Advanced Scripting   
Use Windows APIs from PowerShell

Last Updated: 2/19/2024 6:09 PM Version 1  
Document Prepared for: CYBER360 Student

# Name Click here to enter name ID Click here to enter id

# Instructions

Answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

With the Windows operating system, you can use PowerShell to access functions in Windows APIs. The first task in this exercise explores the **Win32 API**. The second looks at **.NET Framework**, the Windows predecessor to today’s cross-platform **.NET Core**.

# Requirements

Windows

# Setup

Launch PowerShell Core.

# Task 1 — Use P/Invoke to call **MessageBoxA**

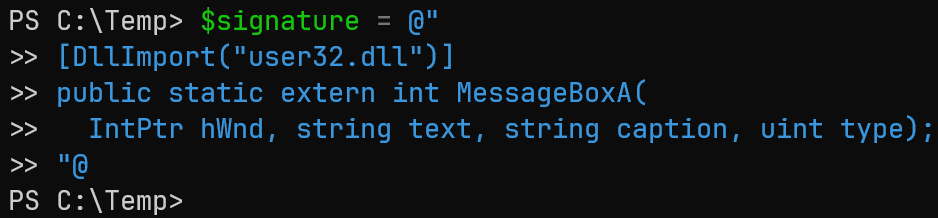
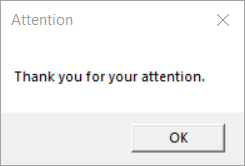
The *Windows 32-bit Application Programming Interface* (Win32 API) was first introduced in 1993. A significant contributor to Microsoft’s success in the IT marketplace has been their continued support of this API for over 30 years. Many developers select and use the Win32 API to implement and publish their software. Among the benefits they get from Microsoft’s API longevity include:

* extended returns on time and money invested in development, and
* reduced time and costs of ongoing maintenance of their software.

Microsoft built the Win32 API using C and C++, and they configured its functions so that they may be called or invoked using code written in *other* programming languages. (*Note:* for this task, you do ***not*** need to have any prior experience with C or C++.)

*Platform Invocation Services* (also known as *P/Invoke*) is a feature of .NET’s *Common Language Runtime* (CLR), and PowerShell can leverage this feature to access the Win32 API. For this exercise, we will write PowerShell code to call **MessageBoxA,** a function in the Win32 API which displays a short alert message in a small pop-up window.

## Steps

1. Enter the following command, which assigns a multiple-line *here-string* to a variable:   
   **$signature = @"**   
   **[DllImport("user32.dll")]**   
   **public static extern int MessageBoxA(**   
    **IntPtr hWnd, string text, string caption, uint type);**   
   **"@**   
     
   As you type it at your PowerShell prompt, notice the **>>** prompts in front of subsequent lines of your multiple-line command.   
   
   1. Question: what is a “*here-string*?” Answer: a here-string is used in PowerShell to specify a single string that contains more than one line. (In other words, it contains an invisible *new-line* (**"`n"**) control character at the end of each line of text in the string.) You might ask why it is called a here-string. Think of the delimiters like this: **@"** means “The string starts *here*” and **"@** means “the string stops *here*.”
   2. The here-string contained in **$signature** has two parts:
      1. The first line is an example of P/Invoke syntax. It signals that what follows is a function to import from a *dynamic link library* (DLL) named **user32.dll**. You’ll find this file in the **\Windows\System32** filesystem path. Figure out a PowerShell command line that will show the size (length) of the **user32.dll** file. Copy your command (***not*** its output) here: Click or tap here to enter text.
      2. The remaining lines are C language *function declaration* syntax for the **MessageBoxA** function, exactly as it might appear in a C or C++ source-code *header* file. (*Reminder*: once again, you ***don’t*** need prior experience with C or C++ to use this in PowerShell).
2. To present this function to PowerShell as if it were a member of a PowerShell class, we’ll use the **Add-Type** cmdlet. Enter the following (all on one line):   
   **Add-Type -MemberDefinition $signature -Name "User32" -Namespace "Win32" -PassThru**   
     
   This command creates a PowerShell class named **[Win32.User32]**. The **MessageBoxA** function specified in **$signature** will be available as a *static method* in that class.
3. Let’s use it! Even if you know nothing about C or C++, you have probably already figured out from the **$signature** that **MessageBoxA** has four parameters.  
   1. Enter the following (all on one line):   
      **[void] [Win32.User32]::MessageBoxA([IntPtr]::Zero, 'Thank you for your attention.', 'Attention', 0)**   
        
      You should now see a message alert box that looks like this:   
      
   2. The alert has control of your PowerShell session. Try typing something else at your PowerShell prompt. You won’t be able to observe your commands or output until *after* you click **[OK]**, so go ahead and clik **[OK]** it to close the alert.
4. Try your own! Make your own custom alert by changing the string arguments. The second of the four arguments is the message string that appears in the alert, and the third argument is the title that appears in the top of the alert window. (Don’t change the first argument **[IntPtr]::Zero**, and don’t change the last argument **0**.)   
   Your customized command line: Click or tap here to enter text.
   1. Notice the **[void]** at the front of the command line. **[void]** is a special PowerShell *unary operator* that discards any output returned by a function. For this task, the returned value doesn’t matter, so we used **[void]** to suppress it. Try your command again, but without the **[void]** operator. After you click **[OK]**, what value does the **MessageBoxA** function return? Click or tap here to enter text.
   2. Using an Internet search engine query or a generative AI chatbot, figure out how to interpret the value that **MessageBoxA** returned. What does the result represent? Click or tap here to enter text.
   3. Use Internet searches or generative AI chats again, this time to figure out what argument to change so that the alert window contains both **[OK]** and **[Cancel]** buttons, instead of just one **[OK]** button. Your customized command line: Click or tap here to enter text.
   4. After you click **[Cancel]**, what value does the **MessageBoxA** function return? Click or tap here to enter text.

# Task 2 — Use PowerShell to compile a .NET Framework C# program

Cross-platform PowerShell Core started with versions 6 and 7. These versions are built on .NET Core, now simply called .NET.

Prior versions of PowerShell (versions 5.1 and earlier) were not cross-platform. Only Windows PowerShell was supported, identified as “Desktop edition,” and it was built on *.NET Framework* (also Windows-only). There will be no further development of .NET Framework and PowerShell Desktop edition; regardless, each continues to be bundled (installed by default) with currently supported Windows operating systems, including Windows 10, Windows 11, and Windows Server.

Moreover, .NET Framework includes a command-line C# compiler. On Windows 10, its path is:  
 **C:\Windows\Microsoft.NET\Framework\4.0.30319\csc.exe**

(*Note:* For this task, you do ***not*** need to have any prior experience with C#.)

## Steps

1. At your PowerShell CLI prompt, change your working directory to **C:\Temp** (or to some other temporary folder of your choice):   
   **Set-Location C:\Temp**
2. Enter the following multiple-line command, which assigns a *here-string* to a variable:   
   **$CSharpSrc = @"**   
   **using System;**   
   **class Program {**   
    **static void Main()**   
    **Console.ForegroundColor = ConsoleColor.Yellow;**   
    **Console.BackgroundColor = ConsoleColor.DarkYellow;**   
    **Console.Write("Yellow");**   
    **Console.ForegroundColor = ConsoleColor.Blue;**   
    **Console.BackgroundColor = ConsoleColor.DarkBlue;**   
    **Console.Write("Blue");**   
    **Console.ForegroundColor = ConsoleColor.Gray;**   
    **Console.BackgroundColor = ConsoleColor.Black;**   
    **Console.WriteLine("Gray");**   
    **}**   
   **}**   
   **"@**   
     
   This here-string contains C# *source code*. (Unlike PowerShell, C# code is case sensitive, so be careful that you enter this code exactly as shown. For example, upcoming steps won’t work if you accidentally type **Using system** instead of **using System**.)
3. To save the source code to a file, enter:   
   **$CSharpSrc | Out-File -FilePath YellowBlueGray.cs**
4. To compile the saved source code using the .NET Framework compiler, enter (all on one line):   
   **\Windows\Microsoft.Net\Framework\v4.0.30319\csc.exe /out:YellowBlueGray.exe .\YellowBlueGray.cs**   
     
   This will put the compiled executable program into a file named **YellowBlueGray.exe**.
5. Execute the compiled program:   
   **.\YellowBlueGray**   
     
   After attackers gain access to a Windows system, they could use the procedure you completed above as a way to deploy executable malware. They don’t need to try to download something suspicious that might be detected and quarantined by antivirus software. Instead, they could just copy and paste strings of malware source code, then compile them on the victim’s machine using its own built-in compiler. (Fortunately, **YellowBlueGray.exe** was harmless, right?)
6. By the way, did you know that you can produce the same output in PowerShell in one command line? Enter (all on one line):   
   **"Yellow","Blue","Gray" | % {Write-Host $\_ -BackgroundColor "Dark$\_"   
   -ForegroundColor $\_ -NoNewLine}; Write-Host "" -BackgroundColor Black**   
   *This works in Windows PowerShell and in all PowerShell Core platforms (macos, Linux, Windows)*.
   1. Carefully compare the output of the compiled C# program to the output of the PowerShell one-liner. Do you notice a difference? If so, what? Click or tap here to enter text.
   2. Rewrite the previous command so that it outputs the color names red, green, and magenta in those respective colors. Your command: Click or tap here to enter text.

# Deliverable

Upload this document with completed answers to I-Learn Canvas.