OREGON STATE UNIVERSITY

CS 373

WINTER 2019

Week 5

Author:
Thomas Noelcke

Instructor:
D. Kevin McGrath

I. Overview

This week we are diving deeper into windows internals. This week we will discuss the details of windows internals and the Windows kernel. This week includes the lowest level discussion we will have in this class. As part of this discussion we are also going to talk about root kits. In this write up I will include notes about the lectures.

II. Lecture Notes

In this section I will include topics that I found interesting or relevant during lecture.

III. A bad Exploit

The first topic we started talking about during the lecture was talking about the sony hack where they tried to block people from stealing the music. This was done by running a file called $sys.bad.exe. This was all fine and good until hackers realized that they could do this too and code was running unchecked by security systems.

IV. This weeks Goal

This week there are only two goals. Day one understand concepts of stealth by means of memory manipulation. Day two debug and explore various stealth techniques. Essentially this week is all about what attackers to do try to hide there software once it on a system and running.

What are rootkits? how are rootkits created? Before we can answer this question we need to understand some operating systems basics. Root kits were really popular some time ago. This is because they were hard to detect via security systems. However, the security industry got better at catching root kits. However, they are still used to try and hide malicious code. Root kits however have become less common. So why ware we still looking at root kits? This is because this gives us a change to examine the windows kernel and various security content in the windows kernel. It's also important to note that root kits aren't just a problem on windows either. There are root kits on mac and Linux as well. ATP attacks are also feature root kits as they allow exploits to stay hidden. Why are root kits less common than they used to be? Its harder for root kits to enter the 64 bit kernel. There have been some security improvements such as code signing. It used to be that you could just have any old code written into the kernel and it would execute no problem. However, today it is required that this code gets signed.

V. Sections of the OS

There are layers to the operating system. Most of the code is running in user land. However, at some point the application will need to interact with the kernel to have certain operations carried out for them. The user land code lives in dlls. The kernel files all live in sys files. Root kits are in sys files in the kernel mode of the OS. This is because this gives the root kit super user like privileges. Generally, we want to avoid root kits from being loaded in the first place. Once they are loaded and running we are on an equal playing field.

VI. Windows Debugging

After talking about some basic windows internals we jumped into the lab. In this lab the goal was to run the sample and try to see the traces of the piece of malware. We can do this using many different types of tools. The point of the exercise is to see if the tools we have can help us find what the malware is doing. This is important because the malware is running at the kernel level and will try to hide so finding it might be difficult. There were some files made on my system as well along with the lecturer. It doesn't seem to me that the malware did a whole lot else given the tools we had I couldn't really derive truly what they program was doing. There was however a sys file near where we ran the malware. This was not found on most of the different tools we had. The dir command was an option that allowed us to find a file that the malware created.

VII. How files are hidden

The malware created some system files that we could not be found via typical dir *.* command. This is because the dir command is run in user mode. We need to pass the dir command some arguments that allow it to look at files that are only available at the system level. That is because this file is an anti query directory file. This may also hide the file from the anti virus as well as other applications. The malware also created an anti enumeration file. This prevents the file from being enumerated and from being seen from the process manager.

VIII. livekd

This is a third party piece of software that allows us to debug the kernel and see things from the kernel level. This will allow us to find out more about this root kit running on our system. This software is capable of reading kernel memory and preforming various different options. This will also allow us to unassemble memory so we can see what is running on our system.

Below are some handy commands for livekd:

- U command -
- .cls - clears the screen.
- lm - Lists all the modules in the kernal memory along with addresses.
- you can dump the kernal memory to a file where you can look at it later.
- dps - gives you a list of all system API's that the system supports.

IX. how to we see the hidden file

Using a tool we can patch back the files to the original return address that prevents the file from hijacking the process using stack manipulation. We will need to do this because livekd can't intercept the process or debug the malware while it is running. We also need to be very careful when patching things back to the kernel. It's possible that something else is calling this function. We won't go into it during this lecture but there are good reasons that we want to make sure we are good to patch the file back. We can use the DPI command to help us detect this type of condition.

X. NAMING

why don't hackers just copy the windows naming convention? Or why can't they hide from us once we are in the kernel? This is because the nt kernel are going to be at the same addresses most of the time. Or at least they are at the same offset.

XI. THREADS

What is a thread? A thread is the smallest unit of execution on a system. The bigger and better question is how is a thread executed on a system? Most of the programs run on a system are programmed in a high level language. These languages will need to be compiled down to assembly code. This will then get linked down to machine code. This may require some memory calculations as the memory references won't match. This is one program and one thread. CPU's can handle a lot more than this, so what happens when there are multiple processes? This is handled by the thread scheduler. Assuming that both threads have same priority the thread scheduler will execute the code for a given unit of time called a time slice. This gets more complicated as we add different priority levels. Lets assume that our two processes had different priorities. Clearly the higher priority thread would get more time slices on the CPU than the lower priority thread. However, the idle thread's priority will elevate as it is sitting and waiting for CPU time. This ensures that it get some amount of CPU time even if its a lower priority.

The process above was farther enhanced using hyperthreading this essentially added a second instruction pipeline. This did help make it faster for some application. However, having more cores is what really made a vast improvement of the efficiency of the system. Applications have taken advantage of this by creating more threads. This is called multi threading. Threads are managed through various different windows internals. We are not going to in depth with this in this lecture. One important concept we need to talk about is thread context. While a thread was running it did some work. This work will need to be saved so that the thread can be picked when it gets CPU time again. This is essential for multi threading. We will talk about this in a bit more detail later.

In livekd there is a td that will show all of the running threads. All of these threads will have their own stack. This is important to note for debugging. Threads are not shared between the kernel level and user land. The only thing that is done so that this thread can be transferred between the kernel and user land is that the data is copied. This is not a one to one copy but rather a transformation of some kind.

Its also important to note that threads and processes are not the same thing. Programs run in process not threads. Process can have multiple threads. Processes need at lease one thread to execute but is not limited to one thread. The threads in a single process will have some shared resources that are available to all the threads.

XII. LAB 2

In this lab we are getting more exposure to what process memory looks like. We are doing this using several different tools. We ca use either process explorer or process hacker. Both of these tools are similar and are suitable for this lab. While looking through the memory for an application we will see a section called image. This is the meat of the memory. All the other memory is either part of the heap or the stack. It is also important to note that in Windows to make the memory more secure windows has tried to give memory pages a permission level. Some are read only, some are write, some are read write and some can be executed. This tries to enforce the separation of executable and the data in the system. While doing lab to we also came across some interesting memory pages on a piece of software. The page did not have a name and was not an executable. This can be one of two things. This is either security code trying to be cleaver or it is something bad. More often than not it is something bad.

XIII. Boot Kits

Viruses started off as boot kits. The first root kit was written by to Pakistani brothers. The first boot kit was called brain and was installed via a floppy drive. Since then there have been many different boot kits. Notably there was a famous incident from Sony. This proves that these boot kits are still threats and we need to be aware of them. Another type of boot kit that has come up recently is a boot kit that password protects the start up process. This isn't that complicated but many people fall for it.

XIV. Boot Process

Below are a list of steps that happen during the boot process.

- The PC is turned on and the BIOS initializes the hardware.
- The bios calls code stored in the MBR at the start of disk 0.
- The mbr loads the code from the bootsector of the active partition.
- the bootsector loads and runs from its file system.
- once the bootloader is finally executed it loads its configuration from files on the same partition.
- The bootloader optionally presents the user with a list of operating systems as loaded from the configuration file.
- the boot loader locates and loads the kernel for the selected OS from the disk and hands off control of th PC to the OS.