
# Iron Scripter 2018: Prequel 6

## The Puzzle

Greetings Iron Scripters. You're past the half way point on your journey to Iron Scripter when you complete this puzzle.

This week's challenge is very simple – on the surface – but has some interesting quirks. You're required to create a PowerShell function that will return the uptime of the local machine or a remote system. The properties of the returned object should include:

- Computer name
- Last Boot Time – as type System.DateTime
- Uptime – as the number of days since the last reboot. The uptime should include partial days rounded to 3 decimal places – for example an uptime of 10 days and 8 hours would be 10.333

Ideally, your code should work with Linux systems as well as Windows. If you don't have access to a Linux system just concentrate on the Windows solution.

Hint: Physical Windows 8 and later machines enter a deep sleep when shutdown rather than being shut down. The standard methods of finding the uptime may not work on these operating systems. Where else could you look for start-up information?

The function should accept pipeline input of computer names. You may want to include operating system in the pipeline object.

Use best practice if it doesn't conflict with your faction's aims. The solution must be acceptable to your faction:

- Daybreak Faction - beautiful code
- Flawless Faction - flawless code
- Battle Faction - good enough to get the job done

Good luck and good coding.

## The commentary

When I work on PowerShell problems I always prefer to start with the basic functionality and then add the layer that supports pipelining and other production features.

So, the first thing is to determine when the machine booted. The simplest way to use the Win32_OperatingSystem class:

```
Get-CimInstance -ClassName Win32_OperatingSystem |
Select-Object -ExpandProperty LastBootUpTime
```

Which returns:

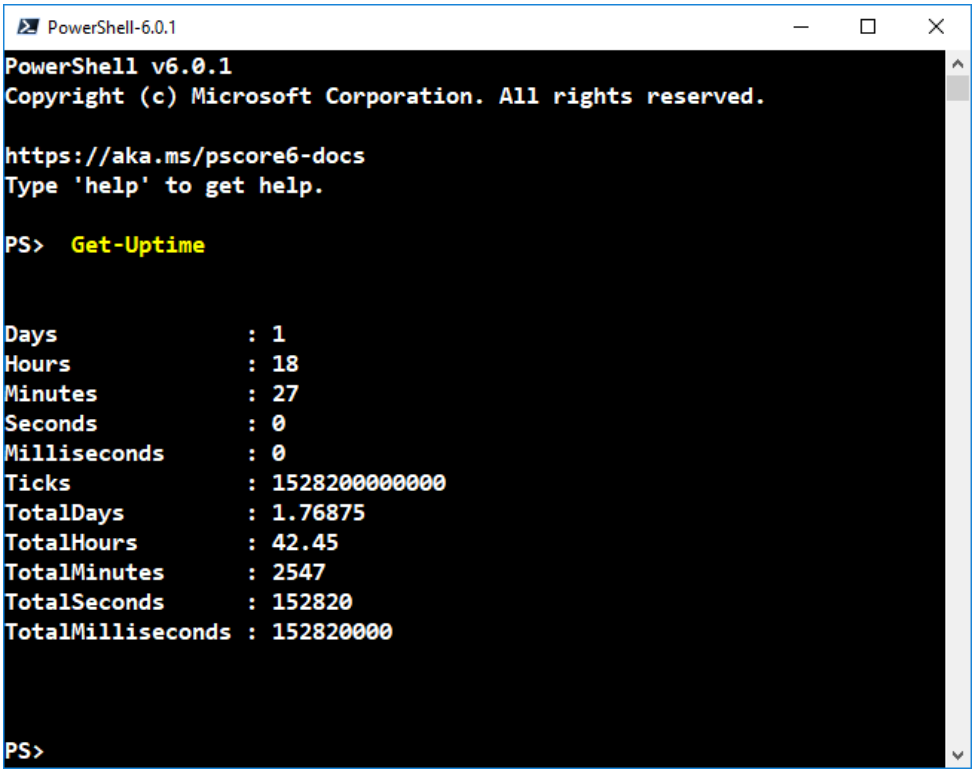
20 February 2018 17:34:09

I know this is wrong because I'm writing this on 21 February and I shutdown the machine last night. The reason it's wrong is that Windows 8 (and later) physical machines don't fully shutdown. The go into a deep sleep. It's the reason they start so quickly compared to Windows 7 and earlier. On these machines the LastBootUptime actually records the last full restart.

For now, I'll accept this result but we'll come back to it later.

As an aside, I always use the CIM cmdlets these days. They support WSMAN rather than DCOM. They convert dates for me and possibly more important the WMI cmdlets aren't in PowerShell v6 at the moment. You should treat the WMI cmdlets as at least deprecated and shout very loudly at people who use them.

Using the CIM class works on Windows but not on Linux. Luckily, PowerShell v6 has a Get-Uptime cmdlet that works on both Windows and Linux.




The cmdlet returns a TimeSpan object. The TotalDays property supplies the time the machine has been running. This also supplies a clue as to get the information required by the puzzle - the date of reboot and the time since the reboot.

One last point to resolve is how you can determine whether you're on PowerShell v5.1 or v6. $PSVersiontable helps with that:

```
PS>  ($PSVersionTable).PSVersion.Major
6
```

Let's start putting these bits of information together.

For Windows my basic function becomes

```
function Get-ComputerStartInfo {

## deliberately assuming that NOT running
##  PowerShell v2 which doesn't have $PSVersionTable
##  if you have v2 should upgrade!
    switch (($PSVersionTable).PSVersion.Major) {
        {$_ -le 5} {
            $lbtime = Get-CimInstance -ClassName Win32_OperatingSystem |
            Select-Object -ExpandProperty LastBootUpTime

            $uptime = (Get-Date) - $lbtime
        }
        default { Throw 'Unrecognised version of PowerShell'}

    }

    $props = [ordered]@{
        ComputerName = $env:COMPUTERNAME
        LastBootTime = $lbtime
        Uptime = [math]::Round($uptime.TotalDays, 3)
    }

    New-Object -TypeName psobject -Property $props

}
```

Get the major version number for PowerShell and using a switch to determine the calculations. I've deliberately ignored PowerShell v2 when writing this code as you should have upgraded. The only reason not to upgrade is if you're running one of the few products like Exchange that don't support the upgrade but then you should have upgraded that version of Exchange by now… Bottom line is I'm not adding PowerShell v2 support. If you do want to add PowerShell v2 support remember that the CIM cmdlets appeared in PowerShell v3 so you'll have to use the WMI cmdlets and convert the date from WMI format to a DateTime object.

Use the CIM class to get the boot time (I know its wrong but we'll come back to that problem – this code works for servers). Subtract that date from the current date to get the uptime.

Create an ordered hash table of properties – use [math]::Round() to get the required precision on the up time. Output the object. This is what you get:

```
PS> Get-ComputerStartInfo
ComputerName LastBootTime        Uptime
------------ ------------        ------
W510W10      20/02/2018 17:34:09  1.784


PS> Get-ComputerStartInfo | gm

   TypeName: System.Management.Automation.PSCustomObject

Name         MemberType   Definition
----         ----------   ----------
Equals       Method       bool Equals(System.Object obj)
GetHashCode  Method       int GetHashCode()
GetType      Method       type GetType()
```

```
ToString     Method       string ToString()
ComputerName NoteProperty string ComputerName=W510W10
LastBootTime NoteProperty datetime LastBootTime=20/02/...
Uptime       NoteProperty double Uptime=1.785
```

The information as required and the correct type information for the properties.

Now, lets add in the Linux part.

```
function Get-ComputerStartInfo {

## deliberately assuming that NOT running
##  PowerShell v2 which doesn't have $PSVersionTable
##  if you have v2 should upgrade!
    switch (($PSVersionTable).PSVersion.Major) {
        {$_ -le 5} {
            $lbtime = Get-CimInstance -ClassName Win32_OperatingSystem |
            Select-Object -ExpandProperty LastBootUpTime

            $uptime = (Get-Date) - $lbtime
        }
        {$_ -eq 6} {
            $uptime = Get-Uptime

            $lbtime = (Get-Date) - $uptime
        }

        default { Throw 'Unrecognised version of PowerShell'}

    }

    $props = [ordered]@{
        ComputerName = $env:COMPUTERNAME
        LastBootTime = $lbtime
        Uptime = [math]::Round($uptime.TotalDays, 3)
    }

    New-Object -TypeName psobject -Property $props

}
```

The difference is that we use the Get-Uptime cmdlet to find the uptime and subtract the TimeSpan from the current date to get the boot time. If you compare the results below with those from the PowerShell v5.1 version you'll notice a few seconds difference in the boot time. I'm assuming that's due to rounding. The results are close enough for my purposes.

```
PS>  Get-ComputerStartInfo

ComputerName LastBootTime        Uptime
------------ ------------        ------
W510W10      20/02/2018 17:34:02  1.797


PS>  Get-ComputerStartInfo | gm


   TypeName: System.Management.Automation.PSCustomObject
```

```
Name         MemberType   Definition
----         ----------   ----------
Equals       Method       bool Equals(System.Object obj)
GetHashCode  Method       int GetHashCode()
GetType      Method       type GetType()
ToString     Method       string ToString()
ComputerName NoteProperty string ComputerName=W510W10
LastBootTime NoteProperty datetime LastBootTime=20/02/2018 17:34:02
Uptime       NoteProperty double Uptime=1.797
```

You've probably worked out that I'm using VSCode for this example. I still prefer ISE (probably due to familiarity more than anything) but VSCode has the great advantage that I can swap the integrated console between PowerShell v5.1 and PowerShell v6 when I have them both installed on the machine.

The puzzle also wanted the function to accept pipeline input and work with remote machines. The code looks like this:

```powershell
function Get-ComputerStartInfo {
[CmdletBinding()]
param (
    [Parameter(ValueFromPipelineByPropertyName=$true)]
    [string]$ComputerName,

    [Parameter(ValueFromPipelineByPropertyName=$true)]
    [ValidateSet('Windows', 'Linux')]
    [string]$OS
)
BEGIN {
    $sb = {
        ## deliberately assuming that NOT running
        ##  PowerShell v2 which doesn't have $PSVersionTable
        ##  if you have v2 should upgrade!
            if (-not $IsLinux) {
                $cos = Get-CimInstance -ClassName Win32_OperatingSystem
                $majorVer = [int]($cos.Version -split '\.')[0]

                $ccs = Get-CimInstance -ClassName Win32_ComputerSystem
            }
            if (($majorver -ge 8) -and ($cos.Caption -notlike "* Server *")  -and ($ccs.Model -
notlike "*Virtual *")) {
                $strtime = Get-WinEvent -FilterHashtable @{Logname='System';
ProviderName='Microsoft-Windows-Power-Troubleshooter'; ID=1} -MaxEvents 1 |
                Select-Object -ExpandProperty properties | Select-Object -Skip 1 -First 1 -
ExpandProperty Value
                $lbtime =   [datetime]$strtime

                $uptime = (Get-Date) - $lbtime
            }
            else {
                switch (($PSVersionTable).PSVersion.Major) {
                    {$_ -le 5} {
                        $lbtime = Get-CimInstance -ClassName Win32_OperatingSystem |
                        Select-Object -ExpandProperty LastBootUpTime

                        $uptime = (Get-Date) - $lbtime
```

```
                    }
                    {$_ -eq 6} {
                        $uptime = Get-Uptime

                        $lbtime = (Get-Date) - $uptime
                    }

                    default { Throw 'Unrecognised version of PowerShell'}

                }
            }

            if ($IsLinux){$comp = hostname} else {$comp = $env:COMPUTERNAME}

            $props = [ordered]@{
                ComputerName = $comp
                LastBootTime = $lbtime
                Uptime = [math]::Round($uptime.TotalDays, 3)
            }

            New-Object -TypeName psobject -Property $props
        }
}

PROCESS {
    if ((-not $ComputerName) -or ($ComputerName -eq $env:COMPUTERNAME)){
        Invoke-Command -ScriptBlock $sb
    }
    else {
        $info = $null
        if (-not $OS){Throw 'No Operating System information'}
        switch ($OS){
            'Windows' {
                $info = Invoke-Command -ScriptBlock $sb -ComputerName $ComputerName -
HideComputerName
            }
            'Linux' {
                if (($PSVersionTable.PSVersion.Major) -ge 6) {
                    $info = Invoke-Command -ScriptBlock $sb -HostName $ComputerName -UserName
root -HideComputerName
                }
                else {
                    Write-Warning -Message "Can NOT connect to Linux system: $computername from
curernt version of PowerShell"
                }

            }
        }
        if ($info){
            $info | Select-Object -Property * -ExcludeProperty RunSpaceId
        }
    }
}
}
```

Getting this far threw out a few jokers. I'll work through the code and explain the issues.

The param block allows you to set the attributes that enable the function to accept pipeline input – [Parameter(ValueFromPipelineByPropertyName=$true)]

The [CmdletBinding()] attribute adds the common parameters -Verbose, -Debug etc.

Two parameters have been defined – the OS parameter has validation set so that only Windows or Linux are allowed as values.

The code has been split into a BEGIN block – it runs once as the first object from the pipeline hits the function – it defines the script block that performs the work. The PROCESS block manages the decisions about making remote connections, calls the script block and outputs the results.

The script block (BEGIN block) first determines if its running on Linux – you could add a test for mac if you want completeness. If its not running on Linux the Win32_OperatingSystem and Win32_ComputerSystem CIM classes are retrieved – these classes don't exist on Linux.

The operating and computer system are tested to determine if the system is Windows 8 or later, NOT a Server and NOT a virtual machine. If these conditions are met then the System event log is interrogated to find the real start up time.

```
PS> Get-WinEvent -FilterHashtable @{Logname='System'; ProviderName='Microsoft-Windows-Power-
Troubleshooter'; ID=1} -MaxEvents 1 | fl *


Message              : The system has returned from a low power state.

                       Sleep Time: 2018-02-21T19:56:51.313331600Z
                       Wake Time: 2018-02-22T11:27:57.157966200Z

                       Wake Source: Unknown
```

By using the Properties of the event log entry you can directly access the start up time. Once you have that you can calculate the uptime as a TimeSpan.

If the machine is not Windows, or a Server, or a Virtual machine you drop into the switch. Using the $PSVersionTable).PSVersion.Major means the code changes for v5.1 or v6. For v5.1 (or earlier) get the LastBootUpTime from Win32_OperatingSystem and subtract that from current date. For v6 use Get-Uptime and subtract it from current date to give boot time.

Getting the computer name is fun. If you're not on Linux you can use $env:COMPUTERNAME. However, PowerShell v6 on Linux doesn't have $env:COMPUTERNAME it has $env:HOSTNAME. That's bad enough but it turns out that $env:HOSTNAME isn't exposed in a PowerShell v6 SSH remoting session to Linux. Just for consistency $env:COMPUTERNAME is exposed in a PowerShell v6 SSH remoting session to Windows. Bottom line is if you're on Linux you need to use the hostname utility. That utility also exists on Windows but that spoils the fun.

Once you have the computer name you can create the hash table and output the object.

In the PROCESS block you run the script block described above.

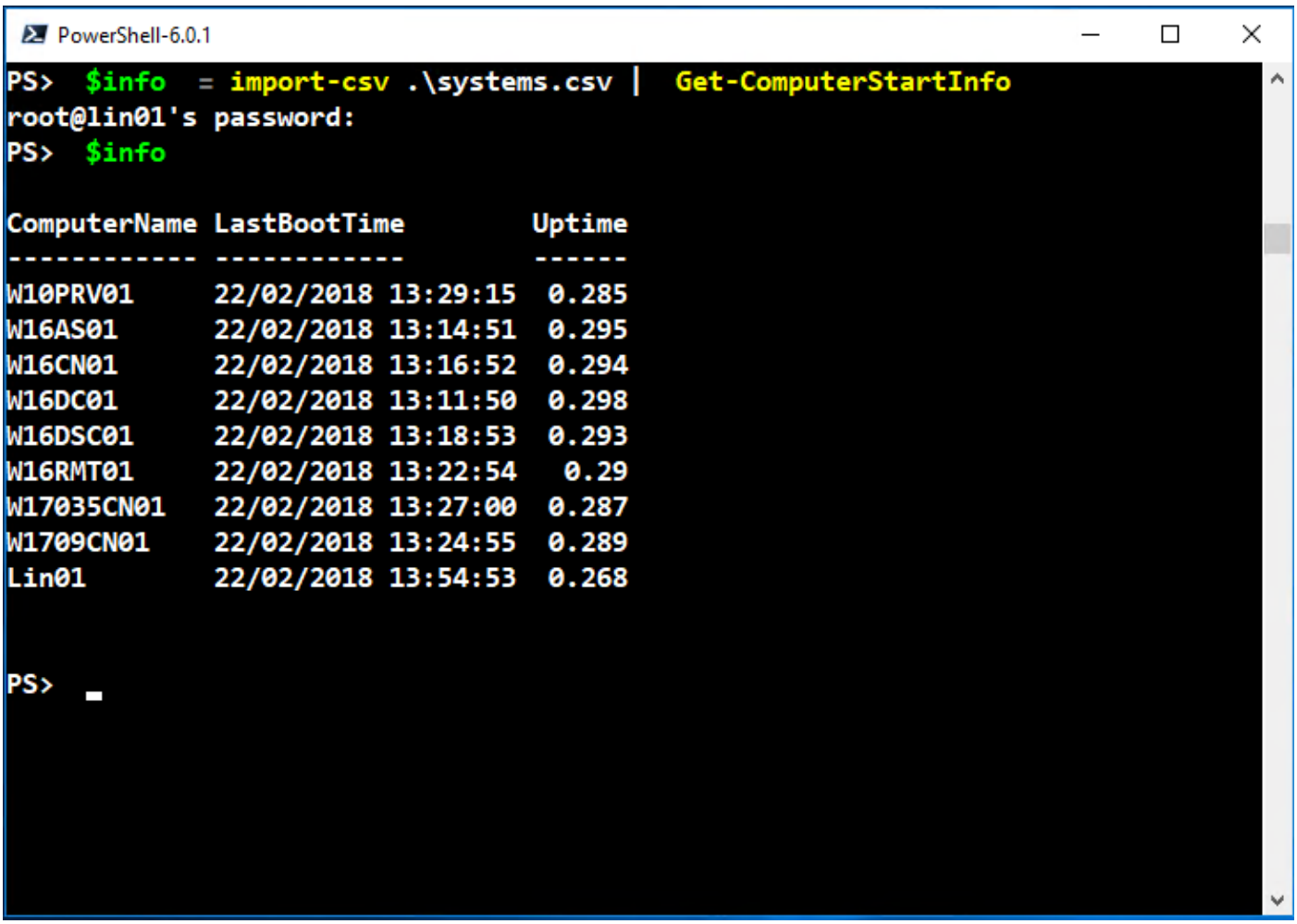
If you're on the local machine – just use Invoke-Command to run the script block. Notice I don't allow for . or localhost. They are abominations that shouldn't be used. I suppose I should add an additional test for running locally on Linux ($Computername -eq $env:HOSTNAME). I don't intend to run locally on Linux so you can add it if you want.

If you're not on the local machine then we're into remoting. For a Windows remote system use standard WSMAN based remoting. If $OS is Linux then test the PowerShell version. If not PowerShell v6 put out a warning otherwise use SSH remoting to get to the Linux system. In both cases suppress the PSComputerName that PowerShell populates. If the remoting was successful output the results suppressing the RunSpaceId.

To run the function, I created a CSV file of systems:

```
ComputerName OS
------------ --
W10PRV01     Windows
W16AS01      Windows
W16CN01      Windows
W16DC01      Windows
W16DSC01     Windows
W16ND01      Windows
W16RMT01     Windows
W17035CN01   Windows
W1709CN01    Windows
Lin01        Linux
```

Running it gives these results:




I'm prompted for the password on Lin01 because I haven't used SSH authentication keys.

You could extend this to cover non-domain Windows machines over SSH remoting if required.

As this point, Battle faction will probably sit back and say job done. We've met the requirements – next problem please.

Daybreak faction will want to format the code so that it looks good. Maybe create a module.

Flawless faction will want to add all the bells and whistles. All the options from Daybreak faction plus help file, parameter validation, Write-Debug and Write-Verbose commands as appropriate, try-catch blocks as necessary, test remote machines are contactable, and anything else that ensures the code executes flawlessly.

I could see Flawless wanting to extend this to cover non-domain machines. Daybreak may want to do that as well.

Enjoy!  Puzzle 7 will be available around the time you're reading this.