How to Leverage PowerShell to Create a User-Friendly Version of WinDump

GIAC (GCIA) Gold Certification

Author: Robert L. Adams, robert.louis.adams@gmail.com
Advisor: Hamed Khiabani, Ph.D.
Accepted: January 11th 2016

Abstract

WinDump is often used to analyze packet captures by incorporating Berkeley Packet Filters, to reduce large captures into manageable subsets. The filtering makes use of macros to easily specify common protocol properties, however, analyzing other properties requires a deeper understanding of the protocol and more complicated expressions. PowerShell is a Windows scripting language that has become increasingly popular within the security community. PowerShell is extremely extensible, and can be used to develop an easy way to interact with WinDump. This paper will demonstrate how to write a custom PowerShell module that serves as a wrapper around WinDump, enabling an easier and more intuitive way of unleashing the power of WinDump.

1. Introduction

Security professionals rely on a myriad of tools to accomplish their job. This is no different than the toolboxes that plumbers, electricians, and other trade professionals carry with them every day. There are specific tools for various aspects of a job. These tools tend to evolve over time and in some cases, new technology influences the creation of a better tool.

Some tools, like a hammer, are timeless. These types of tools are found within the information security industry as well. WinDump, the Windows version of the popular packet capture and analysis tool, tcpdump, has been an ageless asset to security professionals for decades. WinDump is really good at its job: capturing and filtering network packets. It is a command-line tool that succeeds in its minimalistic approach.

A vast amount of time has passed since WinDump was released. Nine years have passed since the latest update was distributed. This begs the question: How can WinDump be better?

Technology influences change. PowerShell is a Windows scripting language that has influenced much change since its original release in 2006. More recently, PowerShell has been affecting the way both attackers and defenders perform their job. PowerShell's appeal is largely attributed to the language's extremely natural syntax, power, and flexibility. It makes complete sense that PowerShell has been the underlying inspiration to many security trends.

PowerShell has influenced PowerSploit—the PowerShell flavor of Metasploit. PowerShell has inspired Posh-Nessus--- the PowerShell take on automating the Tenable Nesses Vulnerability Scanner. Can PowerShell be used to improve WinDump?

WinDump captures and reads packets. It can also be used to filter packet captures into relevant subsets to aid forensic investigation. WinDump makes uses of macros to filter in a natural language-like way. Users can specify "tcp", "src host", "dst host" and so on. However, the learning curve and complexity immediately spike when attempting

Robert L. Adams, robert.louis.adams@gmail.com

to identify specific (less than one byte) values in various protocol header fields: TCP flags, IPv4 attributes, and so forth.

PowerShell seems like the natural choice for creating a custom tool to serve as a wrapper around WinDump. The complexity of more sophisticated filters can be eradicated with extremely easy syntax. PowerShell can be used to provide increased search functionality when working with packet captures. And finally, PowerShell can be leveraged to provide concurrent processing and filtering of multiple packet captures.

2. WinDump – A Brief Overview and History

WinDump is a command-line network capture and analysis tool for Windows. The original tool, tcpdump, was first written for UNIX. The Windows variant captures network traffic through the use of the Windows Packet Capture (WinPcap) library (WinDump, 2013). WinDump is free software and is extremely useful for packet capture and analysis.

WinDump is extremely performant. The tool can process large packet captures and detect patterns in very little time. The tool can be used to analyze live traffic that passes through an interface. It can also be used to process previously captured traffic (in the form of PCAP files).

There are several options built into WinDump. The variety of command-line switches allow users to specify which interfaces to capture, the level of detail in the output, and the amount of traffic to capture (Cane, 2014). Additionally, analysts can use filters in the form of expressions to isolate very specific network traffic patterns.

2.1. Why is WinDump a Valuable Tool?

There are several network analyzers on the market today. Many of these tools have robust graphical interfaces. Wireshark, for example, performs many of the same functions as WinDump. Also, Wireshark is equipped with a powerful interface that represents network data in a clean, organized, and colorful fashion. Wireshark grants power to its users in a point-and-click way.

Where does this leave WinDump? WinDump is a lot less attractive. The data is simply displayed within a command line window. No color. No robust formatting. However, this lightweight approach is what gives WinDump tremendous power. The utility has little overhead and can process a ton of packets in a small amount of time.

Users interact with WinDump via the command line. The structure of commands is comprised of two parts: options and expressions (*See Appendix A for a complete list of WinDump options*).

Options allow one to specify which interface to capture, the level of verbosity, whether or not to perform automatic hostname lookups, among others. Expressions are used to determine *exactly* what type of traffic to display. Users can filter on IP addresses, protocols, and ports. Expressions also support the use of "and", "or", and parenthetical groupings (Van Styn, 2011).

WinDump's minimalistic approach yields much power. The ability to carve packet captures with expressions allows one to identify anomalous patterns. Or to simply troubleshoot bizarre network behaviors.

2.2. What are WinDump's Deficiencies?

WinDump is quick and powerful. WinDump filters, also called Berkeley Packet Filters (BPF), afford the ability to filter on very specific protocol properties (whether individual bits are on within TCP, for example). The problem is that BPF filters require a deep understanding of multiple concepts: byte offsets, bit masking, and binary arithmetic. In other words, there is no easy way to specify, "Show me all packets that have the SYN and ACK flags set within TCP communication".

The statement would have to be specified as: "TCP[13] == Ox12". The protocol, TCP, is first specified. Using bracket notation, the byte offset (13) is then specified which corresponds to the position of the TCP flags within the TCP header. The hexadecimal value, 12, is inserted to represent which bits (flags) to look for. Essentially, the hexadecimal is translated into binary, which then translates into specific bits (that represent TCP flags).

There is a lot of information to process in order to capture a simple concept. It would be extremely beneficial if the same filter could be represented in a natural way: "TCP with SYN and ACK".

3. PowerShell – A Brief Overview and History

PowerShell is a Windows scripting language that debuted in 2006. Since, it has become heavily integrated with multiple core Microsoft technologies. As it relates to the security community, PowerShell has become a trending technology. There have been attacks that leveraged PowerShell. There are also several defender tools written in PowerShell.

Jeffrey Snover, a Technical Fellow at Microsoft, is the founder of PowerShell. In Snover's original Monad Manifesto, the core idea and value proposition are discussed: a scripting language that boasts an extremely easy syntax. All PowerShell commands, called "cmdlets" (pronounced "command lets"), follow a verb-noun syntax: Get-Eventlog. Get-ADForest. Restart-Service. New-GPO.

The result of cmdlets is in the form of objects. This means that users can manipulate results in an object-oriented fashion, as PowerShell objects have properties and methods.

3.1. Mapping PowerShell's Features to WinDump's Deficiencies

PowerShell's natural syntax makes it an excellent candidate for automating WinDump's core features. The complexity of WinDump's common use-cases (specifying TCP flags in the earlier example) can be extrapolated into a PowerShell wrapper. "TCP[13] == Ox12" could be translated into "Invoke-WinDump –TCPFlags 'SYN,ACK'". The byte offsets and hexadecimal numbers could be handled programmatically under the hood.

4. Invoke-WinDump: Value Proposition

Invoke-WinDump is a custom PowerShell module intended to simplify the use of WinDump. The idea focuses on the following value proposition:

- Extraordinarily easy syntax
- Elimination of byte offsets, hexadecimal and bit masking
- Searchable text patterns
- Lightning fast processing

4.1. Getting Started: Mapping Use-Cases

The intended outcome of Invoke-WinDump needs to be crystal clear: "How will users interact with Invoke-WinDump?" "What are the common protocols and header attributes that need to be easily specified?" The blueprint of Invoke-WinDump is designed based on these types of common use-cases.

Implementing a clean syntax is dependent on the underlying WinDump filters. And the underlying WinDump filters require an understanding of the respective protocol headers. What protocol support will Invoke-WinDump include?

- IPv4
- TCP
- UDP
- ICMP

WinDump is already equipped with a mechanism that allows users to specify various protocol attributes in an easy way. Users can use predefined keywords, called primitives (or macros), such as "src host", dst port", "ip6", "tcp", "udp", and so on. An understanding of WinDump's built-in primitives makes it easier to derive use-cases for Invoke-WinDump.

The headers for each of WinDump's target protocols is evaluated during the design phase. Each of the headers are then distilled into matrixes that depict each protocol field, byte offset location, the size of the field, and whether or not WinDump already has a primitive for that field.

Internet Protocol Version 4 (IPv4) has a 20-byte header field that contains various properties:

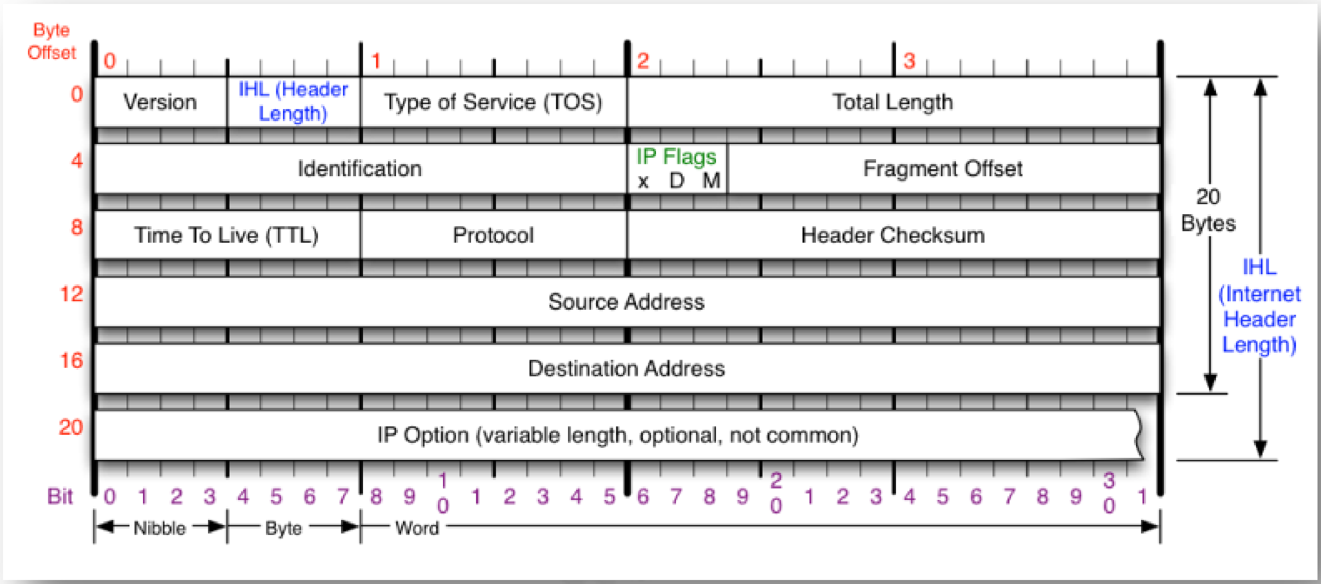


Figure 1 - IPv4 Header (NMAP, n.d.)

The IPv4 header is distilled into the following table:

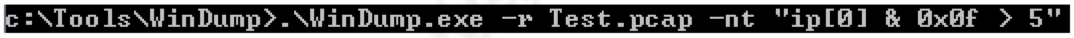
IP Header Attribute	Byte Offset	Size	Primitive
Version	0	4 bits	ip, ip6
IHL (Header Length)	0	4 bits	n/a
Type of Service (TOS)	1	1 byte	n/a
Total Length	2	2 bytes	n/a
Identification	4	2 bytes	n/a
IP Flags	6	3 bits	n/a
Fragment Offset	6	13 bits	n/a

Time to Live (TTL)	8	1 byte	n/a
Protocol	9	1 byte	tcp, udp, icmp
Header Checksum	10	2 bytes	n/a
Source Address	12	4 bytes	src host
Destination Address	16	4 bytes	dst host

Table 1 - IPv4 Header – Mapping WinDump Primitives

Every header field is compared against WinDump's built-in primitives. Column 4 depicts that there are several fields that do not have corresponding primitives. This means that users have to specify a particular field's byte offset position, and in some cases, provide a bit mask, in order to filter when using WinDump. The goal is to identify every protocol header field that does not have a primitive. This provides a roadmap for developing use-cases for Invoke-WinDump.

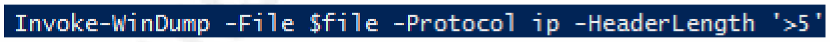
Here is an example of using WinDump to check for packets that include IP options:



```
c:\Tools\WinDump>.\WinDump.exe -r Test.pcap -nt "ip[0] & 0x0f  > 5"
```

Figure 2- WinDump: Looking for IP Options

The Invoke-WinDump equivalent eliminates the need to provide a byte offset and a bitmask:



```
Invoke-WinDump -File $file -Protocol ip -HeaderLength '>5'
```

Figure 3 - Invoke-WinDump: Looking for IP Options

The process of understanding each protocol header, and which fields have a built-in primitive, is pertinent to mapping the functionality of Invoke-WinDump.

(*See Appendix B for TCP, UDP, and ICMP header illustrations, along with primitive mappings.*)

4.1.1. Building the Invoke-WinDump Framework

There is now a clear roadmap for Invoke-WinDump. Invoke-WinDump will support IPv4, TCP, UDP, and ICMP. Each of these protocol headers have been evaluated to understand which have existing macros. One of the primary objectives is providing an easy way to specify any of the protocol header fields.

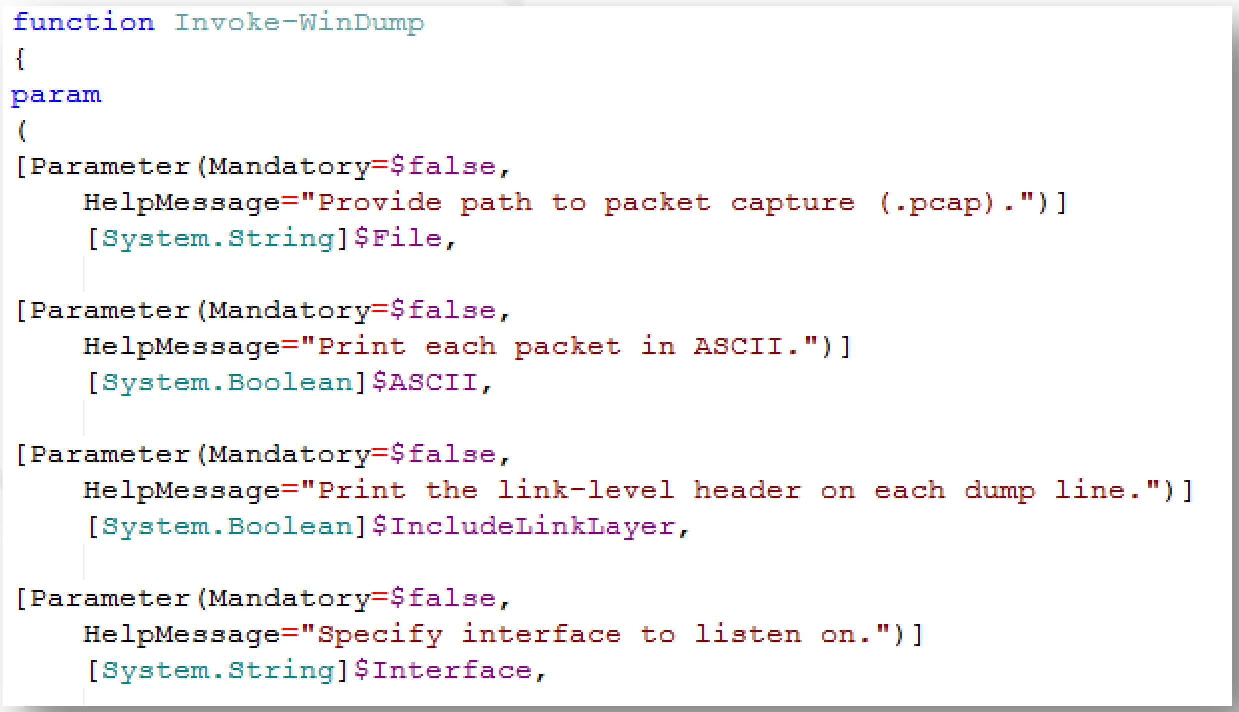
Invoke-WinDump is a collection of PowerShell scripts (.ps1 files) that are wrapped into a module. Users interact with the tool via *Invoke-WinDump.ps1*. The primary script will be responsible for capturing user input and will rely on other script files for processing. The dependent scripts are comprised of helper functions.

Invoke-WinDump.ps1 has several parameter options. These parameters are inspired by the WinDump parameter options. WinDump has many command-line argument options. Invoke-WinDump incorporates the most common ones. The *Invoke-WinDump.ps1* starts off by defining these common parameter options:

```
$File, $ASCII, $IncludeLinkLayer, $Interface, $Quiet, $Verbose
```

Figure 4 - Invoke-WinDump Common Parameters

Here is a snippet of the function parameter definitions:

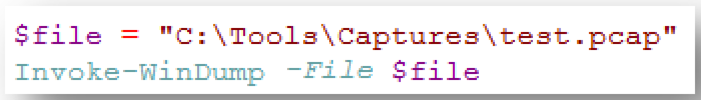


```
function Invoke-WinDump
{
param
(
[Parameter(Mandatory=$false,
    HelpMessage="Provide path to packet capture (.pcap).")]
    [System.String]$File,

[Parameter(Mandatory=$false,
    HelpMessage="Print each packet in ASCII.")]
    [System.Boolean]$ASCII,

[Parameter(Mandatory=$false,
    HelpMessage="Print the link-level header on each dump line.")]
    [System.Boolean]$IncludeLinkLayer,

[Parameter(Mandatory=$false,
    HelpMessage="Specify interface to listen on.")]
    [System.String]$Interface,
```

Figure 5 - Invoke-WinDump Parameter Definitions

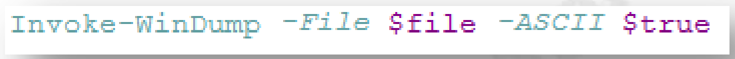
Each parameter is declared as a variable with a data type. In addition, each parameter specifies a help message that is displayed to the user when selecting options. The two datatypes used are System.String and System.Boolean. These datatypes dictate how users interact with tool and what types of values can be provided to the parameters.

For example, users can specify which packet capture file to read from. Users can also depict whether or not to display ASCII while processing packet captures:



```
$file = "C:\Tools\Captures\test.pcap"
Invoke-WinDump -File $file
```

Figure 6 - Providing Value to -File Parameter



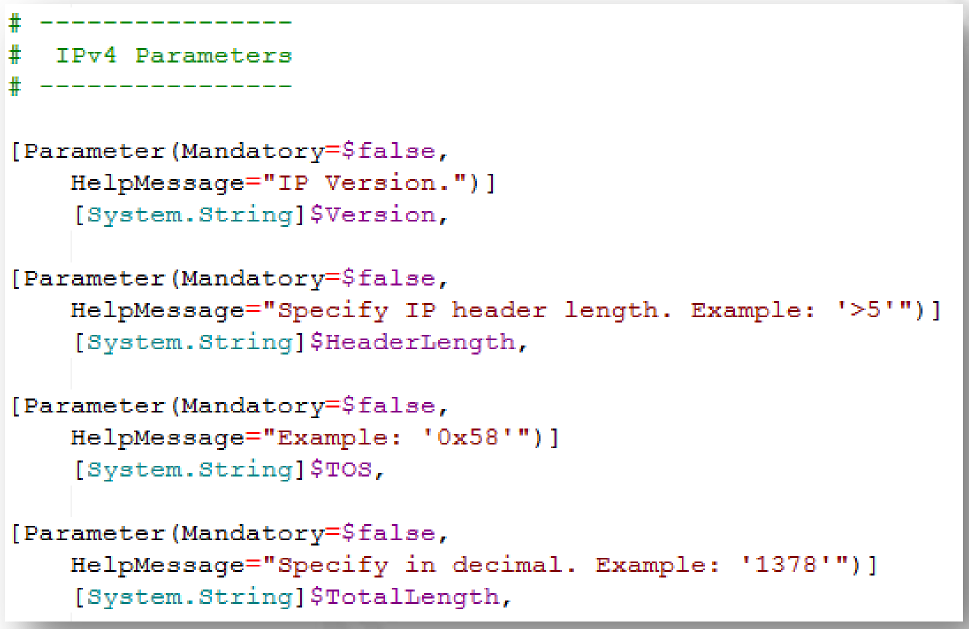
```
Invoke-WinDump -File $file -ASCII $true
```

Figure 7 - Choosing to Display ASCII

Robert L. Adams, robert.louis.adams@gmail.com

5. Implementing Key Feature #1 – Easily Specified IPv4, TCP, UDP, and ICMP Header Fields

The *Invoke-WinDump.ps1* parameter definitions include several other options as well. In fact, there is a parameter for each and every IPv4, TCP, UDP, and ICMP header fields. Here is a snippet of the parameters that are related to IPv4:



```
# ----------------
#  IPv4 Parameters
# ----------------

[Parameter(Mandatory=$false,
    HelpMessage="IP Version.")]
    [System.String]$Version,

[Parameter(Mandatory=$false,
    HelpMessage="Specify IP header length. Example: '>5'")]
    [System.String]$HeaderLength,

[Parameter(Mandatory=$false,
    HelpMessage="Example: '0x58'")]
    [System.String]$TOS,

[Parameter(Mandatory=$false,
    HelpMessage="Specify in decimal. Example: '1378'")]
    [System.String]$TotalLength,
```

Figure 8 - IPv4 Parameter Definitions

The screenshot depicts how the IPv4 $Version, $HeaderLength, $TOS,and $TotalLength parameters are defined. The parameter list continues in this fashion for the remaining IPv4, TCP, UDP, and ICMP header fields.

5.1. How Are All of These Parameters Processed?

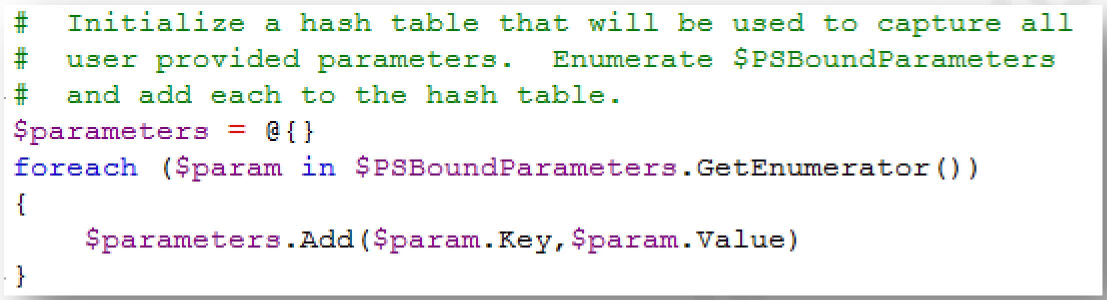
Invoke-WinDump contains numerous parameter options. Users can specify any of the IPv4, TCP, UDP, and ICMP header fields in a natural way. The question remains: “How are these parameters mapped to the underlying execution of WinDump?”

Pseudocode can help answer this question.

1. Enumerate all user-provided parameters

2. Store the parameters into a hash table
3. Create a lookup between the values in the hash table and all protocol header fields
4. Convert the values into a WinDump acceptable expression format

The body of the Invoke-WinDump function begins with the following code:

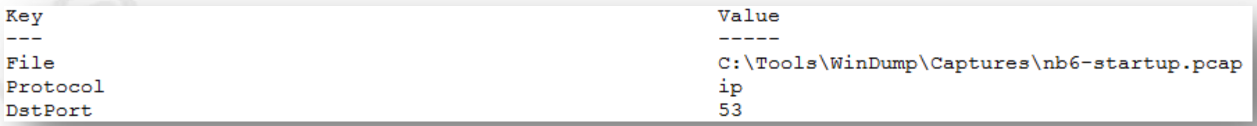


```
#  Initialize a hash table that will be used to capture all
#  user provided parameters.  Enumerate $PSBoundParameters
#  and add each to the hash table.
$parameters = @{}
foreach ($param in $PSBoundParameters.GetEnumerator())
{
    $parameters.Add($param.Key,$param.Value)
}
```

Figure 9 - Start of Invoke-WinDump Function Body

The code starts by initializing the $parameters hash table. The values of $PSBoundParameters is then enumerated and stored into the hash table: in key-value pairs.

$PSBoundParameters is a default PowerShell variable (System.Management.Automation.PSBoundParametersDictionary) that holds all user-provider parameter values. For example, the output of $PSBoundParameters looks like this:

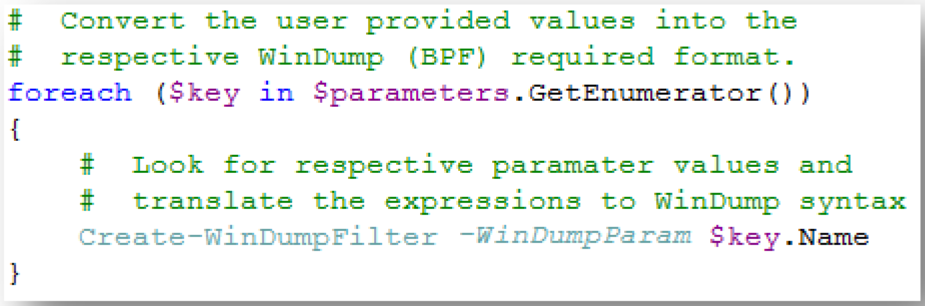


```
Key                                                        Value
---                                                        -----
File                                                       C:\Tools\WinDump\Captures\nb6-startup.pcap
Protocol                                                   ip
DstPort                                                    53
```

Figure 10 - Looking at $PSBoundParameters

The newly created hash table is then enumerated in order to inspect the user provided values. These values are then translated into WinDump-compatible expressions:

Robert L. Adams, robert.louis.adams@gmail.com



```
#  Convert the user provided values into the
#  respective WinDump (BPF) required format.
foreach ($key in $parameters.GetEnumerator())
{
    #  Look for respective paramater values and
    #  translate the expressions to WinDump syntax
    Create-WinDumpFilter -WinDumpParam $key.Name
}
```

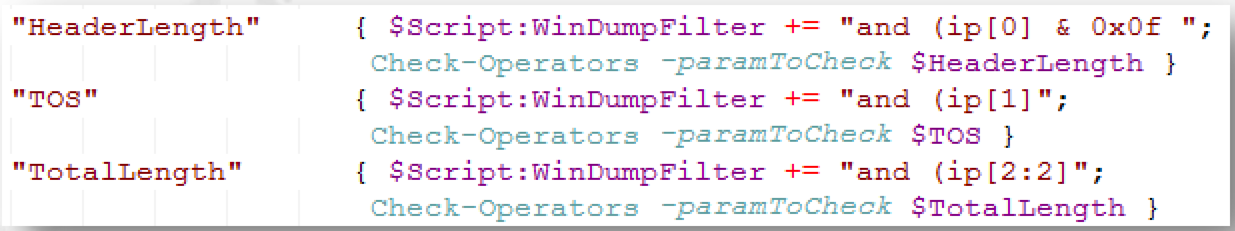
Figure 11 - Enumeration and Translation of Parameter Values

The hash table is enumerated and the name of each parameter is passed to the Create-WinDumpFilter function. Each parameter name (which is stored in the hash table) serves as an index that will be used to conduct translation of the value.

The body of the function is really simple. It contains one switch statement. The case statements map to all of the protocols' header fields. These case statements are organized into sections that represent each protocol: IPv4, TCP, UDP, and ICMP.

The parameter names (in the form of $key.Name) that were passed to the function serve as the switch statement's keyword.

Here is a snippet of the switch statement:

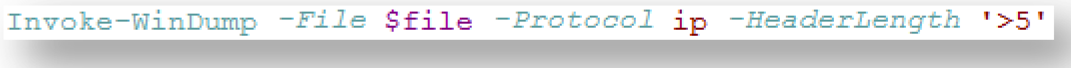


```
"HeaderLength"      { $Script:WinDumpFilter += "and (ip[0] & 0x0f ";
                      Check-Operators -paramToCheck $HeaderLength }
"TOS"               { $Script:WinDumpFilter += "and (ip[1]";
                      Check-Operators -paramToCheck $TOS }
"TotalLength"       { $Script:WinDumpFilter += "and (ip[2:2]";
                      Check-Operators -paramToCheck $TotalLength }
```

Figure 12- Create-WinDumpFilter's Switch Statement

An examination of how one of these case statements gets invoked illustrates how all of the pieces work together. Example: Using Invoke-WinDump to find all packets that have IP options.

Robert L. Adams, robert.louis.adams@gmail.com



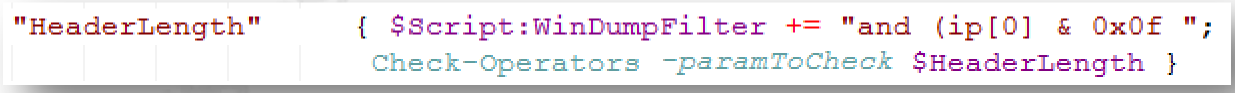
```
Invoke-WinDump -File $file -Protocol ip -HeaderLength '>5'
```

Figure 13 - Finding IP Options with Invoke-WinDump

There are two parameters in our example: protocol and header length. As depicted in the code above, these parameters and values are dissected from $PSBoundParameters and stored in the $parameters hash table. The hash table looks like this:

Name:	***Value:***
Protocol	“ip”
HeaderLength	“>5”

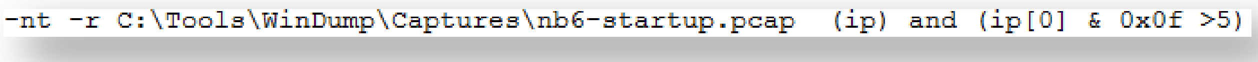
The key-pair values are then passed to the Create-WinDumpFilter function in order to translate the values into WinDump-type expressions. A switch statement is used to depict what processing needs to take place per the parameters specified. HeaderLength, for example:



```
"HeaderLength"       { $Script:WinDumpFilter += "and (ip[0] & 0x0f ";
                        Check-Operators -paramToCheck $HeaderLength }
```

Figure 14 - Taking a Deeper Look at "HeaderLength"

$Script:WinDumpFilter is a string that is used to construct the filter that will get passed to the underlying WinDump.exe. In this case, the results filter looks like this:



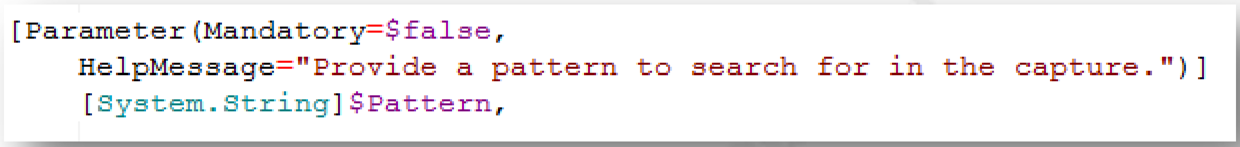
```
-nt -r C:\Tools\WinDump\Captures\nb6-startup.pcap  (ip) and (ip[0] & 0x0f >5)
```

Figure 15 - The Resulting WinDump Filter

6. Implementing Key Feature #2 – Easily Searchable Text Patterns

The functionality of WinDump can be further extended by incorporating text-based search. This could be useful when searching for a domain name, file name, or any other pattern within a packet capture.

A new parameter is added to the Invoke-WinDump function:

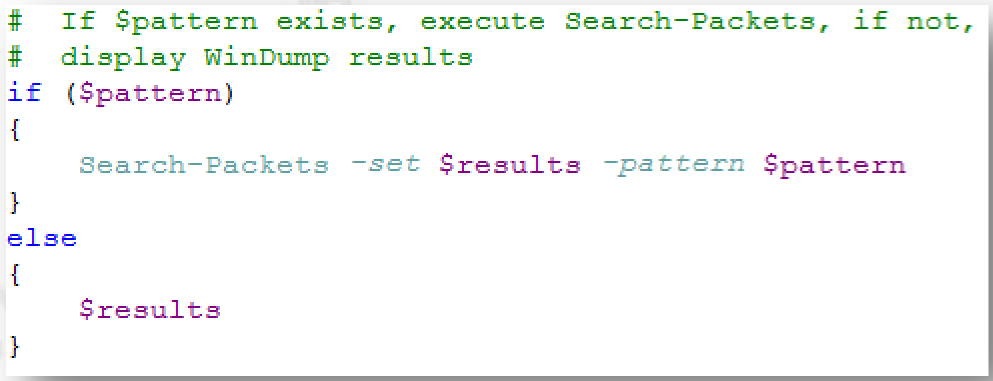


```
[Parameter(Mandatory=$false,
    HelpMessage="Provide a pattern to search for in the capture.")]
    [System.String]$Pattern,
```

Figure 16 - Adding a new function parameter: pattern

The new parameter is responsible for accepting a string. The user-provided string will then be used to search against the packet capture.

The following code has been added to the bottom of the main script, *Invoke-WinDump.ps1*:

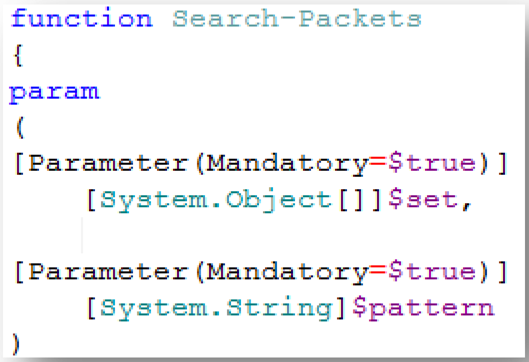


```
#   If $pattern exists, execute Search-Packets, if not,
#   display WinDump results
if ($pattern)
{
    Search-Packets -set $results -pattern $pattern
}
else
{
    $results
}
```

The snippet checks for the presence of the $pattern parameter. If the user specifies a string to search for, the code then relies on the Search-Packets helper function. If not, the results from WinDump are displayed on screen.

Search-Packets is a function that searches the WinDump results for the presence of the user-specified string. For example, a user can specify that they want to search the packet capture for the presence of "evildomain.com".

Search-Packets accepts two parameters:



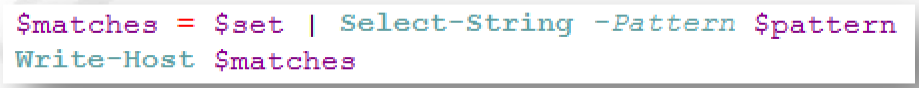
```
function Search-Packets
{
param
(
[Parameter(Mandatory=$true)]
    [System.Object[]]$set,

[Parameter(Mandatory=$true)]
    [System.String]$pattern
)
```

Figure 17 - Search-Packets

$set is used to store the results of WinDump and $pattern is the user-specified search string. Why is $set defined as an array of objects? WinDump returns an array of objects when executed, therefore, the data type remains consistent in the helper function.

The Search-Packets helper function relies on PowerShell's **Select-String** under the hood. The PowerShell help manual describes Select-String as PowerShell's version of Unix's grep.

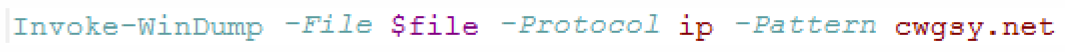
The core of the Search-Packets function is quite simplistic:



```
$matches = $set | Select-String -Pattern $pattern
Write-Host $matches
```

The results of WinDump (represented as $set) is passed to PowerShell's **Select-String**. The user-specified string is fed to Select-String's pattern parameter. The search results are saved to the $matches variable and outputted to the screen.

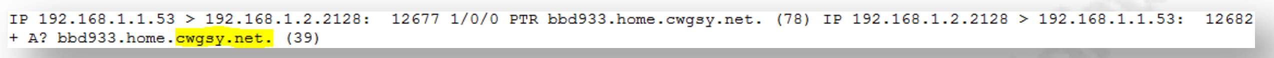
Here is an example of using the search function:



```
Invoke-WinDump -File $file -Protocol ip -Pattern cwgsy.net
```

Figure 18 - Searching a Domain

The results:



```
IP 192.168.1.1.53 > 192.168.1.2.2128:  12677 1/0/0 PTR bbd933.home.cwgsy.net. (78) IP 192.168.1.2.2128 > 192.168.1.1.53:  12682
+ A? bbd933.home.cwgsy.net. (39)
```

Figure 19 – Searching a Domain - Results

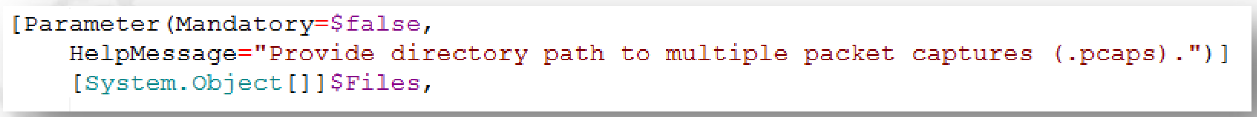
The original packet capture contains 2,263 packets. Using the new search feature yields a single result for the search string, “cwgsy.net”. The returned packet is a DNS query containing the specified search string.

(*The packet capture was derived from WireShark.org’s samples (WireShark, n.d.)*).

7. Implementing Key Feature #3 – Processing Multiple Packet Captures

The final feature of Invoke-WinDump will be focused on processing multiple packet captures. An analyst may be interested in a domain IOC related to DNS traffic. It would be useful to specify the filter once and apply it to a directory full of packet captures.

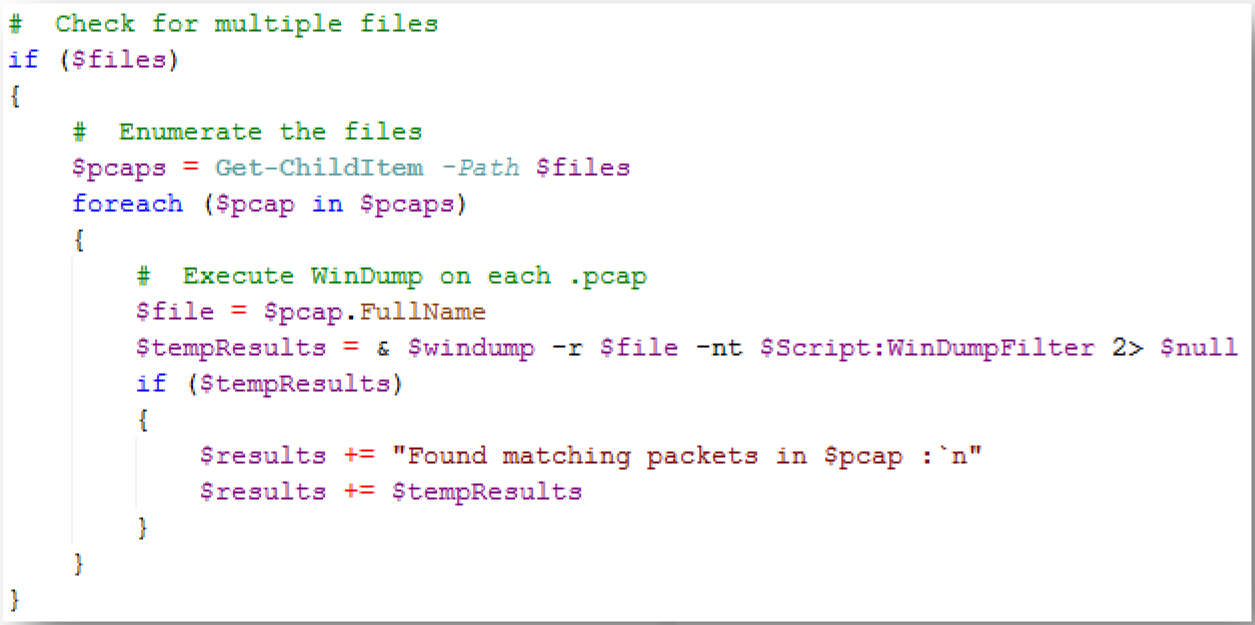
A new parameter is added at the top of the main *Invoke-WinDump.ps1* script:



```
[Parameter(Mandatory=$false,
    HelpMessage="Provide directory path to multiple packet captures (.pcaps).")]
    [System.Object[]]$Files,
```

Figure 20 - New $Files parameter

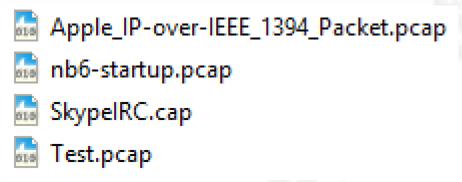
The bottom of the script will introduce a new conditional statement that will check for the presence of the $Files parameter. If the user does specify a directory (containing multiple packet captures), the code will read the files and execute WinDump.exe on each file:



```
#  Check for multiple files
if ($files)
{
    #  Enumerate the files
    $pcaps = Get-ChildItem -Path $files
    foreach ($pcap in $pcaps)
    {
        #  Execute WinDump on each .pcap
        $file = $pcap.FullName
        $tempResults = & $windump -r $file -nt $Script:WinDumpFilter 2> $null
        if ($tempResults)
        {
            $results += "Found matching packets in $pcap :`n"
            $results += $tempResults
        }
    }
}
```

Figure 21- Enumerating Multiple Packet Captures

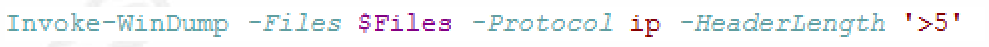
The functionality is tested on a handful of packet captures:



```
Apple_IP-over-IEEE_1394_Packet.pcap
nb6-startup.pcap
SkypeIRC.cap
Test.pcap
```

Figure 22- C : \ Tools \ Captures

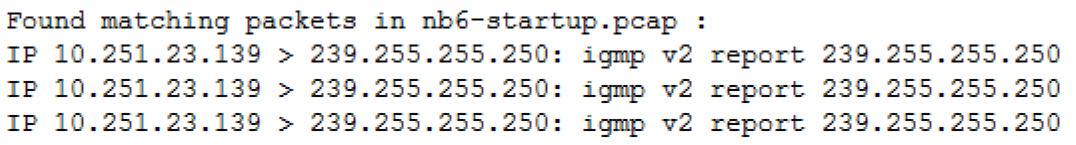
Invoke-WinDump is executed with the newly defined $Files parameter:



```
Invoke-WinDump -Files $Files -Protocol ip -HeaderLength '>5'
```

Figure 23 - Searching Multiple PCAPS

Invoke-WinDump returns any packet (across all files) where there are IP options:



```
Found matching packets in nb6-startup.pcap :
IP 10.251.23.139 > 239.255.255.250: igmp v2 report 239.255.255.250
IP 10.251.23.139 > 239.255.255.250: igmp v2 report 239.255.255.250
IP 10.251.23.139 > 239.255.255.250: igmp v2 report 239.255.255.250
```

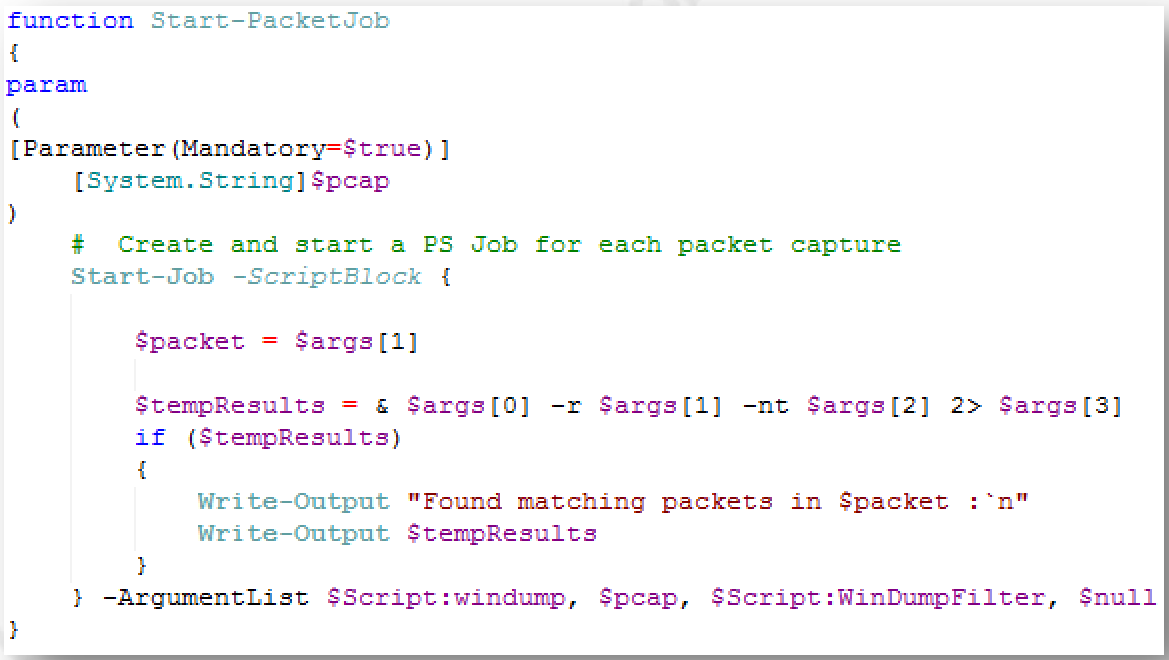
Figure 24- Finding IP Options Results

The script output annotates which packet capture (nb6-startup.pcap) contained the resulting matches, and then displays the 3 corresponding packets.

7.1. Performance Gains

Invoke-WinDump currently processes multiple packet captures sequentially. This works fine for a limited number of reasonably sized captures. The script can be further enhanced by processing packet captures concurrently. This can be achieved by leveraging PowerShell jobs.

Enumerating multiple packet captures and executing WinDump is demonstrated in the previous section. The same code will be repurposed into the body of a new helper function:

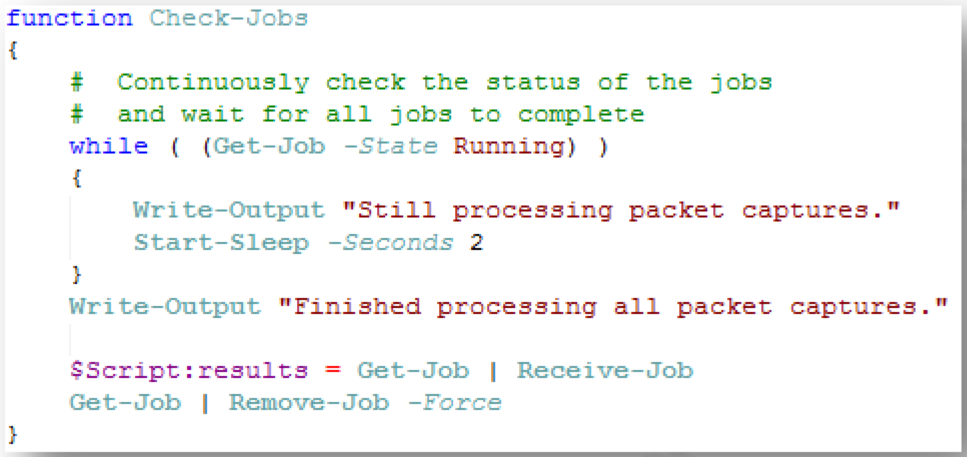


```
function Start-PacketJob
{
param
(
[Parameter(Mandatory=$true)]
    [System.String]$pcap
)
    #  Create and start a PS Job for each packet capture
    Start-Job -ScriptBlock {

        $packet = $args[1]

        $tempResults = & $args[0] -r $args[1] -nt $args[2] 2> $args[3]
        if ($tempResults)
        {
            Write-Output "Found matching packets in $packet :`n"
            Write-Output $tempResults
        }
    } -ArgumentList $Script:windump, $pcap, $Script:WinDumpFilter, $null
}
```

Figure 25 - Start-PacketJob Function

Start-PacketJob is called as the main script (*Invoke-WinDump.ps1*) enumerates the user provided directory containing multiple packet captures. Essentially, the function instantiates a new PowerShell job for each and every packet capture.

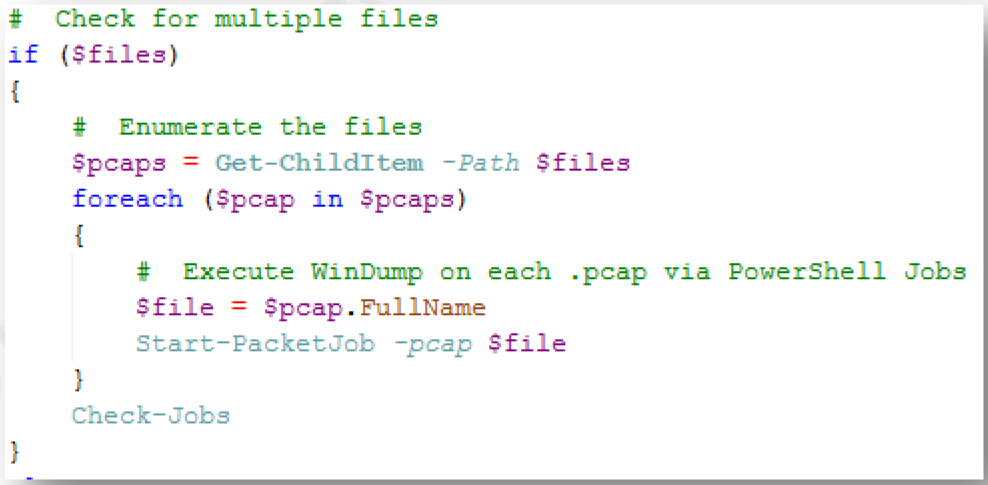
A second helper function is used to keep track of the job statuses and alerts when all the jobs have completed—signifying that WinDump has processed each packet capture:



```
function Check-Jobs
{
    #  Continuously check the status of the jobs
    #  and wait for all jobs to complete
    while ( (Get-Job -State Running) )
    {
        Write-Output "Still processing packet captures."
        Start-Sleep -Seconds 2
    }
    Write-Output "Finished processing all packet captures."

    $Script:results = Get-Job | Receive-Job
    Get-Job | Remove-Job -Force
}
```

Figure 26 - Check-Jobs Function

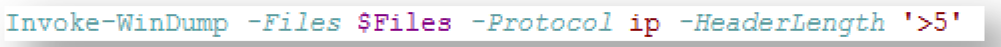
The bottom of the main *Invoke-WinDump.ps1* script has been modified to incorporate the new helper functions. Here is how the main script relies on the new functions:



```
#  Check for multiple files
if ($files)
{
    #  Enumerate the files
    $pcaps = Get-ChildItem -Path $files
    foreach ($pcap in $pcaps)
    {
        #  Execute WinDump on each .pcap via PowerShell Jobs
        $file = $pcap.FullName
        Start-PacketJob -pcap $file
    }
    Check-Jobs
}
```

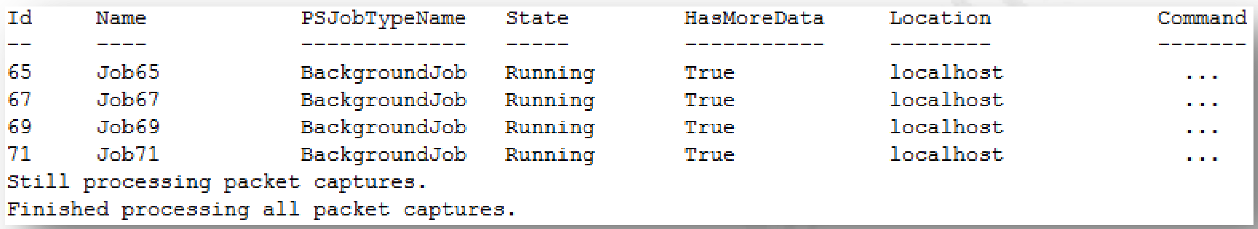
Figure 27 – Enumerating PCAPs and Starting Jobs

Here is the output of executing Invoke-WinDump a second time with the updated code:



```
Invoke-WinDump -Files $Files -Protocol ip -HeaderLength '>5'
```

Figure 28 - Searching Multiple PCAPS



```
Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
65     Job65           BackgroundJob   Running       True            localhost            ...
67     Job67           BackgroundJob   Running       True            localhost            ...
69     Job69           BackgroundJob   Running       True            localhost            ...
71     Job71           BackgroundJob   Running       True            localhost            ...
Still processing packet captures.
Finished processing all packet captures.
```

Figure 29 - Output of Updated Invoke-WinDump

The results of instantiating a job for each packet capture file (4 in total) is now displayed. The status of the jobs is also displayed to the console as a result of the Check-Jobs function.

The matching packets are exactly the same as before:

```
IP 10.251.23.139 > 239.255.255.250: igmp v2 report 239.255.255.250
IP 10.251.23.139 > 239.255.255.250: igmp v2 report 239.255.255.250
IP 10.251.23.139 > 239.255.255.250: igmp v2 report 239.255.255.250
```

Figure 30 - Resulting Packets

Parallel execution increases the power of WinDump and is extremely useful for carving several packet captures simultaneously.

8. Putting It All Together

The Invoke-WinDump.ps1 script is now equipped with the functionality to easily search for various header attributes in IP, TCP, UDP, and ICMP. In addition, the script can perform text-based pattern search and concurrently process packet captures. The main Invoke-WinDump.ps1 file relies on a handful of helper functions that are defined in dependent script files: Create-WinDumpFilter.ps1, Search-Pattern.ps1, and Start-PacketJob.ps1.

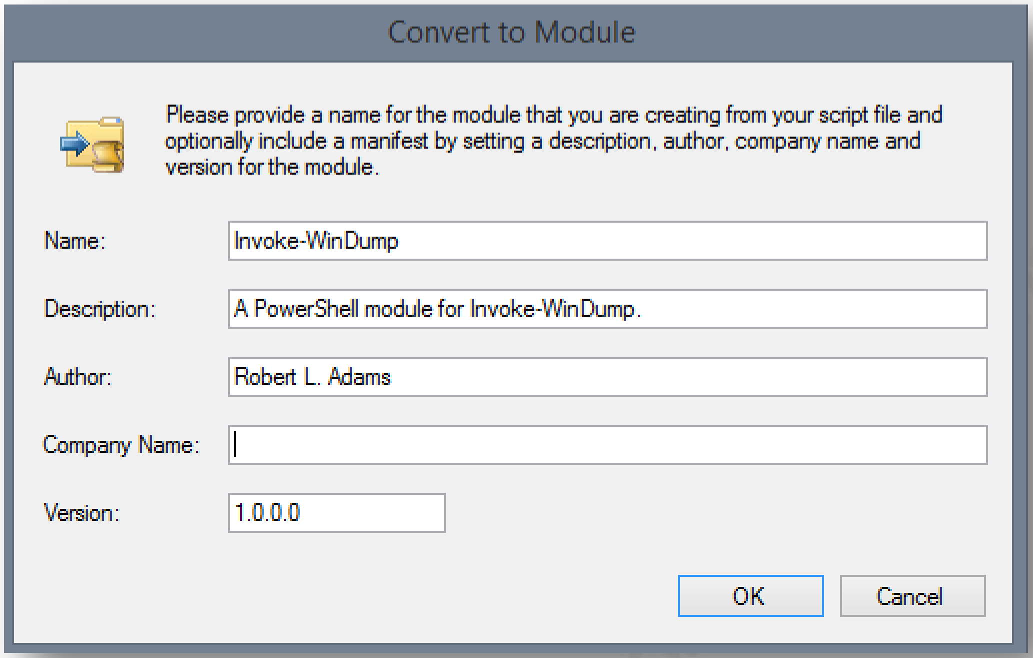
Invoke-WinDump can be distributed to users by sending them all of the script files. However, a better solution would be aggregating all of the files into a PowerShell module. A PowerShell module is essentially a collection of related PowerShell functionalities. There are 4 types of PowerShell modules:

1. Script Modules
2. Binary Modules
3. Manifest Modules
4. Dynamic Modules

A script module is simply a file (.psm1) that contains PowerShell code—to include functions and variables. Users can then load the module which contains all of the relevant code (Microsoft, n.d.).

PowerGUI is a third-party PowerShell IDE (integrated development environment) that has support for converting PowerShell code into a module. The plugin essentially creates the necessary .psm1 file on behalf of the user.

Users can use the "*Convert to Module…*" functionality (from the File dropdown menu) to handle this task:



Please provide a name for the module that you are creating from your script file and optionally include a manifest by setting a description, author, company name and version for the module.

Name: Invoke-WinDump

Description: A PowerShell module for Invoke-WinDump.

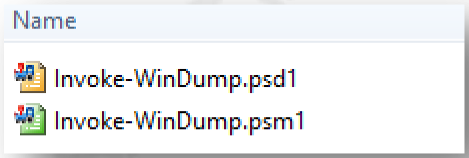
Author: Robert L. Adams

Company Name:

Version: 1.0.0.0

Figure 31 - PowerGUI's Convert to Module

Two files are created within the newly created **Invoke-WinDump** directory: C : \ Users \ [User] \ Documents \ WindowsPowerShell \ Modules \ Invoke-WinDump:

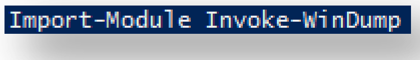


Name
Invoke-WinDump.psd1
Invoke-WinDump.psm1

Figure 32 - Created .psd1 & .psm1 Files

The newly created Invoke-WinDump directory can be copied/moved to C:\Windows\System32\WindowsPowerShell\v1.0\Modules to make it accessible to everyone.

Finally, the newly created module is imported into a new PowerShell session:



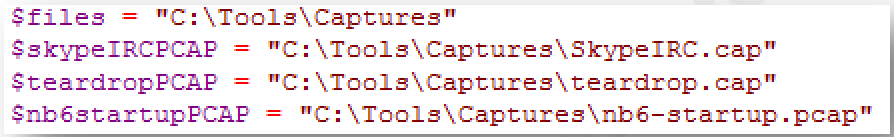
```
Import-Module Invoke-WinDump
```

Figure 33 - Importing Invoke-WinDump Module

All of the module's functions are now available, including the main **Invoke-WinDump** function.

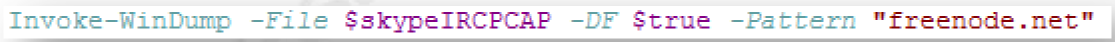
8.1. Invoke-WinDump in Action (Examples)

The following illustrates Invoke-WinDump in action. A couple of variables are initialized and defined to point to the packet captures that will be analyzed:



```
$files = "C:\Tools\Captures"
$skypeIRCPCAP = "C:\Tools\Captures\SkypeIRC.cap"
$teardropPCAP = "C:\Tools\Captures\teardrop.cap"
$nb6startupPCAP = "C:\Tools\Captures\nb6-startup.pcap"
```

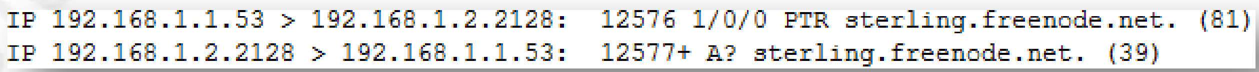
Example #1: Using Invoke-WinDump to look for packets within *SkypeIRC.cap* with the don't fragment (DF) flag set, and containing "freenode.net":



```
Invoke-WinDump -File $skypeIRCPCAP -DF $true -Pattern "freenode.net"
```

Figure 34 - Pattern Searching

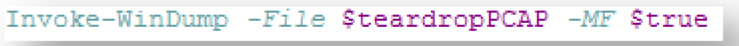
Results:



```
IP 192.168.1.1.53 > 192.168.1.2.2128:  12576 1/0/0 PTR sterling.freenode.net. (81)
IP 192.168.1.2.2128 > 192.168.1.1.53:  12577+ A? sterling.freenode.net. (39)
```

Figure 35 - Pattern Searching - Results

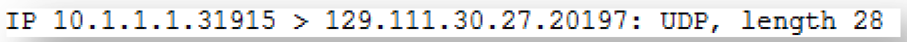
Example #2: Using Invoke-WinDump to look for packets within *teardrop.cap* with the more fragment (MF) flag set:



```
Invoke-WinDump -File $teardropPCAP -MF $true
```

Figure 36 - Looking for MF Flag

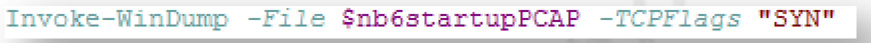
Results:



```
IP 10.1.1.1.31915 > 129.111.30.27.20197: UDP, length 28
```

Figure 37 - Looking for MF Flag - Results

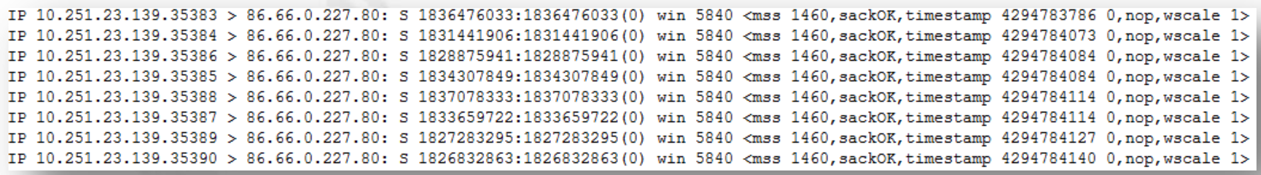
Example #3: Using Invoke-WinDump to look for packets within *nb6-startup.pcap* with only the SYN flag set:



```
Invoke-WinDump -File $nb6startupPCAP -TCPFlags "SYN"
```

Figure 38 - Looking for SYN Flag

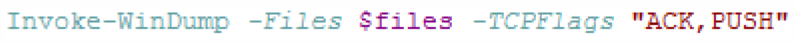
Results:



```
IP 10.251.23.139.35383 > 86.66.0.227.80: S 1836476033:1836476033(0) win 5840 <mss 1460,sackOK,timestamp 4294783786 0,nop,wscale 1>
IP 10.251.23.139.35384 > 86.66.0.227.80: S 1831441906:1831441906(0) win 5840 <mss 1460,sackOK,timestamp 4294784073 0,nop,wscale 1>
IP 10.251.23.139.35386 > 86.66.0.227.80: S 1828875941:1828875941(0) win 5840 <mss 1460,sackOK,timestamp 4294784084 0,nop,wscale 1>
IP 10.251.23.139.35385 > 86.66.0.227.80: S 1834307849:1834307849(0) win 5840 <mss 1460,sackOK,timestamp 4294784084 0,nop,wscale 1>
IP 10.251.23.139.35388 > 86.66.0.227.80: S 1837078333:1837078333(0) win 5840 <mss 1460,sackOK,timestamp 4294784114 0,nop,wscale 1>
IP 10.251.23.139.35387 > 86.66.0.227.80: S 1833659722:1833659722(0) win 5840 <mss 1460,sackOK,timestamp 4294784114 0,nop,wscale 1>
IP 10.251.23.139.35389 > 86.66.0.227.80: S 1827283295:1827283295(0) win 5840 <mss 1460,sackOK,timestamp 4294784127 0,nop,wscale 1>
IP 10.251.23.139.35390 > 86.66.0.227.80: S 1826832863:1826832863(0) win 5840 <mss 1460,sackOK,timestamp 4294784140 0,nop,wscale 1>
```

Figure 39 - Looking for SYN Flag - Results

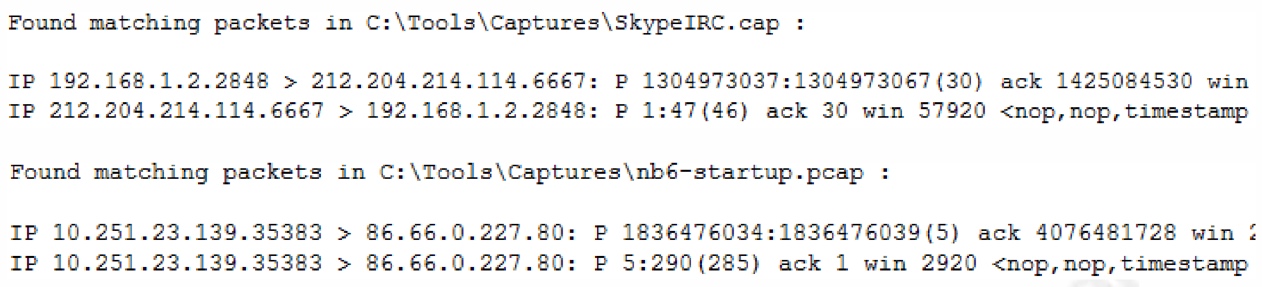
Example #4: Using Invoke-WinDump to look for packets within a directory full of captures, with only the ACK and PUSH flags set:



```
Invoke-WinDump -Files $files -TCPFlags "ACK,PUSH"
```

Figure 40 - Searching Across Multiple PCAPs

Results:



```
Found matching packets in C:\Tools\Captures\SkypeIRC.cap :

IP 192.168.1.2.2848 > 212.204.214.114.6667: P 1304973037:1304973067(30) ack 1425084530 win
IP 212.204.214.114.6667 > 192.168.1.2.2848: P 1:47(46) ack 30 win 57920 <nop,nop,timestamp

Found matching packets in C:\Tools\Captures\nb6-startup.pcap :

IP 10.251.23.139.35383 > 86.66.0.227.80: P 1836476034:1836476039(5) ack 4076481728 win 2
IP 10.251.23.139.35383 > 86.66.0.227.80: P 5:290(285) ack 1 win 2920 <nop,nop,timestamp
```

Figure 41 - Searching Across Multiple PCAPs - Results

9. Conclusion

WinDump is a packet capture and analysis tool that has been around for several years. WinDump is a quintessential security tool that will always have a spot in an analyst's toolbox. WinDump is lightweight and powerful, allowing users to trim down large packet captures in little time. WinDump is extremely easy to use. However, using the tool for granular filtering introduces a level of complexity.

Windows PowerShell is a performant scripting language that is equipped with a very simplistic syntax. Windows PowerShell can be used to automate many security tasks—and this has been the case here: PowerShell to script WinDump's more complicated use-cases.

The challenges that security professionals face today are greater than ever. It is important to work efficiently to combat the evolving threats. Automation should be a key focus in these efforts. PowerShell is extremely versatile and can be used to get the job done in a proficient and clear fashion. Security professionals should continuously look for ways to optimize their tools so that they can remain agile and effective.

References

Cane, B. (2014, October 13). *A Quick and Practical Reference for tcpdump*. Retrieved from Benjamin Cane: http://bencane.com/2014/10/13/quick-and-practical-reference-for-tcpdump/

Dittrich, D. (n.d.). *Notes About tcpdump Filters.* Retrieved from University of Washington: https://staff.washington.edu/dittrich/talks/core02/tools/tcpdump-filters.txt

Microsoft. (2015, August 9). *Get-Job*. Retrieved from Microsoft Developer Network: https://technet.microsoft.com/en-us/library/hh849693.aspx

Microsoft. (n.d.). *Windows PowerShell Module Concepts*. Retrieved from Microsoft TechNet: https://technet.microsoft.com/en-us/%5Clibrary/dd901839(v=vs.85).aspx

Microsoft. (n.d.). *Windows PowerShell Tip of the Week - Using the Switch Statement*. Retrieved from Microsoft TechNet: https://technet.microsoft.com/en-us/library/ff730937.aspx

Microsoft. (n.d.). *Windows PowerShell Tip of the Week - Working with Hash Tables*. Retrieved from Microsoft TechNet: https://technet.microsoft.com/en-us/library/ee692803.aspx

Miessler, D. (n.d.). *A tcpdump Primer with Examples*. Retrieved from Daniel Miessler: https://danielmiessler.com/study/tcpdump/

NMAP. (n.d.). *TCP/IP Reference*. Retrieved from NMAP.org: https://nmap.org/book/tcpip-ref.html

Stack Overflow. (2010, January 19). *Error When Calling 3rd Party Executable from Powershell when Using an IDE*. Retrieved from Stack Overflow: http://stackoverflow.com/questions/2095088/error-when-calling-3rd-party-executable-from-powershell-when-using-an-ide

Stack Overflow. (2011, March 8). *How Do I Get Help Messages to Appear for My Powershell Script Parameters?* Retrieved from Stack Overflowq: http://stackoverflow.com/questions/5237723/how-do-i-get-help-messages-to-appear-for-my-powershell-script-parameters

Stack Overflow. (2012, April 26). *Outputting PSBound Parameters*. Retrieved from Stack Overflow: http://stackoverflow.com/questions/10328083/outputting-psbound-parameters

Van Styn, H. (2011, December 19). *tcpdump fu*. Retrieved from Linux Journal: http://www.linuxjournal.com/content/tcpdump-fu

Wains, S. (2013, June 6). *tcpdump Advanced Filters*. Retrieved from Sebastien Wains: http://www.wains.be/pub/networking/tcpdump_advanced_filters.txt

WinDump. (2013). *WinDump Overview*. Retrieved from WinDump: https://www.winpcap.org/windump/default.htm

WireShark. (n.d.). *Sample Captures*. Retrieved from WireShark Wiki: https://wiki.wireshark.org/SampleCaptures

Appendix A – WinDump Command-Line Options

Option	Description
-A	Print each packet in ASCII.
-c	Exit after receiving *count* packets.
-C	Before writing a raw packet to a savefile, check whether the file is currently larger than *file_size*
-d	Dump the compiled packet-matching code in a human readable form to standard output and stop.
-dd	Dump packet-matching code as a **C** program fragment.
-ddd	Dump packet-matching code as decimal numbers (preceded with a count).
-D	Print the list of the network interfaces available on the system and on which *tcpdump/WinDump* can capture packets.
-e	Print the link-level header on each dump line.
-E	Use spi@ipaddr *algo:secret* for decrypting IPsec ESP packets that are addressed to *addr*and contain Security Parameter Index value *spi*. This combination may be repeated with comma or newline separation.
-f	Print `foreign' IPv4 addresses numerically rather than symbolically.
-F	Use *file* as input for the filter expression. An additional expression given on the command line is ignored.
-i	Listen on *interface*. If unspecified, *tcpdump* searches the system interface list for the lowest numbered, configured up interface (excluding loopback).
-l	Make stdout line buffered.
-L	List the known data link types for the interface and exit.
-m	Load SMI MIB module definitions from file *module*.
-M	Use *secret* as a shared secret for validating the digests found in TCP segments with the TCP-MD5 option (RFC 2385), if present.

-n	Don't convert addresses (i.e., host addresses, port numbers, etc.) to names.
-N	Don't print domain name qualification of host names.
-O	Do not run the packet-matching code optimizer.
-p	*Don't* put the interface into promiscuous mode.
-q	Quick (quiet?) output. Print less protocol information so output lines are shorter.
-R	Assume ESP/AH packets to be based on old specification (RFC1825 to RFC1829).
-r	Read packets from *file* (which was created with the **-w** option). Standard input is used if *file*is ``-''.
-S	Print absolute, rather than relative, TCP sequence numbers.
-s	Snarf *snaplen* bytes of data from each packet rather than the default of 68.
-T	Force packets selected by "*expression*" to be interpreted the specified *type*.
-t	*Don't* print a timestamp on each dump line.
-tt	Print an unformatted timestamp on each dump line.
-ttt	Print a delta (in micro-seconds) between current and previous line on each dump line.
-tttt	Print a timestamp in default format proceeded by date on each dump line.
-u	Print undecoded NFS handles.
-U	Make output saved via the **-w** option ``packet-buffered''.
-v	When parsing and printing, produce (slightly more) verbose output.
-vv	Even more verbose output. For example, additional fields are printed from NFS reply packets, and SMB packets are fully decoded.
-vvv	Even more verbose output. For example, telnet **SB** ... **SE** options are printed in full. With **-X**Telnet options are printed in hex as well.
-w	Write the raw packets to *file* rather than parsing and printing them out.
-W	Used in conjunction with the **-C** option, this will limit the number of files created to the

	specified number.
-x	When parsing and printing, in addition to printing the headers of each packet, print the data of each packet (minus its link level header) in hex.
-xx	When parsing and printing, in addition to printing the headers of each packet, print the data of each packet, *including* its link level header, in hex.
-X	When parsing and printing, in addition to printing the headers of each packet, print the data of each packet (minus its link level header) in hex and ASCII.
-XX	When parsing and printing, in addition to printing the headers of each packet, print the data of each packet, *including* its link level header, in hex and ASCII.
-y	Set the data link type to use while capturing packets to *datalinktype*.
-Z	Drops privileges (if root) and changes user ID to *user* and the group ID to the primary group of *user*.

Appendix B – TCP, UDP, and ICMP Illustrations

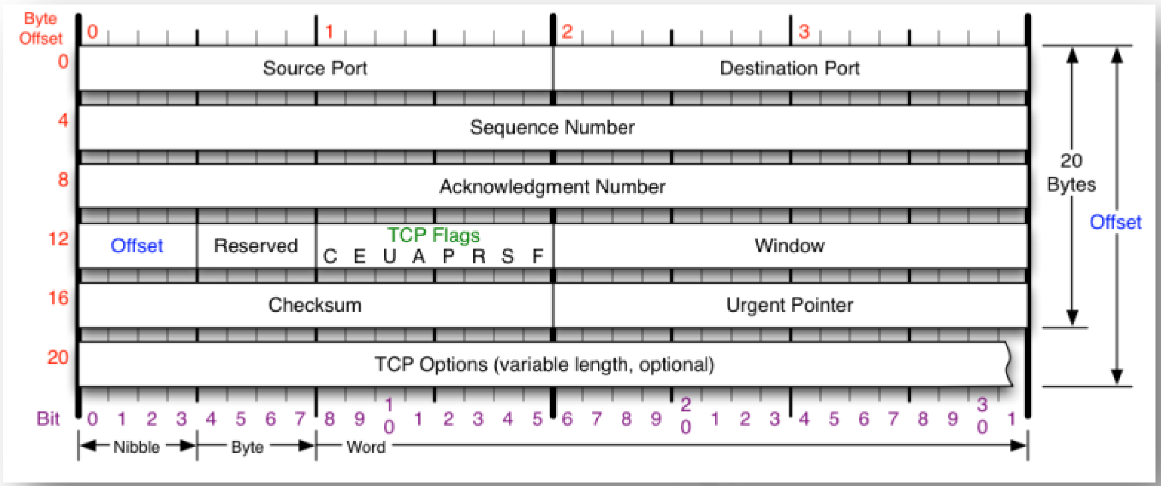


Figure 42 – TCP Header (NMAP, n.d.)

TCP Header Attribute	Byte Offset	Size	Primitive
Source Port	0	2 bytes	src port
Destination Port	2	2 bytes	dst port
Sequence Number	4	4 bytes	n/a
Acknowledgment Number	8	4 bytes	n/a
Data Offset	12	4 bits	n/a
Reserved	12	4 bits	n/a
Flags	13	1 byte	n/a
Window Size	14	2 bytes	n/a
Checksum	16	2 bytes	n/a
Urgent Pointer	18	2 bytes	n/a

Table 2 - TCP Header – Mapping NMAP Primitives

Robert L. Adams, robert.louis.adams@gmail.com

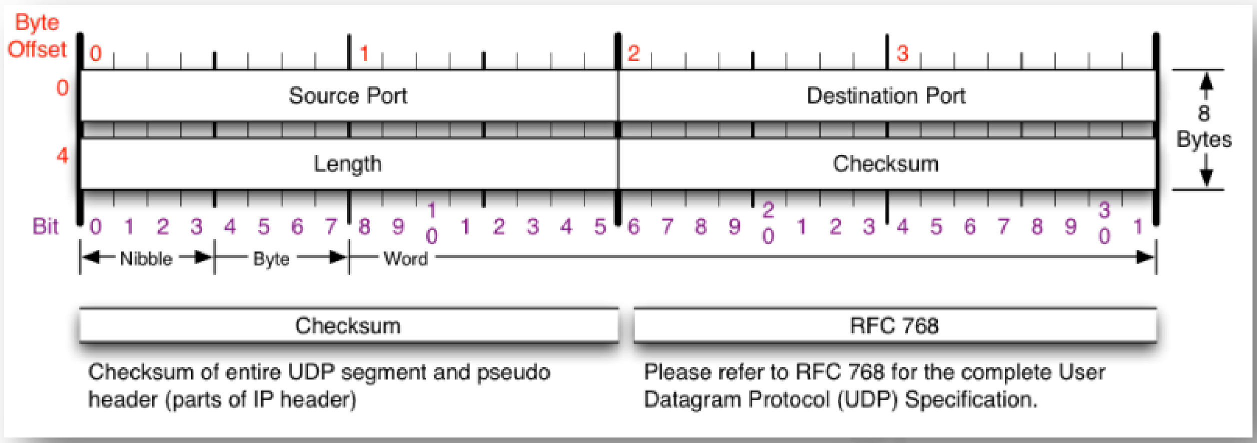

Byte Offset
0
4
Bit
Source Port
Destination Port
Length
Checksum
8 Bytes
Nibble
Byte
Word
Checksum
Checksum of entire UDP segment and pseudo header (parts of IP header)
RFC 768
Please refer to RFC 768 for the complete User Datagram Protocol (UDP) Specification.

Figure 42 - UDP Header (NMAP, n.d.)

UDP Header Attribute	Byte Offset	Size	Primitive
Source Port	0	2 bytes	src port
Destination Port	2	2 bytes	dst port
Length	4	2 bytes	n/a
Cheksum	6	2 bytes	n/a

Table 2 - UDP Header – Mapping NMAP Primitives

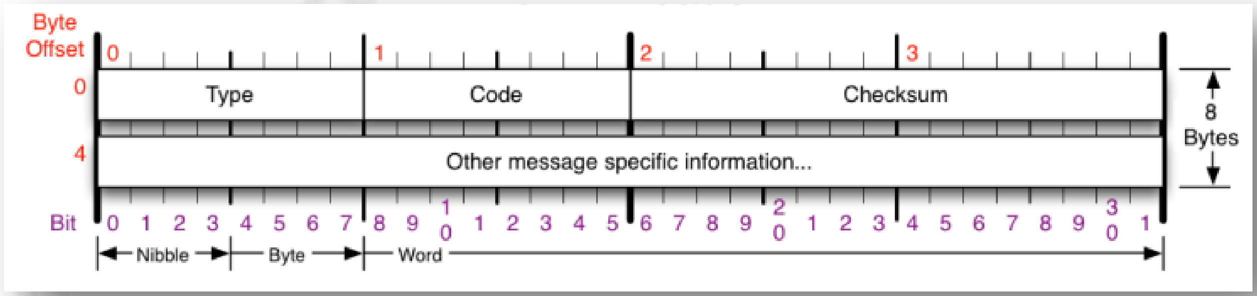


Figure 43 - ICMP Header (NMAP, n.d.)

ICMP Header Attribute	Byte Offset	Size	Primitive
Type	0	1 byte	n/a
Code	1	1 byte	n/a
Checksum	2	2 bytes	n/a

Table 3 - ICMP Header – Mapping NMAP Primitives