deploy a range of operations on other processes, including suspending or terminating the process, or reading and modifying its memory. Under this simple access take over, malware would therefore be in a position to access a banking application and read credit card details, or suspend execution of anti-virus software processes running as the current user.

The Windows OS family contains kernel API functions for modules (drivers) to protect security sensitive processes such as anti-virus software. As currently implemented in Windows OSs, the kernel driver is notified when a user-mode process requiring protection starts. Subsequently, the kernel driver registers a callback procedure in memory, triggered every time another process requests a handle on the protected process. The kernel driver is set up to intercept system calls such as *OpenProcess*, and selectively remove permissions (e.g. read access to the process memory) before passing the handle to the requesting process. Subsequently, the caller obtains a process handle with reduced capabilities that prevent security critical forms of process interaction. In general, the kernel driver feature is widely used in Windows OS to protect critical processes. For instance, anti-virus software utilises the aforementioned feature to prevent malware from disabling anti-virus processes; Virtual Machines (VMs) use it to enforce appropriate isolation preventing access to security critical kernel functions exposed by their drivers from other processes; anti-cheat software uses this feature to prevent cheaters from obtaining access to the game process.

In this paper we show that the previously described protection method can be circumvented, highlighting a fundamental issue within the Windows OSs. We argue that this discovery is not a traditional vulnerability that could be fixed with a simple patch, but rather a core OS security design flaw. Through this work, we demonstrate that arbitrary user-mode processes can obtain fully privileged handles before the kernel driver instruments a callback protection procedure. Consequently, user-mode processes can outrun notification routines destined for the kernel protection driver of the newly created processes. This vulnerability has been acknowledged by Microsoft (see Sect. 6.5); however, Microsoft argues that the issue should be addressed by individual software developers, as addressing it on a kernel level would lead to backward compatibility issues.

The contributions of our work are:

- *Outrunning Kernel Notifications:* we introduce how kernel notification routines can be outrun by an unprivileged user-mode process.
- *Example Exploits:* we show how the flaw can be exploited to (i) disable the anti-malware suite Symantec Endpoint Protection; (ii) take control of VirtualBox protected processes; (iii) circumvent the two major video game anti-cheat protection software solutions, BattlEye and EasyAntiCheat.
- *Mitigating the Flaw by Design:* we recommend that user-mode functions taking a Process identifier (PID) as a parameter should not be able to do so with incomplete initialisation since kernel routines are triggered post initialisation. Although, Microsoft does not plan to implement such functionality as it would create compatibility issues.
- *Handle Invalidation Procedure*: we indicate that on spawn detection of the protected process, the kernel driver could initiate immediate termination, thus invalidating any handle that might have been obtained by exploiting the discovered vulnerability. The driver can then respawn the process from kernel-space and set up callback protection without delay. However, the proposed procedure has limitations and cannot be used in all cases.

The remainder of this paper is structured as follows: Sect. 2 focuses on the required background knowledge to understand the identified vulnerability, associated exploits, and their impact. Section 3 discusses the discovered vulnerability and two example exploits applying alternate exploitation methods. Section 4 demonstrates the vulnerability and our two exploits over three case studies. Related work is presented in Sect. 5, followed by a discussion in Sect. 6 detailing how we discovered the vulnerability, its consequences, and possible solutions. We conclude and summarise the paper in Sect. 7.

2 Background

One of the many roles an OS has, is to enable processes to execute concurrently, securely isolated, with sufficient guarantees that they will not disrupt each other or the overall system. However, in some cases processes require interaction to fulfill their tasks, and must request authorisation to do so from the OS. Traditionally, on Microsoft Windows OSs the function *OpenProcess* is used to request a handle on a target process; the obtained handle represents authorisation.

A handle has a set of privileges [1] allowing it to be used for specific operations. For instance, a process handle may permit the creation of child processes and new threads, duplication of handles, querying of information, setting of quotas as well as suspension, resumption and termination of the process. Moreover, a handle can permit the creation of virtual memory operations, reading and writing of the process virtual memory, and synchronisation with a given target process. Many of these aforementioned privileges can be used to alter adjacent processes, it may therefore be important to apply restrictions. Examples of processes where privilege limitation is necessary include anti-malware software,

software using drivers exposing sensitive kernel functions, banking and point of sale applications, and multiplayer video game processes.

Access to processes can be limited by executing them as different users. However, if processes are executed under the same user, control can be challenging to implement, as by default a user has full access to all of his processes. In addition, processes running as administrator can also access user processes. Microsoft provides a standard method to implement such protection using specialised kernel API functions, to limit handle privileges obtained for a process running as the same user or higher privileged users. To the best of our knowledge, this is the only officially advised method to implement such a security mechanism.

A kernel driver uses the kernel API function *PsSetCreateProcessNotifyRoutine/Ex* to receive notification of new processes. When a new handle on a process is requested, a callback previously registered with *ObRegisterCallbacks* is triggered, and the kernel process can apply filters to limit the privileges of this handle. Thus, fine-grained access control amongst processes owned by the same user or more privileged users can be implemented. This is an important feature, as on a Windows OS most processes run under the user logged into the GUI of the system. This includes processes of security critical applications.

2.1 Notification Routines

To implement process protection, the kernel driver must be notified of new processes in the system. The driver registers a *create process notify routine* using *PsSetCreateProcessNotifyRoutine/Ex* providing a pointer to one of its functions that will be executed when a new process is created or terminated. The pseudocode in Listing 1.1 depicts the key instructions.

```
1 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath) {
2   PsSetCreateProcessNotifyRoutine(
    ProtectionDriverFindProtected, FALSE);
3 }
4
5 NTSTATUS ProtectionDriverFindProtected(HANDLE ParentId,
    HANDLE ProcessId, BOOLEAN Create) {
6   // Code executed at process creation
7 }
```

Listing 1.1. Key steps to register a notify routine

In the *DriverEntry* function, the kernel driver calls *PsSetCreateProcessNotifyRoutine* with *ProtectionDriverFindProtected* as a parameter. This instructs the kernel to execute the driver's function *ProtectionDriverFindProtected* when new processes are created or terminated. The kernel passes parameters to this function: (i) a HANDLE to the parent process, (ii) a HANDLE to the new process, and (iii) a BOOLEAN indicating if the process was started or terminated. Now that the driver is notified of any new processes, it can be decided in *ProtectionDriverFindProtected* if a process requires protection, and what type

of protection to apply. For example, protection might be applied to processes matching a specific image name or signature.

Figure 1 shows the sequence of events when a new process is created. In this diagram, process A starts process B. For example, process A could be *explorer.exe* used to find and then double click on an application to start process B. The parent process (process A) in this example uses *CreateProcess* to create process B, however, the same sequence of events occurs if another function is used to create process B (e.g. *CreateProcessAsUser*, *ShellExecute*, or *system*).

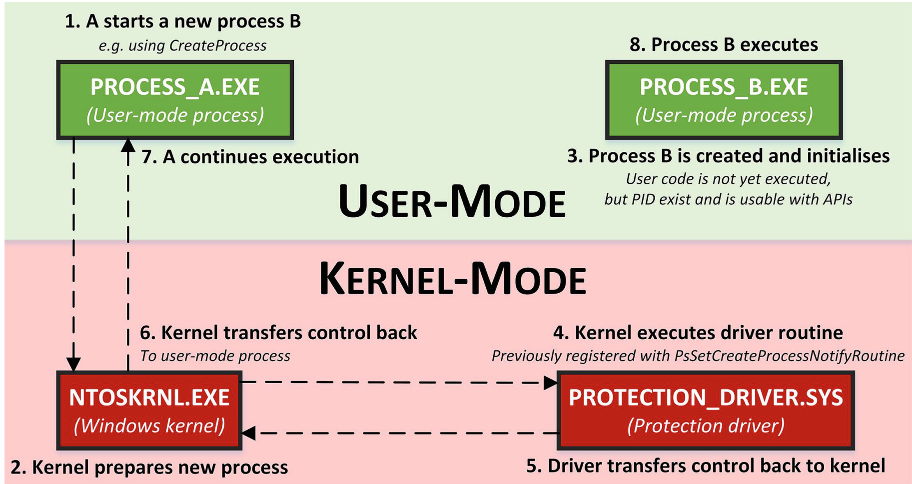

Fig. 1. Creating a new process with a driver's process create notify routine

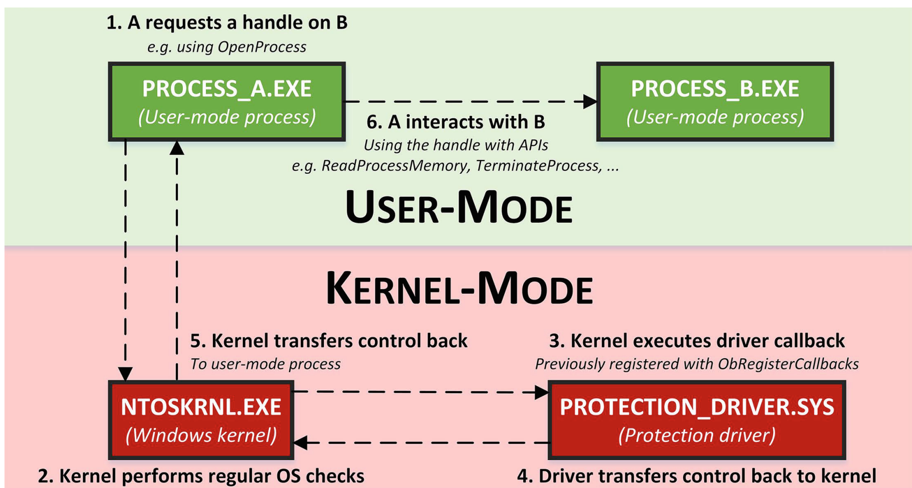

Fig. 2. Handle request with a driver having set up callbacks protection

The kernel performs various checks and operations to prepare process B for execution. Amongst these operations, the kernel creates memory structures describing the new process, such as *KPROCESS* or *EPROCESS*, and attributes a unique PID to the new process, making it reachable with other API functions taking a PID as a parameter, including *OpenProcess*. The kernel then looks at the registered *create process notify routines*, and transfers execution to the drivers having registered one. In our example, control is then passed to our kernel driver, and its *ProtectionDriverFindProtected* function is executed. The code in this function first checks whether protection should be applied, then applies it if necessary.

This specific sequence of events is problematic as the newly created process B is accessible before protection can be applied. Process B can be addressed by other processes in the system before protection is applied in Step 4. This provides a time window for malicious processes to obtain a handle on process B before protection is put in place. The driver then transfers control back to the kernel, then back to the user-mode process, and finally process A continues execution and process B starts executing with protection in place.

2.2 Protection via Callback

The kernel driver registers a callback so that when a handle on a process is requested, a function in the kernel driver is called. This process is depicted in Fig. 2. Before delivering the requested handle, the kernel first transfers execution to the driver, and passes parameters allowing it to retrieve relevant information about the handle operation to perform adequate filtering. This information includes whether the handle is newly created or duplicated, whether it is a kernel handle or not, a pointer to the target process or thread, a pointer to the object type, and a pointer to a memory structure describing operation-specific parameters. With this mechanism, a driver can apply fine grained filtering on handles to the process it protects. The decision on which processes to protect is performed when the driver is notified on the creation of new processes as described in the previous section. The driver can, for example, remove specific rights on a handle, preventing operations including reading or writing the process memory. After the driver's callback function is executed, the driver transfers control back to the kernel, and then back to the user-mode process having initially requested the handle. Listing 2 shows the key steps to register a callback.

```
1 PVOID pCbHandle = NULL;
2 OB_OPERATION_REGISTRATION obCbOp;
3 OB_CALLBACK_REGISTRATION obCbReg;
4 obCbOp.ObjectType = PsProcessType;
5 obCbOp.Operations |= OB_OPERATION_HANDLE_CREATE;
6 obCbOp.PreOperation = PreCbOp;
7 obCbReg.OperationRegistration = &obCbOp;
8 ObRegisterCallbacks(&obCbReg, &pCbHandle);
9 OB_PREOP_CALLBACK_STATUS PreCbOp(PVOID RegistrationContext,
      POB_PRE_OPERATION_INFORMATION OperationInformation) {
```

```
10     // Code executed at handle request
11 }
```

Listing 1.2. Key steps to set up callbacks protection

To register a callback, we initialise and fill the required memory structures, specifying that we want our callback to be triggered when a handle is requested on processes. We set our callback to be executed when a new handle is created with the flag *OB_OPERATION_HANDLE_CREATE*, and supply a pointer to the function to be executed. Finally we register the callback by calling *ObRegisterCallbacks*.

3 Vulnerability

A vulnerability arises from an insecure time period during new process creation. In this initial phase the process is initialised by performing all of the operations required prior to the execution of the program. Among these operations, the OS kernel internal memory structures describing this new process are created, and the process is given a unique PID, enabling API functions taking a PID as a parameter, including *OpenProcess*. This insecure time period is present between stage 2 and 4, as shown in Fig. 1. During initialisation, any driver designed to apply protection has not yet received notification that a new process has been created, and therefore no protection can be applied, while the process is already reachable from API functions. Figure 3 shows a simplified time-line of these events.

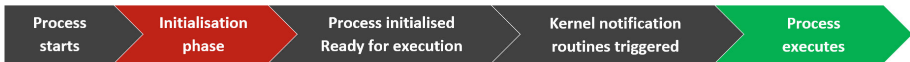

Fig. 3. Time-line of a newly spawned process

Exploiting this insecure period of time is therefore possible if one can (i) know that a newly process has spawned and (ii) know its PID, before the initialisation completes. We have discovered two different methods to accomplish this and exploit the vulnerability.

The First Method is Based on Registering a Job Object on the Parent Process of the Target Process. A job object allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them [2,3]. We noticed that job objects allow a process to be notified of a new child process with its PID directly after it starts, before the initialisation phase completes, and before the driver is notified and applies any protection. Our exploit can then simply call *OpenProcess* to obtain a fully privileged process handle before protection

is applied. We named this first exploit *hFromJob*, since it gives a fully privileged handle (h) using a job object (FromJob), following the Hungarian naming convention that Microsoft traditionally uses.

The Second Method Uses Aggressive PID Guessing. We simply start several threads calling *OpenProcess* on all the possible IDs that the new process could be assigned. A separate thread then analyses all the handles gathered, and stops the exploit once it has obtained a handle to the targeted process. We named this second exploit *hThemAll*, since it obtains a fully privileged handle (h) by attempting to obtain it through all possible PIDs (ThemAll).

3.1 Exploit Using Job Object (hFromJob)

In this first exploit, we create a job object on *explorer.exe* assuming that the target process will spawn as its child process. This configuration is useful as a proof of concept, as it represents starting a process by double-click, the normal procedure a user would adopt to start a new process. This exploit can be adapted to other scenarios by creating a job object on the known parent process. For example, programs started from the command line have *cmd.exe* as parent; the job object should therefore be on this process instead.

The key steps are to create an IO port handle using *CreateIoCompletionPort*, create a handle to the job object with *CreateJobObjectW*, configure the job object with *SetInformationJobObject*, and finally assign the job object to the parent process with *AssignProcessToJobObject*. The exploit process will be notified of new processes being spawned by checking the I/O completion port queue. Therefore, we start a thread on a function that checks the queue as fast as possible, and directly calls *OpenProcess* on any new process. This method outruns kernel process notification routines since the notification from the job object takes place before the end of the initialisation phase. This exploit has the following requirements: (i) the parent process of the targeted process must be known in advance, and (ii) it must be possible to obtain a process handle on the parent with the permissions *PROCESS_SET_QUOTA* and *PROCESS_TERMINATE*. The C++ source code for this exploit can be found on GitHub [4].

3.2 Exploit Using PID Guessing (hThemAll)

An alternative approach is possible. It is feasible to simply predict or "guess" the PID of the target process. To the best of our knowledge, it is not possible to predict with 100% accuracy the PID of the next process to be spawned from user-mode, therefore we took advantage of how Windows manages PIDs to narrow the search space for the next PID: (i) Both process IDs and thread IDs are generated in the same namespace, therefore, they cannot overlap [5]; (ii) PIDs & TIDs are always multiples of 4; (iii) Windows attempts to keep process and thread IDs in low numbers.

The exploit starts by listing all currently existing PIDs and TIDs, excluding them as potential PID for our target process to be spawned. We then create several threads attempting to obtain a fully privileged handle on every possible PID as fast as possible. Every new process handle is placed in a list analysed by a separated thread individually. When a handle to the target process is found, the exploit terminates. The C++ source code can also be found on GitHub. [6]

4 Case Studies

We developed our own protection driver following Microsoft's driver developer guidelines. We first use this demonstrator to clearly showcase the vulnerability. Thereafter, we describe how the vulnerability can be exploited to (i) disable the anti-malware suite Symantec Endpoint Protection; (ii) take control of VirtualBox protected processes; (iii) circumvent the two major video game anti-cheat protection software, BattlEye and EasyAntiCheat. We demonstrate that the vulnerability can be used to bypass protection mechanisms for a wide variety of current applications, highlighting the severity of this issue.

4.1 Bypassing Our Own Protection Driver

Anti-malware solutions, security critical applications, and video game anti-cheat software are not open source and often obfuscate code structure and operations. We decided that our work would benefit from presenting the vulnerability in a context where both attack and target code structure is known. This allows us to precisely pinpoint and clearly describe the vulnerability along with the sequence of events leading to its exploitation. We therefore developed a minimalist kernel driver that installs protection for a specific process that we then target with our exploits; we attempt to obtain a fully privileged handle to the process protected by the kernel driver.

The simplified pseudo-code in Listing 1.3 shows the key steps our driver follows to set up protection. The full code of the driver can be found on GitHub [7].

The driver starts by registering a routine with *PsSetCreateProcessNotifyRoutine*, which causes our driver's function *ProtectionDriverFindProtected* to be called whenever a process is created or terminated, as explained in Sect. 2. It then registers a callback by calling the function *ProtectionDriverSetProtection*, which will cause the function *PreCbOp* to be executed when a new handle is requested on a process. This function, not included in the pseudo code for simplicity, fills the required memory structures before calling *ObRegisterCallbacks* as per Microsoft's guidelines [8].

```
1  HANDLE hProtectedPID = NULL;
2  NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
       PUNICODE_STRING RegistryPath) {
3    PsSetCreateProcessNotifyRoutine(
       ProtectionDriverFindProtected, FALSE);
4    RegisterCallback();
5  }
6  NTSTATUS ProtectionDriverFindProtected(HANDLE ParentId,
       HANDLE ProcessId, BOOLEAN Create) {
7    if (IsProtected(ProcessId, "Protected.exe"))
8      hProtectedPID = ProcessId;
9  }
10 OB_PREOP_CALLBACK_STATUS PreCbOp(PVOID RegistrationContext,
        POB_PRE_OPERATION_INFORMATION OperationInformation) {
11   HANDLE TargetProcessId = PsGetProcessId((PEPROCESS)
       OperationInformation->Object);
12   if (TargetProcessId != hProtectedPID)
13     return OB_PREOP_SUCCESS;
14   if (OperationInformation->Operation ==
       OB_OPERATION_HANDLE_CREATE)
15     OperationInformation->Parameters->
       CreateHandleInformation.DesiredAccess &= ~
       PROCESS_TERMINATE;
16 }
```

Listing 1.3. Key steps of the protection driver

The function *ProtectionDriverFindProtected* checks the new process's image name and compares it to the protected process name, then sets the protected process ID in the global variable *hProtectedPID* when found. In our proof of concept driver we protect the latest instance of any process with the image name *Protected.exe*. This first phase allows the driver to find the protected process ID, so the defence mechanism using callbacks can modify future requested handles.

If the requested handle is on the protected process, the function *PreCbOp* removes the permission *PROCESS_TERMINATE* (0x1) as evidence, demonstrating that the handle permissions were successfully edited by our driver. This last phase represents the defence mechanism implemented by our driver.

As an experiment, we first launch a dummy process named *Protected.exe*, and then attempt to obtain a fully privileged handle from another process. This works as intended, obtaining a fully privileged handle and therefore full access to the target process. Next, we load our protection driver, then execute *Protected.exe* and attempt to obtain a fully privileged handle from another process. This results in the acquisition of a handle without *PROCESS_TERMINATE* (0x1) permission. This indicates that our protection driver has successfully protected the process, and removed the permission to terminate it on the handle.

Finally we terminate *Protected.exe* and execute our two exploits (*hFromJob* and *hThemAll*) described in Sect. 3 before executing *Protected.exe*. This time, both our exploits successfully obtain a fully privileged handle, which indicates that the protection was not in place on time to protect the process. This shows

that kernel notification routines are slow enough that we can obtain a handle using PID guessing (with exploit *hThemAll*), and that user-mode process notifications obtained from job objects are also faster than kernel notification routines (with *hFromJob*). This allows us to bypass any protection set after *PsSetCreateProcessNotifyRoutine/Ex* by outrunning them from unprivileged user-mode processes; in this specific case we bypassed the callback protection.

We provide a short video demonstrating this experience [9].

4.2 Disabling Anti-malware

In this second case study we use our exploits to obtain a fully privileged handle on the service process of Symantec Endpoint Protection. Symantec anti-malware service, *ccSvcHst.exe*, is spawned as a child of *services.exe* under the user *NT AUTHORITY\SYSTEM*, which spawns another instance of the same binary running under the current user.

When requesting a fully privileged handle on the anti-malware user process using *OpenProcess* with *PROCESS_ALL_ACCESS*, a handle is received but with only the following permissions: Query information, Create processes, VM read, Synchronize, Read control, and Write owner (0x1AF490). This demonstrates that Symantec's driver installs protection to modify the handle permissions on its user-mode processes with *ObRegisterCallbacks*. The driver also needs to be notified of the system process being spawned, which is done with *PsSetCreateProcessNotifyRoutine/Ex*. Thus, the anti-malware appears to use the protection method we described and is therefore potentially vulnerable.

For this experiment our *hFromJob* exploit is not usable, as it requires a handle to the parent process of the target with the permissions *PROCESS_SET_QUOTA* and *PROCESS_TERMINATE* to make use of the API *AssignProcessToJobObject*. The system process *services.exe*, being a protected system process (Protected Process Light (PPL)) means that *OpenProcess* would fail. It should be possible to use this exploit if one bypasses PPL and obtains a sufficiently privileged handle on *services.exe*, however for simplicity we will only use the exploit *hThemAll* for this case study where no additional steps are required.

We begin the experiment by configuring *hThemAll* to look for a handle on a process named *ccSvcHst.exe* using 6 threads. We execute the exploit, then start the service of the anti-virus. Once the user processes have spawned, we used Process Hacker [10] to verify the permissions of our process handle, and observe that we have been granted a fully privileged handle.

With this fully privileged handle it is now possible to tamper with the anti-malware system in a variety of ways. Most anti-malware have watchdog systems restarting the user-mode process if terminated, therefore simply terminating it has little interest, however a very simple workaround is to freeze its threads with the API *NtSuspendProcess*, disabling malware detection alerts to the user, and disrupting its real-time protection capabilities. Since this technique exploits an insecure time period at launch, a piece of malware executing early, i.e. during the boot or user login sequence, could take control of the anti-malware processes from the start. Otherwise the malware could terminate the process using a variety of

methods ranging from simply calling TerminateProcess to a denial of service, in which case the anti-malware watchdog would restart a new instance, allowing us to exploit the vulnerability on this newly spawned process. Malware could also simply wait for the process to restart on its own during maintenance and update cycles.

4.3 Bypassing Virtualisation Defences

Virtualisation solutions are complex in that they require device drivers to provide additional functionality at kernel level. They are also required to make these capabilities available to specific user-mode processes. Therefore, virtualisation kernel drivers will often expose potentially sensitive functionality to user-mode processes. For this reason, only the trusted user-mode process of the virtualisation software should be able to access the virtualisation environment's kernel driver. This includes processes running under the same user or a user with higher of privileges. If a privileged handle is obtained on the virtualisation user-mode process, then malware could leverage this handle to obtain access to the kernel driver via the captured user-mode process.

Virtualisation software Virtual Box uses a driver for the protection of its user-mode processes. When a virtual machine is started, another instance of *VirtualBox.exe* is spawned, which has a handle on the driver *VBoxDrvStub* with read and write permissions. The second instance of *VirtualBox.exe* then spawns a thrird instance of *VirtualBox.exe* which is the process running the virtual machine. This process has a handle on the driver *VBoxDrv* with read and write permissions. Both of these processes are protected in such a way that if another process operating under the same user, or a higher privileged user, attempts to obtain a fully privileged handle, the returned handle is modified to only have the access mask 0x131c11 (Query information, VM read, Suspend/resume, Terminate, Synchronize, Delete, Read control).

Through the use of exploits *hThemAll* and *hFromJob*, we can successfully obtain a fully privileged handle on the first instance of *VirtualBox.exe*. This process has a handle on the driver *VBoxDrvStub*, which exposes potentially sensitive kernel functionality. Both exploits successfully provide a fully privileged handle on the second instance of *VirtualBox.exe* that is normally protected. This gives us access to the driver *VBoxDrvStub* that could be leveraged for further exploitation. Interestingly, our exploits do not obtain a fully privileged handle on the third instance of *VirtualBox.exe* protected process. This indicates that this process is either protected, or more likely is *spawned* differently. This latter behaviour could be used as a basis for effective mitigation of the vulnerability. We discuss this later in Sect. 6.

4.4 Bypassing Video Game Anti-cheat Defences

Obtaining access to a game's process allows hackers to manipulate it to gain unfair advantages. Anti-cheat companies offer game developers software to prevent other programs, including programs running at higher privilege levels from

gaining access to the game's process using protection drivers. For our use cases, we have experimented on games protected by the two major anti-cheat systems: BattlEye and EasyAntiCheat.

We observe that a lesser privileged handle is obtained despite requesting all privileges on the game process, indicating the use of the vulnerable APIs. Using our two exploits, we successfully obtain a fully privileged handle on games protected by both anti-cheat software solutions.

We have informed anti-cheat software providers of this vulnerability (see Sect. 6), they have now included additional defences. We discuss these additional defences in more detail in Sect. 6. The vulnerability affected a wide variety of internationally recognised anti-cheat protected games with millions of players.

5 Related Work

This section presents other security mechanisms available in Microsoft Windows and evaluate their validity as mitigation. We also present academic work and patents related to the very specific nature of the vulnerability discussed.

5.1 Other Windows Security Mechanisms

Protected Processes: The Protected Process security mechanism was introduced in Windows Vista and has been expanded in later versions with variants including Protected Process Light (PPL). Protected processes differ from regular processes due to the level of access other processes in the system can obtain on them [11]. When a process is protected by this mechanism, other non-protected processes can only obtain handles on it with tightly restricted rights. If it was possible to spawn a process as protected from its initial inception, including the initialisation phase, this could void our exploits. Unfortunately, this security mechanism is only reserved for system use and is not available for third party software developers.

Anti-malware Services Protection: Microsoft provides a complete guide and specialised tools for anti-malware developers, allowing their driver to launch before other boot-start drivers, and therefore ensure that subsequent drivers do not contain malware [12]. This security measure helps protect against malicious drivers, but does not offer any mitigation against the presented attack, since it does not change the order of operations during process creation.

Mandatory Integrity Control (MIC): MIC is a mechanism for controlling access to securable objects [13]. It uses 4 levels of integrity with the labels low, medium, high, and system, preventing lower integrity processes from accessing the resources of higher levels. The MIC security mechanism was introduced in Windows Vista. This could in very specific circumstances mitigate the presented attack. For example, if the exploits were started with the low integrity label, and the target was allocated a medium integrity label, the exploits would fail. However this security mechanism falls short when defending interactions between

processes of the same integrity level. Since most processes run within the same level, this cannot be considered a reliable counter-measure. For anti-malware solutions specifically, running their critical processes with a high integrity level should provide protection against the exploits when run with the medium default integrity level.

Protected Mode: The protected mode is based on MIC and was originally created to enhance the security of Microsoft's web browser Internet Explorer [14]. This security feature was designed to limit possible attacks from a compromised Internet Explorer process, by running it with greatly reduced privileges. To the best of our knowledge, it is not possible to run third party programs in protected mode natively. Even if this was possible, as a security mechanism it is designed to restrict a specific process, preventing it from interacting with others, not to prevent access from other processes as the vulnerable APIs targeted in this paper do. This security mechanism is therefore not a viable option.

AppContainer Isolation: When creating a program with AppContainer, the process is executed with extreme limitations, allowing only those features critical to the program operations. This security feature functions in a similar way to other mandatory access control implementations in other operating systems, such as Security-Extended Linux (SELinux) or AppArmor in Linux. All non-required resources are kept out of reach, including other processes, therefore a compromised or malicious process cannot take over the rest of the machine [15]. Files, registry, windows, and network resources are also restricted, and access can be managed with fine granularity if required. Finally, process isolation prevents the AppContainer program from influencing other processes. However, after experimenting with AppContainer ourselves, we were able to restrict a process's access to resources and other processes, but could not restrict other processes from accessing itself. Consequently, we could not use AppContainer as a valid form of mitigation.

5.2 Research Efforts and Patents

A multitude of projects and software make use of kernel notify routines and callbacks. *PsSetCreateProcessNotifyRoutine/Ex* and *ObRegisterCallbacks* are often used for automated malware detection and prevention, or program behavioural analysis which permits the creation of tools for reverse engineering such as Capture presented in [16].

There exist only a few usable methods to monitor the behaviour of a program for which the source code is not available. These methods can be categorised as follows: (i) User level API hooking, (ii) kernel level API hooking, and (iii) Kernel callbacks [16]. User-level API hooking can be easily detected or bypassed by unprivileged programs. For this reason the quasi-totality of reputable anti-malware use solutions in kernel space. Many kernel-mode malware and rootkits made use of kernel level API hooking (e.g. SSDT hooking) to hide their presence and execute malicious code stealthily, consequently Microsoft now defends the kernel with various protections including PatchGuard (also known as Kernel

Patch Protection, KPP). A good example of malware making use of *PsSetCreateProcessNotifyRoutine/Ex* is the Rustock Rootkit and Spam Bot studied and documented in [17]. Due to its potential for abuse, it is no longer possible to enact kernel level API hooking on the modern versions of Windows, leaving only user level API hooking and a set of kernel API functions to implement security.

A presentation on how anti-viruses implement their monitoring, detection, and defences can be found in [18]. As recommended by Microsoft, all 5 of the major anti-viruses investigated in this work make use of the kernel callbacks and routines, including *PsSetCreateProcessNotifyRoutine/Ex* to obtain process creation and termination notifications to then run analysis and mitigations. This makes these anti-viruses vulnerable to the attack presented in this paper. One of the anti-viruses tested in [18] is Norton Security 2015, which is the anti-virus we selected as a use case.

A multitude of academic projects and patents in the field of malware analysis heavily rely on *PsSetCreateProcessNotifyRoutine/Ex*. In [19] a set of monitoring drivers are presented, including a process monitoring driver that uses *PsSetCreateProcessNotifyRoutine* to obtain information on newly created or terminated processes. In [20] the researchers attempted to correlate network traffic with user applications using the vulnerable API. Injecting data flow control object into processes using the same system as in our protection driver (using first *PsSetCreateProcessNotifyRoutine* then *ObRegisterCallbacks*) is presented in [21]. In [22], a system stored on a mass storage device is presented that registers a process notification routine to then hook functions in processes. Two researchers have designed a portable dynamic malware analysis tool following Microsoft recommendations in [23], therefore using the vulnerable *PsSetCreateProcessNotifyRoutine* to monitor process activities. A system aiming at identifying processes responsible for system slow downs making use of process notification routines is presented in [24]. In [25], a method relying on hooking/detouring the execution flow of *PsSetCreateProcessNotifyRoutine/Ex* to prevent malware from de-registering notify routines is presented. Because of their reliance on the vulnerable API, all of these projects could be disrupted or bypassed entirely.

6 Discussion

6.1 Discovery

The vulnerability was discovered whilst investigating how several system processes obtained privileged handles on video games despite active anti-cheat using protection drivers. Three system processes held privileged handles: *csrss.exe* (all privileges), *lsass.exe* (read/write), and *PcaSvc*'s *svchost.exe* (all privileges).

We investigated how *PcaSvc* obtained its handle, and quickly identified that it was accommodated through a normal *OpenProcess* call. We also noticed that if a delay is placed before calling *OpenProcess*, when execution is resumed the handle is modified as intended by the protection driver. This indicated that Windows system processes receive new process notifications before the kernel notification routine are triggered.

To verify this hypothesis, we developed a proof of concept exploit by hooking *OpenProcess* in *PcaSvc*'s process. With this hook, the PID of the new process is passed to another process using shared memory and its execution is resumed using a semaphore, allowing it to call *OpenProcess* and obtain a fully privileged handle. This exploit confirmed that *PcaSvc* was outrunning the kernel process notification routines used by the anti-cheat driver.

By analysing the internals of *PcaSvc*, we discovered that job objects are used to receive notifications. We then created the standalone exploit *hFromJob* replicating *PcaSvc*'s behaviour. Finally, we developed the second exploit *hThemAll*, which affords fewer restrictions and further confirms that the vulnerability emerges from an insecure time period during process initialisation.

6.2 Vulnerability Time Period Measurement

Measuring the vulnerability time period is not easy, since it depends on hardware characteristics such as CPU frequency, number of cores, threads, and also current system state and other parameters hard to fully control. We have conducted all our experiment in a 2.40 GHz mono-core VM on an idle system.

To measure the vulnerability time period we have counted how many fully privileged handles could be retrieved before the protection is set up. We have used the first **non**-fully privileged handle obtained as a sign that the exploitation time window has finished. We have measured using 3 methods: (1) using the *RDTSC* (Read Timestamp Counter) CPU instruction to get a number of CPU cycles, (2) *QueryPerformanceCounter*, which is a Microsoft supplied high resolution time stamp that can be used for time-interval measurements, and (3) *GetTickCount64* which uses the CPU clock to give an interval in milliseconds.

We first modified *hFromJob*, so that when its first fully privileged handle is obtained, it calls a measuring function that keeps calling *OpenProcess* requesting all permissions and verifying if the returned handles correctly has them using *NtQueryObject*. When the first lesser privileged handle is obtained, the measuring function calculates the time difference using the methods listed above. hFromJob successfully obtained between 63 and 105 fully privileged handles during our tests, occuring during 21 to 35 million cycles (from RDSTC), while *QueryPerformanceCounter* returned between 89 and 140 k (with a base frequency of 10 million retrieved with *QueryPerformanceFrequency*). GetTickCount64 doesn't provide enough accuracy and returned 0 in all our tests, indicating that the vulnerability is faster than its accuracy (Microsoft estimates this accuracy to be between 10 and 16 ms). Using the RDTSC readings, the vulnerability time period was measured to be between 8.75 and 14.5 ms, while using the readings of the performance counter the vulnerability time period is measured between 8.9 and 14 ms.

hThemAll is harder to measure, since we use a set of threads blindly attempting to get handles and a control thread looking for our target. By configuring the exploit to be extremely aggressive and using up most of the system resources by using 16 threads for exploitation we notice that many more (thousands) fully

privileged handles are obtained. Since the resources that should be used to initialise the process are redirected to the exploit, the vulnerability time period is extended due to the initialisation phase being slowed down. We attempted to measure the vulnerability time period with a single-threaded version. Since this exploit does PID guessing, we ran the experiment multiple times aiming at guessing the PID correctly soon to reveal the vulnerability time period. In the best result, *hThemAll* successfully obtained more 218 fully privileged handles before being stopped. This indicates that the vulnerability was present for at least 29 ms.

6.3 Implications

The discovered vulnerability poses the question of how to best implement protections for user-mode processes. Microsoft provides routines to obtain notifications in kernel drivers, however we demonstrated that they can be outrun by user-mode processes. Therefore a malicious process can outrun and consequently bypass any protections set up following reception of such notifications, simply through execution prior to its target. The protection can also be defeated after the target has been started if it can be forcibly restarted (e.g. by terminating it or crashing it).

The identified vulnerability allows outrunning thread notification routines set up with *PsSetCreateThreadNotifyRoutine/Ex*, and load image notification routines set up with *PsSetLoadImageNotifyRoutine*. Since writing to the process memory is possible with the process handle, a malicious program can force the execution of any code with a simple detour or hook within the context of the target process before any notification routine is triggered.

The most severe consequences of this vulnerability are for anti-malware solutions. Due to the fact that a process with malicious intent can interact with other processes before the routines trigger, it is feasible to fully modify and control them before protection has been applied. Malware can, for example, execute malicious code within the context of another process, or hijack the process completely with techniques such as process hollowing [26]. As demonstrated, if malware can be started early enough, or can force the user-mode process of the anti-virus to restart. Thus malware can control, disable, or prevent it from alerting the user of any present threats.

We argue that applications requiring exposure of sensitive kernel mode functions to their user-mode process, such as virtualisation software, are also at risk. These applications limit access to their user-mode processes obtaining a handle on the driver, thus preventing other processes from using these critical kernel functions. If an external process gains access to the permitted user-mode process possessing a valid handle on the driver, it can then can be exploited as demonstrated by our Virtual Box use case.

In general, a number of applications with high security requirements may be at risk; examples include other virtualisation and anti-malware software, but also banking applications or point-of-sales systems. In the latter two examples,

applications store security-critical information (e.g. full credit card details) in memory which could be retrieved with memory scanning.

While Microsoft does not explicitly promotes *PsSetCreateProcessNotifyRoutine/Ex* to set up this security mechanism, it is to the best of our knowledge required to set up such protection. The different case studies of real-world software presented in this paper confirms this to an extent.

Setting up this protection solely with callbacks is possible, however it is not possible to retrieve information on the process requesting the handle. Consequently, all handles get their permissions modified, including handles for Windows system processes such as *csrss* or *lsass* which either respectively prevent the new process to initialise and run or create various instability and/or crashes. This worryingly indicates that the operating system itself requires the behaviour leading to the vulnerability to function correctly, making patching even more challenging for the kernel developers.

Because the timeline of the different notifications and triggers are not documented, developers may have written vulnerable code by wrongly assuming that kernel notifications should trigger before user-mode notifications, which is not precised on Microsoft API documentation. It is our opinion that kernel notifications should always trigger before user-mode following the protection rings hierarchy.

We thoroughly tested this vulnerability through the case studies discussed in Sect. 4 on Windows 10 x64 and Windows 7 x64 up to date as of November 2019. Furthermore, we hypothesise the vulnerability is most likely present in other Windows versions including Windows 8, all Windows editions and architectures included.

6.4 Responsible Disclosure

We first disclosed the vulnerability to Microsoft in July 2018 following their guidelines [27]. The formal disclosure provided a description of the vulnerability, the code of both exploits [4,6], and the protection driver [7], along with the compiled binaries to allow for the recreation of our experiments. Moreover, we provided a video of the vulnerability in action [9]. The response from Microsoft was produced almost a year after our disclosure and is provided in Sect. 6.5.

In parallel, we also disclosed this vulnerability to the remaining stakeholders from our case studies. In fact, after disclosing our finding to anti-cheat companies, we noticed that they implemented new countermeasures aiming at preventing the exploitation of this vulnerability. Our analysis shows that the affected anti-cheat companies developed a procedure that terminates the first instance of the game launched, then respawns it from kernel space to obtain the handle instantly and set up the callback without delay. This behaviour appears similar to our observations with VirtualBox. Such a solution is sub-optimal for a number of reasons. Overall, a malicious program can still briefly obtain a fully privileged handle on the first instance. Moreover, it is not applicable for all programs. Since this solution requires forceful termination of the protected process, and subsequent respawning from kernel-space, some applications may not

function correctly after such an operation. Furthermore, in order to implement this solution, developers are required to have a signed kernel mode driver, this is not common for most developers. Note however that this solution should be applicable on the use cases previously presented in Sect. 4.

We have also identified the development of an additional defence and detection mechanism that implements a periodic walk-through of the handle table for all running processes. The goal of this mechanisms is to search for open handles on protected processes. Thus, if a handle to a protected process is found, the implemented procedure modifies the granted permissions. It is unclear how anti-cheat companies implement such a process, since it requires using undocumented kernel functions and memory structures to do Direct Kernel Object Modification (DKOM), which is discouraged by Microsoft.

6.5 Microsoft's Response

Microsoft replied to our responsible disclosure and have acknowledged the vulnerability. Unfortunately, Microsoft *"will not be addressing this scenario for in market operating systems via a security update"*. The response decision extends further stating that Microsoft's *"assessment considers this scenario to be a defense in depth against third party products"*. Finally Microsoft acknowledges that fixing this vulnerability *"for in market OS's would potentially result in significant application compatibility issues"*. Microsoft also gave us permission to publicly disclose this vulnerability.

In our opinion, fixing this vulnerability would require changes to the functions themselves, including the parameters they take. Eventually, such an approach would most likely not be retro-compatible and cause problems. However, a new function could be made available with a new name, most likely with the suffix *Ex* or *Ex2* as per Microsoft's tradition, with a security notice placed on the older functions indicating that a newer, more secure function is available as it has been done many times in the past for other vulnerable functions.

In order to efficiently address the discovered vulnerability, the affected kernel function must be modified. Ironically this is made impossible by default due to various security mechanisms preventing any kernel modifications such as Kernel Patch Protection (KPP, also known as PatchGuard). Unfortunately, without Microsoft upgrading the kernel API functions, this vulnerability cannot be adequately fixed in all circumstances. Nonetheless, in the next section we explore possible solutions that, while they will not be able to remove the vulnerability, could significantly mitigate it without requiring kernel modifications.

6.6 Possible Solutions

The most appropriate solution would be to modify the kernel so that user-mode functions taking a PID as a parameter either instantly fails or, more elegantly, get delayed until the process finishes its initialisation, and the kernel routines trigger. This solution can only be implemented by Microsoft.

A solution could be to use notification routines allowing notification of the process to protect being spawned, but then directly terminating the process and spawning it again from kernel space using a function that immediately returns either the PID or a handle, such as *ZwCreateProcess*. This appears to be the solution anti-cheat softwares have set up following our disclosure, and how VirtualBox is spawning the virtual machine's *VirtualBox.exe* dedicated process. This solution is effective but requires a signed kernel driver, which most developers do not have and comes at a cost, in addition to the expertise required for its implementation. It may also be inapplicable in many scenarios, especially if attempting to protect third party processes that are not designed to be forcibly terminated then restarted in a different way. Fortunately, the software in our use cases satisfy these requirements and can implement this solution.

Based on our reverse engineering, anti-cheat software appears to have implemented an additional mitigation in addition to the aforementioned solution. They periodically scan the object table of all processes running on the system, and if a handle to the protected process is found the permissions are then edited accordingly to the desired filtering rules. This is not in our opinion a viable solution, as our exploits would still obtain privileged handles and could use them for a brief moment which is sufficient to tamper with the process. This mitigation also requires the use undocumented internal kernel functions and memory structures, as well as Direct Kernel Object Modification (DKOM) which are highly discouraged by Microsoft. Note that while this solution doesn't fully protect against the vulnerability, it allows detection of exploitation.

It is possible to set up a handle permission filter using callbacks without being notified of newly spawned processes, and therefore without using the vulnerable API functions. Using this method, all handles are filtered and have their permissions modified. Unfortunately, in this case even Window's vital systems processes such as *csrss.exe* have their handle permissions modified, which prevent the protected process to execute correctly. Quite ironically, it seems that Windows itself makes use of this insecure time period to operate correctly. This may be prevented if the driver could collect information about the process requesting the handle from within the callback function, and let Windows system processes acquire unmodified handles, along with possible other white-listed processes. Unfortunately, with the current kernel callback functions it is not possible to retrieve such information, making this solution impossible. Microsoft could implement this solution without compatibility issues by modifying the kernel memory structure *POB_PRE_OPERATION_INFORMATION* to include information about the requesting process. Alternatively, Microsoft could modify the kernel callback API functions, to have an additional parameter allowing for the retrieval of this information, however this would unavoidably lead to compatibility issues due to function parameters and memory structures changes.

7 Conclusions

In this paper we introduce a fundamental security design flaw within the Microsoft Windows OSs. We demonstrated the feasibility of outrunning

Windows kernel process notification routines from unprivileged user-mode processes. Thus, effectively bypassing any protection set in kernel mode following notification routines. Consequently, Microsoft's standard method of protecting user processes via a kernel driver is ineffective. We verified our work on current Windows 7 x64 and Windows 10 x64 up to date up to date as of November 2019.

In order to validate our findings, we implemented our own protection driver and assessed its features. Our findings highlight that the discovered vulnerability can be exploited to bypass protection built for sensitive and widely used applications. We have assessed and demonstrated the aforementioned property by studying the behaviour of the (i) Symantec Endpoint Protection anti-malware suite; (ii) virtualisation environments such as VirtualBox; (iii) anti cheat protection software such as BattlEye and EasyAntiCheat. In addition, solutions to address the vulnerability were presented, namely to change the Windows API for handle requests, respawning the protected process from kernel space to immediately set up protection, scanning object tables system-wide for detection and protection, and providing sufficient information to callback driver functions to avoid using routines.

We disclosed the vulnerability to Microsoft. Microsoft acknowledged the problem but decided against a OS patch. As shown, in response to our work application developers have reacted and implemented unique fixes to their applications. However, we feel that this is an inefficient strategy as the solutions are incomplete, different from case to case, and have to be re-designed for each situation. A comprehensive solution in the form of an OS update from Microsoft would effectively mitigate this vulnerability, however there would be an undesirable cost from a compatibility perspective. Maybe this work serves as a well documented example where security improvements cannot be easily balanced with other industry requirements.

We have made the source code of every binary discussed in this paper publicly available on GitHub [4,6] so developers can assess if their solutions are vulnerable, and attempt to implement additional security on a minimalist protection driver [7] before adding it to their products. A compiled version is also available for quick testing and experimenting. Finally we made a video showing the API functioning normally, then the effects of our exploits [9].

Overall, we argued that the discovered vulnerability is not caused by a simple development bug, but rather a fundamental security flaw deeply ingrained in the OS core design, laying the foundations for a new generation of OS-level attacks.

References

1. Microsoft. Msdn, process security and access rights. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684880(v=vs.85).aspx
2. Russinovich, M.E., Solomon, D.A., Yosifovich, P., Ionescu, A.: Windows Internals, 7th edn. Microsoft Press, Redmond (2018)
3. Microsoft. Msdn, about processes and threads. https://docs.microsoft.com/en-us/windows/desktop/procthread/about-processes-and-threads

4. Anonymous. Github, hfromjob exploit. https://github.com/Anonymous-3ab41c/FastAndFurious/blob/master/hFromJob.cpp
5. Margosis, A., Russinovich, M.E.: Troubleshoot with the Windows Sysinternals Tools. Microsoft Press, Redmond (2018)
6. Anonymous. Github, hthemall exploit. https://github.com/Anonymous-3ab41c/FastAndFurious/blob/master/hThemAll.cpp
7. Anonymous. Github, protectiondriver. https://github.com/Anonymous-3ab41c/FastAndFurious/blob/master/ProtectionDriver.c
8. Microsoft. Msdn, obregistercallbacks function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-obregistercallbacks
9. Anonymous. Youtube, outrunning windows kernel notification routines from user-mode. https://youtu.be/dp_GqDfUUxA
10. Liu, W.J.: (wj32). Process hacker. https://processhacker.sourceforge.io/
11. Microsoft. Protected processes. http://download.microsoft.com/download/a/f/7/af7777e5-7dcd-4800-8a0a-b18336565f5b/process_Vista.doc
12. Microsoft. Early launch antimalware. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/early-launch-antimalware
13. Microsoft. Mandatory integrity control. https://docs.microsoft.com/en-us/windows/desktop/secauthz/mandatory-integrity-control
14. Microsoft. Understanding and working in protected mode internet explorer. https://technet.microsoft.com/en-us/windows/bb250462(v=vs.60)
15. Microsoft. Appcontainer isolation. https://docs.microsoft.com/en-us/windows/desktop/SecAuthZ/appcontainer-isolation
16. Seifert, C., Steenson, R., Welch, I., Komisarczuk, P., Endicott-Popovsky, B.: Capture-a behavioral analysis tool for applications and documents. Digit. Invest. **4**, 23–30 (2007)
17. Chiang, K., Lloyd, L.: A case study of the rustock rootkit and spam bot. HotBots **7**, 10 (2007)
18. Liang, S.C.: Understanding behavioural detection of antivirus (2016)
19. Jeong, H.C., Im, C.T., Oh, J.H.: Malware auto-analysis system and method using kernel callback mechanism, US Patent App. 12/942,700, 29 March 2012
20. Liu, Z., Chen, P.: Improved method of packet filtering. In: Proceedings, The 2009 International Symposium on Web Information Systems and Applications (WISA 2009), p. 294. Academy Publisher (2009)
21. Perez, D.S., Balinsky, H., Simske, S.J.: Injection of data flow control objects into application processes, US Patent App. 14/917,839, November 24 2016
22. Aussel, J.-D.: Portable mass storage device with hooking process, US Patent App. 12/667,196, 22 July 2010
23. Pektaş, A., Acarman, T.: Portable dynamic malware analysis with an improved scalability and automatisation. In: Kurzynski, M., Wozniak, M., Burduk, R. (eds.) CORES 2017. AISC, vol. 578, pp. 211–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-59162-9_22
24. Shochat, E., Elkind, D.: Real time monitoring of computer for determining speed of various processes, US Patent 8,307,246, 6 November 2012
25. Teller, T., Segal, A.: Method of defending a computer from malware, US Patent 9,536,090, 3 January 2017
26. Leitch, J.: Process hollowing (2013)
27. Microsoft. Microsoft, report an issue faq. https://www.microsoft.com/en-us/msrc/faqs-report-an-issue