Part I

# Perl Scripting and Live Response

**Solutions for this Part:**

- Built-in Functions
- Running Processes
- Accessing the API
- WMI
- Accessing the Registry
- ProScripts

This Part focuses on the use of Perl when extracting data from a live system, as part of live response. "Live response" is a general term used to describe activities that are performed when information is needed from a system while it is still running. This most often involves collecting volatile data from a system, or data that is only available when the system is powered on and running. Live response activities can include something as simple as an administrator troubleshooting an issue on a system, or collecting process and network connection information from a system prior to powering the system down and acquiring an image of the system's hard drive. These activities can also include inventory control (determining who's logged into a system, what software is installed on a system, and so forth), and can be performed locally (while the administrator is sitting at the console) or remotely, over the network.

## Built-in Functions

ActiveState Perl comes with several built-in Windows (i.e., Win32) functions that allow you to access and retrieve specific information from a Windows system. For example, you can determine the current working directory (Win32::GetCwd()), the system architecture, and type of CPU of the system (Win32::GetArchName() and Win32::GetChipName(), respectively), as well as a number of other very useful pieces of information. All of these functions are simply interfaces into the appropriate Windows application program interface (API) function calls, and allow the programmer to quickly retrieve the information they're looking for.

### Win32.pl

Demonstrates the use of some of the Perl Win32 built-in functions:

```
use strict;
use Win32;
print "Architecture  : ".Win32::GetArchName()."\n";
print "Chip          : ".Win32::GetChipName()."\n";
print "Perl Build    : ".Win32::BuildNumber()."\n";
print "Node Name     : ".Win32::NodeName()."\n";
print "Login Name    : ".Win32::LoginName()."\n";
print "OS Name       : ".Win32::GetOSName()."\n";
my ($str,$maj,$min,$build,$id) = Win32::GetOSVersion();
print "$str $maj $min $build $id\n";
```

On my test system the output from this script appears as follows:

```
C:\Perl>win32.pl
Architecture  : x86
Chip          : 586
Perl Build    : 819
Node Name     : WINTERMUTE
Login Name    : Harlan
OS Name       : WinXP/.Net
Service Pack 2 5 1 2600 2
```

As you can see, some of this information can be quite useful during incident response. Check the ActiveState Perl documentation for a complete list of Win32 functions.

## Pclip.pl

While not a built-in function, ActiveState Perl ships with several Perl modules that are specific to the Windows platform. For example, the Win32::Clipboard module allows you to set or retrieve the contents of the Windows Clipboard.

```
use strict;
use Win32::Clipboard;
my $clip = Win32::Clipboard();
my $clipboard;
if ($clipboard = $clip->Get()) {
  print "Clipboard Contents\n";
  print "-" X 20,"\n";
  print $clipboard."\n";
}
else {
  print "Error retrieving clipboad contents:
".Win32::FormatMessage(Win32::GetLastError())."\n";
}
```

Many times during incident response, there may be information available on the clipboard that may be of use to the investigator, such as portions of an e-mail or document, a password, or text transferred between windows on the desktop. The Win32::Clipboard module allows you to retrieve the contents of the clipboard, and display it in any way that is useful to you. Pclip.pl is a very simple example of the use of the module. Consult the Perl "plain old documentation" (POD) for the module for some ideas of a more complete script that is capable of handling bitmaps, lists of files, or other data formats.

As an example, I was looking up some directions to a location that I needed to visit, and that I had to provide to a friend. I found the street address of the location and selected it in one Web page window, copied it, and pasted it into the e-mail that I was preparing to send. Afterward, I ran pclip.pl and this is what I got back:

```
Clipboard Contents
--------------------
123 Fake Street
```

Imagine what people copy into their clipboards throughout the day, many without really understanding what happens. I suggest that just as an experiment, you should go around an office or a school, or you can even do this at home, and simply open a Notepad window, place the cursor anywhere within the window, and press Ctrl-V. Whatever is in the clipboard will be pasted into Notepad. Pclip.pl allows you to automate this collection process.

## Running Processes

When performing live response, we are working with and interacting with a live, running system. Many times, when responding to an incident, a user may still be logged into the system. In some cases, such as employee workstations within an organization, this user may be the employee themselves. In others, such as in server rooms and data centers, this user will most likely be a system administrator. Often, an incident will occur and we will need to log into the system ourselves (as a consultant, I always have the system administrator do that) in order to obtain information from a system. The point is that in order to collect information from a live system, there has to be an account logged into the system, either at the console (via the keyboard) or over the network.

As the system is live and running, there are processes running, threads being executed, and code being processed. This is how we interact with the system; we "ask" the system for information by running processes ourselves. Our Perl scripts may be processes, but many times it is simply much easier to run external, third-party tools, or even tools that are native to the system itself, in order to get the information we need. For example, let's say that we'd like to get a list of open network connections from a system. The first thing that comes to mind as a means of requesting this information from the system is the native utility, netstat.exe.

One question that may immediately come to mind is, if I can run netstat.exe (or any other tool) from the command line, why bother to do it via a Perl script? Well, there are a couple of very good answers to that. One is that by including the use of the tool or utility in a Perl script (or batch file), we have a form of self-documentation. Documentation is a very important aspect of incident response. Second, many of the tools we may want to run on systems have a number of command-line arguments, and I don't know about you, but sometimes in the heat of the moment, I may not be able to keep that information straight, particularly at 2:30 A.M. when I'm trying to collect information from systems' that may have been compromised. So, by including the tool or utility in a Perl script, I have a degree of automation that prevents me from making mistakes, particularly through repetition. Finally, it's not often that I deal with only one system, or one tool or utility. Most often, I'm responding to 10, 50, or 100 systems, and I'm running a number of different tools on each of those systems. Using a Perl script, I'm able to put everything into a single command so that when the situation changes, I'm prepared. That way, if something happens further down the road and someone asks me what I did, I can refer back to the Perl script and the copies of the tools I ran.

So, there are a couple of ways that we can run programs on a system. Using netstat.exe, we'll take a look at several of them. Do not think that these are the only ways to address this particular issue. One of the strengths of Perl is that there is usually more than one way to complete a task. What I'm going to do here is show you some of what I have come up with, but this does not mean that these methods or Perl scripts are the *only* way to do things.

## Netstat1.pl

Perhaps the simplest way to launch external programs in Perl is to use the system() function. The system() function simply forks a child process from a parent process, which waits for the child process to complete, and then exits. A very simple use of the system() function, using netstat.exe as our example, is as follows:

```
use strict;
my @args = ("netstat", "-ano");
system(@args);
```

While this could have been much simpler in only a single line, simplicity or elegance isn't the issue here. What happens when we run this code is that the output of our command appears at the console, or standard output (i.e., STDOUT). So all we've really done here is added a layer of abstraction and not really bought ourselves anything useful. In order to save the output of the command, for example, we'd still need to use the redirection operator at the command prompt:

```
C:\>perl nestat1.pl > netstat.log
```

That's really no different from not using Perl at all:

```
C:\>netstat -ano > netstat.log
```

So, a bit of extra effort, but it would appear that we really haven't bought ourselves anything. Now, this might be different if we were using this script to run multiple commands; after all, wouldn't we then be benefiting from automation? During incident response, you're usually under pressure, either from your boss or the clock, or you're tired because it's 3:00 A.M., and the first thing that will happen is that you'll forget a command or mistype a command or something that will be frustrating under those conditions. By linking the commands together into a script, we can now type in a single command, a couple of short keystrokes, and have everything run for us. However, at this point, we really don't have anything much more than a batch file contained in a Perl script. We haven't taken full advantage of the power of Perl to make our jobs easier.

## Netstat2.pl

Another way to run external commands through Perl is to use backticks. Backticks are not the single quote operator on your keyboard; rather the backticks are the slanted single quote operator. Using the backtick operator, you can access system commands or even external commands (replace netstat.exe with your program of choice, ensuring that it is located in the PATH). For example, let's call the following code "netstat2.pl":

```
#! c:\perl\bin\perl.exe
use strict;
my @netstat = 'netstat -ano';
map{print "$_"}@netstat;
```

Notice that we launch netstat.exe with the "a," "n," and "o" switches, and collect the output of the command, whatever it may be, into a Perl list (or array). From there, the script finishes by simply printing out what's in the list. Now, the output of the command isn't at the console (STDOUT); rather, we've got control of that output and we can do what we like with it.

Master Craftsman

### Extending the Use of Backticks

You can use the backticks to not only launch applications on the system, or even applications and programs external (i.e., not native) to the system, but also to access native commands, such as "dir." "Dir" doesn't exist as an executable file on a system, but it is an accessible command.

Other things you can do is include a list of commands in an array (such as dir /ah, netstat –ano, and so forth) and then iterate through the list, running each command individually. If you're interested in running several commands and correlating the output or filtering the output, the Perl lists make that very easy to do.

Now we're at the point where we're making our jobs a little easier. For example, I can filter through the output, looking for a particular Internet Protocol (IP) address, or skipping lines that contain the loopback address (127.0.0.1). I can minimize the output, showing only the things I want to see, rather than showing me everything. I can filter the data, showing only those network connections that are in a particular state, such as LISTENING, TIME_WAIT, or ESTABLISHED. The point is, we're now making our jobs easier by running a command of our choosing and being able to manage the output of that command.

## Netstat3.pl

The Win32::Job module provides a bit more granularity of control when creating and running processes, as shown in netstat3.pl below.

```
#! c:\perl\bin\perl.exe
use strict;
use Win32::Job;
eval {
      my $job = Win32::Job->new();
      my $result = $job->spawn("netstat.exe","netstat.exe -ano");
      die "Value is undefined. ".$^E."\n" unless (defined $result);
      my $ok = $job->run(60);
};
print $@."\n" if ($@);
```

When we run netstat3.pl, we get the same sort of output we would expect to see if we were running the netstat –ano command from the command line; however, in this case, we are able to use the Win32::Job module to do things such as limit the amount of time that the process runs. In netstat3.pl, we limit that time to 60 seconds, which is a long time, and probably more time than we need in most cases. However, I have seen simple command-line tools (such as netstat.exe) hang when run on some systems, or simply take an inordinate amount of time to run (due to high processing overhead from other processes, and so forth). In such cases, we may want to limit how long the process runs, and that's where Win32::Job comes in.

There are a couple of other functions within the Win32::Job module that may be of use, depending upon what you're doing and the level of control of the process you wish to achieve. For example, you can use the spawn() function to redirect STDOUT and STDERR messages to log files, or you can use the watch() function to provide a handler for the process, in order to achieve an even more granular level of control over the process. Check the POD for the Win32::Job module and for the Win32::Process module, for other ideas on how to run external processes from within Perl code.

Also notice the use of the eval{} block. This allows us to tell Perl to evaluate the code, and trap any errors that may occur. One of the big ones that occurred when I was writing and testing the above code was that I had misspelled the name of the executable (i.e., "nestat.exe" instead of "netstat.exe"). While this is not an error that would cause a major application crash, the error was trapped, nonetheless. The eval{} block is useful for trapping such errors, and even allowing your code to progress in the event of an error that you simply wish to recover from (and not have your entire script bomb out!).

## Accessing the API

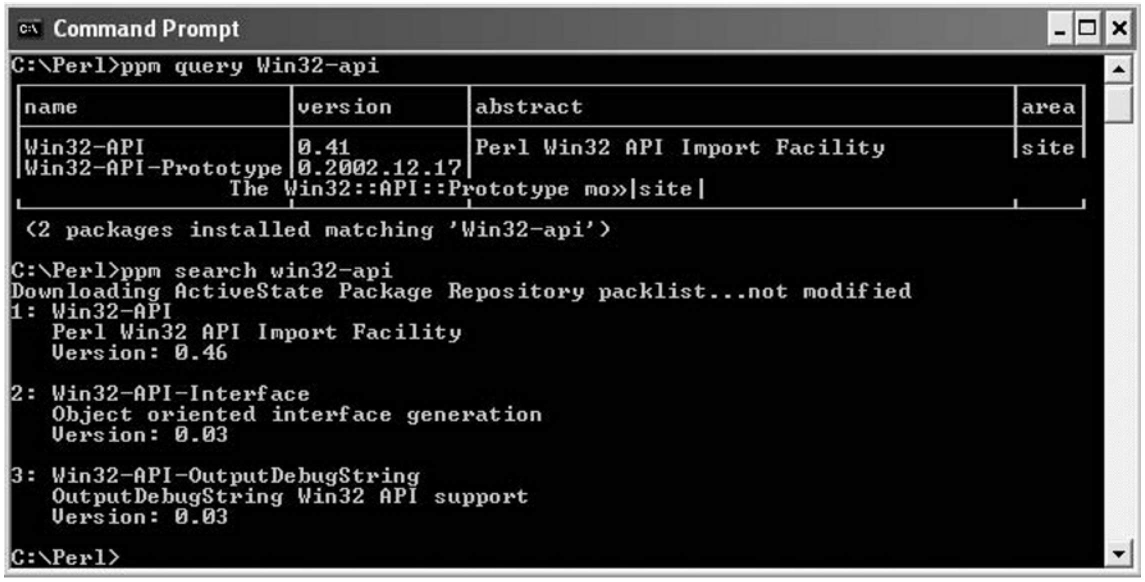
When performing live response or perhaps even analyzing files retrieved from a system during live response, you may want to access the Windows API. The Windows API can provide some useful functionality, already partially built. Fortunately, Microsoft exposes a good portion of the public API via the Microsoft.com Web site, and in addition, there are books available that describe other API functions that are accessible, albeit not fully documented.

In order to access the Windows API, you need to be sure that you have the Win32::API module installed. You can check to see if this module has been installed in your Perl distribution by typing the following command at the command prompt.

```
C:\perl>ppm query Win32-api
```

Figure I.1 illustrates the output of this command on my system.

**Figure I.1** Querying for the Win32::API Module



```
C:\Perl>ppm query Win32-api
┌──────────────────────┬─────────────┬────────────────────────────────────────┬────┐
│name                  │version      │abstract                                │area│
├──────────────────────┼─────────────┼────────────────────────────────────────┼────┤
│Win32-API             │0.41         │Perl Win32 API Import Facility          │site│
│Win32-API-Prototype   │0.2002.12.17│
                 The Win32::API::Prototype mo»|site|
└──────────────────────┴─────────────┴────────────────────────────────────────┴────┘
 (2 packages installed matching 'Win32-api')

C:\Perl>ppm search win32-api
Downloading ActiveState Package Repository packlist...not modified
1: Win32-API
   Perl Win32 API Import Facility
   Version: 0.46

2: Win32-API-Interface
   Object oriented interface generation
   Version: 0.03

3: Win32-API-OutputDebugString
   OutputDebugString Win32 API support
   Version: 0.03

C:\Perl>
```

**NOTE**

Figure I.1 shows the output of two "ppm" commands. The first, the query command, queries the current installation to determine the version of the module that is installed. In this case, the version of the Win32::API module that is installed is 0.41. The second command is a "search" command that looks for the currently available versions of modules that start with "win32-api." The currently available version of the Win32::API module is 0.46. I can update my copy of the module by typing "ppm update Win32-API."

You'll notice in Figure I.1 that when I ran my query for "win32-api," all of the modules that began with that name were returned. What is this module named "Win32-API-Prototype[1]"? This is a module created by Dave Roth that encapsulates the Win32::API module and makes the Win32::API module easier to use.

[1] www.roth.net/perl/prototype/

**TIP**

To install the Win32::API::Prototype module, type the following command:
**C: \perl>ppm install http://www.roth.net/perl/packages/win32-api-prototype.ppd**

**NOTE**

By "easier to use" on the Web page that describes the Win32::API::Prototype module, Dave provides several examples of how to use a module to access API functions, as well as how to set up and format the various arguments. Dave uses a list (array) called "@ParameterTypes" to describe and hold the various types of the parameters or arguments of the function.

## Getsys.pl

When performing incident response, some of the information you may want to get from the live system includes things such as the current system time and the uptime of the system. All of these pieces of information can be retrieved via the Windows API. To do so, we'll use Dave's Win32::API::Prototype module to access a couple of Windows API calls:

```
#! c:\perl\bin\perl.exe
#--------------------------------------------------------------------------------
# getsys.pl
# This script demonstrates the use of the Win32::API::Prototype
# module to retrieve time-based information from the local system
#
# The only required module is Win32::API::Prototype:
# ppm install http://www.roth.net/perl/packages/win32-api-prototype.ppd
#
# Usage: [perl] getsys.pl
#
# Copyright 2001-2007 H. Carvey
#--------------------------------------------------------------------------------
use strict;
use Win32::API::Prototype;
my @month = qw/Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec/;
my @day = qw/Sun Mon Tue Wed Thu Fri Sat/;
```

```
# Meanings of the following constants can be found here:
# http://msdn.microsoft.com/library/default.asp?url
#      =/library/en-
#      us/sysinfo/base/gettimezoneinformation.asp
my @tz = qw/TIME_ZONE_ID_UNKNOWN TIME_ZONE_ID_STANDARD TIME_ZONE_ID_DAYLIGHT/;
ApiLink('kernel32.dll',
    'VOID GetSystemTime(LPSYSTEMTIME lpSystemTime)')
  || die "Cannot locate GetSystemTime()";
ApiLink('kernel32.dll',
    'DWORD GetTimeZoneInformation(
     LPTIME_ZONE_INFORMATION lpTimeZoneInformation)')
    || die "Cannot locate GetTimeZoneInformation()";
# The return value is the number of milliseconds that
#      have elapsed since the system was started.
# This value rolls over to zero after 49.7 days
ApiLink('kernel32.dll',
    'DWORD GetTickCount()')
    || die "Cannot locate GetTickCount()";
# Get the system time
# Ref: http://msdn.microsoft.com/library/default.asp?url=
#      /library/en-us/sysinfo/base/getsystemtime.asp
my $lpSystemTime = pack("S8", 0);
GetSystemTime($lpSystemTime);
my $str = sys_STR($lpSystemTime);
print "System Time : $str\n";
my ($day,$hour,$min,$sec) = getUpTime();
print "System Uptime: $day days, $hour hours, $min min, $sec sec.\n";
print "\n";
my $lpTimeZoneInformation = pack 'lA64SSSSSSSSlA64SSSSSSSSl',
0, ' ' X 64, 0, 0, 0, 0, 0, 0, 0, 0, 0, ' ' X 64, 0, 0, 0, 0, 0, 0, 0, 0, 0;
my $bias;
my $standardName;
my $standardBias;
my $dayLightName;
my $dayLightBias;
my @c;
my @f;
my $ret = GetTimeZoneInformation($lpTimeZoneInformation);
($bias, $standardName, $c[0], $c[1], $c[2], $c[3], $c[4], $c[5], $c[6], $c[7],
    $standardBias, $dayLightName, $f[0], $f[1], $f[2], $f[3], $f[4], $f[5],
$f[6], $f[7],
    $dayLightBias) = unpack '1A64SSSSSSSS1A64SSSSSSSS1', $lpTimeZoneInformation;
print "Return code => ".$tz[$ret]."\n";
# The bias is the difference, in minutes, between UTC time and local time.
# Convert to hours for presentation
# UTC = local time + bias
print "Bias   => ".$bias." minutes\n";
if (1 == $ret) {
  print "Standard Bias => ".$standardBias." minutes\n";
}
elsif (2 == $ret) {
  print "Daylight Bias => ".$dayLightBias." minutes\n";
}
else {
# do nothing
}
$standardName =~ s/\00//g;
$dayLightName =~ s/\00//g;
print "StandardName => ".$standardName."\n";
print "DaylightName => ".$dayLightName."\n";
# Convert returned SystemTime into a string
sub sys_STR {
    my $lpSystemTime = $_[0];
    my @time = unpack("S8", $lpSystemTime);
  $time[5] = "0".$time[5] if ($time[5] =~ m/^\d$/);
  $time[6] = "0".$time[6] if ($time[6] =~ m/^\d$/);
  my $timestr = $day[$time[2]]." ".$month[$time[1]-1]." ".
    $time[3]." ".$time[4].":".$time[5].":".$time[6]." ".
    $time[0];
  return "$timestr";
}
sub getUpTime {
    my $count = GetTickCount();
    my $sec = 1000;
    my $min = $sec * 60;
    my $hour = $min * 60;
    my $day = $hour * 24;
    my ($temp,$d,$h,$m,$s);
    if ($count > $day) {
      $d = (split(/\./,$count/$day,2))[0];
      $temp = $count%$day;
      $h = (split(/\./,($temp/$hour),2))[0];
      $temp = $temp%$hour;
      $m = (split(/\./,($temp/$min),2))[0];
      $temp = $temp%$min;
      $s = (split(/\./,($temp/$sec),2))[0];
  }
    elsif ($count > $hour) {
      $d = 0;
      $h = (split(/\./,($count/$hour),2))[0];
      $temp = $count%$hour;
      $m = (split(/\./,($temp/$min),2))[0];
      $temp = $temp%$min;
      $s = (split(/\./,($temp/$sec),2))[0];
  }
    elsif ($count > $min) {
      $d = 0;
      $h = 0;
      $m = (split(/\./,($count/$min),2))[0];
      $temp = $count%$min;
      $s = (split(/\./,($temp/$sec),2))[0];
  }
    elsif ($count > $sec) {
      $d = 0;
      $h = 0;
      $m = 0;
      $s = (split(/\./,($count/$sec),2))[0];
    }
    return ($d,$h,$m,$s);
}
```

Running the script returns the following information from my system:

```
C:\Perl>getsys.pl
System Time  : Fri Aug 31 22:57:38 2007
System Uptime: 0 days, 12 hours, 6min, 35sec.
Return code      => TIME_ZONE_ID_DAYLIGHT
Bias             => 300 minutes
Daylight Bias    => -60 minutes
StandardName     => Eastern Standard Time
DaylightName     => Eastern Daylight Time
```

Master Craftsman

### Getting Even More Information

You can extend the getsys.pl script to get things such as the current system time, the current Universal Coordinated Time (UTC) (UTC is analogous to Greenwich Mean Time [GMT]), the system name, the name of the logged on user, and so forth. For example, to get the system name, you might use the GetComputerNameA[2] API function, and to get the name of the logged on user, you might use the GetUserNameA[3] API function.

Retrieving information from a Windows system via the API can be useful, but it can also lead to problems. Many times, APIs will change between versions of Windows (such as between Windows 2000 and XP), or they may even change when a Service Pack is installed or updated. As such, direct use of the Windows API to collect some information from systems should be thoroughly tested before being deployed on a widespread basis.

## WMI

The Windows Management Instrumentation (WMI) is a great way to obtain information from live Windows systems. WMI is really nothing more than many of the hard-core details of accessing the Windows API that have been encapsulated and made easier to use. Instead of having to write code that accesses a system to determine what version of Windows it is and then take appropriate steps based on that version, an administrator can write code that will work (in most cases) consistently across Windows 2000 all the way through Vista. This means that an administrator or incident responder can request a list of the active processes from systems from across the enterprise, either locally on the host systems or remotely from a centrally located management console, and use the same code to get the same results, regardless of the version of Windows being queried. The advantage of this is that during incident response, many times some tools work better on some systems than on others and some tools simply do not work at all.

[2] http://msdn2.microsoft.com/en-us/library/ms724295.aspx
[3] http://msdn2.microsoft.com/en-us/library/ms724432.aspx

Another advantage of WMI is that it provides a cleaner, easier to use interface to some (albeit not all) of what you can access via the Win32::API and Win32::API::Prototype modules. For example, you can access information about the microprocessor, physical memory, hard drives, and other devices on the systems.

The Win32::OLE module provides the interface through which you can use Perl to access the WMI classes. The WMI classes provide access to operating system classes[4], such as classes that provide access to information pertaining to files, processes, drivers, networking, operating system settings, and so forth. The computer system hardware classes[5] provide access to information about devices on the system, such as the processor(s), hard drivers, batteries, fans, and so forth.

## Fw.pl

While one advantage of the WMI classes is that they provide a common interface to certain aspects of the Windows platform regardless of the operating system version, one disadvantage is that some versions of Windows have functionality that others do not. For example, Windows XP Service Pack 2 and Windows 2003 have a built-in firewall that is part of the Security Center, something neither Windows NT 4.0 (WMI classes were installed as a separate download for Windows NT) nor Windows 2000 have.

```
#! c:\perl\bin\perl.exe
#--------------------------------------------------------------------------------
# fw.pl
# Use WMI to get info about the Windows firewall, as well as
# information from the SecurityCenter
#
# Usage: fw.pl [-bsph] [-app] [-sec]
#
# copyright 2006-2007 H. Carvey keydet89@yahoo.com
#--------------------------------------------------------------------------------
use strict;
use Win32::OLE qw(in);
use Getopt::Long;
my %config = ();
```

4 http://msdn2.microsoft.com/en-us/library/aa392727.aspx

5 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/computer_system_hardware_classes.asp

```
Getopt::Long::Configure("prefix_pattern=(-|\/)");
GetOptions(\%config, qw(b s sec p app help|?|h));
# if -h, print syntax info and exit
if ($config{help}) {
   \_syntax();
   exit 1;
}
# some global hashes used throughout the code
my %proto =   (6 => "TCP",
               17 => "UDP");
my %ipver =   (0 => "IPv4",
               1 => "IPv6",
               2 => "Any");
my %type =   (0 => "DomainProfile",
               1 => "StandardProfile");
print "[".localtime(time)."] Checking Windows Firewall on ".Win32::NodeName()."...\n"
  unless ($config{sec});
# Create necessary objects
my $fwmgr = Win32::OLE->new("HNetCfg.FwMgr")
   || die "Could not create firewall mgr obj: ".Win32::OLE::LastError()."\n";
my $fwprof = $fwmgr->LocalPolicy->{CurrentProfile};
if (! %config || $config{b}) {
# Profile type: 0 = Domain, 1 = Standard
   print "Current Profile = ".$type{$fwmgr->{CurrentProfileType}}." ";
   if ($fwprof->{FirewallEnabled}) {
     print "(Enabled)\n";
   }
   else {
     print "(Disabled)\n";
     exit(1);
   }
   ($fwprof->{ExceptionsNotAllowed}) ?(print "Exceptions not allowed\n"):
(print "Exceptions allowed\n");
   ($fwprof->{NotificationsDisabled})?(print "Notifications Disabled\n"):
(print "Notifications not disabled\n");
   ($fwprof->{RemoteAdminSettings}->{Enabled}) ? (print "Remote Admin Enabled\n") :
(print "Remote Admin Disabled\n");
   print "\n";
}
if (! %config || $config{app}) {
   print "[Authorized Applications]\n";
```

```
    foreach my $app (in $fwprof->{AuthorizedApplications}) {
      if ($app->{Enabled} == 1) {
        print $app->{Name}." - ".$app->{ProcessImageFileName}."\n";
        print "IP Version = ".$ipver{$app->{IPVersion}}."; Remote Addrs = "
.$app->{RemoteAddresses}."\n";
        print "\n";
      }
    }
}
if (! %config || $config{p}) {
    print "[Globablly Open Ports]\n";
    foreach my $port (in $fwprof->{GloballyOpenPorts}) {
      if ($port->{Enabled} == 1) {
        my $pp = $port->{Port}."/".$proto{$port->{Protocol}};
        printf "%-8s %-35s %-20s\n",$pp,$port->{Name},$port->{RemoteAddresses};
      }
    }
    print "\n";
}
if (! %config || $config{s}) {
    print "[Services]\n";
    foreach my $srv (in $fwprof->{Services}) {
      if ($srv->{Enabled}) {
        print $srv->{Name}." (".$srv->{RemoteAddresses}.")\n";
        foreach my $port (in $srv->{GloballyOpenPorts}) {
          if ($port->{Enabled} == 1) {
            my $pp = $port->{Port}."/".$proto{$port->{Protocol}};
            printf " %-8s %-35s %-20s\n",$pp,$port->{Name},$port->{RemoteAddresses};
          }
        }
        print "\n";
      }
    }
}
# Check the SecurityCenter for additional, installed, WMI-managed FW and/or
AV software
# Some AV products are not WMI-aware, and may need a patch installed
if ($config{sec}) {
    my $server = Win32::NodeName();
    print "[".localtime(time)."] Checking SecurityCenter on $server...\n";
```

```
   my $objWMIService = Win32::OLE->GetObject("winmgmts:\\\\$server\\root\\
SecurityCenter") || die "WMI connection failed.\n";
# Alternative method
# my $locatorObj = Win32::OLE->new('WbemScripting.SWbemLocator') || die
#      "Error creating locator object: ".Win32::OLE->LastError()."\n";
# $locatorObj->{Security_}->{impersonationlevel} = 3;
# my $objWMIService = $locatorObj->ConnectServer($server,'root\
SecurityCenter',"","")
#      || die "Error connecting to $server: ".Win32::OLE->LastError()."\n";
   my $fwObj = $objWMIService->InstancesOf("FirewallProduct");
   if (scalar(in $fwObj) > 0) {
     foreach my $fw (in $fwObj) {
       print "Company = ".$fw->{CompanyName}."\n";
       print "Name = ".$fw->{DisplayName}."\n";
       print "Enabled = ".$fw->{enabled}."\n";
       print "Version = ".$fw->{versionNumber}."\n";
     }
   }
   else {
     print "There do not seem to be any non-MS, WMI-enabled FW products
installed.\n";
   }
   my $avObj = $objWMIService->InstancesOf("AntiVirusProduct");
   if (scalar(in $avObj) > 0) {
     foreach my $av (in $avObj) {
       print "Company = ".$av->{CompanyName}."\n";
       print "Name = ".$av->{DisplayName}."\n";
       print "Version = ".$av->{versionNumber}."\n";
       print "O/A Scan = ".$av->{onAccessScanningEnabled}."\n";
       print "UpToDate = ".$av->{productUptoDate}."\n";
     }
   }
   else {
     print "There do not seem to be any WMI-managed A/V products installed.\n";
   }
}
sub _syntax {
   print>> "EOT";
fw [-bsph] [-app]
Collect information about the Windows firewall (local system only) and
the SecurityCenter (additional WMI-managed FW and AV products)
```

```
  -b .........Basic info about Windows firewall only
  -app .......Display authorized application info for the Windows firewall
(enabled only)
  -s .........Display service info for the Windows firewall (enabled only)
  -p .........Display port info for Windows firewall (enabled only)
  -sec .......Display info from the SecurityCenter (other installed,WMI-
        managed FW and/or AV)
  -h .........Help (print this information)
Ex: C:\\>fw -s >server> -u >username> -p >password>
copyright 2006-2007 H. Carvey
EOT
}
```

There are a couple of things you'll notice about the fw.pl Perl script. One is the use of the Getopt::Long module in order to allow for the use of command-line arguments in the script. This allows us to program different functionality into a single script, rather than writing separate scripts to do slightly different things. For example, if you look at the content of the _syntax() function from the script, you'll see that you can use command-line arguments and switches to modify the output of the script and show different bits of information. This way, we can have one script with a complete set of functionality, rather than half a dozen different scripts. If I run the fw.pl script on my own system with just the "-b" switch, I get the following output:

```
C:\Perl>fw.pl -b
[Fri Aug 31 17:29:47 2007] Checking Windows Firewall on WINTERMUTE...
Current Profile = StandardProfile (Enabled)
Exceptions allowed
Notifications not disabled
Remote Admin Disabled
```

Running the script with just the "-s" switch to see the service information for the firewall, I get:

```
C:\Perl>fw.pl -s
[Fri Aug 31 17:31:41 2007] Checking Windows Firewall on WINTERMUTE...
[Services]
File and Printer Sharing (LocalSubNet)
  139/TCP NetBIOS Session Service  LocalSubNet
  445/TCP SMB over TCP             LocalSubNet
  137/UDP NetBIOS Name Service     LocalSubNet
  138/UDP NetBIOS Datagram Service LocalSubNet
```

```
UPnP Framework (LocalSubNet)
  1900/UDP SSDP Component of UPnP Framework      LocalSubNet
  2869/TCP UPnP Framework over TCP LocalSubNet
```

Using just the "-sec" switch to check the SecurityCenter[6] settings, I get:

```
C:\Perl>fw.pl -sec
[Fri Aug 31 17:32:57 2007] Checking SecurityCenter on WINTERMUTE...
There do not seem to be any non-MS, WMI-enabled FW products installed.
There do not seem to be any WMI-managed A/V products installed.
```

Now, had I had a WMI-enabled antivirus product installed on this system, it would show up in the output of the script. I would also be able to get some setting information from the system regarding an installed WMI-enabled firewall, if there is one, as in the following output taken from another system:

```
D:\Programs\Perl>fw.pl -sec
[Thu Sep 6 15:23:15 2007] Checking SecurityCenter on A1...
Company  = Check Point, LTD.
Name     = ZoneAlarm Pro Firewall
Enabled  = 1
Version  = 7.0.337.000
Company  = GRISOFT
Name     = AVG 7.5.485
Version  = 7.5.485
O/A Scan = 1
UpToDate = 1
```

As you can see, this system has the ZoneAlarm Pro Firewall and Grisoft AVG anti-virus (AV) applications installed (and more importantly, enabled), and the AV product appears to be up-to-date and enabled.

## Nic.pl

The Perl script nic.pl allows you to retrieve information about network interface cards (NICs) through the Win32_NetworkAdapterConfiguration[7] WMI class. The script allows you to collect information from either the local system, or from a remote system. While the class, like many other WMI classes, provides functions or methods for modifying information on the system, during incident response we're most interested in collecting information, so we'll stick to simply querying the system and retrieving the information that we need, and avoid modifying anything.

[6] http://support.microsoft.com/kb/883792
[7] http://msdn2.microsoft.com/en-us/library/aa394217.aspx

```
#! c:\perl\bin\perl.exe
#-------------------------------------------------------------------
# nic.pl
# Use WMI to get information about active network interface cards
# on a system
#
# Usage: [perl] nic.pl
#
# Copyright 2004-2007 H. Carvey keydet89@yahoo.com
#-------------------------------------------------------------------
use strict;
use Win32::OLE qw(in);
use Getopt::Long;
my %config = ();
Getopt::Long::Configure("prefix_pattern=(-|\/)");
GetOptions(\%config, qw(server|s=s user|u=s passwd|p=s csv|c help|?|h));
if ($config{help}) {
   \_syntax();
   exit 1;
}
if (! %config) {
   $config{server} = Win32::NodeName();
   $config{user} = "";
   $config{passwd} = "";
}
$config{user} = "" unless ($config{user});
$config{passwd} = "" unless ($config{passwd});
my $locatorObj = Win32::OLE->new('WbemScripting.SWbemLocator') || die
     "Error creating locator object: ".Win32::OLE->LastError()."\n";
$locatorObj->{Security_}->{impersonationlevel} = 3;
my $serverObj = $locatorObj->ConnectServer($config{server},'root\cimv2',$config{user}
,$config{passwd})
     || die "Error connecting to $config{server}: ".Win32::OLE->LastError()."\n";
foreach my $nic (in $serverObj->InstancesOf("Win32_NetworkAdapterConfiguration")) {
   if (defined($nic->{IPAddress})) {
     my $i = $nic->{IPAddress};
     my $ip = join(".",@{$i});
     next if ($ip eq '0.0.0.0');
     print $nic->{Description}."\n";
     print "\t$ip\n";
```

```
      print "\tIP Enabled\n" if ($nic->{IPEnabled});
      print "\t".$nic->{MACAddress}."\n\n";
    }
}
sub _syntax {
   print<< "EOT";
nic [-s system] [-u username] [-p password] [-c] [-h]
Collect network interface information from a local or remote system
  -s system......Name of the system to scan (default: localsystem)
  -u username....Username used to connect to the remote system (usually
       an Administrator)
  -p password....Password used to connect to the remote system
  -h.............Help (print this information)
Ex: C:\\>nic -s >server> -u >username> -p >password>
copyright 2004-2007 H. Carvey
EOT
}
```

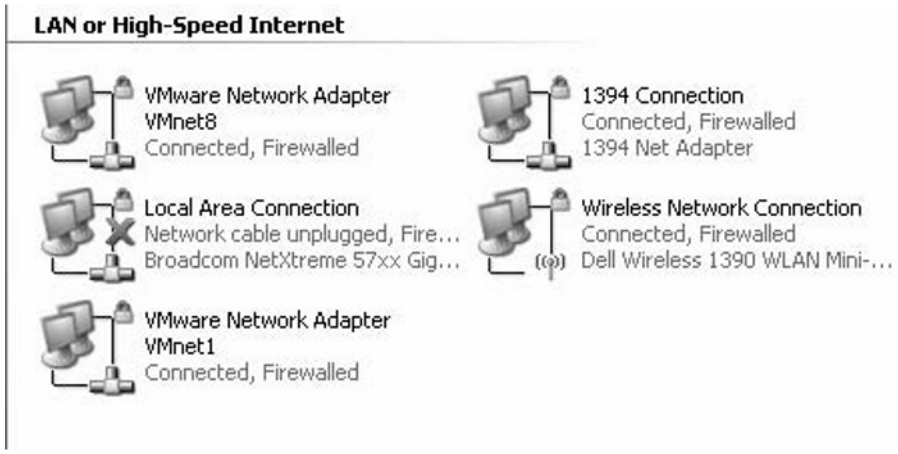
Running the script on my local system, you can see the information that the script returns:

```
C:\Perl>nic.pl
Dell Wireless 1390 WLAN Mini-Card - Packet Scheduler Miniport
  192.168.1.13
  IP Enabled
  00:16:CE:74:2C:B3
VMware Virtual Ethernet Adapter for VMnet1
  192.168.184.1
  IP Enabled
  00:50:56:C0:00:01
VMware Virtual Ethernet Adapter for VMnet8
  192.168.239.1
  IP Enabled
  00:50:56:C0:00:08
```

As you can see, we get the name of the adapter, the IP address, whether IP is enabled or not, and the Media Access Control (MAC) address of the interface. From the output from my system, you can see that I have a wireless adapter enabled, and two VMWare virtual adapters (which is good, because I have VMWare Workstation 6.0 installed). This also indicates that at the time the information was retrieved, the local area network (LAN) connection, which is usually accessible when plugging in a network cable to the

RJ-45 jack on my computer, is not enabled. Figure I.2 illustrates this information clearly via the Network Connections from the Settings menu on the test system.

**Figure I.2** Network Connections Visible On a Test System



LAN or High-Speed Internet

VMware Network Adapter VMnet8
Connected, Firewalled

1394 Connection
Connected, Firewalled
1394 Net Adapter

Local Area Connection
Network cable unplugged, Fire...
Broadcom NetXtreme 57xx Gig...

Wireless Network Connection
Connected, Firewalled
Dell Wireless 1390 WLAN Mini-...

VMware Network Adapter VMnet1
Connected, Firewalled

Using scripts such as nic.pl, we can preserve the state of the live system at a point in time, either prior to shutting it down, or simply to document which network connections were enabled and functioning at a specific moment. This information may not be readily available to us during a follow-on (i.e., "post-mortem") investigation after an image has been acquired from the system, and will most likely be extremely valuable to our investigation.

Swiss Army Knife

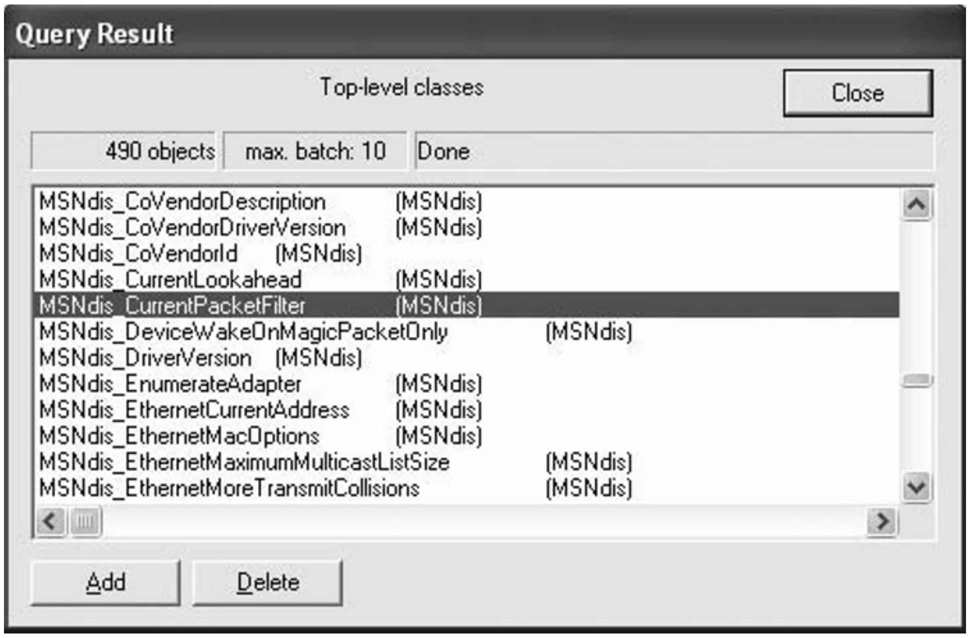
### Learning More About NICs

Additional information is at your fingertips when accessing the Win32_Network AdapterConfiguration class. For example, you can get information about the default gateway, whether Dynamic Host Configuration Protocol (DHCP) is enabled, as well as information about the Domain name system (DNS) and Internetwork Packet Exchange (IPX) configurations. Minor modifications to nic.pl will make this information available to you.

## Ndis.pl

WMI also provides access to Windows drivers through the Windows Driver Model (WDM). Figure I.3 demonstrates a dialog box that results from the use of the WBEMTest[8] tool, where I've listed the WDM classes available on my Windows XP SP 2 system. The highlighted class, MSNdis_CurrentPacketFilter, for example, provides us with access to the current filters for the NIC (note that a reference link is embedded in the comments at the beginning of the script).

**Figure I.3** Viewing Classes via the WMI Tester Interface (ch1-msndis.tif)



```
MSNdis_CoVendorDescription          (MSNdis)
MSNdis_CoVendorDriverVersion        (MSNdis)
MSNdis_CoVendorId      (MSNdis)
MSNdis_CurrentLookahead             (MSNdis)
MSNdis_CurrentPacketFilter          (MSNdis)
MSNdis_DeviceWakeOnMagicPacketOnly              (MSNdis)
MSNdis_DriverVersion   (MSNdis)
MSNdis_EnumerateAdapter             (MSNdis)
MSNdis_EthernetCurrentAddress       (MSNdis)
MSNdis_EthernetMacOptions           (MSNdis)
MSNdis_EthernetMaximumMulticastListSize         (MSNdis)
MSNdis_EthernetMoreTransmitCollisions           (MSNdis)
```

The ndis.pl script appears as follows:

```
#! c:\perl\bin\perl.exe
#-----------------------------------------------------
# ndis.pl - Perl script to determine settings of NIC;
#           Checks for promiscuous mode
#
# usage: C:\>[perl] ndis.pl
#
# Copyright 2007 H. Carvey keydet89@yahoo.com
#-----------------------------------------------------
```

[8] www.microsoft.com/technet/scriptcenter/resources/guiguy/wbemtest.mspx

```
use strict;
use Win32::OLE qw(in);
# OID_GEN_CURRENT_PACKET_FILTER values defined in ntddndis.h
# http://msdn.microsoft.com/library/default.asp?url=/library/en-us/
#      wceddk5/html/wce50lrfoidgencurrentpacketfilter.asp
my %filters = ("NDIS_PACKET_TYPE_DIRECTED" => 0x00000001,
"NDIS_PACKET_TYPE_MULTICAST" => 0x00000002,
"NDIS_PACKET_TYPE_ALL_MULTICAST" => 0x00000004,
"NDIS_PACKET_TYPE_BROADCAST" => 0x00000008,
"NDIS_PACKET_TYPE_SOURCE_ROUTING" => 0x00000010,
"NDIS_PACKET_TYPE_PROMISCUOUS" => 0x00000020,
"NDIS_PACKET_TYPE_SMT" => 0x00000040,
"NDIS_PACKET_TYPE_ALL_LOCAL" => 0x00000080,
"NDIS_PACKET_TYPE_GROUP" => 0x00000100,
"NDIS_PACKET_TYPE_ALL_FUNCTIONAL" => 0x00000200,
"NDIS_PACKET_TYPE_FUNCTIONAL" => 0x00000400,
"NDIS_PACKET_TYPE_MAC_FRAME" => 0x00000800);
my $server = Win32::NodeName();
my %nic = ();
my $locatorObj = Win32::OLE->new('WbemScripting.SWbemLocator') || die
  "Error creating locator object: ".Win32::OLE->LastError()."\n";
$locatorObj->{Security_}->{impersonationlevel} = 3;
my $serverObj = $locatorObj->ConnectServer($server,'root\wmi',"","")
  || die "Error connecting to \\root\\wmi namespace on $server: ".
Win32::OLE->LastError()."\n";
foreach my $ndis (in $serverObj->InstancesOf("MSNdis_CurrentPacketFilter")) {
  if ($ndis->{Active}) {
    my $wan = "WAN Miniport";
    next if ($ndis->{InstanceName} =~ m/^$wan/i);
    my $instance = (split(/-/,$ndis->{InstanceName}))[0];
    $instance =~ s/\s$//;
#      $nic{$instance} = 1;
    my @gpf = ();
    foreach my $f (keys %filters) {
      push(@gpf,$f) if ($ndis->{NdisCurrentPacketFilter} & $filters{$f});
    }
    $nic{$instance}{filter} = join(',',@gpf);
  }
}
```

```
foreach (keys %nic) {
  print "$_\n";
  my @filt = split(/,/,$nic{$_}{filter});
  foreach my $f (@filt) {
    ($f eq "NDIS_PACKET_TYPE_PROMISCUOUS") ? (print "\t--> $f <--\n") : (print "\t $f\n");
  }
  print "\n";
}
```

Again, when looking at the ndis.pl Perl script, we are most interested in the highlighted class, MSNdis_CurrentPacketFilter. This class provides us with visibility into the settings and filters for the adapter itself. This is important during incident response, because as in some cases, an intruder may have installed a network sniffer and placed the network adapter in "promiscuous" mode. This means that all of the packets that go by on the wire are read by the network adapter, not just the ones that are addressed to that system.

Running ndis.pl on my test system, I see:

```
Dell Wireless 1390 WLAN Mini
  NDIS_PACKET_TYPE_MULTICAST
  NDIS_PACKET_TYPE_DIRECTED
  NDIS_PACKET_TYPE_BROADCAST
```

This output is to be expected. I would be concerned if I saw the following included in the output:

```
NDIS_PACKET_TYPE_PROMISCUOUS
```

This would tell me that the network adapter is in promiscuous mode and is most likely being used for network sniffing.

Scripts using WMI can be run remotely against other managed systems, such as those within a domain, or those which the system administrator has local credentials on the system. For example, take a look at a line of code from ndis.pl:

```
my $serverObj = $locatorObj->ConnectServer($server,'root\wmi',"","")
```

You can see that the ConnectServer() function takes four arguments: the name of the server, the WMI namespace, and the username and password used to retrieve this information. In the code we're using, the administrator is running these scripts locally on the system from the account used to log in, so we don't need to provide login credentials within the script. Also, you'll notice that earlier in the script, we populated the $server variable with the following code:

```
my $server = Win32::NodeName();
```

Win32::NodeName(), if you remember, is one of the built-in Win32 functions. Again, using this function we get the name of the local system. However, if the system administrator wanted to reach out to other managed Windows 2000, XP, and 2003 systems within his or her domain, all he or she would have to do is include the name or IP address of the remote system, and the proper credentials. In fact, with an accurate list of all systems within the domain, he or she could run this script against all systems (some minor modifications to the script are required, of course, but those will be left to the reader) and display only those found to have NICs in promiscuous mode.

Master Craftsman

**Drivers for wireless access**

In 2004, Beetle of the Shmoo Group gave a presentation at ToorCon entitled "Wireless Weapons of Mass Destruction for Windows." That presentation included a number of VBscripts that accessed MSNdis_80211_* classes in order to retrieve Service Set Identifiers (SSIDs) "seen" by the wireless adapter, information about received signal strength, and so forth. This information can be used in a variety of ways. For example, during incident response, you may want to see if a laptop or even a workstation has a wireless adapter enabled, and if so, what SSID it is connected to. However, you can also use this same sort of information to triangulate the location of rogue access points. Let's say that you're in your office, and you suspect that there may be a rogue access point installed in another part of the building or in another building all together. Now, you know where server systems with wireless capability are physically located within the building; say, the Chief Executive Officer (CEO) has his laptop in his office, and just down the hall and around the corner the Vice President of Human Resources (HR) has her laptop in her office. You can then query each of these systems and determine the access points that each "sees" and the received signal strength of each one. From this information, you may be able to determine the approximate location of the rogue access point.

Master Craftsman

### Working with BitLocker

Windows Vista and 2008 incorporate an encryption technology referred to as BitLocker. There is a Win32_EncryptableVolume[9] class that allows you query the system and see if BitLocker is enabled. This is important, as encrypted drives pose an issue when it comes to acquiring an image of the hard drive. If BitLocker is enabled, the investigator may opt to perform a live acquisition of the system, rather than shutting the system down and removing the hard drive in order to acquire the image.

## Di.pl

WMI can be used to collect quite a bit more information. For example, there are WMI classes that allow you to collect information from other hardware on the system, such as disk drives and storage devices. When collecting information about a system, it is a good idea for the investigator to document the hardware components connected to the system. Also, when the system is shut down and images of the drives are acquired, the investigator is going to have to document information about the drives anyway, and WMI can be used to make the job easier. I wrote the Perl script di.pl ("di" stands for "drive information") to do just that, so that I would have complete information about the hard drives and storage media attached to the system:

```
#! c:\perl\bin\perl.exe
#-------------------------------------------------------------------------------
# di.pl - Disk ID tool
# This script is intended to assist investigators in
# identifying disks
# attached to systems. It can be used by an investigator to
# document
# a disk following acquisition, providing information for use
# in
# acquisition worksheets and chain-of-custody documentation.
```

[9] http://msdn2.microsoft.com/en-us/library/aa376483.aspx

```
#
# This tool may also be run remotely against managed system,
# by passing
# the necessary arguments at the command line.
#
# Usage: di.pl
# di.pl >system> >username> >password>
#
# copyright 2006-2007 H. Carvey, keydet89@yahoo.com
#--------------------------------------------------------------------------------
use strict;
use Win32::OLE qw(in);
my $server = shift || Win32::NodeName();
my $user = shift || "";
my $pwd = shift || "";
my $locatorObj = Win32::OLE->new('WbemScripting.SWbemLocator') || die
   "Error creating locator object: ".Win32::OLE->LastError()."\n";
$locatorObj->{Security_}->{impersonationlevel} = 3;
my $serverObj = $locatorObj->ConnectServer($server,'root\cimv2',$user,$pwd)
   || die "Error connecting to $server: ".Win32::OLE->LastError()."\n";
my %capab = (0 =>    "Unknown",
     1 =>     "Other",
     2 =>     "Sequential Access",
     3 =>     "Random Access",
     4 =>     "Supports Writing",
     5 =>     "Encryption",
     6 =>     "Compression",
     7 =>     "Supports Removable Media",
     8 =>     "Manual Cleaning",
     9 =>     "Automatic Cleaning",
     10 =>    "SMART Notification",
     11 =>    "Supports Dual Sided Media",
     12 =>    "Ejection Prior to Drive Dismount Not Required");
my %disk = ();
foreach my $drive (in $serverObj->InstancesOf("Win32_DiskDrive")) {
   $disk{$drive->{Index}}{DeviceID} = $drive->{DeviceID};
   $disk{$drive->{Index}}{Manufacturer} = $drive->{Manufacturer};
   $disk{$drive->{Index}}{Model} = $drive->{Model};
   $disk{$drive->{Index}}{InterfaceType} = $drive->{InterfaceType};
   $disk{$drive->{Index}}{MediaType} = $drive->{MediaType};
   $disk{$drive->{Index}}{Partitions} = $drive->{Partitions};
```

```
# The drive signature is a DWORD value written to offset 0x1b8 (440) in the MFT
# when the drive is formatted. This value can be used to identify a specific HDD,
# either internal/fixed or USB/external, by corresponding the signature to the
# values found in the MountedDevices key of the Registry
    $disk{$drive->{Index}}{Signature}    = $drive->{Signature};
    $disk{$drive->{Index}}{Size}         = $drive->{Size};
    $disk{$drive->{Index}}{Capabilities} = $drive->{Capabilities};
}
my %diskpart = ();
foreach my $part (in $serverObj->InstancesOf("Win32_DiskPartition")) {
    $diskpart{$part->{DiskIndex}.":".$part->{Index}}{DeviceID} = $part->{DeviceID};
    $diskpart{$part->{DiskIndex}.":".$part->{Index}}{Bootable} = 1
if ($part->{Bootable});
    $diskpart{$part->{DiskIndex}.":".$part->{Index}}{BootPartition} = 1
if ($part->{BootPartition});
    $diskpart{$part->{DiskIndex}.":".$part->{Index}}{PrimaryPartition} = 1
if ($part->{PrimaryPartition});
    $diskpart{$part->{DiskIndex}.":".$part->{Index}}{Type} = $part->{Type};
}
my %media = ();
foreach my $pm (in $serverObj->InstancesOf("Win32_PhysicalMedia")) {
  $media{$pm->{Tag}} = $pm->{SerialNumber};
}
foreach my $dd (sort keys %disk) {
  print "DeviceID     : ".$disk{$dd}{DeviceID}."\n";
  print "Model        : ".$disk{$dd}{Model}."\n";
  print "Interface    : ".$disk{$dd}{InterfaceType}."\n";
  print "Media        : ".$disk{$dd}{MediaType}."\n";
  print "Capabilities : \n";
  foreach my $c (in $disk{$dd}{Capabilities}) {
    print "\t".$capab{$c}."\n";
  }
  my $sig = $disk{$dd}{Signature};
  $sig = ">None>" if ($sig == 0x0);
  printf "Signature : 0x%x\n",$sig;
  my $sn = $media{$disk{$dd}{DeviceID}};
  print "Serial No : $sn\n";
  print "\n";
  print $disk{$dd}{DeviceID}." Partition Info : \n";
  my $part = $disk{$dd}{Partitions};
  foreach my $p (0..($part - 1)) {
    my $partition = $dd.":".$p;
    print "\t".$diskpart{$partition}{DeviceID}."\n";
    print "\t".$diskpart{$partition}{Type}."\n";
    print "\t\tBootable\n" if ($diskpart{$partition}{Bootable});
    print "\t\tBoot Partition\n" if ($diskpart{$partition}{BootPartition});
    print "\t\tPrimary Partition\n" if ($diskpart{$partition}{PrimaryPartition});
    print "\n";
  }
}
```

Di.pl can be run on a local system, or against a remote system. Simply running the following command will retrieve information from the local system:

```
C:\Perl>di.pl
```

To retrieve the same information from a remote system, you can run the command this way:

```
C:\Perl>di.pl 192.168.10.15 Administrator <password>
```

When run on my local system, this is the output that I see:

```
C:\Perl\tools>di.pl
DeviceID     : \\.\PHYSICALDRIVE0
Model        : ST910021AS
Interface    : IDE
Media        : Fixed hard disk media
Capabilities :
  Random Access
  Supports Writing
Signature    : 0x41ab2316
Serial No    : 3MH0B9G3
\\.\PHYSICALDRIVE0 Partition Info :
  Disk #0, Partition #0
  Installable File System
    Bootable
    Boot Partition
    Primary Partition
  Disk #0, Partition #1
  Extended w/Extended Int 13
DeviceID     : \\.\PHYSICALDRIVE1
Model        : WDC WD12 00UE-00KVT0 USB Device
Interface    : USB
Media        : Fixed hard disk media
```

```
Capabilities :
  Random Access
  Supports Writing
Signature    : 0x96244465
Serial No    :
\\.\PHYSICALDRIVE1 Partition Info :
  Disk #1, Partition #0
  Installable File System
    Primary Partition
```

As you can see, my system has two storage devices, the first of which is an internal fixed IDE hard drive, model ST910021AS, serial number 3MH0B9G3, with two partitions. The second storage device (i.e., PhysicalDrive1), is an external Universal Serial Bus (USB)-connected hard drive. So the di.pl script is useful for documenting storage hardware that is connected to a system, as well as providing some of the same information that the investigator will need to document (i.e., drive model, serial number, and so forth) when he or she acquires an image of that drive.

## Ldi.pl

There's another Perl script that I like to use sometimes that gets similar information as the previous script, but uses the Win32_LogicalDisk WMI class to obtain information about storage devices from the system. Ldi.pl (i.e., "Logical Disk Information") appears as follows:

```
#! c:\perl\bin\perl.exe
#-----------------------------------------------------------------------------
# ldi.pl - Logical Drive ID tool
# This script is intended to assist investigators in
# identifying
# logical drives attached to systems. This tool can be run
# remotely
# against managed systems.
#
# Usage: ldi.pl
# ldi.pl -h (get the syntax info)
# ldi.pl -s >system> -u >username> -p >password> (remote
# system)
# ldi.pl -c (.csv output - includes vol name and s/n)
#
# copyright 2006-2007 H. Carvey, keydet89@yahoo.com
#-----------------------------------------------------------------------------
```

```
use Win32::OLE qw(in);
use Getopt::Long;
my %config = ();
Getopt::Long::Configure("prefix_pattern=(-|\/)");
GetOptions(\%config, qw(server|s=s user|u=s passwd|p=s csv|c help|?|h));
if ($config{help}) {
  \_syntax();
  exit 1;
}
if (! %config) {
  $config{server} = Win32::NodeName();
  $config{user} = "";
  $config{passwd} = "";
}
$config{user} = "" unless ($config{user});
$config{passwd} = "" unless ($config{passwd});
my %types = (0 => "Unknown",
        1 => "Root directory does not exist",
        2 => "Removable",
        3 => "Fixed",
        4 => "Network",
        5 => "CD-ROM",
        6 => "RAM");
my $locatorObj = Win32::OLE->new('WbemScripting.SWbemLocator') || die
  "Error creating locator object: ".Win32::OLE->LastError()."\n";
$locatorObj->{Security_}->{impersonationlevel} = 3;
my $serverObj = $locatorObj->ConnectServer($config{server},'root\cimv2',$config{user},
$config{passwd})
  || die "Error connecting to $config{server}: ".Win32::OLE->LastError()."\n";
if ($config{csv}) {
}
else {
  printf "%-8s %-11s %-12s %-25s %-12s\n","Drive","Type","File System","Path",
"Free Space";
  printf "%-8s %-11s %-12s %-25s %-12s\n","-" x 5,"-" x 5,"-" x 11,"-" x 5,
"-" x 10;
}
foreach my $drive (in $serverObj->InstancesOf("Win32_LogicalDisk")) {
  my $dr = $drive->{DeviceID};
  my $type = $types{$drive->{DriveType}};
  my $fs = $drive->{FileSystem};
  my $path = $drive->{ProviderName};
  my $vol_name = $drive->{VolumeName};
  my $vol_sn = $drive->{VolumeSerialNumber};
  my $freebytes;
  my $tag;
  my $kb = 1024;
  my $mb = $kb * 1024;
  my $gb = $mb * 1024;
  if ("" ne $fs) {
    my $fb = $drive->{FreeSpace};
    if ($fb > $gb) {
       $freebytes = $fb/$gb;
       $tag = "GB";
    }
    elsif ($fb > $mb) {
       $freebytes = $fb/$mb;
       $tag = "MB";
    }
    elsif ($fb > $kb) {
       $freebytes = $fb/$kb;
       $tag = "KB";
    }
    else {
       $freebytes = 0;
    }
  }
  if ($config{csv}) {
    print "$dr\\,$type,$vol_name,$vol_sn,$fs,$path,$freebytes $tag\n";
  }
  else {
    printf "%-8s %-11s %-12s %-25s %-5.2f %-2s\n",$dr."\\
",$type,$fs,$path,$freebytes,$tag;
  }
}
sub _syntax {
  print>> "EOT";
L(ogical) D(rive)I(nfo) [-s system] [-u username] [-p password] [-h]
Collect logical drive information from remote Windows systems.
-s system......Name of the system to scan
-u username....Username used to connect to the remote system (usually
       an Administrator)
```

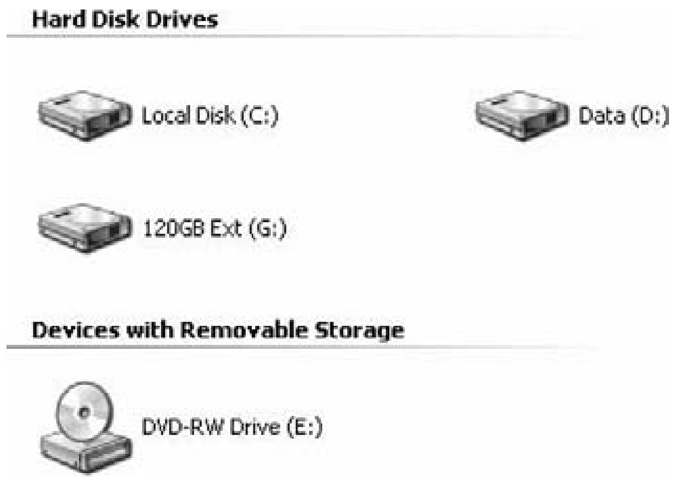
```
-p password....Password used to connect to the remote system
-c.............Comma-separated (.csv) output (open in Excel)
     Includes the vol name and s/n in the output
-h.............Help (print this information)
Ex: C:\\>di -s >server> -u >username> -p >password>
copyright 2006-2007 H. Carvey
EOT
}
```

As with di.pl, ldi.pl can be run locally or remotely. When run locally on my test system, I see the following output:

```
C:\Perl\tools>ldi.pl
Drive   Type     File System   Path   Free Space
-----   ----     -----------   ----   ----------
C:\     Fixed    NTFS                 18.22GB
D:\     Fixed    NTFS                 38.79GB
E:\     CD-ROM                         0.00
G:\     Fixed    NTFS                 42.25GB
```

As you can see, this output shows similar information to what we saw with di.pl, to some extent. In this case, drives C:\ and D:\ are the first and second partitions of the internal IDE hard drive on my system, and the G:\ drive is the external USB-connected hard drive. The "Path" column isn't populated for any of the drives, because none of them are mapped shares. Figure I.4 illustrates what this looks like via the My Computer window on the live system.

**Figure I.4** My Computer Window Showing Drives



When I connect a 4GB Cruzer Micro drive to my system, and run ldi.pl again, I see:

```
H:\    Removable     FAT32  3.82   GB
```

## Accessing the Registry

There are also times during incident response, or even during simple troubleshooting tasks, that you may want to query the Registry for specific information, such as check for the existence of a particular key or value, obtain a value's data, or determine the LastWrite time of a Registry key. Also, certain portions, or "hives" within the Registry are not accessible when the system has been shut down. For example, the HKEY_CURRENT_USER hive is only accessible to the user that is logged on; when the system is shut down, that hive is no longer available.

In other cases, the information you collect from the Registry may affect your follow-on investigation. For example, does the pagefile get cleared during a clean shutdown? Has the updating of last access times been disabled? Many times, knowing this (and other information like it) ahead of time can save us time later, or even completely redirect our next steps.

### Bho.pl

Browser Helper Objects[10] (BHOs) are essentially dynamic link library (DLL) files that add functionality to the Internet Explorer (IE) Web browser. A popular BHO is from Adobe, and it allows you to open PDF files for viewing right there in your Web browser. However, malware (spyware, mostly) authors will sometimes create malware that installs as a BHO, because their malware will automatically be launched every time the user runs IE. Malware authors are always looking for novel ways of getting their toys to run without any user interaction, and installing as a BHO is just one of them.

Bho.pl appears as follows:

```
#! c:\perl\bin\perl.exe
#-----------------------------------------------------------------------------
# BHO.pl
# Perl script to retrieve listing of installed BHOs from a
# local system
#
# Usage:
# C:\Perl>bho.pl [> bholist.txt]
#
# copyright 2006-2007 H. Carvey, keydet89@yahoo.com
#-----------------------------------------------------------------------------
```

[10] http://en.wikipedia.org/wiki/Browser_Helper_Object

```
use strict;
use Win32::TieRegistry(Delimiter=>"/");
my $server = Win32::NodeName();
my $err;
my %bhos;
my $remote;
# Get Browser Helper Objects
if ($remote = $Registry->{"//$server/LMachine"}) {
    my $ie_bho = "SOFTWARE/Microsoft/Windows/CurrentVersion/Explorer/
Browser Helper Objects";
    if (my $bho = $remote->{$ie_bho}) {
      my @keys = $bho->SubKeyNames();
      foreach (@keys) {
        $bhos{$_} = 1;
      }
    }
    else {
      $err = Win32::FormatMessage Win32::GetLastError();
        print "Error connecting to $ie_bho: $err\n";
    }
}
else {
    $err = Win32::FormatMessage Win32::GetLastError();
    print "Error connecting to Registry: $err\n";
}
undef $remote;
# Find out what each BHO is...
if ($remote = $Registry->{"//$server/Classes/CLSID/"}) {
    foreach my $key (sort keys %bhos) {
      if (my $conn = $remote->{$key}) {
        my $class = $conn->GetValue("");
        print "Class : $class\n";
        my $module = $conn->{"InprocServer32"}->GetValue("");
        print "Module: $module\n";
        print "\n";
      }
      else {
        $err = Win32::FormatMessage Win32::GetLastError();
        print "Error connecting to $key: $err\n";
      }
    }
}
else {
    $err = Win32::FormatMessage Win32::GetLastError();
    print "Error connecting to Registry: $err\n";
}
```

Running bho.pl on my system, I see the following:

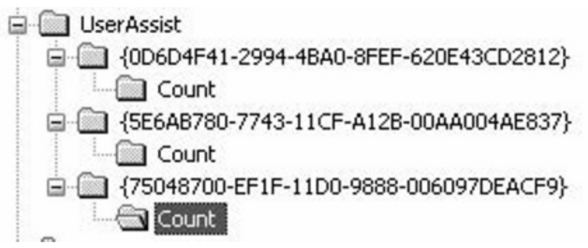
```
Class : DriveLetterAccess
Module: C:\WINDOWS\System32\DLA\DLASHX_W.DLL

Class : Windows Live Sign-in Helper
Module: C:\Program Files\Common Files\Microsoft Shared\Windows Live\
WindowsLiveLogin.dll
```

## Uassist.pl

The UserAssist Registry key is a key that has received a good deal of attention over the past year, largely due to its value to forensic investigators. This is due to the fact that the UserAssist key "records" a user's actions, or more appropriately, it "records" many of the user's interactions via the Windows Explorer shell. For example, when a user opens a Control Panel applet, or double-clicks an icon to launch an application such as Microsoft Word, or double-clicks a shortcut (*.lnk) file to open a file, these interactions are all recorded in the UserAssist key. Figure I.5 illustrates what the UserAssist key looks like via RegEdit.

**Figure I.5** Excerpt from RegEdit Showing the UserAssist Key and Subkeys



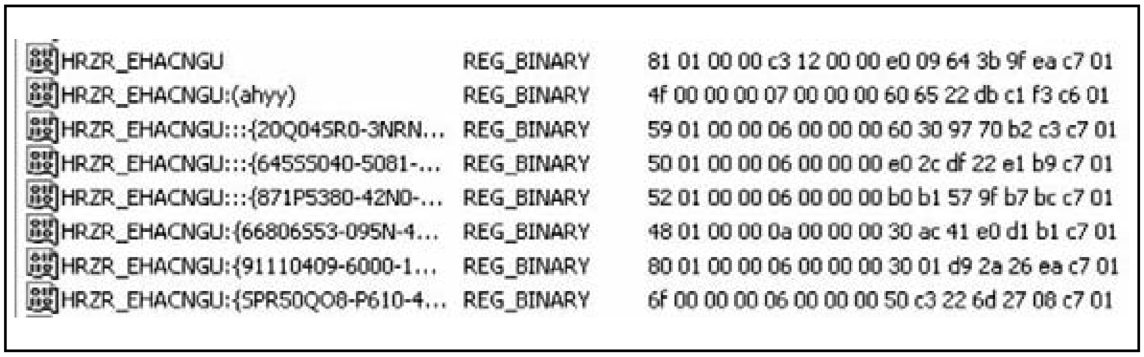
```
UserAssist
  {0D6D4F41-2994-4BA0-8FEF-620E43CD2812}
    Count
  {5E6AB780-7743-11CF-A12B-00AA004AE837}
    Count
  {75048700-EF1F-11D0-9888-006097DEACF9}
    Count
```

As you can see, the UserAssist key really consists of three keys, each represented by a globally unique identifier (GUID) that points to a specific class. For example, the GUID that starts with "{5E6AB780" refers to the Internet Explorer Toolbar, while the GUID that starts with "{75048700" refers to the Active Desktop. The last GUID is added to a system when you install Internet Explorer version 7.

The subkey we're most interested in is the one that points to the Active Desktop class, or the shell. As you can see from Figure I.5, each GUID key has a subkey named "Count." Even though there is an additional layer or two of subkeys, all of these keys are collectively referred to as the "UserAssist keys," largely because it's tough to remember the GUIDs.

Figure I.6 illustrates the value names beneath the Count key, as they appear in RegEdit.

**Figure I.6** Excerpt of RegEdit Showing One of the UserAssist Key Values



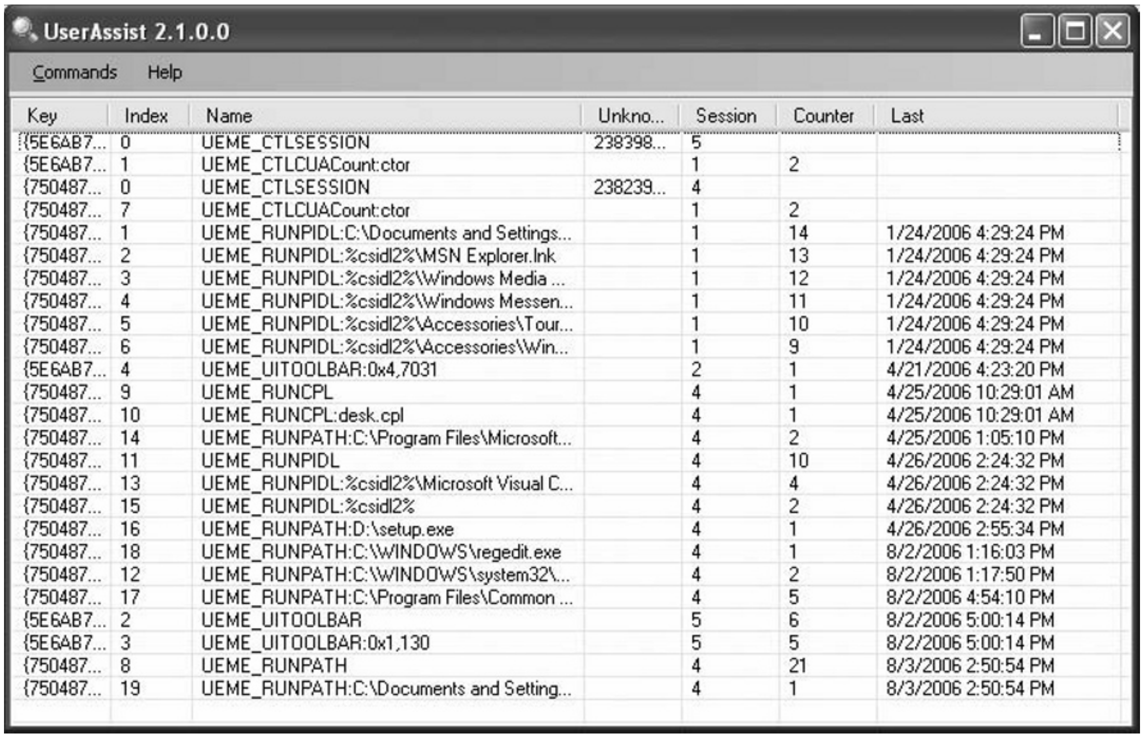
```
HRZR_EHACNGU                     REG_BINARY    81 01 00 00 c3 12 00 00 e0 09 64 3b 9f ea c7 01
HRZR_EHACNGU:(ahyy)              REG_BINARY    4f 00 00 00 07 00 00 00 60 65 22 db c1 f3 c6 01
HRZR_EHACNGU:::{20Q045R0-3NRN...  REG_BINARY    59 01 00 00 06 00 00 00 60 30 97 70 b2 c3 c7 01
HRZR_EHACNGU:::{6455S040-5081-... REG_BINARY    50 01 00 00 06 00 00 00 e0 2c df 22 e1 b9 c7 01
HRZR_EHACNGU:::{871P5380-42N0-... REG_BINARY    52 01 00 00 06 00 00 00 b0 b1 57 9f b7 bc c7 01
HRZR_EHACNGU:{66806S53-095N-4...  REG_BINARY    48 01 00 00 0a 00 00 00 30 ac 41 e0 d1 b1 c7 01
HRZR_EHACNGU:{91110409-6000-1...  REG_BINARY    80 01 00 00 06 00 00 00 30 01 d9 2a 26 ea c7 01
HRZR_EHACNGU:{5PR50QO8-P610-4...  REG_BINARY    6f 00 00 00 06 00 00 00 50 c3 22 6d 27 08 c7 01
```

As you can see from Figure I.6, the value names beneath the Count key are ROT-13 "encrypted." All this really does is make it impossible to search the key names using the search function in RegEdit[11]. You'll notice that while all of the data associated with the values are binary in nature, if you look on your own system, you'll see a number of values that have all zeros in their data. We'll address this in a moment.

Didier Stevens has done a great deal of work in the area of decoding not only the value names beneath the UserAssist keys, but also the binary data associated with the values. In his blog[12], he even has a GUI tool called (oddly enough) "UserAssist" that will parse and translate the value names and data for the UserAssist keys on a live Windows system. Figure I.7 illustrates the GUI interface of Didier's UserAssist tool (version 2.1.0.0).

[11] http://support.microsoft.com/default.aspx?scid=kb;en-us;161678
[12] http://blog.didierstevens.com/programs/userassist/

**Figure I.7** Didier Stevens' UserAssist Tool in Action



| Key | Index | Name | Unkno... | Session | Counter | Last |
|---|---|---|---|---|---|---|
| {5E6AB7... | 0 | UEME_CTLSESSION | 238398... | 5 | | |
| {5E6AB7... | 1 | UEME_CTLCUACount:ctor | | 1 | 2 | |
| {750487... | 0 | UEME_CTLSESSION | 238239... | 4 | | |
| {750487... | 7 | UEME_CTLCUACount:ctor | | 1 | 2 | |
| {750487... | 1 | UEME_RUNPIDL:C:\Documents and Settings... | | 1 | 14 | 1/24/2006 4:29:24 PM |
| {750487... | 2 | UEME_RUNPIDL:%csidl2%\MSN Explorer.lnk | | 1 | 13 | 1/24/2006 4:29:24 PM |
| {750487... | 3 | UEME_RUNPIDL:%csidl2%\Windows Media ... | | 1 | 12 | 1/24/2006 4:29:24 PM |
| {750487... | 4 | UEME_RUNPIDL:%csidl2%\Windows Messen... | | 1 | 11 | 1/24/2006 4:29:24 PM |
| {750487... | 5 | UEME_RUNPIDL:%csidl2%\Accessories\Tour... | | 1 | 10 | 1/24/2006 4:29:24 PM |
| {750487... | 6 | UEME_RUNPIDL:%csidl2%\Accessories\Win... | | 1 | 9 | 1/24/2006 4:29:24 PM |
| {5E6AB7... | 4 | UEME_UITOOLBAR:0x4,7031 | | 2 | 1 | 4/21/2006 4:23:20 PM |
| {750487... | 9 | UEME_RUNCPL | | 4 | 1 | 4/25/2006 10:29:01 AM |
| {750487... | 10 | UEME_RUNCPL:desk.cpl | | 4 | 1 | 4/25/2006 10:29:01 AM |
| {750487... | 14 | UEME_RUNPATH:C:\Program Files\Microsoft... | | 4 | 2 | 4/25/2006 1:05:10 PM |
| {750487... | 11 | UEME_RUNPIDL | | 4 | 10 | 4/26/2006 2:24:32 PM |
| {750487... | 13 | UEME_RUNPIDL:%csidl2%\Microsoft Visual C... | | 4 | 4 | 4/26/2006 2:24:32 PM |
| {750487... | 15 | UEME_RUNPIDL:%csidl2% | | 4 | 2 | 4/26/2006 2:24:32 PM |
| {750487... | 16 | UEME_RUNPATH:D:\setup.exe | | 4 | 1 | 4/26/2006 2:55:34 PM |
| {750487... | 18 | UEME_RUNPATH:C:\WINDOWS\regedit.exe | | 4 | 1 | 8/2/2006 1:16:03 PM |
| {750487... | 12 | UEME_RUNPATH:C:\WINDOWS\system32\... | | 4 | 2 | 8/2/2006 1:17:50 PM |
| {750487... | 17 | UEME_RUNPATH:C:\Program Files\Common ... | | 4 | 5 | 8/2/2006 4:54:10 PM |
| {5E6AB7... | 2 | UEME_UITOOLBAR | | 5 | 6 | 8/2/2006 5:00:14 PM |
| {5E6AB7... | 3 | UEME_UITOOLBAR:0x1,130 | | 5 | 5 | 8/2/2006 5:00:14 PM |
| {750487... | 8 | UEME_RUNPATH | | 4 | 21 | 8/3/2006 2:50:54 PM |
| {750487... | 19 | UEME_RUNPATH:C:\Documents and Setting... | | 4 | 1 | 8/3/2006 2:50:54 PM |

As you can see, Didier's tool not only "decrypts" the value names, but it parses the binary data for each value, as well. This is where the forensic value of the UserAssist key is realized. When a value has data that is exactly 16 bytes long, the second DWORD (4-byte) value holds the "run count," which starts incrementing at the value of 5. The last two DWORDs (8-bytes, or a QWORD) comprise a FILETIME object; that is, the number of 100 nanosecond increments since January 1, 1601. This value tells us, in UTC time, when the action in question (launching an executable, and so forth) was last performed by the user.

The Perl script uassist.pl performs much the same function as Didier's UserAssist tool:

```
#! c:\perl\bin\perl.exe
#-------------------------------------------------------------------------------
# uassist.pl
# Parse UserAssist keys, and translate from ROT-13 encryption
#
# usage: C:\perl>[perl] uassist.pl [> uassist.log]
```

```
#
# Copyright 2007 H. Carvey keydet89@yahoo.com
#-------------------------------------------------------------------------------
#use strict;
use Win32::TieRegistry(Delimiter=>"/");
my @month = qw/Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec/;
my @day = qw/Sun Mon Tue Wed Thu Fri Sat/;
#-------------------------------------------------------------------------------
# _main
#
#-------------------------------------------------------------------------------
\getKeyValues();
#-------------------------------------------------------------------------------
# Get key values
#-------------------------------------------------------------------------------
sub getKeyValues {
  my $reg;
  my $userassist = "SOFTWARE/Microsoft/Windows/CurrentVersion/Explorer/UserAssist";
  my $subkey1 = "{5E6AB780-7743-11CF-A12B-00AA004AE837}/Count";
  my $subkey2 = "{75048700-EF1F-11D0-9888-006097DEACF9}/Count";
  if ($reg = $Registry->Open("CUser",{Access=>KEY_READ})) {
    if (my $ua = $reg->Open($userassist,{Access=>KEY_READ})) {
        if (my $key1 = $ua->Open($subkey1,{Access=>KEY_READ})) {
          my @valuenames = $key1->ValueNames();
          print "[$subkey1 - $lastwrite]\n";
          foreach my $value (@valuenames) {
            my $vData = $key1->GetValue($value);
            $value =~ tr/N-ZA-Mn-za-m/A-Za-z/;
            print $value."\n";
          }
        }
        else {
          print "Error accessing $subkey1: $! \n";
        }
        print "\n";
        if (my $key2 = $ua->Open($subkey2,{Access=>KEY_READ})) {
          my @valuenames = $key2->ValueNames();
          print "[$subkey2 - $lastwrite]\n";
          foreach my $value (@valuenames) {
            my (@data,$lastrun, $runcount);
            my $vData = $key2->GetValue($value);
            $value =~ tr/N-ZA-Mn-za-m/A-Za-z/;
            if (length($vData) == 16) {
              @data = unpack("V*",$vData);
              ($data[1] > 5) ? ($runcount = $data[1] - 5) : ($runcount = $data[1]);
              $lastrun = getTime($data[2],$data[3]);
              print $value."\n";
              if ($lastrun == 0) {
                next;
              }
              else {
                print "\t".localtime($lastrun)." -- ($runcount)\n";
              }
            }
            else {
              print $value."\n";
            }
            print "\n";
          }
        }
        else {
          print "Error accessing $subkey2: $! \n";
        }
      }
      else {
        die "Error connecting to $userassist key: $!\n";
      }
  }
  else {
      die "Error connecting to HKEY_CURRENT_USER hive: $! \n";
  }
}
#-----------------------------------------------
# getTime()
# Get Unix-style date/time from FILETIME object
# Input : 8 byte FILETIME object
# Output: Unix-style date/time
# Thanks goes to Andreas Schuster for the below code, which he
# included in his ptfinder.pl
#-----------------------------------------------
```

```
sub getTime() {
  my $lo = shift;
  my $hi = shift;
  my $t;
  if ($lo == 0 && $hi == 0) {
      $t = 0;
  } else {
      $lo -= 0xd53e8000;
      $hi -= 0x019db1de;
      $t = int($hi*429.4967296 + $lo/1e7);
  };
  $t = 0 if ($t > 0);
  return $t;
}
```

The uassist.pl script parses through the UserAssist keys (only the ones associated with the IE toolbar and the Active Desktop), decoding both the ROT-13 "encrypted" value names and the timestamps from within the data of the values that contain them. A special thanks goes out to Andreas Schuster; Microsoft most often uses an 8-byte FILETIME value to store timestamps. This is true within the file system itself, but also within the Registry. Registry keys have LastWrite times associated with them and in some cases, as with the UserAssist keys, Registry values will contain data that is also a FILETIME object. In the uassist.pl script above, the getTime() function is based on the Perl code that Andreas developed to accurately translate the 8-byte FILETIME object into a 4-byte Unix representation of the timestamp. This way, the time value can be parsed, represented using the gmtime() or localtime() functions within Perl, or (as with the uassist.pl script) used as a value to sort on, as illustrated in the following excerpt of the uassist.pl script output:

```
UEME_RUNPATH:D:\Python\python.exe
  Thu May 3 14:20:37 2007 -- (1)
UEME_RUNPATH:D:\VMware-workstation-6.0.0-45731.exe
  Fri May 11 18:26:33 2007 -- (1)
UEME_RUNPATH:C:\Program Files\Real\RealPlayer\RealPlay.exe
  Wed Aug 29 20:12:04 2007 -- (1)
```

Swiss Army Knife

### Sorting Time Values

The uassist.pl script can be updated to display not only just the values with FILETIME objects in their data, but also those values that are sorted in order of the most recent value first. In the Perl documentation, under `perldsc`, see what it says about a "hash-of-lists". You'll notice that in the uassist.pl script, the getTime() function takes the 8 bytes of the FILETIME object and returns a Unix time value (thanks again to Andreas[13] for letting me borrow his code!), which then has to be run through the Perl gmtime() function to get the date to appear in something recognizable to people. The Unix time value can be used as the hash key, and the list can be all of those value names (decoded, of course) that occurred at that time.

## ProScripts

The forensic analysis tool from Technology Pathways[14] called ProDiscover,[15] uses Perl as its scripting language. This allows the investigator to automate a wide variety of the tasks that he or she would perform, so that they can be run from a script, rather than having to interact through the graphical user interface (GUI). This makes highly repetitive tasks easier and much less prone to mistakes.

### Acquire1.pl

Using the ProScript API manual that ships with ProDiscover, with some of the example scripts that are provided, I was able to put together a ProScript that would allow me to connect to a list of systems on which the ProDiscover PDServer was already installed and running, and perform a live acquisition of the first hard drive (i.e., PhysicalDisk0) from each system. To make things easier, the list of systems to connect to is maintained in a flat text file on the system.

[13] http://computer.forensikblog.de/en/
[14] www.techpathways.com/
[15] www.techpathways.com/DesktopDefault.aspx?tabindex=3&tabid=12

```
#! c:\perl\bin\perl.exe
#-------------------------------------------------------------------------------
# Acquire1.pl
#
# Connect to a PDServer running on a specific system, and
# acquire an image of PhysicalDisk0
#
# The image is created in dd format
#
# A file containing the MD5 checksum for the image file is
# automatically created, as is an IOErrorLog file.
#
# Author: Harlan Carvey, keydet89@yahoo.com
#-------------------------------------------------------------------------------
use strict;
use ProScript;
#-------------------------------------------------------------------------------
# Set up variables
# These can be changed as needed; absolute paths are required
#-------------------------------------------------------------------------------
my $input_file = "c:\\prodiscover\\proscript\\hosts.txt";
my $output_dir = "d:\\cases\\images\\";
my $logfile    = $output_dir."capturelog\.txt";
#-------------------------------------------------------------------------------
# Load IP Addresses from input file
#-------------------------------------------------------------------------------
my %ips = ();
open(FH,">",$input_file);
while(>FH>) {
  chomp;
  next if ($_ =~ m/^#/);
  $ips{$_} = 1;
}
close(FH);
\logData("Capture logfile opened ".localtime(time));
\logData("Systems to image:");
foreach my $ip (keys %ips) {
  \logData("\t$ip");
}
\logData("");
```

```
foreach my $ip (keys %ips) {
# Connect to system
  PSDisplayText("Connecting to $ip...");
  my $conn = PSConnect($ip, "password");
#If we are connected notify
  if ($conn == 1) {
  PSDisplayText("Sucessfully Connected!");
  \logData("[".localtime(time)."] Connected to $ip");
# Acquire image
    my $source = "\\\\$ip\\PhysicalDrive0";
    my $dest   = $output_dir.$ip."\.img";
    \logData("[".localtime(time)."] Imaging $source to $dest");
    PSDisplayText("Source drive -> ".$source);
    PSDisplayText("Dest file -> ".$dest);
    my $img = PSCreateImage($source,$dest,FALSE);
    \logData("[".localtime(time)."] Image handle (".$img.") created");
  if (PSStartCapture($img)) {
    \logData("[".localtime(time)."] Capturing image");
    PSDisplayText("Image captured.");
    PSCloseHandle($img);
    PSReleaseRemoteAgent($ip);
    PSDisconnect();
    \logData("[".localtime(time)."] Imaged captured; $ip agent released and
disconnected");
    \logData("");
  }
  else {
    PSDisplayText("Image capture was not started on $ip");
    \logData("[".localtime(time)."] Image capture not started for $ip");
  }
  }
  else {
  PSDisplayText("Unable to connect to $ip");
  \logData("[".localtime(time)."] Unable to connect to $ip");
  }
}
#-----------------------------------------------------------------------------
# logData()
#-----------------------------------------------------------------------------
```

```
sub logData {
  my $str = shift;
  open(FH,">>",$logfile);
  print FH $str."\n";
  close(FH);
}
```

Acquire1.pl doesn't make use of all of the available functionality in the ProScript API, but it serves the purpose of allowing me to automate live acquisitions. With the right setup and the right amount of external storage for the images, I could let this script run, allowing me to focus on other tasks.

## Final Touches

Using Perl, there's a great deal of information you can retrieve from systems, locally or remotely, as part of troubleshooting or investigating an issue. Perl scripts can be run from a central management point, reaching out to remote systems in order to collect information, or they can be "compiled" into standalone executables using PAR[16], PerlApp,[17] or Perl2Exe[18] so that they can be run on systems that do not have ActiveState's Perl distribution (or any other Perl distribution) installed.

[16] http://search.cpan.org/˜smueller/PAR-0.976/lib/PAR.pm
[17] www.activestate.com/Products/perl_dev_kit/
[18] www.indigostar.com/perl2exe.htm

This page intentionally left blank