Laboratory 1: Rootkit Creation

Part I: Creating a rootkit on Linux

There are different ways to hide processes on Linux, the most reliable and commonly used by rootkits passes by creating a kernel module to accomplish this. However, there are other, more simplistic ways to achieve this. Then, to develop this Linux rootkit we will do a very simple hack, consisting on PRELOADING system libraries and override some basic system calls.

OK, since we want to hide a process, first should see how process listing applications work. To do so, first let's investigate the calls to our beloved **libc** involved on an execution of the command. For that we will use the **strace** application:

```
$ strace ps
execve("/usr/bin/ps", ["ps"], 0x7ffffabfbfe0 /* 79 vars */) = 0
brk(NULL)                               = 0x5614affe5000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffca0dd5cf0) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=231324, ...}) = 0
mmap(NULL, 231324, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f337abe1000
close(3)                                = 0
openat(AT_FDCWD, "/lib64/libprocps.so.8", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\\\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=87360, ...}) = 0
…
```

Investigate the whole trace by running the above command on a Linux console.

If you look closely you'll realize that the command tries to list the contents on the **/proc** directory. In fact, as you may (should?) know, all processes in Linux have an entry on the **/proc/$PID** directory, where all the running process information is stored.

So, it seems that there are some calls that allow ps to iterate through all the entries on **/proc**. In particular, **ps** uses the **openat** and **getdents** calls. Nice, however these are... system calls, not **libc** functions. It is important to notice that **libc** wraps those system calls to ease its use, and depending on the situation, cache some queries to optimize its access. We may want to read the **libc** documentation, there we may see, that the **libc** functions wrapping **getdents** and **openat** are **opendir** and **readdir**. Nice!

So, we may want to override these functions.

Now, how can we override the **libc** functions with some stub that allows us to override its normal behavior?

Quite easy actually, we just need to do the following (pseudo-code ahead):

1. Define a static global function pointer on your lib (initially set to null) of a **“orig_readdir”** function with the same signature as **readdir** → use man to discover the signature
2. Define a **readdir** function following the original specification
3. The first time the **readdir** function is invoked we should initialize the **orig_readdir** function pointer to the original **readdir** function, we will use this as hook point as both functions have the same name
    - This can actually be achieved by using the dynamic loading function symbol resolution, e.g., **dlsym**, while assigning the return value to **orig_readdir**, go ahead and check the man page to understand how **dlsym** works
4. Now the easy part, invoke the **orig_readdir** function as it was legitimate
5. Check the given process name, if the process name is the process we want to hide, just iterate again to the next dir entry and return it, effectively skipping the “hidden” process

Nice!, go and code this application. It shouldn’t be more than 100 lines of C code. We know you love it!

Since this needs to be a library we should compile it using the proper command line. Feel free to create a **Makefile** if that’s OK for you:

```
$ gcc -fPIC -shared -o library_name.so source.c -ldl
```

Of course change **library_name** and **source** with fancy names you like.

Now, compile the application and run the **ps** function as follows:

```
LD_PRELOAD=$(pwd)/library_name.so ps
```

It should hide the process you defined on your code!

If you want to make the change persistent forever, just edit the file **/etc/ld.so.preload** appending at the end the full path of your sexy process hider!

It is very important to notice that this technique is not perfect, if any application uses a different method to access to the process list, for example through kernel modules, or using another mechanism to open the directory and reading from it, even if the binary is statically linked!, the solution wouldn’t work. Luckily the vast majority of software uses this method to list the system processes.

For further reference, think that apps such as **netstat**, use similar methods over different entries on **/proc**, hence we could apply this technique to further hide our backdoor!

Part 2: Creating a rootkit on Windows

OK, this second part of the lab, will consider how can we implement a similar rootkit, well, the hiding part at least, on a Windows 10 system.

It is important to notice, that sadly enough, the rootkit we will show on this lab will not work in general. Windows has a subsystem, namely, Kernel Path Protection (KPP), which detects the hack we are about to run. Luckily, this runs every few minutes, hence showing the patch working, but later crashing the machine with **`CRITICAL_STRUCTURE_CORRUPTION`** error.

This highlights a couple of things about security:

1. It's a race of good versus bad
2. Even a secured system may have opportunities to be exploited, even if partially

For good or bad, to do this lab we will need some background and to install some stuff on our system. Let's go step by step.

(Hands on) Background

The goal of this lab is to hide a process within the Windows operating system. To do that we will use a technique called *Direct Kernel Object Modification* (DKOM).

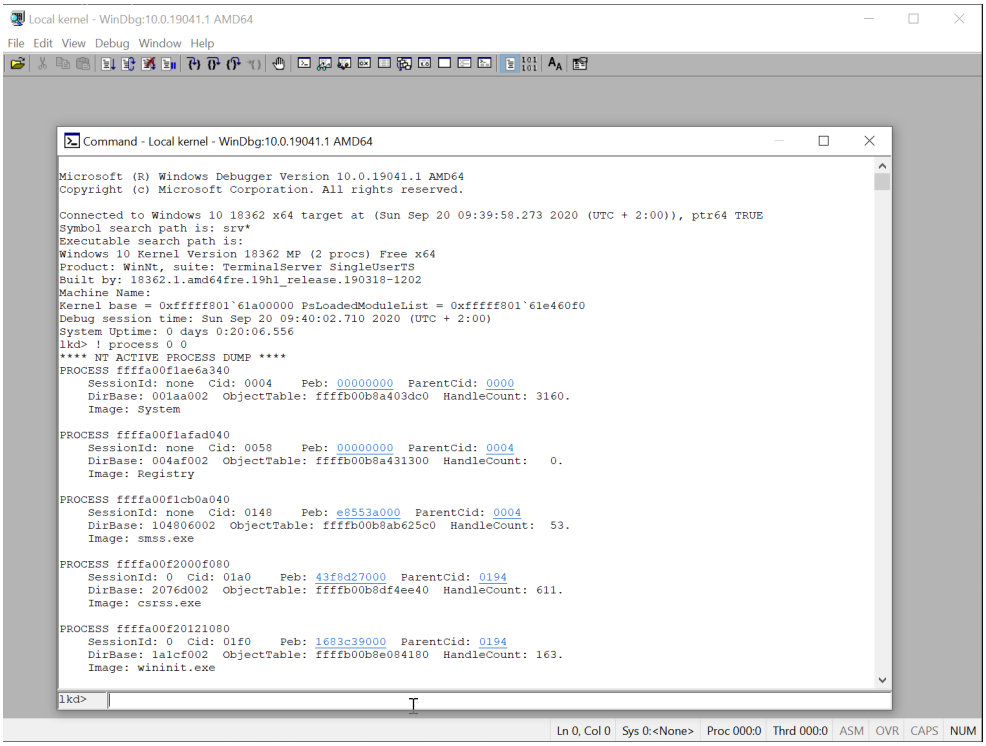
To do this, we need to understand the **`EPROCESS`** structure, and most importantly, how can we obtain the list of all processes through it programmatically.

First we have to fire up the Windows kernel Debugger: WinDBG, which is part of the Windows SDK. Be sure you launch the version that matches your running architecture x86 or x64, and don't forget to run it with Administrator privileges. You may find the user manual in here.

Once there you have to select kernel debugging (Ctrl–K) and select local machine (you can do that remotely as well from your physical machine if it's windows).

Then, we can investigate the **`EPROCESS`** list very easily using:

```
lkd> !process 0 0
```



Then, if we want to investigate further one of those processes:

```
lkd> dt -b -v nt!_EPROCESS [address]
```

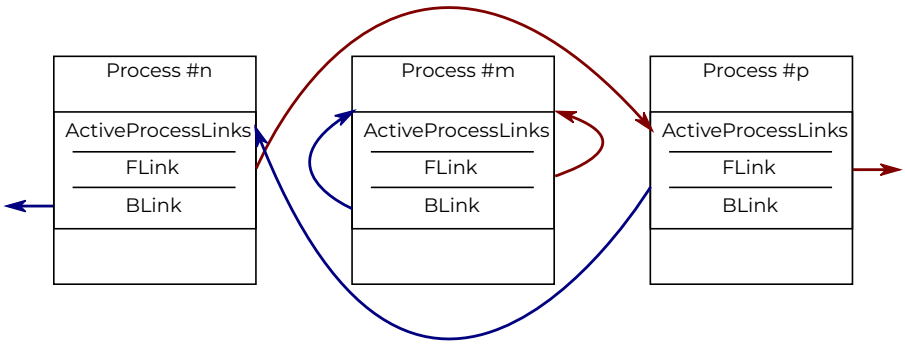
```
lkd> dt -b -v nt!_EPROCESS ffffa00f1ae6a340
struct _EPROCESS, 226 elements, 0x880 bytes
   +0x000 Pcb              : struct _KPROCESS, 46 elements, 0x2e0 bytes
      +0x000 Header           : struct _DISPATCHER_HEADER, 58 elements, 0x18 bytes
         +0x000 Lock             : 0n12058627
         +0x000 LockNV           : 0n12058627
         +0x000 Type             : 0x3 ''
         +0x001 Signalling       : 0 ''
         +0x002 Size             : 0xb8 ''
         +0x003 Reserved1        : 0 ''
         +0x000 TimerType        : 0x3 ''
         +0x001 TimerControlFlags : 0 ''
```

It seems our system has 226 fields, of course depending on your Windows version this value may change.

For our purpose we will focus on several fields, which you should check on your own: **`UniqueProcessId`**, **`ImageFileName`**, and the **`ActiveProcessList`**, a **`_LIST_ENTRY`** with **`Flink`** and **`Blink`**.

Particularly we are interested on their offsets (so we can later access them from our rootkit). Remember that processes on windows are stored as a circular list.

Then, to summarize, what we want to achieve is to iterate over all the different processes, checking the ImageFileName for matches with some predefined value (is not the purpose of this lab to construct a full userspace kernel communication system), and then removing the element from the list. For example our final goal is depicted on the following figure:



Software installation

Before starting we will assume you have access to a VM with Windows 10. You can get one for free from:

- https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/
- https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/
- Or even at: https://distribuciosoftware.upc.edu

This lab is a little tricky, because talking to the kernel may only be achieved through enabled interfaces, basically, **`ntdll.dll`** or **`kernel32.dll`**, or through the development and install of a kernel driver.

To have the capabilities of compiling kernel drivers, we need several Microsoft tools to do so. Let's get our hands a little dirty:

- **`Visual Studio Community 2019`**: https://visualstudio.microsoft.com/es/downloads/
- **`Microsoft SDK`**, version 10.0.19041.1:

  https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk/
- **`Windows Driver Kit`** – Windows 10.0.19041.1:

  https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk

Good luck :). We are not fond of making you install all this bloat. However, this also highlights the tools generally needed to develop this kind of software we are doing.

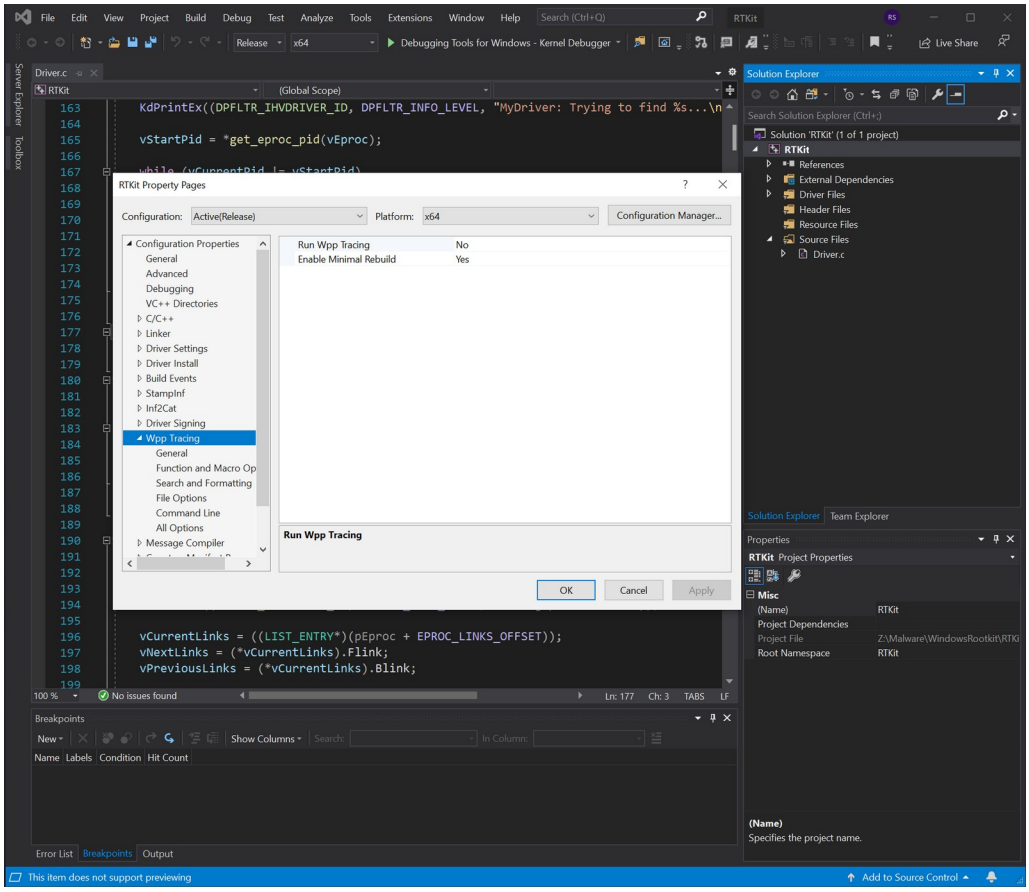
By the way, don't forget to install the Visual Studio integration present on WDK will make your life a lot easier. For more information about how to create kernel drivers feel free to check here.

Kernel Driver Development

Let's start the fun. We will assume you installed all the above software and are familiar with Lesson 1 and Background section above.

Now, to develop the Driver, it is actually pretty simple, Visual Studio should provide you with some templates. In this case, we will use the most basic of all Windows Kernel Mode Driver Frameworks: **`Kernel Mode Driver, Empty (KMDF)`**.

Once there, you can go to the project settings and disable tracing:



As we will not be using it. Now you should create your driver using both the offsets and the logic we explained before. Your whole code should not exceed 250 lines of code, probably way less. How hard can be?

Our recommendation is that you hard-code as many parameters as possible, since interacting with a kernel driver is something we will not cover on this course. But you can browse around the network in case you are interested on how to do it!

Deploying the driver

OK, now things will start to get messy. If you want to debug your code you have various options, neither beautiful. You'll basically need a second machine, one for development (the

physical machine?), or two different VMs. If not, well, you are on your own, since any crash will actually crash your machine probably. A debug helper may be the **KdPrintEx** primitive whichwill leave useful logs into the Event Log in Windows.
To deploy the driver follow these instructions:
https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/provision-a-target-computer-wdk-8-1

The problem is that since we don't have a digital certificate, windows will complain (hence on of the complexities of this kind of exploit) and not install the certificate. The only option is to enter Testing mode. To do that just run on a Administrator enabled Powershell session and run:

```
bcdedit.exe /set testsigning on
```

Then reboot the system. You'll see the system in testing mode now, then you can just deploy the driver either by right clicking on the INF file created by Visual Studio and selecting install or by using the CLI and running the **devcon.exe** application located in the Windows Kits directory on **Program Files (x86)**.

Before closing the laboratory two important considerations:

- Probably your machine will crash after a while given the KPP discussed above.
- Depending on how you developed the rootkit, it will run just once on initialization time (**KmdfHelloWorldEvtDeviceAdd**), this means that you'll only be able to hide the process once, please make sure you start the application (for example **notepad.exe**) BEFORE starting the kernel driver.