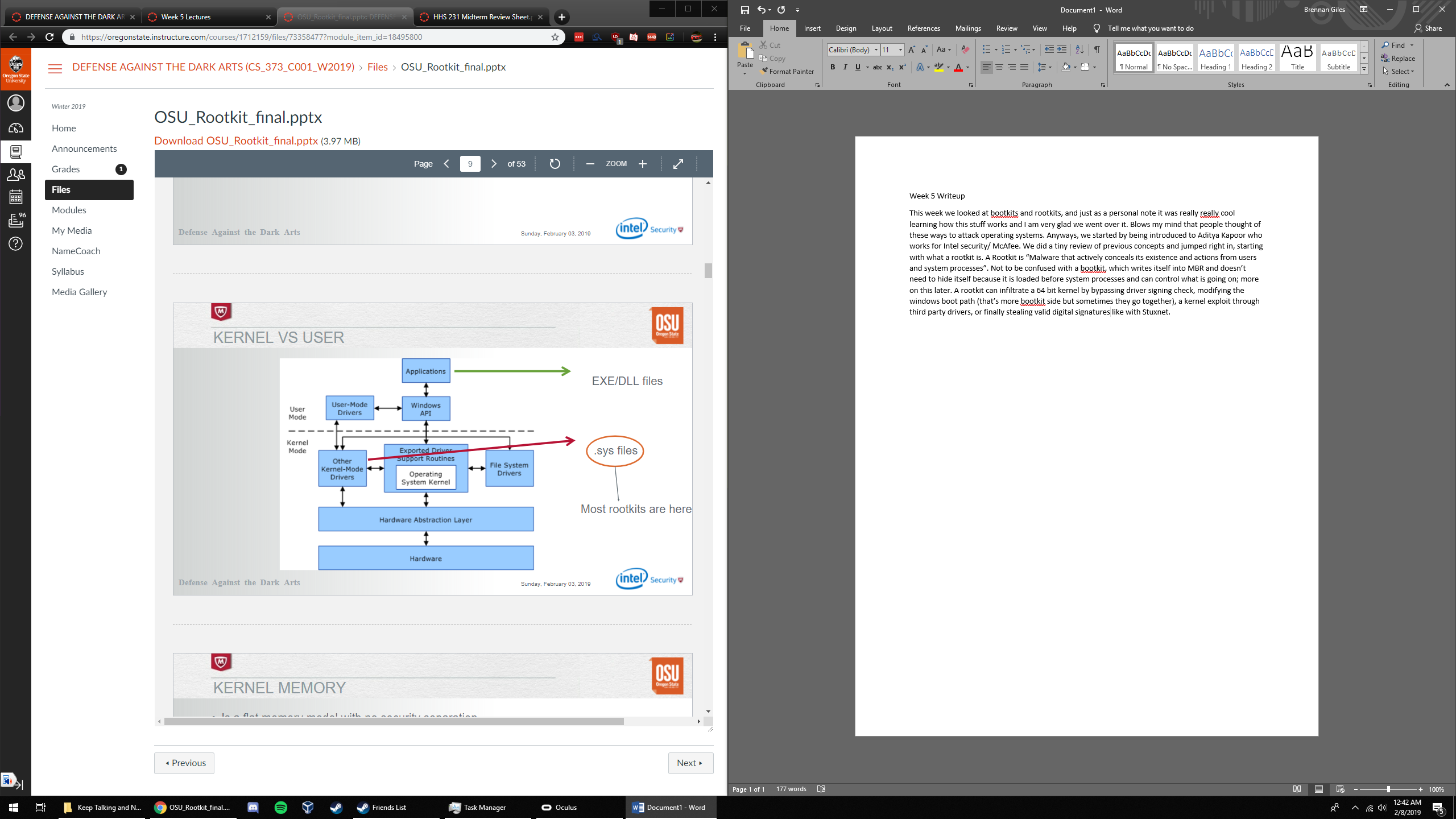
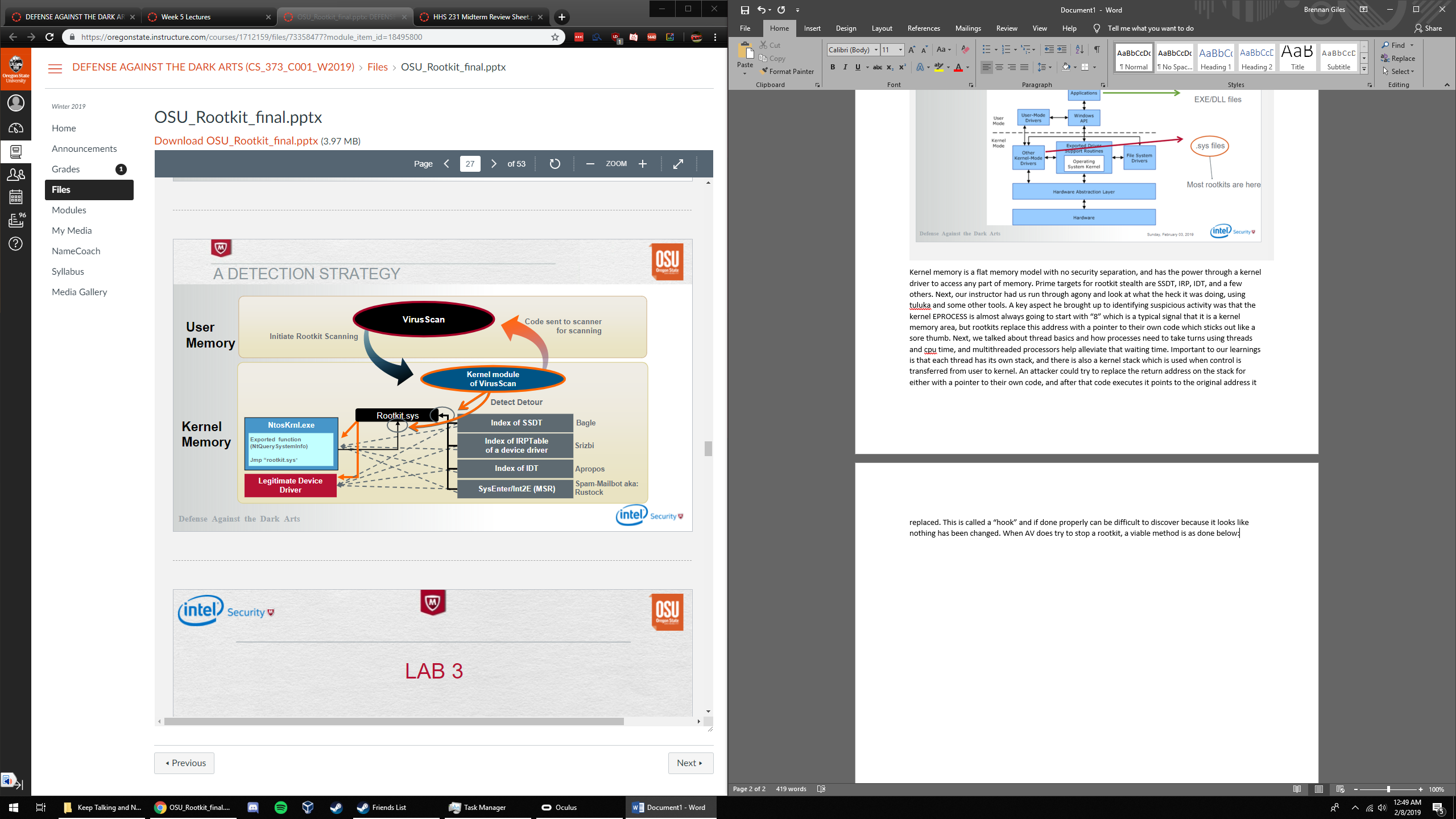
Week 5 Writeup

This week we looked at bootkits and rootkits, and just as a personal note it was really really cool learning how this stuff works and I am very glad we went over it. Blows my mind that people thought of these ways to attack operating systems. Anyways, we started by being introduced to Aditya Kapoor who works for Intel security/ McAfee. We did a tiny review of previous concepts and jumped right in, starting with what a rootkit is. A Rootkit is “Malware that actively conceals its existence and actions from users and system processes”. Not to be confused with a bootkit, which writes itself into MBR and doesn’t need to hide itself because it is loaded before system processes and can control what is going on; more on this later. A rootkit can infiltrate a 64 bit kernel by bypassing driver signing check, modifying the windows boot path (that’s more bootkit side but sometimes they go together), a kernel exploit through third party drivers, or finally stealing valid digital signatures like with Stuxnet.

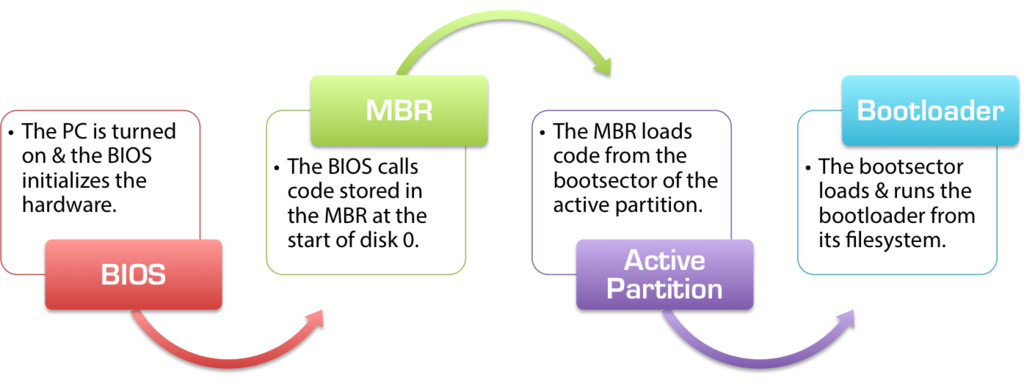


Kernel memory is a flat memory model with no security separation, and has the power through a kernel driver to access any part of memory. Prime targets for rootkit stealth are SSDT, IRP, IDT, and a few others. Next, our instructor had us run through agony and look at what the heck it was doing, using tuluka and some other tools. A key aspect he brought up to identifying suspicious activity was that the kernel EPROCESS is almost always going to start with “8” which is a typical signal that it is a kernel memory area, but rootkits replace this address with a pointer to their own code which sticks out like a sore thumb. Next, we talked about thread basics and how processes need to take turns using threads and cpu time, and multithreaded processors help alleviate that waiting time. Important to our learnings is that each thread has its own stack, and there is also a kernel stack which is used when control is transferred from user to kernel. An attacker could try to replace the return address on the stack for either with a pointer to their own code, and after that code executes it points to the original address it replaced. This is called a “hook” and if done properly can be difficult to discover because it looks like nothing has been changed. When AV does try to stop a rootkit, a viable method is as done below:



After this, we went over some more examples and WinDBG work.

Next, we started looking at how the system boot process works. He had a great image that had the entire process but It is not in the slides and I can only find the top half on the internet:



The good news is it covers MBR which is all I need to talk about anyways. So, Pre-Boot there is a power on self test to diagnose memory, hardware, and load the bios. The bios initializes the hardware and calls MBR, which is at the start of disk 0. MBR finds the active partition and loads the boot sector in memory to execute it. It turns out there is a small amount of space within the chunk allocated to MBR that can be replaced without causing any problems, and bootkits seek to take advantage of this. Bootkits overwrite on disk 0 where MBR is called to execute their own code, which happens so early in the boot up process that OS can’t even stop it because it hasn’t been started up yet! The bootkit (may) set up a rootkit and hides it cleverly. Furthermore, it can record original addresses and memory and redirect user queries so that every time we want to look at where the rootkit is, it intercepts the request and sends us the original data (hiding itself in the process). Some rootkits/ bootkits even hide/deactivate themselves whenever AV, ProcMon, VM software, or other security tools are present. Sometimes all it takes is the time of day!

I really enjoyed the material this week, can’t wait for the next one.

-Brennan