Windows Internals Exercises

Debugging, Data Model

1. Attach a local kernel debugger (or a remote debugger if you have a Windows 10 Virtual Machine) and configure symbol path.
    a. Using a remote debugger will allow you to set breakpoints but your host probably has more interesting things running on it.
2. Variables and casts:
    a. Declare a new variable and save an integer inside it.
       dx @$myInt = 5
    b. Declare a new variable and save a string inside it.
       dx @$myStr = “Hello!”
3. Get the base address of NT (easiest to do using the legacy command “lm vm nt” – at this point). Using DX cast the base address to nt!_IMAGE_DOS_HEADER and save the result in a new variable.
   lm vm nt → copy “start” address
   dx @$kernel = <Address you copied>
   dx @$dosHeaderNt = (nt!_IMAGE_DOS_HEADER*)@$kernel
    a. Step 2: cast the value of the e_magic field into a string to show the magic “MZ”.
       dx ((char*)&(@$dosHeaderNt->e_magic)).ToDisplayString("s").Substring(0, 3)
    b. Advanced: Use DX to parse the image header of an image of your choice to print its exports.
       dx @$curprocess.Modules[4].Contents.Exports → 4 can be replaces with anything else
    c. Using .Modules attribute of a process object: instead of getting the base address using “lm” and then hard-coding it, use @$curprocess.Modules to find the base address of NT, cast it into the needed type and save it in a new variable.
       dx @$kernel = @$curprocess.Modules.Where(m => m.Name.StartsWith("nt")).First().BaseAddress → then use @$kernel in commands shown earlier.
4. Processes and threads:
    a. Print the name and address of the KernelObject for the current process.
       dx new {Name = @$curprocess.Name, KernelObject = &@$curprocess.KernelObject}
    b. Print the CreateTime field for all the threads running in the current process (ETHREAD->CreateTime).
       dx @$curprocess.Threads.Select(t => t.KernelObject.CreateTime)
        i. Want to show human-readable time? Use ExecuteCommand:
           dx @$curprocess.Threads.Select(t => Debugger.Utility.Control.ExecuteCommand("!filetime " + t.KernelObject.CreateTime.QuadPart.ToDisplayString("x"))[0])
    c. Print a grid view with the ID and start address of all threads running in the current process (for start address: ETHREAD->StartAddress).
       dx -g @$curprocess.Threads.Select(t => new {Id = t.Id, StartAddress = t.KernelObject.StartAddress})

   d. Print the address of the current process token (found in EPROCESS->Token).
      dx @$curprocess.KernelObject.Token
      i. Advanced: cast the token pointer to a TOKEN structure and print the process' EnabledPrivileges.
         Tip: the token is represented as an EX_FAST_REF union. The bottom 4 bits are the RefCnt field – you need to zero them out to get the actual token pointer.
         dx ((nt!_TOKEN*)(@$curprocess.KernelObject.Token.Object & ~0xf))->Privileges.Enabled

5. Using Select() and Where():
   a. Print all processes that have DisableNonSystemFonts mitigation enabled.
      dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlagsValues.DisableNonSystemFonts)
   b. Using @$curprocess.Modules project a table of all the module names and base addresses that are loaded into the current process
      dx -g @$curprocess.Modules.Select(m => new {Name = m.Name, Base = m.BaseAddress})
      i. Add to the grid the number of imports, exports and delay imports for each module and order the grid by number of delay imports.
         dx -g @$curprocess.Modules.Select(m => new {Name = m.Name, Base = m.BaseAddress, Imports = m.Contents.Imports.Count(), DelayImports = m.Contents.DelayImports.Count()}).OrderBy(m => m.DelayImports)
      ii. Choose one module and create a grid view of all the functions it exports.
         dx -g @$curprocess.Modules[0xf].Contents.Exports.Select(e => new {Name = e.Name, CodeAddress = e.CodeAddress})
   c. For every process in the system, print its name, ID and whether it has CFG enabled or not
      dx -g @$cursession.Processes.Select(p => new {Name = p.Name, Id = p.Id, CFG = p.KernelObject.MitigationFlagsValues.ControlFlowGuardEnabled})
   d. Print the impersonation level of every thread in the current process (ETHREAD->ClientSecurity.ImpersonationLevel)
      dx @$curprocess.Threads.Select(t => t.KernelObject.ClientSecurity.ImpersonationLevel)
   e. Using the PEB (process.Environment) print the ImageBaseAddress of every process in the system.
      dx @$cursession.Processes.Select(p => p.Environment.EnvironmentBlock.ImageBaseAddress) → This might not always give correct results since the debugger doesn't always switch to the context of the target process correctly

6. Arrays:
   a. @$curprocess.Environment.EnvironmentBlock.KernelCallbackTable is an array of function pointers. Cast it to an array and save the result in a new variable. (notice: there are some processes that don't have a PEB or a KernelCallbackTable so make sure to pick one that has both)
      dx @$kernelCallbackTable = (__int64(*)[100])@$curprocess.Environment.EnvironmentBlock.KernelCallbackTable, x

            i. Advanced: use the helper function that you'll write later using ExecuteCommand() to print the symbol for each of these pointers.
            dx @$getSym = (x => Debugger.Utility.Control.ExecuteCommand(".printf \"%y\", " + x.ToDisplayString("x"))[0])
            dx @$kernelCallbackTable->Select(x => @$getSym(x))

7. Synthetic methods:
    a. create a method that takes in a process (a natvis process object) and returns the number of threads it contains. Can either use .Threads.Count() or .KernelObject.ActiveThreads.
    dx @$getThreadsCount = (p => p.Threads.Count())
    b. Advanced – using ExecuteCommand() – Write a function that takes in an address and prints the symbol matching it. Can use "ln" or "printf %y" to achieve this. (try to do this, it will be useful for many things later)
    dx @$getSym = (x => Debugger.Utility.Control.ExecuteCommand(".printf \"%y\", " + x.ToDisplayString("x"))[0])
    c.
8. Lists:
    a. Print the list of processes in the system starting with nt!PsInitialSystemProcess.
    dx Debugger.Utility.Collections.FromListEntry((*(nt!_EPROCESS**)&nt!PsInitialSystemProcess)->ActiveProcessLinks, "nt!_EPROCESS", "ActiveProcessLinks")
    b. Starting from @$curthread.KernelObject.ThreadListEntry print a list of all threads in the current process (type = ETHREAD).
    dx Debugger.Utility.Collections.FromListEntry(@$curthread.KernelObject.ThreadListEntry, "nt!_ETHREAD", "ThreadListEntry")
    c. Starting from @$curprocess.KernelObject.Job->JobLinks, print a list of all jobs in the system (type = EJOB).
    dx Debugger.Utility.Collections.FromListEntry(@$curprocess.KernelObject.Job->JobLinks, "nt!_EJOB", "JobLinks")
    d. Starting from @$curprocess.KernelObject.Job->ProcessListHead, print a list of all processes in the current job (type = EPROCESS, linking field: JobLinks).
    dx Debugger.Utility.Collections.FromListEntry(@$curprocess.KernelObject.Job->ProcessListHead, "nt!_EPROCESS", "JobLinks")
        i. Try this for different processes and see how many processes different jobs contain.
        dx -g @$cursession.Processes.Select(p => new { Name = p.Name, ProcessesInJob = Debugger.Utility.Collections.FromListEntry(p.KernelObject.Job->ProcessListHead, "nt!_EPROCESS", "JobLinks").Count()})
    e. Try to find different process lists and see if there's a difference in the amount of processes you get in each list. Try the process list starting at nt!KiProcessListHead.
9. Breakpoints – set a breakpoint on Nt!NtCreateFile when called by explorer.exe only.
dx /w "@$curprocess.Name == \"explorer.exe\"" nt!NtCreateFile

    a. Advanced – every time the breakpoint is hit print the name of the file being created.
    dx /w "@$curprocess.Name == \"explorer.exe\"" nt!NtCreateFile "dx ((nt!_OBJECT_ATTRIBUTES*)@r8)->ObjectName; g"

10. TTD:
    a. Dump all memory access to a specific field in the shared user data and which instruction accessed it.
    b. Dump all calls to NtCreateFile and which file was opened.

JavaScript

1. Write a script to add a field to every process with the number of active threads in the process.
2. Write a script to add a field to every thread indicating if it's currently impersonating (use ETHREAD. ClientSecurity.ImpersonationData).
    a. Advanced – add another class to add a field to each EPROCESS to indicate the number of impersonating threads in the process.
3. Write a JS script with a function that receives a process and returns the number of loaded modules.
    a. Advanced – for each module, print the number of functions it imports from each imported module.
    To print to the debugger output window use host.diagnostics.debugLog.

Code Integrity

1. Best Practices / Analysis – If you have Hyper-V enabled: Try to enable HVCI on your machine – are you successful? If not, what drivers are causing issues? Try to figure out why.
2. Debugging UMCI – For every process in the system: dump its name, signature level and section signature level. Dump as a grid view and order by descending for signature level.
   dx -g @$cursession.Processes.Select(p => new {Name = p.Name, Id = p.Id, SIgnatureLevel = p.KernelObject.SignatureLevel & 0xf, SectionSignatureLevel = p.KernelObject.SectionSignatureLevel & 0xf}).OrderByDescending(p => p.SIgnatureLevel)
3. Debugging – changing code integrity config – If you have a VM / older Windows 10 system without Hyper-V, try modifying nt!SeILSigningPolicy to different values and see what still runs. Notice – this is monitored by PG so might crash your system if you don't change it back fast enough!
    a. With hyper-V this variable is protected by another mitigation so it cannot be tampered with.
4. Forensics / Analysis – HVCI also adds a capability to block malicious drivers found in an sdb file: drvmain.sdb. Using sdbexplorer.exe open this sdb and find the list of drivers that will be blocked. What do you see there? Why is each of the drivers on the list?

Vendor Security Features

1. Debugging / Scripting – Try to dump the process/thread/module load callbacks from WinDBG. There are no symbols for the internal structures so this could be tricky.
    a. Try writing a JS script to dump the callbacks instead.
    This will be a bit ugly as it requires working around the 53-bit pointer JS limitation.

```
function getProcNotifyRoutines()
{
    var routines = host.getModuleSymbol("nt", "PspCreateProcessNotifyRoutine",
"_EX_FAST_REF");
    host.diagnostics.debugLog("PspCreateProcessNotifyRoutine: ", routines.address, "\n");
    var routinesArray = host.createPointerObject(routines.address, "nt", "_EX_FAST_REF*");
    for (var i = 0; i < 64; i++)
    {
        //
        // we just need to zero out the bottom nibble of the address, but because JS can't work
        // with 64-bit numbers we need to do all this stuff and generate a new 64-bit
        // value from the 2 ULONGs that build the address
        //
        if (routinesArray[i].Value.convertToNumber() != 0)
        {
            host.diagnostics.debugLog(((host.Int64(routinesArray[i].Value.getLowPart() & 0xfffffff0,
routinesArray[i].Value.getHighPart())).add(8)), "\n");

host.diagnostics.debugLog((host.createTypedObject((host.Int64(routinesArray[i].Value.getLowPa
rt() & 0xfffffff0, routinesArray[i].Value.getHighPart())).add(8), "nt", "void(*)()")).dereference(),
"\n\n");
        }

    }
}
```

   b. Advanced – use synthetic types (we did not go over these in class, there are a few public examples online)
   c. Using open-source tools: – Use wincheck to dump all registered callbacks, or WinObjEx64 -> System Callbacks if running on a machine with debug mode enabled.

2. Advanced – RE – Is your machine running any AV that has an ELAM driver? Windows Defender does, as well as a few others. Can you find the location of the boot driver? Can you find the registry key containing the blocked hashes? What hashes are listed there?

PatchGuard / HyperGuard

1. Advanced – Debugging / Driver Dev – using windbg (or a custom driver, if you're comfortable writing drivers) try triggering as many PG / HG bugchecks as possible. A few ideas – setting breakpoints, modifying callbacks, changing CRs, modifying process list…
2. RE – Advanced – try finding out what driver registers for the PG callback and what driver notifies it.

Processor Security Features

1. RE – what auditing information can you receive for CET? In IDA search for ETW events that might be sent on CET failures. What failure triggers each of the different events?

Interrupts, Timers, APCs, DPCs

1. Dump the array of KINTERRUPTs found in the KPRCB. For each interrupt dump the service routine, message service routine and irql. For the functions add columns for their symbols.

```
dx -g @$prcb->InterruptObject.Where(i => i != 0).Select(i => (nt!_KINTERRUPT*)i).Select(i =>
new {ServiceRoutine = i->ServiceRoutine, ServiceRoutineSym = @$getSym(i->ServiceRoutine),
```

MessageServiceRoutine = i->MessageServiceRoutine, MessageServiceRoutineSym = @$getSym(i->MessageServiceRoutine), Irql = i->Irql})

2. Use WinDBG to print all APCs in the system and their normal/rundown/kernel routines.
   ```
   dx -r0 @$apcsForThread = (t => new {TID = t.Id, Object = (void*)&t.KernelObject, Apcs = Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&t.KernelObject.Tcb.ApcState.ApcListHead[0], "nt!_KAPC", "ApcListEntry").Select(a => new { Kernel = @$getsym(a.KernelRoutine), Rundown = @$getsym(a.RundownRoutine)})})
   dx -r0 @$procWithKernelApc = @$cursession.Processes.Select(p => new {Name = p.Name, PID = p.Id, Object = (void*)&p.KernelObject, ApcThreads = p.Threads.Where(t => t.KernelObject.Tcb.ApcState.KernelApcPending != 0)}).Where(p => p.ApcThreads.Count() != 0)
   dx -r6 @$procWithKernelApc.Select(p => new { Name = p.Name, PID = p.PID, Object = p.Object, ApcThreads = p.ApcThreads.Select(t => @$apcsForThread(t))})
   ```
   a. JS – try to do the same thing with a JS script.

System Calls

1. Dump a grid view of all system calls with their symbols and the number of arguments they take (approximately)
   ```
   dx @$serviceTable = (int(*)[0x1e1])&nt!KiServiceTable, x
   dx @$serviceTableAddr = (__int64)&nt!KiServiceTable, x
   dx -g @$serviceTable->Select(s => new {Address = @$serviceTableAddr + (s >> 4), Offset = s >> 4, Syscall = @$getSym(@$serviceTableAddr + (s >> 4)), Args = (s & 0xf) + 4}), x
   ```
   a. Do that for NT system calls and then win32k system calls.
      First make sure to switch to the context of a process that is using GUI, or the table will not be mapped. Then use "dps nt!KeServiceDescriptorTableShadow" to learn how many win32k system calls exist on your system.
      ```
      dx @$serviceTable = (int(*)[0x5b0])&win32k!W32pServiceTable, x
      dx @$serviceTableAddr = (__int64)&win32k!W32pServiceTable, x
      dx -g @$serviceTable->Select(s => new {Address = @$serviceTableAddr + (s >> 4), Offset = s >> 4, Syscall = @$getSym(@$serviceTableAddr + (s >> 4)), Args = (s & 0xf) + 4}), x
      ```
   b. Advanced – scripting – Write a JS script to parse the system call array and print all system calls and arguments.
2. Win32k syscalls and font loading mitigations:
   a. Print all processes that enable the mitigation flag EnableFilteredWin32kAPIs.
      ```
      dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlagsValues.EnableFilteredWin32kAPIs)
      ```
   b. Print all processes that enable the mitigation flag DisallowWin32kSystemCalls.
      ```
      dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlagsValues.DisallowWin32kSystemCalls)
      ```
   c. Print all processes that enable the mitigation flag DisableNonSystemFonts.
      ```
      dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlagsValues.DisableNonSystemFonts)
      ```
3. Choose interesting system calls and set a breakpoint to hit if they are called from a kernel driver with PreviousMode == UserMode
   ```
   bp nt!NtCreateFile "dx @$curthread.KernelObject->PreviousMode"
   ```

   a. hint: use @$curstack to know if call came from a kernel driver or the system call handler.
   b. Now try with PreviousMode == KernelMode: in what cases does this happen? When do we want to preserve the PreviousMode and when do we want to overwrite it?
4. RE – using IDA / other RE tool find instances of INT 29 and the code paths leading to them. How do you trigger each of them?

Object Manager

1. Use DX to dump all the different object types that services.exe has open handles to.
   dx @$cursession.Processes.Where(p => p.Name == "services.exe").First().Io.Handles.Select(h => h.Type).Distinct()
2. Dump all the processes that have open handles to mutant objects.
   dx @$cursession.Processes.Where(p => p.Io.Handles.Where(h => h.Type == "Mutant").Count() != 0)
   a. Advanced: dump the names of the mutants that processes have open handles to.
      dx -r2 @$cursession.Processes.Select(p => p.Io.Handles.Where(h => h.Type == "Mutant").Select(h => h.ObjectName))
   b. Use Process Hacker to view all mutants that processes have open handles to. Double-click a process and look under the “Handles” tab to find all open handles that the process has – only to named objects! You’ll see the object type and name for each open handle.
3. Dump the names of all files that each process has an open handle to.
   dx -r2 @$cursession.Processes.Select(p => p.Io.Handles.Where(h => h.Type == "File").Select(h => h.Object.UnderlyingObject.FileName))
4. Process Object callbacks:
   a. Track registered callbacks manually through windbg (no symbols so just try to dump pointers and look for pre- and post- callback functions).
   b. Write a JS script that dumps all registered callbacks for Thread, Process and Desktop.
5. Use the linked list in the handle tables to enumerate all processes on the system – the handle tables are all connected through a HandleTableList field, and the QuotaProcess field points to the EPROCESS of the process that this handle table belongs to.
   dx Debugger.Utility.Collections.FromListEntry((*(nt!_HANDLE_TABLE**)&nt!ObpKernelHandleTable)->HandleTableList, "nt!_HANDLE_TABLE", "HandleTableList")->Select(t => t.QuotaProcess)
6. Dump all object types. Including name, number of objects, Open/Close procedure.
   dx -g @$objTypes->Select(o => new {Name = o->Name, Number = o->TotalNumberOfObjects, OpenProcedure = o->TypeInfo.OpenProcedure, CloseProcedure = o->TypeInfo.CloseProcedure})
   a. Add symbols for the functions pointed to by the open/close procedures.
   b. Add additional fields: pool type and object size.
      dx -g @$objTypes->Select(o => new {Name = o->Name, Number = o->TotalNumberOfObjects, OpenProcedure = o->TypeInfo.OpenProcedure, CloseProcedure = o->TypeInfo.CloseProcedure, PoolType = o->TypeInfo.PoolType, ObjectSize = o->TypeInfo.Length})
7. Print all object types that have a SecurityProcedure in their TypeInfo. Print the name of each of these object types and their SecurityProcedure. Which ones have a non-default security

procedure?
dx -g @$objTypes->Where(o => o->TypeInfo.SecurityProcedure != 0).Select(o => new {Name = o->Name, SecurityProcedure = o->TypeInfo.SecurityProcedure})

8. Write a JS script that takes in the address of an object header and returns the object type and which optional headers it has.

ALPC

1. How many ALPC ports exist in the system? Use SelectMany to iterate over the handle tables of all the processes and find handles to ALPC ports. Cast the underlying object to ALPC_PORT and print the name of the owner process.

Memory Mitigations

1. Using Process Hacker find all processes running on your system that don't have ASLR enabled
    a. Which processes are running ASLR but not HEASLR?
2. Using WinDBG print all processes that don't have CFG enabled.
   dx -g @$cursession.Processes.Where(p => p.KernelObject.MitigationFlagsValues.ControlFlowGuardEnabled == 0)
3. Using Process Hacker look at processes that have CFG enabled – what functions are suppressed? Try to find the reasons why they would be suppressed (what benefit to they give to attackers?)
4. Which processes disable dynamic code? Why? What other mitigations do these processes enable?
5. Use process hacker to inspect different processes and see what mitigations that have enabled. Which processes enable more mitigations? Which ones enable less?
    a. Find how different mitigations can be enabled for processes when compiling – through Visual studio / command line – what flags need to be set to enable CFG? Or CET?