Wireshark Developer and User Conference

PowerShell and TShark

26 June 2012

Graham Bloice

Software Developer | Trihedral UK Limited

SHARKFEST '12

UC Berkeley

June 24-27, 2012

Introduction

- R&D Software Developer with Trihedral UK Limited, a SCADA/HMI vendor.
- Trihedral's software products run on Windows.
- Use Wireshark (and Tshark) for analyzing industrial protocols and telemetry.
- First contributed to Wireshark source code in 1999, core developer since 2003.
- PowerShell user since 2006.

Topics

- PowerShell, an introduction.
- Using Tshark with PowerShell, converting *nix commands.
- Advanced PowerShell functionality.
- Q&A.

Audience Participation

- How many are Windows “bound”?
- How many have heard of PowerShell?
- How many have used PowerShell at all?
- How many use PowerShell frequently?

PowerShell Introduction

- PowerShell is the task automation framework for Windows.
- Comprised of:
  - Command-line shell.
  - Scripting language.
  - Integrated Scripting Environment (ISE).
- Built on, and integrated with, .NET Framework
- Access to COM and WMI.
- Enables administrators to perform admin tasks on local and remote machines.

Why PowerShell?

- Replaces limited functionality of DOS and CMD shells.
- Allows scripting of Windows systems and products, e.g OS, Exchange, Active Directory.
- Incorporated into all new Windows products.

PowerShell Versions

- V1.0, arrived with Server 2008 in Nov 2006, available for XP SP2/3, Server 2003 and Vista.
- V2.0, arrived with Windows 7 and Server 2008R2 in August 2009, available for XP SP3, Server 2003 SP2, Vista SP1 and Server 2008 SP1.
- V3.0 coming with Windows 8 and Server 2012.
- This presentation discusses V2.0.

PowerShell Overview

- PowerShell can execute:
  - Cmdlets, .NET programs built for PowerShell.
  - Script Files containing PowerShell commands.
  - PowerShell functions.
  - Any other executable program.
- Has a pipeline, but is object based. Pipeline elements don't need to reparse textual output.
- Scripting language is dynamically typed.

PowerShell Help

- Built in cmdlet 'Get-Help', aliased to 'help' and 'man' ☺
- Can use wildcards such as “*”, e.g. 'help *var*'.
- Help for each command, e.g. 'help Get-Help' or 'Get-Help -?'.
- Several levels of help, add switches for more, e.g. –Examples, -Detailed, -Full.
- Conceptual help, e.g. 'help about*' lists topics.

PowerShell Keywords

- PowerShell “language”, ‘help about_Language_Keywords’.
- Functions; function; param; return; begin; process; end; dynamicparam; data; filter.
- Branching; if; else; elseif; switch.
- Loops; do while|until; while; for; foreach; in; break; continue.
- Exception handling; try; catch; finally; trap; throw.

PowerShell Basics I

- Variables are prefixed with “$”, are not case-sensitive and are dynamically typed. Variable scope is global, script or private.
- Variables are:
  - User created variables such as $myVar, created or changed by assigning a value, e.g. '$myVar = 42'.
  - Automatic variables such as $args and $_, store the state of PowerShell.
  - Preference variables are “options” and are set default values by PowerShell and can be changed, e.g. $MaximumHistoryCount.
- Current variable type can be found by piping to Get-Member, e.g. '$myVar | Get-Member'.
- See help topics 'about_Variables' and 'about_Automatic_Variables' and 'about_Environment_Variables'.

PowerShell Basics II

- Cmdlets are based on a Verb-Noun pattern, e.g 'Get-Help'.
- To see all commands use 'Get-Command'.
- Object properties accessed using “.” operator, e.g. '$object.property'.
- Object member functions accessed using “.” operator and passing parameters in parenthesis, e.g. '$object.member($aParameter)'.
- To inspect object methods and parameters, pipe to Get-Member, e.g. '$a | Get-Member'.

PowerShell Basics III

- Function arguments can be position based or named, e.g. 'Get-Help help' or 'Get-Help –Name help'.
- Function calls use space to separate arguments, e.g. 'myFunction parm1 parm2'.
- Operators include arithmetic, assignment, comparison, logical, redirection, split/join, type, unary, special. See 'help about_operators'.
- Access to all of .NET, e.g. '[System.Math]::Sqrt(9)'.

PowerShell Data I

- Structured data types, arrays:
  - construct with “,” operator, e.g. '$a = 1,2,3,4,"data",5'. Can also construct a range with “..” operator, e.g. '$a = 1..10'.
  - Access via subscript operator, e.g. '$a[1]'. Negative numbers count from the end, e.g. '$a[-1]' for last element.
  - Length given by “length” or “Count” properties, e.g. '$a.length'.
  - Set an element using assign, e.g. '$a[4] = 111' or “SetValue” method, e.g. '$a.SetValue(111, 4)'.

PowerShell Data II

- Structured data types, hashtables:
  - Hashtable (or dictionary or associative array) holds name/value pairs.
  - construct with “@{},” operator, e.g.

    '$h = @{int1=42;float1=24.4;string1=“help”}'.
  - Access via “.” or “[]” operators, e.g.

    '$h.int1', '$h["int1"].
  - Sort by piping enumerator to sort, e.g. '$h.getenumerator() | sort key'.

PowerShell Aliases

- An alias is an alternate name or nickname for a cmdlet or command element. See 'help about_aliases'.
- Many “built-in” aliases, e.g. 'help' and 'man' are aliases for the 'Get-Help' cmdlet.
- See current aliases with 'Get-Alias' or it’s alias 'alias'.
- Create an alias with 'New-Alias', e.g. 'New-Alias tshark 'C:\Program Files (x86)\Wireshark\tshark.exe''.

PowerShell Providers I

- Providers enable access to data stores, e.g. FileSystem, Registry, Environment, Active Directory and more.

- Providers use a “drive” prefix and then a path. To list providers use 'Get-PSProvider'.

- Filesystem provider uses standard drive letters and paths, e.g. 'C:\Temp\Afile.txt'.

PowerShell Providers II

- Other drives; Alias, Function, Environment (Env), HKLM, Variable and more. To list drives use 'Get-PSDrive'.

- Can create “drives” for own use, e.g. my “captures” drive 'New-PSDrive –name CAPS –PSProvider FileSystem –Root "C:\Temp\Caps" –Description "Captures Directory"'.

- PowerShell “drives” are only usable from PowerShell cmdlets and functions. Use 'Convert-Path' to pass to other executables.

Common use commands I

- Lots of aliases that make things familiar from dos or other shells.
- Directory manipulation:
  - dir, cd, md, rd, pushd, popd.
  - ls, cd, mkdir, rmdir.
- File manipulation:
  - type, copy, del, echo.
  - cat, cp, diff, rm.
- Output to a file (output is Unicode by default):
  - Redirection using “>”, e.g. '"help " + "me" > test.txt'.
  - Using Out-File, e.g. '"help " + "me" | Out-File test2.txt'.
  - Out-File can set encoding, e.g. '"help " + "me" | Out-File –Encoding ASCII test3.txt'.

Common use commands II

- Counting things in a pipeline, e.g.

  '$a | measure'.

- Sorting in a pipeline, e.g.

  '$a | sort'.

  optionally reversed with '-Descending'.

- Selecting the first or last set of elements in a pipeline, e.g.

  '$a | select –First 5' or 'select –Last 5'.

Common use commands III

- Removing duplicates from a pipeline, e.g.

  '$a | select –Unique'.

- Filtering in a pipeline, e.g.

  '$a | where { $_.SomeProperty –eq "SomeValue" }'.

- Iterate over elements in a pipeline performing an operation on the elements, e.g.

  '$a | % { $_ + 10 }'.

- Note use of '$_' to denote current input object.

Data Format conversion.

- PowerShell can handle csv format, cmdlets to import/export objects to files, and pipeline values, e.g. Export-CSV, Import-CSV, ConvertTo-CSV, ConvertFrom-CSV.

- Csv input can define field names using first row, or they can be supplied by a '–Header' parameter to ConvertFrom-CSV.

- '$a = "first,second", "1,2", "8,9","3,4" | ConvertFrom-CSV'.

PowerShell and Tshark

- How to use PowerShell to process tshark output.
- Examples shamelessly stolen, with permission, from Sake Blok's SharkFest 2010 presentation ☺.
- Captures used (example.cap & http.cap) are on https://surf.cloudshark.org (username sharkfest, password sharkfest).

Access to Tshark

- By default, the Windows installer doesn't put the Wireshark directory on the path.

- Two options:
  - Create an alias:

    'New-Alias tshark "path\to\tshark.exe"'.
  - Modify the path:

    '$env:Path += ";path/to/wireshark/directory".

- Modifying the path gives access to all the other tools, e.g. capinfos etc.

Communicating with Tshark

- PowerShell can call tshark, passing in parameters, and then process the output.
- Parameters can be awkward, due to delimiters, just quote them all with ".
- Tshark produces text, PowerShell interprets this output as an array of text strings.
- PowerShell likes csv so tshark parameters for fields are often used, e.g.

  '–T Fields –E "separator=," –e xxx'.

Counting http response codes I

- Problem:
  - Need http response codes and counts.
- Approach:
  - Print only http response code.
  - Make sorted table.
  - Count.

Counting http response codes II

- The command:

```
① tshark –r example.cap –R http.response `
② –T fields –e http.response.code |
③ sort –Unique | measure
```

Pipeline breakdown:

1. Calls tshark, provides the input file and the filter.
2. Specifies field output, and the required field.
3. Pipes the output to sort, selecting unique entries and then pipes that to measure.

Top 10 requested URL's I

- Problem:
  - Need a list of visited URL's.
- Approach:
  - Print full request uri.
  - Strip all parameters (e.g. after "?").
  - Count URL's.
  - Return top 10.

Top 10 requested URL's II

- The command:

```
① tshark -r example.cap -R http.request `
② -T fields -e http.request.full_uri |
③ % { ($_.Split("?"))[0] } | group |
④ sort Count –Descending | select Name,Count –First 10
```

Breakdown:

1. Calls tshark, provides the input file and the filter.
2. Specifies field output and the required field.
3. Iterates over the entries, forming a new string from the first part of the URI up to the "?". Pipes result to group results.
4. Pipes to sort using member Count in descending order, and then pipes to select the fields and the first 10 results.

All sessions with cookie XXXX I

- Problem:
  - Need to see only data from a specific session, i.e. have a cookie "PHPSESSID=c0bb9d04cebbc765bc9bc366f663fcaf"
- Will return a new capture file with the required data.
- Approach:
  - Select packets that contain the cookie.
  - Grab the port numbers.
  - Create a new filter with these port numbers.
  - Use filter to extract sessions.
  - Save packets to a new file.

All sessions with cookie XXXX II

- The command:

```
① tshark -r .\example.cap -w cookie.cap `
② -R ((tshark -r .\example.cap -T fields -e tcp.srcport `
③       -R "http.request and http.cookie contains `
    c0bb9d04cebbc765bc9bc366f663fcaf" |
④       % { "tcp.port=={0}" -f $_} ) -join "||")
```

Breakdown:

1. Calls tshark, provides the input and output files.
2. Specifies a filter, contents in (). Filter is a call to tshark using same input file and printing the tcp.srcport field.
3. And has a filter for the session identifier.
4. Iterates over the entries, forming a new string from the port number. Joins each of these strings using "||".

All sessions for user XXXX I

- Problem:
  - A particular user has multiple sessions and I need to see all sessions for that user.
- Output:
  - Will return a new capture file with the required data.
- Approach:
  - Print all session cookies for user xxxx.
  - Create a new capture file per session cookie as per previous example.
  - Merge files to new output file.

All sessions for user XXXX II

```
# Sessions for user
param(
  $file, $user
)

$userFilter = "http.request and http contains $user"
$cookies = tshark -r "$file" -R $userFilter `
                  -T fields -e "http.cookie" | % { $_.split("=")[2] }
foreach ($cookie in $cookies)
{
  $tmpfile = "tmp_$cookie.cap"
  write-host "Processing session cookie $cookie to $tmpfile“

  $cookieFilter = "http.request and http.cookie contains $cookie"
  tshark -r $file -w $tmpFile -R ((tshark -r $file -R $cookieFilter `
                                  -T fields -e "tcp.srcport" |
                                  % { "tcp.port=={0}" -f $_ }) -join "||")
}

& mergecap -w "$user.cap" tmp_*.cap
rm tmp_*.cap
```

Advanced PowerShell functionality

- PowerShell can interface with any COM object, e.g. Excel.

- PowerShell has full access to the .Net framework, e.g. Windows Forms and Charts.

Driving Excel I

- PowerShell has built in support for interaction with COM objects, e.g. Excel.
- Again we'll use the top URL's, but this time the whole list and push it into Excel.
- Create the Excel object and obtain references to some further objects in Excel.

```
$xl = New-Object -ComObject 'Excel.Application'
$wb = $xl.Workbooks.Add()
$ws = $wb.ActiveSheet
$cells = $ws.Cells
```

Driving Excel II

- Set the title cell, including formatting

```
$cells.item(1,1) = "Top URL's"
$cells.item(1,1).font.bold = $true
$cells.item(1,1).font.size = 18
```

Driving Excel III

- Iterate over data setting values into cells.

```
$row = 1

foreach ($url in $topURL) {
  $row++
  $col = 1
  $cells.item($row,$col) = $url.Name
  $col++
  $cells.item($row,$col) = $url.Count
}

$xl.Visible = $true
$wb.SaveAs("C:\Temp\SharkFest12\urls.xlsx")
```

Driving Excel IV

- Show Excel UI, and save the file.

```
$xl.Visible = $true
$wb.SaveAs("C:\Temp\SharkFest12\urls.xlsx")
```

- Cleanup and exit.

```
$wb.Close()
$xl.Quit()
```

.Net Charting

- .Net Charting is a download for .Net 3.5, included with .Net 4.0.
- PowerShell defaults to running under .Net 2.0.
- To check version use '[environment]::Version'.
- To run powershell under 4.0 requires application configuration file (powershell.exe.config) next to powershell.exe ($pshome):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- http://msdn.microsoft.com/en-us/library/
w4atty68.aspx -->
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" />
    <supportedRuntime version="v2.0.50727" />
  </startup>
</configuration>
```

.Net Charting – Creating a chart

- Using LibraryChart.ps1 from PoshCode.
- Using the top 10 URL's from earlier in the presentation, pipe to Out-Chart:

```
$top10 | Out-Chart –xField Name –yField Count
–chartType 'Pie'
```

- Can also save image to a file:

```
$top10 | Out-Chart –xField Name –yField Count `
–chartType 'Pie' `
–filename C:\temp\SharkFest12\top10pie.png
```

Resources

- MSDN – Start here for PowerShell:
  http://msdn.microsoft.com/en-us/library/windows/desktop/dd835506%28v=vs.85%29.aspx
- Technet:
  http://technet.microsoft.com/en-us/scriptcenter/powershell.aspx
- PoshCode – repository of code:
  http://poshcode.org
- Don Jones ShellHub:
  http://shellhub.com/
- Windows 7 Resource Kit PowerShell Pack:
  http://archive.msdn.microsoft.com/PowerShellPack
- PowerShell Community Extensions (PSCX):
  http://pscx.codeplex.com/

Questions?

- Maybe even some answers.