Interested in learning
more about security?

SANS Institute
InfoSec Reading Room

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

Capturing and Analyzing Packets with Perl

While many useful tools exist to aid the intrusion analyst in reviewing packet information such as Wireshark and Snort, sometimes they don't do exactly what you need them to do. What if you need to manipulate the data in a field before logging it, need to use your own timestamp format, or need to get more granular in the capture logic than the capturing or logging than the tool allowed? Getting full control over the logic and output usually means creating your own program, which many analysts will find too hard ...

Copyright SANS Institute
Author Retains Full Rights

AD

Streamline IT security environments
and compliance processes.

Transforming
IT Management.

Capturing and Analyzing Packets with Perl

GIAC (GCIA) Gold Certification

Author: John Brozycki, john@trueinsecurity.com
Advisor: Chet Langin

Accepted: January 2010

Abstract

While many useful tools exist to aid the intrusion analyst in reviewing packet information such as Wireshark and Snort, sometimes they don't do exactly what you need them to do. What if you need to manipulate the data in a field before logging it, need to use your own timestamp format, or need to get more granular in the capture logic than the capturing or logging than the tool allowed? Getting full control over the logic and output usually means creating your own program, which many analysts will find too hard to do. This paper will go through the steps in setting up a Windows system with Perl and the necessary add-ons to be able to run and create packet capturing Perl scripts. It will develop and demonstrate some sample Perl scripts that can be run or modified by the reader to accomplish a packet analysis task. In short, it will provide a foundation for how to capture and analyze packets with Perl.

© 2010 SANS Institute, Author retains full rights.

Introduction

Have you ever needed to do something during a packet capture that the tool you were using didn't do? Perhaps you looked for another tool only to find that none did exactly what you needed. You may have thought about writing something on your own, then realized it would be a lot more complicated than you were willing to take on. Packet capturing libraries are freely available such as WinPcap (Winpcap, 2009) for Windows operating systems and libpcap for Unix-like operating systems (Tcpdump, 2009) but how do you interface with them? Most system tools and libraries are written in C or a C variant (such as C++) and it may be necessary to code in C to make use of the low level access. Programming in C can be intimidating if you're not already a C programmer and can require a significant investment of time to develop and debug your project. This has undoubtedly been enough to stop many from continuing any further.

There are some commercial solutions to work with higher-level languages. For example, if you have programming experience in a language supporting ActiveX, beeSync makes PacketX (available at http://www.beesync.com/packetx/index.html) that allows you to interface a language like Visual Basic with WinPcap (the Windows version of libpcap, the packet capturing library.) This costs money and is tied to a specific platform, as ActiveX is Microsoft's technology for Windows platforms. Although this may work in some situations, the ideal would be a solution that is platform independent, relatively easy to be productive with, and freely available or at least inexpensive.

Analysts with system administration experience may have some experience with Perl. Perl is a high-level language, originally developed to assist system administrators automate tasks and reports. Perl runs on all major platforms and is freely available with optional commercial solutions. Programming in Perl involves writing a script (basically a text file) and then invoking it with a Perl interpreter. Perl evaluates the syntax of the script right before running it so there is no full compile process as there is with languages like C. Generally, it is easier and less time consuming to script something in Perl than to write it in C. And, if you've experience in neither Perl nor C, it'll probably be easier to start creating something in Perl.

John Brozycki, john@trueinsecurity.com

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Perl is widely supported and extended by way of modules. These modules allow Perl to interface with a plethora of systems and standards. One key module for capturing packets is NET::PCAP (Aperghis-Tramoni, 2008), which allows a Perl script to interface with libpcap or WinPcap. Additionally, NET::PCAPUTILS provides a simplified interface into NET::PCAP, making packet capturing even easier. Perl modules, ports of the Perl interpreter, and sample Perl code are available on the Comprehensive Perl Archive Network (CPAN) at www.cpan.org. These factors make Perl a good solution for creating your own packet-capturing tool. With an introduction, some examples, and a little time gaining experience with Perl, you could be creating or customizing your own Perl-based utilities to capture or display packet information for whatever need might arise. In this paper, I'll first establish the environment and tools that will be required to get started with Perl. Next, I'll introduce some Perl scripts to accomplish a specific task and also serve as a template for customizing to other uses. Finally, I'll look at some more advanced Perl topics that may help you in working with network packets.

In the Perl language, there are often many ways to accomplish the same task. Advanced Perl programmers often write small, eloquent code that is difficult for a new user to understand. The scripts here will not be eloquent but will focus on being direct and easy to understand.

1. Configuring the Environment

1.1. Getting started

Perl is generally included in Unix-like environments, including most Linux distributions and Mac OS X. A review of the CPAN site (www.cpan.org) reveals that a great number of operating systems, including many that are not actively supported, have Perl ports available. Windows operating systems do not come with native Perl support. This paper will focus on setting up Perl under Windows, but most of this is applicable to the Unix-like environments as well. Getting packet capturing to work on alternate platforms will come down to the ability to obtain functional ports of the libpcap packet capturing library and Net::Pcap and Net::PcapUtils, the Perl modules for interfacing with libpcap.

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

1.2. Installing a Perl environment on Windows systems

Although there are several versions of the Perl interpreter available for Windows, the two most popular are probably ActiveState's ActivePerl (www.activestate.com/activeperl) and Strawberry Perl (www.strawberryperl.com.) There are many positive attributes to both. Both offer free Perl interpreters. However, Strawberry Perl is offered as a Windows only solution. ActiveState is available for the major platforms (Unix/Linux, Windows, Mac OS X) and offers additional commercial offerings such as a programming development kit and integrated development environment. ActiveState's commercial offerings allow you to create stand-alone executables (really scripts wrapped with a Perl runtime) for supported platforms and the ability to create a Windows service executable. For these reasons, ActivePerl will be used as the Perl environment here.

To download ActivePerl, go to www.activestate.com/activeperl and click on the link for ActivePerl. Download and run the installer, following the instructions. Installation of ActivePerl is quite easy, but the capturing environment is not ready yet as there are a few more components that are needed to capture packets. If it's not already installed, you will need WinPcap, which is available at http://www.winpcap.org/install/default.htm. According to the WinPcap site, the WinPcap package consists of the driver and library to capture packets at the link layer and a Windows version of the libpcap api (Winpcap, 2009). After installing the WinPcap package, a Perl module is needed to allow Perl to access it. This can be done through an included utility.

When ActivePerl is installed, it also installs ppm, the Perl Package Manager. This utility allows you to download Perl modules for use on your system. Where CPAN provides module source that you must compile before using, ppm provides a binary that is ready to run in your environment. This is

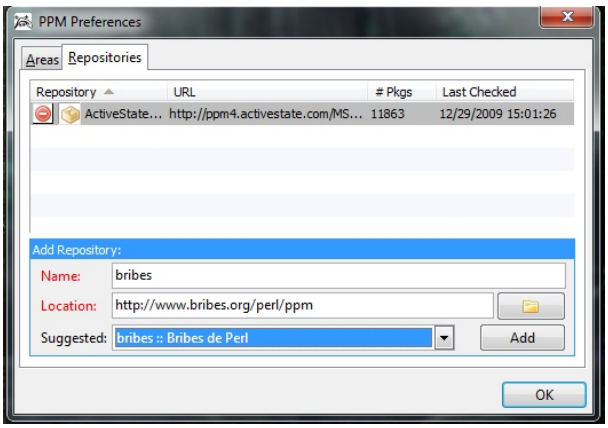

Figure 1: Adding a repository in PPM

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

convenient and a real time saver! The ActivePerl program group contains a shortcut to the Perl Package Manager from the Program menu under the Start menu. ActiveState maintains its own package repository, but you can add additional repositories within ppm. In fact, at least as of the time of writing, it is necessary to add a repository to ppm in order to get the Windows ports of NET::PCAP and NET::PCAPUTILS. To accomplish this, after launching ppm select "preferences" from the Edit menu. From the PPM Preferences window click on the Repositories tab. You should see "Add Repository" near the lower half of the window, with entry fields directly below. Click on the drop down arrow for the "Suggested field" and look for "bribes :: Bribes de Perl" and select it, then click the Add button then click OK. If you don't see it, you can enter it manually. Enter "bribes" in the name field and enter http://www.bribes.org/perl/ppm/package.xml" in the location field. Click the Add button then click OK. Once added, you should see the bribes repository added to the ActiveState respository in the list at the top of the window under the Repositories tab. In the upper left of the Perl Package Manager, click on the View All Packages button. Search for Net-Pcap and Net-PcapUtils and install them by clicking each package and then selecting Install from the Action menu.

Installed packages show up with an orange colored icon before their names. Uninstalled packages will have a greyed-out icon. Once you've downloaded a module it is ready to use. Near the bottom of the Perl Package Manager window, underneath the list of packages, there is a details tab. Contained within this tab is a link to the CPAN page about the selected package. You can use this link to get more information about the package, not the least of which is the instructions on how to invoke the module from within Perl. At this point, Perl is installed, WinPcap is installed, and the NET::PCAP package is installed, making the Windows system ready to capture packets. Before exiting the Perl Package Manager add the TIME::HIRES module. It will be discussed and used later.

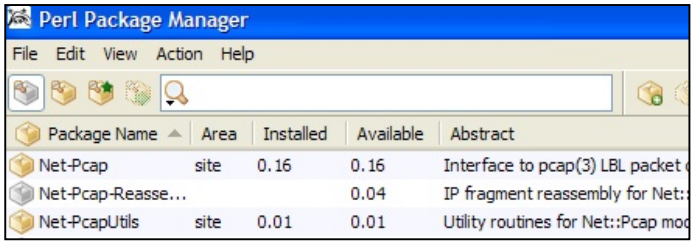

Figure 2: Adding modules with PPM

John Brozycki, john@trueinsecurity.com

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

2. Performing a packet capture

2.1. Using NET::PCAPUTILS

To start off and verify the setup, let's write a very simple Perl script that will capture packets and print a message to the console each time it captures a packet. Open any text editor, such as Notepad, and type in the commands given in this section (or copy and paste them from the appendix.) Save the script as "tinycap.pl."

Sometimes copying text from a paper and pasting it into an editor results in character conversion problems or missed characters at the beginning or end of the section being copied. All of the Perl source scripts from this paper are also posted online at http://www.trueinsecurity.com/perlcap/perlcap.html. If you experience syntax errors with any of examples in this paper, please try getting them from this web address.

To begin, the Net:PcapUtils module needs to be specified. (Net::PcapUtils is used because it simplifies capturing over using Net::Pcap.) The "use" command tells Perl to use the module name specified.

```
use Net::PcapUtils; #packet cap module for WinPcap
```

Next, the Net::PcapUtils::loop loop call function is invoked. The first parameter it takes is the name of the subroutine that will process each packet retrieved, which is named "process_pkt" here. The remaining parameters are parameters that are sent to WinPcap to control the capture and correspond to the commands you would use with tcpdump or windump. For now, a SNAPLEN will be set to 65536 (to capture the entire packet, even if it's big) and the network adapter will be put into promiscuous mode by setting it to a 1. (You can capture without putting the adapter into promiscuous mode, but you will only see broadcast traffic or traffic specifically destined for your network address. Promiscuous mode instructs the adapter to pass on all packets it sees, not just those specifically addressed to it.) You can specify additional parameters here

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

controlling the capture, just as you would if using tcpdump or windump. They will be discussed in more detail later.

```
Net::PcapUtils::loop(\&process_pkt,
       SNAPLEN => 65536, #Size of packet.  Can also use 0
       PROMISC => 1, ); #Look at ALL packets
```

Finally, we need the subroutine that is going to process the data from each packet.

```
sub process_pkt       #Packet processing routine.
      {
         print("Got a packet!\n");
      }
```

To run the script, first open a command prompt. Then, from the directory where you saved the script, type: *perl tinycap.pl*. Each time a packet is captured by WinPcap and passed up to the Perl script, the script outputs the line "Got a packet!" to the console. Open a browser or perform a similar activity to generate some traffic if you don't see any results. If you still don't see anything, you'll learn the most likely reason why shortly. This is not a particularly useful script and a similar one is demonstrated in the documentation for NET::PCAPUTILS on the CPAN site. However, it demonstrates how easy it is to get link layer packets into Perl, demonstrates the basic mechanisms, and verifies that everything is set up correctly. A copy of the Perl script is in appendix A. If you didn't see any output we'll resolve the most likely issue in the next section.

If your Windows system has more than one network adapter (including virtual or VPN adapters) and the network interface you wanted to capture from isn't the first adapter, you probably didn't see any "Got a packet!" messages. By default, the first adapter is used if one isn't specified. To specify the adapter, you need to enter the adapter name as a parameter supplied to Net::PcapUtils::loop. Windows uses an "unfriendly" name for network adapters, and you have to specify that name exactly. This isn't particularly well explained in any documentation I've come across on CPAN or elsewhere. Even the Bribes website, where the Windows port of Net::Pcap is maintained at http://www.bribes.org/perl/wnetpcap.html doesn't clearly explain this. I initially tried

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

specifying numeric values, such as 2 for the second adapter, but I couldn't capture anything. I ended up contacting Jean-Louis Morel, the maintainer of Net::Pcap at Bribes, and he was kind enough to explain the need to use the Windows adapter name (which is something in the format of "\Device\NPF_{11598F65-F342-4D6D-88B8-E04700BAEFF8}"). Jean-Louis even sent me sample code to interrogate the device names. To specify an interface other than the first, you would add the "DEV" parameter as follows:

```
DEV => "\Device\NPF_{11598F65-F342-4D6D-88B8-E04700BAEFF8}",);
```

To make this more flexible, I expanded upon what Jean-Louis sent me into a script called *pickinterface.pl* that will enumerate the network adapters and let you select one of the adapters. The selection the user makes is saved to a text file. When writing a capture script, I read this file's single entry into a variable and use that variable in the DEV parameter. This allows a capture script to easily be moved to other systems as well as to accommodate network adapter changes in a system. This also works out great if you are running a virtualized environment that changes or an environment that can run either physically or virtually. An example is Windows running on a Macintosh computer via Bootcamp, Apple's environment for easily allowing for dual booting Mac computers, where the hardware physically boots up a Windows Operating System. It is also possible to run the Bootcamp partition as a virtual machine under VMware Fusion or Parallels. Depending upon which environment Windows was launched from, the adapter will have a different name and this makes it easy to use scripts without having to modify them. A copy of this script is in Appendix B. A revised version of the first script, *tinycap.pl*, which uses the interface information from *pickinterface.pl,* is in Appendix C.

One additional troubleshooting note: if you should get an error message when running *pickinterface.pl* (or subsequent script) that is similar to the following:
print() on closed filehandle SETTINGSFILE at pickinterface.pl line 39
it most likely means that the file is write protected or you don't have sufficient permissions to create files in the destination directory. Ensure you can create a file manually and adjust your permissions accordingly if you can't.

John Brozycki, john@trueinsecurity.com

© 2010 SANS Institute, Author retains full rights.

2.2. Examining what is being captured in $packet

In the previous Perl example, the packet contents were passed into a variable called $packet, but no processing of it occurred. We first need to understand what the data looks like in $packet. By adding the line:

```
print ("A packet:\n#####\n".$packet."#####\n");
```

to the previous capture script, we can print to the screen the contents of a packet as it's captured. Below is a displayed packet from a browser accessing Google:

```
A packet:
#####
 →pDä0 ♀)çº E ☻S↓D@ Ç♠3Y└¿☺♫@Θ─h♣☼ P%♠{[?ûP↑⌂ì}4  GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0;
.NET
 CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022; .NET CLR
3.0.4506.21
52; .NET CLR 3.5.30729)
Accept-Encoding: gzip, deflate
Host: www.google.com
Connection: Keep-Alive
Cookie:
PREF=ID=8fa9d7a955181748:U=1c17f55495302dae:TM=1202964795:LM=1241269051:
S=9Sdb5Qz8sldmT6BX; NID=25=JgNv7Go87lUG0L8ErJjR3aaJlSAmYaHWCjSSZXQjcnkwQjkXVb-
a3
6ffZ2bMAaoX8lVv0282H-dk8itFsF78B80PXqonLoJxXASaBD0PoVpjcgQ4C6-qP5nBbLPl5d_b


#####
```

Figure 3: Example packet capture

The 5 "#" characters serve as delineators so we can see the beginning and end of the packet. Note that the payload data, which in this case is an HTTP GET request, is passed to $packet as character data that's easy to read and work with. Also note that the header information shows up as strange characters. Unlike the output from tcpdump or windump, the header information is in a raw format and will need to be converted to work with it.

To get a better idea of what is in the header and how to convert it, we can use a small Perl script named *showheader.pl*, which is included in appendix D. This utility

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

prints out the raw header, then goes through each byte, printing the byte number followed by the hex, decimal, and binary values of each byte. Worth noting here is the Perl unpack operator. Pack and unpack are two Perl functions that can be used to store and retrieve binary data to a string in Perl format (Quigley, 2002, p. 78). A template is selected as well as how many characters to be formatted with the template. An explanation of its use follows the line from the script.

```
$hexval =
unpack('H2',substr($packet,$i,1));
```

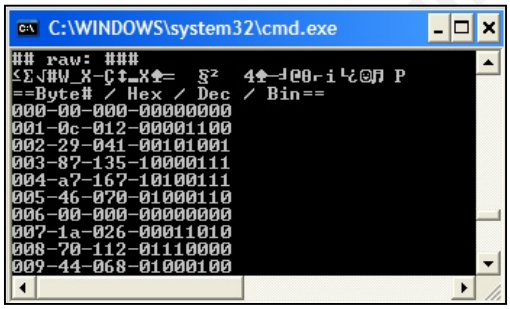

Figure 4: Displaying packet values in hex/dec/binary

'H2' - unpack a hex string, H is the template for high nybble first, 2 for the number of positions (Hall & Schwartz, 1998, p. 220).

The value to be unpacked is the substring containing the starting position specified by the variable $i, the loop value, for a length of 1, from the variable $packet, which contains the packet data passed from NET::PCAP. The result, $hexval, is the hexadecimal value of the byte at position $i. To convert this into a decimal representation the hex operator is added, and the line looks like this:

```
$decval = hex(unpack('H2',substr($packet,$i,1)));
```

The hexadecimal value is retrieved using the unpack operator as done previously, only the hex function is used to convert it to a decimal value[http://www.misc-perl-info.com/perl-hex.html]. Another way to convert is to use the printf function, which provides printed output and the ability to display a value in a wide variety of formats. To display the variable $decval as a binary value, simply output it with printf as follows:

```
printf "%08b\n", $decval;
```

% - indicates the start of the conversion string.

08 - the 8 means format the output to 8 positions, the 0 means pad with leading 0s if less

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

than 8 positions long.

b - indicates the value should be converted to binary.

\n - indicates that a new line should be printed so future output doesn't print immediately after it.

$decval - represents the variable or value to be converted.

To use printf to convert a value to be stored in a variable for later manipulation:

$binaryval = printf "%08b\n", $decval;

However, printf will give you an extra "1" at the end. For example, $binaryval might look like this:

01101001

1

What is happening here is that the printf function is returning a "0" for failure and a "1" for success, and $binaryval is getting that return code (TutorialsPoint.com, 2009). To strip out the return code, the chop function, which removes the last character from a value, is used:

chop($binaryval);

However, there is a *much* better way. The sprintf function was specifically designed to return a string value that is formatted and ready for variable assignment. In general, you don't want to make assignments with printf and even chopping off the last character can still leave unwanted newline or carriage return characters. So, when you want to format data that is going to be assigned to a variable remember to add the "s" to the beginning of printf. If you forget or miss the "s" because of a typo, and get extra data, I hope you'll remember this section.

Showheader.pl enumerates byte 0 through byte 54, which will show the complete header for most standard packets. If the header length is larger, you will need to increase this value. Referring to an IP reference chart, such as the *SANS TCP/IP and TCPDUMP*

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Pocket Reference Guide (available at http://www.sans.org/resources/tcpip.pdf) will assist in determining the relative positioning of fields in the header.

To confirm what you're seeing, you may run windump or Wireshark to capture the same packets simultaneously to verify where and what a field is. Notice that there are two fields that are part of an Ethernet frame, but not part of an IP header. These are the source and destination MAC addresses, which are the low-level addresses of the network adapter that devices on a LAN use to communicate with each other. As each MAC address is 6 bytes, combined they represent the first 12 bytes (bytes 0 through 11.) The 12th byte starts the IP header (that is, if the packet is IP) and follows the expected sequence.

While fields like the MAC addresses are composed of multiple bytes, the bytes are distinct values. Fields such as IP header length are multiple bytes and the bytes must be combined to arrive at the field value. For the IP header length field, the second byte represents a value that is 2 raised to the 0, which is 1. The first byte represents a value that is 2 raised to the 8, or 256, so this value is the number of 256s are part of the field value. To make this a little clearer, the formula for the length field would be ((LengthByte1 x 256) + LengthByte2)). If the field is composed of a three-byte value, the first byte would be multiplied by 2 raised to the 16, or 65,636. Following this formula you can reconstitute values that are stored across multiple bytes into their decimal equivalents.

John Brozycki, john@trueinsecurity.com

© 2010 SANS Institute, Author retains full rights.

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Byte #	Description	Comment
0	DstMAC1	
1	DstMAC2	
2	DstMAC3	
3	DstMAC4	
4	DstMAC5	
5	DstMAC6	
6	SrcMAC1	
7	SrcMAC2	
8	SrcMAC3	
9	SrcMAC4	
10	SrcMAC5	
11	SrcMAC6	
12	Type1	(x0800=IP)
13	TYpe2	
14	Version & Hdr Len	4=IP v4, 5(*4) = 20 hdr len
15	Differentiated Service Field	
16	Length1	(L1x256 + L2)
17	Length2	
18	ID1	(ID1x256 + ID2)
19	ID2	
20	(0x040=Don't fragment)	
21	Fragment Offset	
22	TTL	
23	Protocol	(6=TCP)
24	HdrChksm1	(H1x256 + H2)
25	HdrChksm2	
26	SrcIP1	
27	SrcIP2	
28	SrcIP3	
29	SrcIP4	
30	DstIP1	
31	DstIP2	
32	DstIP3	
33	DstIP4	
34	SrcPrt1	(S1x256 + S2)
35	SrcPrt2	
36	DstPrt1	(D1x256 + D2)
37	DstPrt2	
38	SeqNo1	(SN1 x 16777216 +
39	SeqNo2	SN2 x 65536 +
40	SeqNo3	SN3 x 256 +
41	SeqNo4	SN4
42	AckNo1	AN1 x 16777216 +
43	AckNo2	AN2 x 65536 +
44	AckNo3	AN3 x 256 +
45	AckNo4	AN4
46	HeaderLength	(1st 4 bits, so divide by 16)
47	Flags	
48	WinSize1	(WinSize1 x 256 + WinSize2)
49	WinSize2	
50	ChkSm1	(Chksm1 x 256 + ChkSm2)
51	ChkSm2	
52	UrgPtr1	(UrgPty1 x 256 + UrgPtr2)
53	UrgPtr2	

Figure 5: Layout of a standard TCP header in $packet

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

2.3. Recording packet time

When logging or processing packet data, you generally want to record the precise time each packet arrived. However, NET::PCAP doesn't pass any time information back to the Perl script. One option is to use Perl's built-in time function, *localtime*, but it only provides time resolution down to the second. When capturing network traffic, there will likely be a large number of packets captured in any given second. To gain more granularity in the reporting of time, another Perl module can be used. While there are several options available, TIME::HIRES works well. If you didn't install it when you installed NET::PCAP and NET::PCAPUTILS, open the Perl Package Manager and install it now.

The syntax to add TIME::HIRES to the Perl script, taken from the CPAN page for the module (Hietaniemi, J., Wegscheid, D., Schertler, R., & Aas, G., 2008) is as follows:

```
use Time::HiRes qw(gettimeofday);
```

To query the exact time the following assignment is made:

```
@HiResTime = gettimeofday();
```

Notice the "@" in front of the HiResTime variable name instead of "$"? This indicates that the variable is an array and is one of the many handy features in Perl. When *gettimeofday* is called it returns two values. The first is the number of seconds since epoch and the second is the number of microseconds. To convert the seconds into values that people can understand, the *localtime* function can be used to convert and separate the time values, like an array from *gettimeofday*. Another handy feature of Perl is the ability to assign multiple variables in a single assignment when the source has multiple values. The assignment looks like this:

```
($sec,$min,$hour,$day,$month,$year) = localtime($HiResTime[0]);
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

In the proceeding line, six time variables (second, minute, hour, day, month, and year) are assigned values from the *localtime* function, as the *localtime* function converts the first element of $HiResTime (which is seconds since epoch) that was just received. There are two caveats that must be addressed before the retrieved time is correct. First, the month variable retrieved starts with 0 for January and goes to 11 for December. Therefore, you need to add 1 to the month value to adjust it to the standard 1-12 values people typically use. Second, the year is returned as years since 1900. For example, the year 2009 is returned as 109. To address this just add 1900 to the value in year. Finally, add on the microseconds field, which is stored in $HiResTime[1], and format the entire value. To format, use sprintf (remember, not printf!) to control the spacing and formatting. The final code to assess the time is as follows:

```
$month = $month + 1;
$year = $year + 1900;
my $time = sprintf "%02d/%02d/%4d %02d:%02d:%02d:%06d",
           $month,$day,$year,$hour,$min,$sec,$HiResTime[1];
```

This code can be placed in a function. The function can be called each time a packet is retrieved to get a more accurate representation of time. Full code for this time function will appear in the URL capturing example in the next section. An example of the output from the sprintf formatting is as follows:

08/16/2009 15:54:36:512250

2.4. A Project to Log Browsed URLs

So far we have covered setting up the Perl environment and modules, installing the packet capturing library, specifying the interface to capture packets from, determining and retrieving values from the packet, and getting time granularity in Perl below a second. In this section a real world example and a solution are presented.

Imagine you have been tasked with capturing all the URLs that users are accessing through an Internet link. Why might you need to do this? Perhaps your firewall or similar device doesn't provide sufficient logging or you don't have administrative access to the device or its log files. You could try tshark, the command line version of Wireshark, which can be made to capture and display this information.

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

However, in my experience the overhead of tshark with its complement of dissectors and the data it wants to cache lead to memory exhaustion and program crashing. This is documented on the Wireshark wiki (Meier, B., 2009). Wireshark and tshark are very useful tools, but are not only overkill for this task, they are unlikely to be able to do it reliably. One answer is to write your own Perl script that captures and logs time, source ip and port, destination ip and port, and full URL. Further, this script should be able to run for long periods of time (probably non-stop,) output to a text file, and automatically output a new file name each day to make reviewing the output easier. This project will allow the introduction of some key tasks including outputting to text files and dynamically changing the file handle, and scheduling a Perl script so it can run without requiring a logged on user and can survive reboots.

The first script output a line that a packet was captured, but it didn't display any of the data from the packet. To do anything meaningful, a script will need to be able to parse and convert data in the packet. The first thing we need to do is create a variable to hold our packet data as it is passed to the script. In previous scripts "@_" was used but not explained. In Perl, @_ is a special variable (Schwartz, R., & Phoenix, T., 2001, p60). It is an array that holds the parameter list from the NET::PCAPUTILS loop that invokes the subroutine, and it only applies within the subroutine. Names are given to the parameters. (If you're wondering how to know how many parameters there are or where these particular names come from, they are from the NET::PCAPUTILS documentation.) These variables are preceded by the word "my." In Perl, this is done to create private variables that only apply to the current block of code (Schwartz, R., & Phoenix, T., 2001, p. 62). If you want them to be globally available throughout the script then don't use "my" in the variable assignment. Using @_ may seem a little odd, but it makes for less coding. With just a few lines, raw packet data from the link layer has been placed into a variable. The next steps are to parse out individual fields and convert to decimal (or hex or binary) where needed.

Since the goal is to capture all of the URLs accessed, the following fields will be needed: Destination IP, Destination port, Source IP, source port, host (the name of the web site being requested, such as "www.google.com",) and URI (Uniform Resource Indicator, such as "/about.html".) Host and URI will be contained in the payload. The

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

other fields will be in header. The time the packet was received will also be needed. As the only traffic desired is HTTP traffic, we can set a capture filter to only pass HTTP GET requests to the script. Windump or Wireshark can again be used to capture some sample traffic and make note of the characteristics of the traffic to capture. In the NET::PCAPUTILS::LOOP, where capture parameters are specified, the following BPF (Berkley Packet Filter) format filter will cause only HTTP GET packets to be processed by the script:

```
FILTER => 'tcp[20:4] = 0x47455420'
```

This filter specifies that in TCP packets starting at position 20 look for a match of four positions to the hex value 0x47455420. In hexadecimal 47 represents "G", 45 represents "E", 54 represents "T", and 20 represents the space character. If hexadecimal conversion isn't your strong point, you can examine a sample Wireshark capture and expand out the packet. Find the character representation of "GET" and click on it. It comes immediately after the TCP header. A standard TCP header is 20 bytes and starts at byte 0, so it ends at byte 19. The "G" starts at position 20. The image below shows the TCP header highlighted in blue. Immediately afterward are the characters representing "GET", starting at position 20. Remember that this filter is being passed down to WinPcap to filter the data that it will sent to the script and bares no relationship to the position of the data returned in $packet.

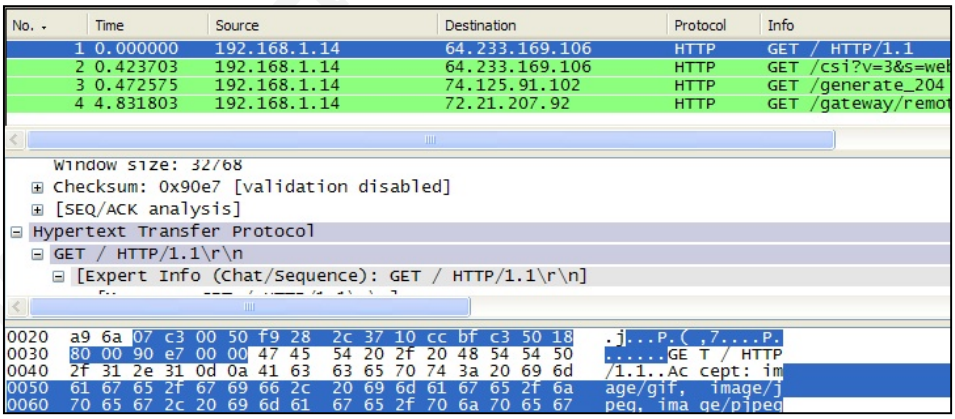

Figure 6: Using Wireshark to help write filters

To give an example of the use of this filter, the parameters sent for the capture would look like this:

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
Net::PcapUtils::loop(\&process_pkt,
                     SNAPLEN => 65536,
                     PROMISC => 1,
                     FILTER => 'tcp[20:4] = 0x47455420',
                     DEV => $interface, );
```

The fields being retrieved from the header are all straight forward. For example, to pull the destination IP address the following code snippet can be used:

```
my $src_1st_octet = hex(unpack('H2',substr($packet,26,1)));
my $src_2nd_octet = hex(unpack('H2',substr($packet,27,1)));
my $src_3rd_octet = hex(unpack('H2',substr($packet,28,1)));
my $src_4th_octet = hex(unpack('H2',substr($packet,29,1)));
my $dst_1st_octet = hex(unpack('H2',substr($packet,30,1)));
```

To pull the payload data, get the length of the packet and then go from the first byte and then extract for a length of PacketLength - StartOfPayload. For simplicity, the assumption will be made that the header has a standard length of 20 and that the HTTP data we're extracting starts at the 54th byte. (Handling larger header lengths will be discussed later.) The code for this could look as follows:

```
my $len = length $packet;

my $httpdata = substr($packet,54,$len-54);
```

Now that the payload has been pulled out, the host and URI need to be extracted from it. For the host, the word "host:" appears in the GET request. The URI appears a few spaces directly after the GET command. While there are multiple ways of extracting these values, one straightforward way is to use the SUBSTR operator to pull out substrings and the INDEX operator to locate values in the data. Code to do this could look as follows:

```
#Extract the "host", i.e.: www.google.com
my $startofhost = index($httpdata,"Host:")+6;
my $endofhostlength = (index($httpdata,"\r\n",$startofhost) -
$startofhost);
my $host = substr($httpdata,$startofhost,$endofhostlength);
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
#Extract the URI, which is the resource being requested from the host,
example "/"
my $startofuri = index($httpdata,"GET ")+4;
my $endofurilength = (index($httpdata,"HTTP/",$startofuri)-4);
#skip back past the newline and return characters
my $uri = substr($httpdata,$startofuri,$endofurilength);
```

The script will need to output to a text file for easy review. In this project, we want the script to run continuously. To keep the file to a more manageable size, it should start outputting to a new file when the date changes. Another feature of Perl is that it is easy to redirect the output of a file handle, which makes this easy. First, here is how a dated output file would be set up.

```
$formatdate = sprintf "%04d%02d%02d",$year,$month,$day;
$logdate = $formatdate."_webcap\.txt";
open (LOGFILE,">>$logdate");
```

In the first line, the sprintf function is used to format the year, month, and day into a uniform length variable. In the second line, a suffix of "_webcap.txt" is appended and the results assigned to a new variable. (The "\" is used to indicate that the "." is just a character as it has a special meaning otherwise.) In the third line, the OPEN command is used to open the file. LOGFILE is the handle name it is being given. A single ">" would indicate that the file is to be written. A ">>" indicates that the file is to be appended. This way, should the process crash and restart, the script will add new data to the log file instead of overwriting it.

To make the script log to a new output file each day, a comparison can be done on the $day variable. When it has changed, the $formatdate and $logdate variables should be updated. To change the file handle to make it start logging to the new file, simply run the OPEN assignment again and the LOGFILE handle is now pointed to the new file. The day can be checked by storing the previous day into a variable and then comparing it to the current day when the time function updates it. Because it's necessary to pull the date first before it can be compared, logic is added to handle the first time the script is run. The code looks like this:

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
if ($firstrun == 1)  # If 1st run, OPEN logfile & set orig date var
      {
        $formatdate = sprintf "%04d%02d%02d",
                                $year,$month,$day;
        $logdate = $formatdate."_webcap\.txt";
        open (LOGFILE,">>$logdate");
        $origdate = $formatdate;
        $firstrun = 0;
      }
    if ($firstrun == 0)      # If not 1st run, get curr date &
    #compare to orig date.  If date changed output to new file.
      {
        my $newdate = sprintf "%04d%02d%02d",
                                $year,$month,$day;
        if ($newdate ne $origdate)
          {
            $logdate = $newdate."_webcap\.txt";
            open (LOGFILE,">>$logdate");
            $origdate = $newdate;
          }
      }
```

At this point, the script is nearly done. If the purpose is to log external website access and there are internal web resources that should not be captured, they can be filtered out by comparing the destination IP addresses. Many companies utilize private IP address ranges internally using a common set of ranges that are not routable on the public Internet (Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., & Lear, E., 1996). RFC 1918 covers three ranges; 10.0.0.0/8, 192.168.0.0/16, and 172.16.0.0/12 (172.16.0.0-172.31.255.255.) By adding in code to check if the destination is within these three IP address ranges, the script can prevent any logging when a match is made. The code to do this could look as follows:

```
$nooutput = 0;
  $intranet = $dst_1st_octet.$dst_2nd_octet;
  if (($intranet eq "192168") || ($dst_1st_octet eq "10"))
     {
        $nooutput = 1;  # Set to 1 to prevent output
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
     }
  if (($dst_1st_octet eq "172") && (($dst_2nd_octet > 15) &&
($dst_2nd_octet < 32)))
     {
        $nooutput = 1;
     }
```

The complete script for the web access script, webcap.pl, is in appendix E.

2.5. Handling non-standard headers and flags

On most Ethernet networks, webcap.pl should work just fine as it is. But what if header options are used and the header is not a standard length? How could this be handled? Recall back to figure 5 that the IP header length is in byte 14. To retrieve it, simply assign the 14th byte to a variable. The first four bits contain the IP version, while the last four bits contain the header length, which must be multiplied by 4 to give the full value (valid ranges are 5,6,7,…15 to yield header lengths of 20,24,28,…60.) Since the first four bits shouldn't be counted toward the length, one option would be to mask them out. To do a bitwise comparison, a binary value of "00001111" could be logically "ANDed" to the entire byte to effectively zero of the first four bits (0 AND 1 yields 0, 1 AND 1 yields 1.) This can be done easily in Perl with the "&" operator, which works at the binary level even if the variable is a decimal value. In decimal, "00001111" is 15. After the binary "ANDing", multiply by 4. To retrieve the IP header length, the following code may be used:

```
my $iphdrlen = hex(unpack('H2',substr($packet,14,1)));
$iphdrlen = $iphdr & 15;
$iphdrlen = $iphdrlen * 4;
print $iphdrlen."\n";
```

In similar fashion, the TCP header is in the first four bits of byte 46. Although the last four bits are officially reserved, they should be "ANDed" out to ensure no stray or malicious value is present. Since the last four bits need to be "ANDed" out, the binary comparison would be to "11110000", which is 240 in decimal. Zeroing out the last four bits ensures that no extra bits are included in the TCP header length. Since the header

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

length is in the first four bits, it needs to be shifted down to get its real value. As it needs to be shifted down four binary positions, this can be accomplished by dividing it by 16. Sample code to retrieve the TCP header length (assuming a standard IP header length) is as follows:

```
my $tcphdrlen = hex(unpack('H2',substr($packet,46,1)));
$tcphdrlen = $tcphdrlen & 240;
$tcphdrlen = $tcphdrlen / 16;
print $tcphdrlen."\n";
```

Once the header lengths are known, these values can be used to adjust the retrieval of other fields. For example, if the IP header length has a value of 6, which makes the IP header 24 bytes, fields that follow the IP header will come four bytes later than if it had a standard IP header. To accommodate this, a variable named $offset could be created and set to be equal to the IP header length - 20. Then $offset would be added to the value of the fields that followed.

Performing binary "ANDs" is also useful for extracting flag information. An example is the flag field in the TCP header, which is found in a standard IP packet in byte 47. TCP header flags are only the 6 least significant bits of the 47th byte, and each position indicates a flag. The Reset flag is in the 3rd bit, so a packet with only the Reset flag on would have a flag field that looked like: "xx001000." To isolate the Reset flag the 47th byte would be "ANDed" with "00001000" or 8 in decimal. It is not necessary to then divide the field by 8 to get the "1" value of the flag. Since only the selected bit is being preserved (regardless of the bit being compared) having any value greater than 0 means the bit is set. Sample code to display the state of the Reset flag follows:

```
my $rstflag = hex(unpack('H2',substr($packet,47,1)));
$rstflag = $rstflag & 8;
if ($rstflag > 0)
  {
     print "Reset flag is on.\n";
  }
else
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
{
   print "Reset flag is off.\n";
}
```

Using these techniques, the correct byte locations of fields can be determined when header lengths aren't standard and flags or other bit-level values can be compared.

2.6. Making scripts persistent and executable/portable

If the purpose of your script is long-term monitoring, alerting, or processing of specific packets, the Perl script should be run as a service to allow it to survive user logouts, reboots, and crashes. There are various utilities, many commercial, that wrap an executable with the necessary components to properly respond to the operating system when it checks the service (as running a plain executable as a service will result in it being terminated by the OS.) A freely available utility that I use is NSSM, the "Non-Sucking Service Manager," written by Iain Patterson, and available for download at http://iain.cx/src/mssm/. Unlike some of the other utilities, NSSM monitors executables and restarts them if they crash, an important requirement when you need a script to keep running. The NSSM web page explains how to install a service with it, and this can be accomplished via a GUI or from the command line. Remember that a Perl script is merely a text file and can't run by itself. To run a Perl script as a service you need to invoke the Perl interpreter executable and pass the Perl script name as an argument/option.

NSSM documentation is available on Iain Patterson's website, but an example will be given here for clarity. This example assumes that the service will be called "webcapture", NSSM.EXE has been copied to the System32 directory (or added to the Window path), that the Perl executable is in c:\perl\bin\perl.exe (the default for ActiveState Perl), and that the script is named webcap.pl and is located in c:\webcap.

To install the service, type:

runas /user:Administrator "nssm.exe install webcapture"

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

The "runas" command allows you to install this as an administrator even if you're logged on as a non-administrative user (it can be omitted if you are already running as an administrator). Otherwise, the "runas" command will prompt you for the password of Administrator and then run NSSM under an administrator context. When the GUI pops up, enter "c:\perl\bin\perl.exe" in the Application field and then enter the full path of the Perl script, "c:\webcap\webcap.pl" in the Options field. The service name should have already pre-filled Service Name with "webcapture" if you included it on the command line, but you can edit it here if you want. Click on the button to "Install Service" to install the service, but it won't run until you go into the Windows services applet and start it. In the Services applet, set the type to "Automatic" if you want it to start each time the system starts.

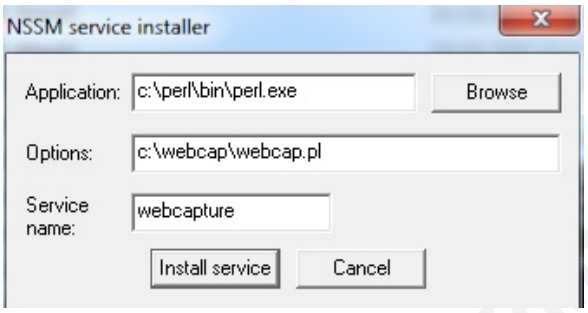

Figure 7: Installing a Perl script as a service with NSSM

What if portability is important? In order to run a packet capture script on another machine, WinPcap will, of course, need to be present to capture the packets. Additionally, to run a Perl script, a Perl interpreter must be installed as well as any non-standard Perl modules used in the script. It is possible to make a Perl executable but there are some limitations. Executable support for Perl on Windows has mostly been experimental, has not been well supported, and has been dropped from newer Perl releases (Perlmonks.org, 2007). This requires you to use older versions, which might not produce working executables from your scripts anyway, or seek out alternatives. There are some commercial alternatives. ActiveState sells a Perl Developer's Kit that includes PerlApp, a utility that turns a Perl script into an executable. It is important to note that it doesn't create native code, but rather wraps the Perl script with a Perl runtime interpreter. This means that you don't see any increase in performance, but it also means that all of the Perl functionality that benefits from running in an interpreted environment is still there and you have solid compatibility. The two primary benefits are the ability to obscure your scripts and the ability to run on another system without having to first

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

install a duplicate Perl environment, including any added Perl modules. (Remember that WinPcap is still a requirement and would need to be installed if it is not.) If you make an executable from PerlApp and want to use NSSM to run it as a service, you can call it directly, as you don't need to invoke the Perl interpreter and pass the script name as an option. A word of caution from my experience: sometimes PerlApp doesn't properly recognize and include the dependent Perl packages. When you attempt to run the executable on a system that doesn't already contain these missing modules, it will fail. In these instances, you will need to specify these modules manually as you are creating the executable.

Finally, the ActiveState Perl Developer's Kit also contains a utility called PerlSvc, which allows you to create an executable that will run directly as a Windows service, without the need of a utility such as NSSM or SRVANY.

3. Conclusion

Perl is a powerful language with greatly expanded capabilities thanks to the many modules supported in the community, including NET::PCAP and NET::PCAPUTILS. Using these modules to interface with Libpcap or Winpcap to process packets gives an analyst flexibility in capturing and analyzing network traffic.

This paper has described the initial setup of the Perl environment and additional modules and utilities required to capture packets. It has discussed how a network interface is selected and provided a small script utility for simplifying the selection. It has examined what packet data is available to a Perl script, how it is formatted, and how it can be easily be retrieved and manipulated within Perl for performing comparisons, format changes, logging, and other tasks. An analyst using the techniques and sample scripts described here, along with some existing or newly developed Perl experience, should be able to use Perl to capture and analyze network packets for whatever unique requirements he or she faces in examining network traffic.

John Brozycki, john@trueinsecurity.com

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

4. References

Aperghis-Tramoni, S. (2008). Net::Pcap. Retrieved from http://search.cpan.org/~saper/Net-Pcap-0.16/Pcap.pm. (accessed 9/16/2009).

Hall, J., & Schwartz, R. (1998). *Effective Perl Programming: Writing Better Programs with Perl*. Reading, MA: Addison-Wesley Professional.

Hietaniemi, J., Wegscheid, D., Schertler, R., & Aas, G. (2008). *Time-HiRes-1.9719*. Retrieved from http://search.cpan.org/~jhi/Time-HiRes-19.719/HiRes.pm (accessed 9/15/2009).

Meier, B. (2009). *KnownBugs - OutOfMemory*. Retrieved from http://wiki.wireshark.org/KnownBugs/OutOfMemory. (accessed 8/21/2009).

Perlmonks.org (2007). *What Happened to Perlcc?* Retrieved from http://www.perlmonks.org/?node_id=654568. (accessed 9/16/2009).

Quigley, E. (2002). *Perl by Example, Third Edition*. Upper Saddle River, NJ: Prentice Hall PTR.

Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., & Lear, E., (1996). *Address Allocation for Private Internets*. Retrieved from http://tools.ietf.org/html/rfc1918. (accessed 10/9/2009).

Schwartz, R., & Phoenix, T. (2001). *Learning Perl, 3rd Edition*. Sebastopol, CA: O'Reilly & Associates.

Tcpdump.org (2009). Tcpdump/libpcap. Retrieved from http://www.tcpdump.org. (accessed 8/3/2009).

TutorialsPoint.com (2009). *PERL printf Function*. Retrieved from http://www.tutorialspoint.com/perl/perl_printf.htm. (accessed 9/4/2009).

Winpcap.org (2009). *Description*. Retrieved from http://www.winpcap.org/default.htm. (accessed 8/3/2009).

John Brozycki, john@trueinsecurity.com

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Appendix A: tinycap.pl (first example of capturing packets with Perl)

```
#!/usr/bin/perl -w
  # ##########################################################################
  #
  use Net::PcapUtils;    #packet capture module to allow use of WinPcap or LibPcap
  #
  # ##########################################################################
  sub process_pkt       #Packet processing routine.
  {
     print("Got a packet!\n");
  }
  # ##########################################################################
  # Main part of program
  # Here we are invoking the NetPcap module and looping through forever.
  Net::PcapUtils::loop(\&process_pkt,
                       SNAPLEN => 65535,   #Size of data to get from packet
                       PROMISC => 1,);    #Promiscuous means look at ALL packets
```

John Brozycki, john@trueinsecurity.com

© 2010 SANS Institute, Author retains full rights.

Appendix B: pickinterface.pl (utility to specify and save the network interface name)

```
#!/usr/bin/perl
# John Brozycki 08/11/2009
# This program allows you to select the interface to use as the listening interface and saves
# the interface name to a settings file to be read in by webcap.pl as an input.  This is very
# handy as the interface names, especially in Windows, can be quite long and have
punctutation
# characters in them, making them hard to specify manually.
# This code is modified from an example Jean-Louis Morel sent to me when I was having trouble
# figuring out how to get capturing working under Windows.  I have modified it to save the
# selected interface to a settings file.  We can thank Jean-Louis for the
# Windows port of Net::Pcap.  His page is at www.bribes.org/perl.  Thanks, Jean-Louis!
#  use strict;
  use warnings;
  use Net::PcapUtils;
  $, = ' ';
  $|++;
  $settings = "c:\\webcap\\interface.txt";
  open (SETTINGSFILE,">$settings");
  my ( $error, %description );
  my @adapter = Net::Pcap::findalldevs( \$error, \%description );
  @adapter > 0 or die "No adapter installed !\n";
  my $i = 1;
  if ( @adapter > 1 ) {  #Change 1 to 0 if you want prompt even if only 1 adapter

   print "\nThis utility needs to be run before running webcap for the first time\n";
   print "and then when you change the network adapters in your system or want to\n";
   print "capture from a different adapter.\n\n";
   print "It outputs the selected adapter to a settings file.  Webcap reads\n";
   print "this file at startup.\n\n";
   print "Here are the adapters found:\n\n";
   print $i++, " - $description{$_}\n $_\n" foreach @adapter;
   do {
     print "\nPlease select the number of the adapter to set as the capture device:";
     $i = <STDIN>;
     chomp $i;
   } until ( $i =~ /^(\d)+$/ and 0 < $i and $i <= @adapter );
  }
  print "\nSet to Listen to $description{$adapter[$i-1]}\n\n";
  print "...which is referenced by the system as:\n\n".$adapter[ $i - 1]."\n";
  print SETTINGSFILE ($adapter[ $i - 1]);
  close SETTINGSFILE;
```

John Brozycki, john@trueinsecurity.com

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Appendix C: tinycap2.pl (Modified to use the interface file from pickinterface.pl)

```
#!/usr/bin/perl -w
  # ##########################################################################
  #
  use Net::PcapUtils;    #packet capture module to allow use of WinPcap or LibPcap
  my $settingfile = "c:\\webcap\\interface.txt"; #name of file to read our listening
  open (SETTINGS, $settingfile) or die "Cannot open setting file: $!";
  #
  # ##########################################################################
  sub process_pkt       #Packet processing routine.
  {
     print("Got a packet!\n");
  }
  # ##########################################################################
  while (<SETTINGS>)    #Read in the input adapter.  It was saved to a file when
                         #PickInterface.pl was run.
   {
    $interface = $_;
   }
  close SETTINGS;
  # Here we are invoking the NetPcap module and looping through forever.
  Net::PcapUtils::loop(\&process_pkt,
                       SNAPLEN => 1541,   #Size of data to get from packet
                       PROMISC => 1,      #Promiscuous means look at ALL packets
                       DEV => $interface, );
```

John Brozycki, john@trueinsecurity.com

© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Appendix D: showheader.pl (Display packet header bytes in hex, decimal, and binary.)

```
#!/usr/bin/perl -w
# ##################################################################
#
use Net::PcapUtils;    #packet cap module to allow use of WinPcap/LibPcap
my $settingfile = "c:\\webcap\\interface.txt"; #name of file with cap dev
open (SETTINGS, $settingfile) or die "Cannot open setting file: $!";
#
# ##################################################################
sub process_pkt       #Packet processing routine
{
   my ($user_data,$header, $packet) = @_;
   my $minipacket = substr($packet,0,54);
   print ("\n## raw: ###\n");
   print ($minipacket);
   print ("\n==Byte# / Hex / Dec / Bin==\n");
   for ($i=0;$i<55;$i++)
     {
       $hexval = unpack('H2',substr($packet,$i,1));
       $decval = hex(unpack('H2',substr($packet,$i,1)));
       printf ("%03s-%02s-%03s-%08b\n", $i, $hexval, $decval, $decval);
     }
}
# ##################################################################
# Main part of program
while (<SETTINGS>)   #Read in the input adapter. PickInterface.pl to set
 {
  $interface = $_;
 }
close SETTINGS;
# Here we are invoking the NetPcap module and looping through forever.
Net::PcapUtils::loop(\&process_pkt,
                     SNAPLEN => 65536,    #Size of data to get from packet
                     PROMISC => 1,       #Put in promiscuous mode
                    FILTER => 'tcp',    #only pass TCP packets
                     DEV => $interface, );
```

John Brozycki, john@trueinsecurity.com

© 2010 SANS Institute, Author retains full rights.

Appendix E: webcap.pl (capture and log external web access)

```
#!/usr/bin/perl -w
# ##########################################################################
# Title:         webcap.pl
# Purpose:       Captures all web sites accessed by PCs seen via a SPAN port,
#                hub, or tap to a text-based log.
# Author:        John Brozycki
# Date:          8/10/2009
# Last Modified: 12/11/2009 added variable for logfile directory
# ##########################################################################
#
use Net::PcapUtils;    #packet capture module to allow use of WinPcap or LibPcap
use Time::HiRes qw(gettimeofday); #Time module to provide time resolution < seconds
my $loopforever = 0;
my $firstrun = 1;     #variable to indicate 1st run through program for init. setup
my $origdate = 0;     #variable to store date to compare for date change
my $sep = ",";        #variable to allow field separator to be easily changed
my $loglocation = "c:\\webcap\\"; #Location to write logs to
my $settingfile = "c:\\webcap\\interface.txt"; #name of file to read our listening
                                               #interface.  run PickInterface.pl to set.
open (SETTINGS, $settingfile) or die "Cannot open setting file: $!";
#
# ##########################################################################
sub get_time          #Time processing.  Mash up DateTime and Time::HiRes to get
{                     #what we want (sys time + nanoseconds)
 @HiResTime = gettimeofday();
 ($sec,$min,$hour,$day,$month,$year)=localtime($HiResTime[0]);
 # Month starts with 0 for Jan, so add one.
 $month = $month + 1;
 # Year is offset from 1900, so add 1900 to make it actual.
 $year = $year + 1900;
 my $time = sprintf "%02d/%02d/%4d %02d:%02d:%02d:%06d",
$month,$day,$year,$hour,$min,$sec,$HiResTime[1];
}
# ##########################################################################
sub process_pkt       #Packet processing routine.
{
 my $time = &get_time;  #Call get_time subroutine to get the current time
 my ($user_data, $header, $packet) = @_;
 my $len = length $packet;
 # Extract Source and Destination IP addresses
 my $src_1st_octet = hex(unpack('H2',substr($packet,26,1)));
 my $src_2nd_octet = hex(unpack('H2',substr($packet,27,1)));
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
my $src_3rd_octet = hex(unpack('H2',substr($packet,28,1)));
my $src_4th_octet = hex(unpack('H2',substr($packet,29,1)));
my $dst_1st_octet = hex(unpack('H2',substr($packet,30,1)));
my $dst_2nd_octet = hex(unpack('H2',substr($packet,31,1)));
my $dst_3rd_octet = hex(unpack('H2',substr($packet,32,1)));
my $dst_4th_octet = hex(unpack('H2',substr($packet,33,1)));
#Extract the source and destination ports
my $srcport = hex(unpack('H4',substr($packet,34,2)));
my $dstport = hex(unpack('H4',substr($packet,36,2)));
# Extract the http data
my $httpdata = substr($packet,54,$len-54);
#Extract the "host", i.e.: www.google.com
my $startofhost = index($httpdata,"Host:")+6;
my $endofhostlength = (index($httpdata,"\r\n",$startofhost) - $startofhost);
my $host = substr($httpdata,$startofhost,$endofhostlength);
#Extract the URI, which is the resource being requested from the host, example "/"
my $startofuri = index($httpdata,"GET ")+4;
my $endofurilength = (index($httpdata,"HTTP/",$startofuri)-4); #skip back past the
newline and return characters
my $uri = substr($httpdata,$startofuri,$endofurilength);
# This section handles file output.  We want the filename to contain the date and we
#want a new file to be created when the date changes.
 if ($firstrun == 1)      # If 1st run, OPEN logfile and set original date variable
   {
     $formatdate = sprintf "%04d%02d%02d",
                             $year,$month,$day;
     $logdate = $loglocation.$formatdate."_webcap\.txt";
     open (LOGFILE,">>$logdate");
     $origdate = $formatdate;
     $firstrun = 0;
   }
 if ($firstrun == 0)      # If not 1st run, collect curr date & compare to orig date.
                          #If date changed output to new file
   {
     my $newdate = sprintf "%04d%02d%02d",$year,$month,$day;
     if ($newdate ne $origdate)
       {
         $logdate = $loglocation.$newdate."_webcap\.txt";
         open (LOGFILE,">>$logdate");
         $origdate = $newdate;
       }
   }
 # We don't want to log access to devices on the Intranet, so check destination IP
 # range.  We will check for "10" in the first octet, "192.168" in the first two octets,
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

```
  # or "172" in the first octet and the range of 16-31 in the second octet.
$nooutput = 0;
$intranet = $dst_1st_octet.$dst_2nd_octet;
if (($intranet eq "192168") || ($dst_1st_octet eq "10"))
   {
      $nooutput = 1;  # Set to 1 to prevent output
   }
if (($dst_1st_octet eq "172") && (($dst_2nd_octet > 15) && ($dst_2nd_octet < 32)))
   {
      $nooutput = 1;
   }
# Next we will output the packet information we wanted to the logfile.  That info is:
# Date/Time, SRC IP, SRC Prt, DEST IP, DEST Prt, Requested URL (Host + URI).
if ($nooutput == 0) #If = 1 then don't output it, as it's an Intranet access
   {
      print LOGFILE
($time.$sep.$src_1st_octet."\.".$src_2nd_octet."\.".$src_3rd_octet."\.".$src_4th_octet.$s
ep.$srcport.$sep);
      print LOGFILE
($dst_1st_octet."\.".$dst_2nd_octet."\.".$dst_3rd_octet."\.".$dst_4th_octet.$sep.$dstport
.$sep.$host.$uri."\n");
   }
}
# ##########################################################################
# Main part of program
my ( $error, %description );
my @adapter = Net::Pcap::findalldevs( \$error, \%description );
@adapter > 0 or die "No adapter installed !\n";
while (<SETTINGS>)    #Read in the input adapter.  It was saved to a file when
                      #PickInterface.pl was run.
 {
   $interface = $_;
 }
close SETTINGS;
# Here we are invoking the NetPcap module and looping through forever.
Net::PcapUtils::loop(\&process_pkt,
                     SNAPLEN => 65536,                  #Size of data to get from packet
                     PROMISC => 1,                     #Promiscuous = ALL packets
                     FILTER => 'tcp[20:4] = 0x47455420', #This is a BPF capture filter.
                                                       # TCP and "GET"
                     DEV => $interface, );             #This is the int. to capture from
```


© 2010 The SANS Institute As part of the Information Security Reading Room Author retains full rights.

Last Updated: May 19th, 2013

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Malaysia @ MCMC 2013	**Cyberjaya, MY**	**Jun 03, 2013 - Jun 08, 2013**	**Live Event**
SANS Pen Test Berlin 2013	**Berlin, DE**	**Jun 03, 2013 - Jun 08, 2013**	**Live Event**
Industial Control Systems Security Training - Houston	**Houston, TXUS**	**Jun 10, 2013 - Jun 15, 2013**	**Live Event**
Security Impact of IPv6 Summit 2013	**Washington, DCUS**	**Jun 14, 2013 - Jun 16, 2013**	**Live Event**
SANSFIRE 2013	**Washington, DCUS**	**Jun 14, 2013 - Jun 22, 2013**	**Live Event**
SANS Canberra 2013	**Canberra, AU**	**Jul 01, 2013 - Jul 13, 2013**	**Live Event**
SANS London Summer 2013	**London, GB**	**Jul 09, 2013 - Jul 16, 2013**	**Live Event**
Digital Forensics & Incident Response Summit 2013	**Austin, TXUS**	**Jul 09, 2013 - Jul 16, 2013**	**Live Event**
SANS Rocky Mountain 2013	**Denver, COUS**	**Jul 14, 2013 - Jul 20, 2013**	**Live Event**
SANS Mumbai 2013	**Mumbai, IN**	**Jul 22, 2013 - Jul 27, 2013**	**Live Event**
SEC528 SANS Training Program for the CompTIA® New Advanced Security Practitioner (CASP) Certification	**Washington, DCUS**	**Jul 22, 2013 - Jul 26, 2013**	**Live Event**
SANS San Francisco 2013	**San Francisco, CAUS**	**Jul 29, 2013 - Aug 03, 2013**	**Live Event**
SANS SEC 560: Network Penetration Testing @ Bangalore 2013	**Bangalore, IN**	**Aug 05, 2013 - Aug 10, 2013**	**Live Event**
SANS Boston 2013	**Boston, MAUS**	**Aug 05, 2013 - Aug 10, 2013**	**Live Event**
Critical Security Controls Summit	**Washington, DCUS**	**Aug 12, 2013 - Aug 18, 2013**	**Live Event**
Industrial Control Systems Security Training - DC	**Washington, DCUS**	**Aug 12, 2013 - Aug 16, 2013**	**Live Event**
Mobile Device Security Summit 2013	**OnlineCAUS**	**May 30, 2013 - Jun 06, 2013**	**Live Event**
SANS OnDemand	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**