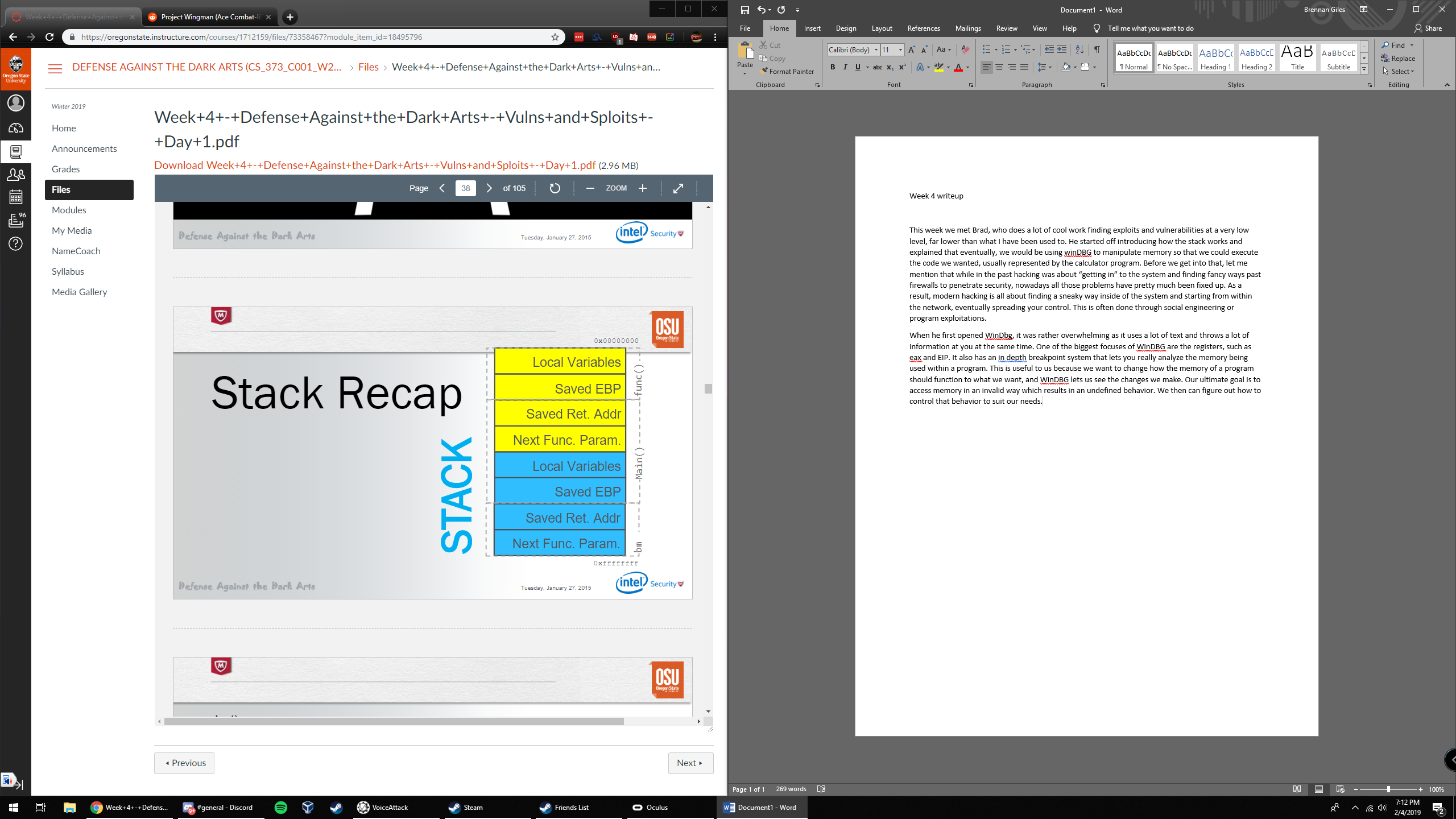
Week 4 writeup

Brennan Giles

This week we met Brad, who does a lot of cool work finding exploits and vulnerabilities at a very low level, far lower than what I have been used to. He started off introducing how the stack works and explained that eventually, we would be using winDBG to manipulate memory so that we could execute the code we wanted, usually represented by the calculator program. Before we get into that, let me mention that while in the past hacking was about “getting in” to the system and finding fancy ways past firewalls to penetrate security, nowadays all those problems have pretty much been fixed up. As a result, modern hacking is all about finding a sneaky way inside of the system and starting from within the network, eventually spreading your control. This is often done through social engineering or program exploitations.

When he first opened WinDbg, it was rather overwhelming as it uses a lot of text and throws a lot of information at you at the same time. One of the biggest focuses of WinDBG are the registers, such as eax and EIP. It also has an in depth breakpoint system that lets you really analyze the memory being used within a program. This is useful to us because we want to change how the memory of a program should function to what we want, and WinDBG lets us see the changes we make. Our ultimate goal is to access memory in an invalid way which results in an undefined behavior. We then can figure out how to control that behavior to suit our needs.

I thought this diagram was useful so I’ll include it here. Brad made a “human stack” to help explain what we were trying to do, which was basically this: Cause a stack overflow so that EBP points to what we want (4141414141141) then we can use the crash triage. The crash triage is

1. Determine the return address offset
2. Position our shellcode
3. Find the address of our shellcode.

After we pop ebp and return the 4 bytes on top, Brad tells the EIP to jmp to esp where the shellcode is situated. Once the function returns to EIP it runs the shellcode instead!

Later, Brad spends a surprisingly long amount of time reliving the 90’s. Eventually we get to the heap, and how heapalloc() will use different versions according to the size required. For our purposes the most important is VirtualAlloc() but we will get there in a second. In order to defeat the stack we will use the Use-After-Free method, which goes

1. Free the object
2. Replace the object with ours (need size)
3. Position our shellcode
4. Use the object again

Ok so in order to position our shellcode in an unpredictable world, we need to make it predictable. We do this by allocating with VirtualAlloc() a huge string (1MB) which results in a big piece of filler on top of the heap. At a certain point all this filler starts a predictable pattern of where in memory it is, and this allows us to put our shellcode inside of it. Once we call the object again it will point to somewhere in the filler and we put the shellcode at that spot. Now we can pull up the calculator!

This week was a little tough for me so if I am misunderstanding these concepts please let me know.