GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials (Security 401)"
at http://www.giac.org/registration/gsec

Programmed Vigilance

Monitoring your Windows NT/2000 network with Perl and command-line utilities

Alex Yu

Assignment Version 1.4b

© SANS Institute 2003, Author retains full rights.

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Abstract

Monitoring your network environment is critical to its security and stability. Security holes are constantly discovered, logs can overwhelm reviewers, users and administrators make mistakes, and viruses and crackers purposely try to penetrate your defenses. Fortunately, there are many products available for auditing a Windows NT/2000 network. On the other hand, it takes a lot of time and discipline to run these tools on a regular basis, and graphically oriented utilities usually require user interaction. Worse, tight purse strings and layoffs reflect our current economic condition. How is a security professional supposed to stay on top of things despite limited resources?

The goal of this paper is to address basic security principles, demonstrate how to use easily obtainable command line utilities, and orchestrate their use with the Perl language to effectively watch your network. With the sample scripts provided herein, it becomes easy to begin automating the scanning of Windows NT/2000 machines, look for signs of trouble, and alert personnel.

Perl

Larry Wall introduced us to Perl in 1987. Officially known as the Practical Extraction and Report Language, Perl has, among other things, very robust text manipulation functions. Programmers will find that Perl has much in common with C, BASIC, and awk. Your time investment in learning Perl would be well spent; it has been ported to many operating systems such as Mac OS, Linux, Unix, VMS, and Windows, just to name a few.

The examples in this paper focus on using Perl 5.8.0 from ActiveState on Windows NT/2000 machines. The debugger in the Perl Development Kit (PDK) will help you trace through your code and allow you to see the values of your program variables one instruction at a time. Perl and the Perl Development Kit may be freely downloaded from ActiveState at http://www.activestate.com. If you are using a different distribution or operating system, you will still benefit by learning how to apply security principles on your OS by using the commands and utilities that it supports.

Before we continue, allow me to state that there are many ways of writing the scripts we are about to see to perform the desired results. In fact, the motto of Perl is "There's More Than One Way To Do It," or TMTOWTDI. (Kirrily). Every effort has been made to present the code in a straightforward manner to demonstrate the language to a new Perl programmer, but it would be helpful to have some prior programming experience. The purpose of this paper is not to teach Perl, so I strongly recommend the reader to peruse Programming Perl, by Larry Wall, Tom Christiansen, and Jon Orwant. Also known as the Camel book, this work is widely accepted as the definitive book on Perl, and is referenced often as we encounter new Perl concepts.

This paper will address a number of security topics. The scripts provided are intended to show how running scripts may make it easier to promptly address security issues. If you wish to try these scripts, please seek permission from the authorities of your network. Each program is heavily commented to show the logic and lead you through the process. While the programs have been designed

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

for pedagogical purposes, running them may divulge potentially sensitive information. Furthermore, as important as error checking is, adding such code would distract us from the concepts at hand and inflates the already abundant lines of code. At the end of each section, think of ways add enhancements and implement error checking.

These scripts functioned correctly on the test machines on which they were run, and did not seem to produce adverse effects; however, the author cannot accept responsibility for damages that may occur. Let this be a reminder to always read and understand the code before blindly executing programs. How do you know I'm not trying to compromise your systems? Remember, according to Law #1 of Microsoft's Ten Immutable Laws of Security, "If a bad guy can persuade you to run a program on your computer, it's not your computer anymore." (Microsoft) With that said, let us proceed onward. Let us begin to explore Perl by using a simple example – network connectivity.

Connectivity

Network servers that provide a useful function need to stay up and running. An administrator needs to know when connectivity to a host is lost. One would normally type the following at a command prompt to send an ICMP Echo Request to see if a host is reachable on a TCP/IP network:

```
C:\>ping 192.168.0.1
```

If the host in question is alive, and there are no intervening devices that block ICMP traffic, you may receive the following output:

```
Pinging 192.168.0.1 with 32 bytes of data:

Reply from 192.168.0.1: bytes=32 time<10ms TTL=127
Reply from 192.168.0.1: bytes=32 time<10ms TTL=127
Reply from 192.168.0.1: bytes=32 time<10ms TTL=127
Reply from 192.168.0.1: bytes=32 time<10ms TTL=127

Ping statistics for 192.168.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum =  0ms, Average =  0ms
```

Executing this command from Perl is just as easy. Start a text editor such as Notepad and type the single line below. Make sure the command is enclosed with backticks (`), also known as command input operators (Wall, p. 80). Comments appear after the hash mark (#) and are presented to clarify the code; since the Perl interpreter ignores these comments, it is not necessary to type them in (Wall, p. 49)

```
`ping 192.168.0.1`;           # [1] execute shell command
```

Save this file as ping.pl. (Make sure the file is saved with a .pl extension, not .txt, which is the default in Notepad.) Then return to the command prompt and run it by typing:

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
C:\>perl ping.pl
```

This simple script will attempt to send ICMP packets to the device at 192.168.0.1. When you run this program, you experience the usual four-second delay it takes to complete the command, but do not see any results because the Perl script still needs to capture the output. Modify the script like so:

```
@results = `ping 192.168.0.1`;        # [1] store in array
foreach $r (@results) {               # [2] loop through array
    print $r;                         # [3] print each element
}                                     # [4] end of loop
```

The first line, denoted by [1], stores each line of the output into an array called @results (Wall, p.51-52). [2] assigns each output line to the scalar variable $r (Wall, p.51-52), [3] prints $r to the screen, and this loop repeats (Wall, p. 33) until each output line is printed. You may wish to record the results for future reference by writing the data to a log file. Take note of the additional bolded lines in the following code:

```
open (LOG, ">>file.txt");             # [1] open file.txt for append
@results = `ping 192.168.0.1`;        # [2] store ping results in array
foreach $r (@results) {               # [3] loop through array elements
       print $r;                      # [4] print each element
       print LOG $r;                  # [5] write to file.txt
}                                     # [6] end of loop
close LOG;                            # [7] close file.txt
```

[1] prepares data to be appended to a file called file.txt (Wall, p.21). The file is referenced by its file handle, LOG. [5] is responsible for writing each value of $r to file.txt. [7] saves the data by closing the file.

Let us now be more selective in displaying and recording the data. It is not necessary to show or record each reply or time-out line. The important data lies in the line that contains “Packets:” This is where we use Perl’s regular expressions, or regexes, to help us achieve this end (Wall, p. 139). The bolded lines below show the changes to pings.pl.

```
open (LOG, ">>file.txt");             # [1] open file.txt for append
@results = `ping 192.168.0.1`;        # [2] store ping results in array
foreach $r (@results) {               # [3] loop through array elements
    if ($r =~ /Packets:/) {           # [4] check if element contains “Packets:”
        print $r;                     # [5] print each element
        print LOG $r;                 # [6] write to file.txt
    }                                 # [7] end of if block
}                                     # [8] end of foreach loop
close LOG;                            # [9] close file.txt
```

[4] uses the =~ operator to determine if the regex between the pattern match operator (/ /) appears anywhere in $r (Wall, p.144). If so, $r is printed to the screen in [5], and to the log file in [6].

Next, it would be useful to extract the data in the statistics line, and use it within your program. Modify the regular expression as shown below.

```
open (LOG, ">>file.txt");             # [1] open file.txt for append
@results = `ping 192.168.0.1`;        # [2] store ping results in array
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
foreach $r (@results) {           # [3] loop through array elements
    if ($r =~ /Packets: Sent = (\d+), Received = (\d+), Lost = (\d+)/) {
                                  # [4] capture statistics
        $sent = $1;               # [5] save 1st (\d+) in $sent
        $recd = $2;               # [6] save 2nd (\d+) in $recd
        $lost = $3;               # [7] save 3rd (\d+) in $lost
        print $r;                 # [8] print each element
        print LOG $r;             # [9] write to file.txt
    }                             # [10] end of if block
}                                 # [11] end of foreach loop
close LOG;                        # [12] close file.txt
print "You sent $sent packets, received $recd back, and lost $lost\n";
                                  # [13] show statistics
```

In [4], we are matching the string to include sent, received, and lost packet information. Notice the three occurrences of (\d+). This can be deciphered as follows: \d represents a digit (Wall, p.167); the + quantifier indicates one or more of each digit (Wall, p. 176); the parentheses ( ) store the enclosed value into special variables called backreferences, starting with $1, $2, etc. (Wall, p. 182-184). In summary, this statement compares the line read to the expression within the pattern match operators, and place the numbers immediately following the "Sent = ", "Received = ", and "Lost = " strings into the backreferences $1, $2, and $3, respectively. These values are stored in scalar variables in [5]-[7]. [13] displays these values to the screen. (Note: as scripts become more complicated, they become harder to trace. You might want to install ActiveState's Perl Development Kit at this point. Once installed, the Perl Debugger may be invoked with the –d switch: `C:\>perl -d ping.pl`)

So far, we have a working program that pings an IP address, then checks the output for sent, received, and lost packets. Wouldn't it be a good idea to continually ping a list of devices and alert an administrator if any device went down?

```
use Net::SMTP;                                  # [1] use SMTP library
use Win32::Sound;                               # [2] use Sound library

$num = 0;                                       # [3] initialize server count
open (IN, "<servers.txt") || die "Can't open file";
                                                # [4] read list of server IPs
while ($line = <IN>) {                          # [5] read line by line
    chomp $line;                                # [6] remove CRLF
    $server[$num++] = $line;                    # [7] store server IP in variable
}                                               # [8]

while (1) {                                     # [9] loop until program stops
    open (LOG, ">>file.txt");                   # [10] open file.txt for append
    for ($i=0; $i<$num; $i++) {                 # [11] loop through IP list
        @results = `ping -n 1 $server[$i]`;     # [12] make one request per server
        foreach $r (@results) {                 # [13] loop through array elements
          if ($r =~ /Packets: Sent = (\d+), Received = (\d+), Lost = (\d+)/){
                                                # [14] get statistics
            $sent = $1;                         # [15] save 1st (\d+) in $sent
            $recd = $2;                         # [16] save 2nd (\d+) in $recd
            $lost = $3;                         # [17] save 3rd (\d+) in $lost
            print $r;                           # [18] print each element
            print LOG $r;                       # [19] write to file.txt
          }                                     # [20] end of if block
        }                                       # [21] end of foreach loop
        print "You sent $sent packets, received $recd back, and lost $lost\n";
                                                # [22] show statistics
        close LOG;                              # [23] close file.txt

        if ($recd == 0) {                       # [24] if no replies
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
            `net send Janet "Lost connectivity to $server[$i]"`;
                                                # [25] send popup
            email();                            # [26] email message
            Win32::Sound::Play("warning.wav");  # [27] play a sound file
            print LOG "No reply from $server[$i] \n";
                                                # [28] record in log file
        }                                       # [29] end if block
    }                                           # [30] for loop
    close LOG;                                  # [31] close file.txt
    sleep(120);                                 # [32] pause for 2 minutes
}                                               # [33] end while loop

sub email                                       # [34] define subroutine
{                                               # [35]
    $smtp = Net::SMTP->new('smtp.company.com'); # [36] define SMTP server
    $smtp->mail("ping.pl");                     # [37] sender name
    $smtp->to('janet@company.com');             # [38] send to Janet
    $smtp->data();                              # [39] start data section
    $smtp->datasend("To: Janet\n");             # [40] send To line
    $smtp->datasend("From: Ping.pl\n");         # [41] send From line
    $smtp->datasend("Subject: Connectivity problem for $server[$i]\n");
                                                # [42] send subject
    $smtp->datasend("\n");                      # [43] send CRLF
    $smtp->datasend("Cannot ping $server[$i]\n");# [44] send body
    $smtp->dataend();                           # [45] done with data send
    $smtp->quit;                                # [46] close connection
}                                               # [47] end subroutine
```

This code shows the use of two modules (Wall, p. 299), Net::SMTP by Graham Barr in [1] and Win32::Sound by Aldo Calpini in [2]. Net::SMTP enables a Perl script to send e-mail messages via the Simple Mail Transport Protocol; Win32::Sound will let the script play sounds in Win32 platforms. Make sure these lines exist; otherwise, Perl will generate errors since it will not know how to run the associated methods. Natively, Perl does not support SMTP and audio playback. Instead, these functionalities depend upon modules contributed by Perl programmers from all over the world. They may already exist in your Perl distribution, but if they are not, they may be downloaded from CPAN, the Comprehensive Perl Archive Network at http://www.cpan.org, or from various mirror sites (Wall, pp.xxi-xxii). Alternatively, one may easily obtain and install a module through the Programmer's Package Manager (ppm) utility, as we will see in a later section. [3]-[8] reads a text file called servers.txt, which contains the list of IP addresses we wish to monitor for connectivity. Such a file may simply consist of the following text:

```
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
192.168.0.5
192.168.0.6
192.168.0.7
192.168.0.8
192.168.0.9
```

[9] is the start of a while loop (Wall, p.32), which executes for the remainder of this program run. [11] loops through each IP address (Wall, p.33) and makes one echo request to each server in [12]. [24] checks to see if no replies were returned; if this is the case, the administrator is alerted of this situation by e-mail [26],[34]-[47], popup message [25], audio clip [27], and log

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

entry [28]. The subroutine in [34]-[47] uses functions defined in the Net::SMTP library referenced in [1]. The commands within this subroutine were taken from ActiveState's on-line documentation and modified for our purposes. In [32], the sleep command (Wall, p.788) causes the process to wait for 120 seconds before the loop starts over.

With some basic tools, we have created a simple monitoring and alerting system. We produced a meaningful program to read a list of IP addresses and alert the administrator when any of the IP addresses fail to respond to an echo request. Try to make improvements to this script, such as customized audio warnings, pager support, and reducing the warning frequency after detecting an unreachable host. The options are unlimited and exciting, but to avoid wandering off-topic, let us now move on to monitoring user accounts.

User Account Status

If you do not have a policy to govern your user accounts, it would be wise to start the process. The SANS Institute has compiled policy templates covering a variety of topics. To assist in increasing Internet security and its awareness, these resources are freely shared at http://www.sans.org/resources/policies.

For starters, it is a good idea to lock out a user account after so many logon attempts, to enforce passwords of a certain length, and to require a password change every predetermined number of days. The script below can answer questions about the user accounts: Is anyone using the account? Might someone have tried to break into that account? How old is its password?

Locked accounts may indicate password-cracking attempts; accounts that don't expire are more attractive to crackers than those that do; when passwords are not required, there is a chance that the password is blank; and accounts that have never been logged on to might have a simple default password.

Much information can be obtained from the command prompt:

```
C:\>net user janet /domain
User name                    Janet
Full Name
Comment                      IT
User's comment
Country code                 000 (System Default)
Account active               Yes
Account expires              Never

Password last set            12/13/2002 7:36 PM
Password expires             Never
Password changeable          12/13/2002 7:36 PM
Password required            Yes
User may change password     Yes

Workstations allowed         All
Logon script
User profile
Home directory
Last logon                   1/29/2003 3:45 PM

Logon hours allowed          All

Local Group Memberships      *Administrators
Global Group memberships     *Domain Admins
The command completed successfully.
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Occasionally executing the net command on a few usernames to give you immediate account status is not a difficult task. However, to do this for every member of your organization on a regular basis may leave an administrator with little time to do much else. Such a task would demand an automated solution.

Before scripting, consider how to manually accomplish the task. One may first enumerate the users on the domain, then run the net command on each account on the list, read the relevant output information, alert the administrator if necessary, and move on to the next account. Now that we have a plan, we need to know how to obtain the information, and in what format it is presented. The following command will generate the list of users in a domain:

```
C:\>net user /domain

User accounts for \\SERVER

-------------------------------------------------------------------------------
Guest                  IUSR_SERVER01       Janet
User1                  User2               User3
User4                  User5               User6
IWAM_SERVER01          TsInternetUser
The command completed successfully.
```

How would a script be written to extract the usernames? By examining the output, one could make the following observations: The list of user accounts is presented after the series of dashes (-). This list appears in three columns and ends with the line, “The command completed successfully.” In each line, the first column is always populated; the second and third columns in the line just before the success message do not necessarily have a username. This information is sufficient for us to start scripting.

```
while (1) {                                     # [1] loop until terminated
    @results = `net user /domain`;              # [2] run net user
    $flag=0;                                    # [3] clear attn flag
    foreach $r (@results) {                     # [4] cycle through output
        if ($r=~/----------/) { $flag=1; }      # [5] pay attn after dashes
        if ($r=~/The command completed successfully./) { $flag=0; }
                                                # [6] til done
        if ($flag==1) {                         # [7] if flag is raised,
            $r =~ /(\S+)\s+(\S*)\s+(\S*)/;      # [8] get usernames
            $col1 = $1;                         # [9]   and store
            $col2 = $2;                         # [10]  them into
            $col3 = $3;                         # [11]  variables
            push @users, $col1;                 # [12] push 1st into array
            if ($col2 ne "") { push @users, $col2; }
                                                # [13]  push 2nd and 3rd if
            if ($col3 ne "") { push @users, $col3; }
                                                # [14]  not blank
        }                                       # [15] end if block
    }                                           # [16] end foreach loop
    foreach $u (@users) {                       # [17] cycle through users
        print "$u\n";                           # [18] print username
    }                                           # [19] end foreach loop
    sleep(60*60*24);                            # [20] wait for 24 hours
}                                               # [21] end of while loop
```

We begin this code by executing the net command in [2] to generate a list of users. In [3] a flag variable is set to 0. The reason for doing so is to signal when it is time to gather the usernames. Since the usernames aren’t listed until

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

the dashes are encountered, the script does not need to look for them until that point. Similarly, the processing of usernames should stop when we encounter the line, “The command completed successfully.” We set up the script to loop through each line $r of output [4] until the dashes are reached and set the flag [5] to start paying attention to usernames.

Once the flag is set [7], $r is compared to a string of nonwhitespace characters (indicated by \S) followed by two sets of alternating whitespace (indicated by \s) and optional nonwhitespace characters (Wall, p.167). Whitespace refers to spaces, tabs, line feeds, carriage returns, etc. The first string of nonwhitespace is captured into the special $1 variable. After one or more whitespaces, there may or may not be more nonwhitespace characters; if there are, store it in $2. The \S* indicates there may be 0 or more non-whitespace characters (Wall, p. 176), so if there are not, the null string is stored in $2. Similarly, the same idea applies to $3 [8].

The backreferences are then stored into temporary variables, $col1, $col2, and $col3 [9]-[11]. Since we observed the first column should always contain some text, $col1 is pushed (Wall, p. 268) in the into the @users array [12]. $col2 and $col3, if not null, are similarly added to the same array [13]-[14].

The last lines of code [17]-[19] cycle through the usernames and print them on the screen. Let us now focus on this section to generate more useful output:

```
while (1) {                                     # [1] loop until termin'd
    @results = `net user /domain`;              # [2] get user list
    $flag=0;                                    # [3] clear attn flag
    foreach $r (@results) {                     # [4] cycle through output
        if ($r=~/----------/) { $flag=1; }      # [5] pay attn after dashes
        if ($r=~/The command completed successfully./) { $flag=0; }
                                                # [6] until done
        if ($flag==1) {                         # [7] if flag is raised,
            $r =~ /(\S+)\s+(\S*)\s+(\S*)/;      # [8] get usernames
            $col1 = $1;                         # [9]  and store them in
            $col2 = $2;                         # [10]  variables
            $col3 = $3;                         # [11]
            push @users, $col1;                 # [12]  push 1st into array
            if ($col2 ne "") { push @users, $col2; }
                                                # [13]  push 2nd and 3rd if
            if ($col3 ne "") { push @users, $col3; }
                                                # [14]  not blank
        }                                       # [15] end if block
    }                                           # [16] end foreach loop

    open (OUT, ">alert.txt");                   # [17] create an alert file
    foreach $u (@users) {                       # [18] cycle through each user
        @info = `net user $u /domain`;          # [19] get info on each user
        foreach $i (@info) {                    # [20] cycle through info
            if ($i =~ /Account active\s+Locked/) {# [21] Look for Locked acct
                print OUT "Locked account: $u\n"; # [22] Record in file
            }                                   # [23]
            if ($i =~ /Account expires\s+Never/) {# [24] Look for never-expire
                print OUT "Never-expiring account: $u\n";
                                                # [25] Record in file
            }                                   # [26]
            if ($i =~ /Password required\s+No/) { # [27] Look for no pswd req'd
                print OUT "No password required: $u\n";
                                                # [28] Record in file
            }                                   # [29]
            if ($i =~ /Last logon\s+Never/) {   # [30] Look for never logged on
                print OUT "Maybe never logged on: $u\n";
                                                # [31] Record in file
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
            }                                       # [32]
        }                                           # [33] end of inner foreach loop
    }                                               # [34] end of outer foreach loop
    close OUT;                                      # [35] save information
    `blat alert.txt -t janet@company.com`;          # [36] e-mail file to admin
    sleep(60*60*24);                                # [37] do again tomorrow
}                                                   # [38] end while loop
```

Line 17 opens a file to log the problems we encounter. Cycling through the list of users [18], we execute the net user command and gather the results in the @info array [19].

Next, we wish to watch for potential security concerns in [21], [24], [27], and [30]. If there is a match, the line is written to the alert file and may be e-mailed to an administrator [36]. The process repeats every 24 hours [37].

This time the e-mail mechanism is through Blat instead of Net::SMTP. Blat is a convenient command line utility from Mark Neal, Pedro Mendes, Gilles Vollant, and Tim Charron. With this utility, files are easily sent as e-mails from the command line. It is available for download from http://www.interlog.com/~tcharron/blat.html (Charron).

Group Membership

Along the same lines as checking for account status, it would be prudent to periodically check the members of groups with elevated privileges, such as Domain Administrators. Note that this command may only be run on Windows Domain Controllers.

```
C:\net group "Domain Admins"
Group name     Domain Admins
Comment        Designated administrators of the domain

Members

-------------------------------------------------------------------------------
Janet                    Bad_Guy                  William
The command completed successfully.
```

This output is very similar to that of net user, so we may modify parts of the previous script to extract user names:

```
open (IN, "<admins.txt");                           # [1] read admin file
while ($line = <IN>) {                              # [2] read line
    chomp $line;                                    # [3] remove CRLF
    push @admins, $line;                            # [4] store into admin array
}                                                   # [5] end while loop

while (1) {                                         # [6] loop until termination
    @users = ( );                                   # [7] clear users array
    $flag=0;                                        # [8] clear warning
    @results = `net group "Domain Admins"`;         # [9] get group members
    foreach $r (@results) {                         # [10] cycle through output
        if ($r=~/----------/) { $flag=1; }          # [11] pay attn at dashes
        if ($r=~/The command completed successfully./) { $flag=0; }
                                                    # [12] no need to pay attn
        if ($flag==1) {                             # [13] if attn required,
            $r =~ /(\S+)\s+(\S*)\s+(\S*)/;          # [14] look for members
            $col1 = $1;                             # [15] assign into variable
            $col2 = $2;                             # [16] assign into variable
            $col3 = $3;                             # [17] assign into variable
            if ($col1 ne "") { push @users, $col1; }
                                                    # [18] if not null,
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
            if ($col2 ne "") { push @users, $col2; }
                                                    # [19] then store into
            if ($col3 ne "") { push @users, $col3; }
                                                    # [20] users array
        }                                           # [21] end if block
    }                                               # [22] end foreach loop

    open (OUT, ">alert.txt");                       # [23] write alert file
    foreach $u (@users) {                           # [24] cycle through members
        $u =~ tr/A-Z/a-z/;                          # [25] convert to lower case
        $flag = 1;                                  # [26] assume user doesn't belong
        foreach $a (@admins) {                      # [27] cycle through admin list
            $a =~ tr/A-Z/a-z/;                      # [28] convert admins to lower case
            if ($u eq $a) {                         # [29] if user is admin,
                $flag = 0;                          # [30] then all is well
            }                                       # [31] end if block
        }                                           # [32] end foreach loop
        if ($flag == 1) {                           # [33] if flag is still set,
            print OUT "$u is in Domain Admins\n"; # [34] record in log
            `net send Janet "$u is in Domain Admins"`;
                                                    # [35] send popup
        }                                           # [36] end if block
    }                                               # [37] end foreach loop
    close OUT;                                      # [38] save the data
    `blat alert.txt -t janet@company.com`;          # [39] email the log
    sleep(60*60);                                   # [40] wait 1 hour
}                                                   # [41] end while loop
```

This code first reads the admins.txt file for a list of accepted administrators that you provide, in the format shown below [1]-[5]:

```
Janet
William
```

When the main loop starts [6], we capture the group members from the output of the net command [9]-[22]. Next, we cycle through each member [24], convert the name to lower case [25] (Wall, p.144) and compare the name to each name in your admin list [29]. We will assume that a username does not belong in the Domain Admin group until that name is encountered in the list by first setting a flag to indicate a problem [26]. Only when the username matches an admin username [29] is the flag cleared [30]. If the flag is still set [33] after all the comparisons, then log the incident and notify somebody [34]-[35].

In this script, an e-mail log is sent after each hourly cycle. One may enhance the code so that the alert is sent only if there is a security violation.

Services

Servers open up ports to await client requests. One way to make sure your critical services are running is to use the NETSVC utility from the Windows 2000 Resource Kit. This utility is not limited to querying the local system; it can be used to determine the status of network hosts, provided one is connected to the remote host with administrative privileges. The command and its output appear below:

```
C:\>netsvc \\server service /query
Service is running on \\server
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

The script we formulate should issue the command and look for the string "Service is running"

```
while (1) {                                          # [1] loop until terminated
    $dt = localtime;                                 # [2] record date and time
    open (LOG, ">>services.log");                    # [3] append to log
    open (IN, "<services.txt");                      # [4] read service list
    while ($line = <IN>) {                           # [5] read a line
        chomp $line;                                 # [6] remove CRLF
        @part = split (",",$line);                   # [7] split the line at comma
        $server = $part[0];                          # [8] server precedes comma
        $service = $part[1];                         # [9] service is after comma
        @results = `netsvc \\\\$server $service /query `;
                                                     # [10] run command
        $flag=1;                                     # [11] assume service is stopped
        foreach $i (@results) {                      # [12] cycle through output
            if ($i !~ /Service is running/) {        # [13] if service is running,
                $flag=0;                             # [14] all is well
            }                                        # [15] end of if block
        }                                            # [16] end foreach loop
        if ($flag==1) {                              # [17] if all is not well,
            print LOG "$dt: Service $service is not running on $server\n";
                                                     # [18] log the problem
            `net send Janet "$service is not running on $server"`;
                                                     # [19] notify admin
        }                                            # [20] end of if block
    }                                                # [21] end of inner while loop
    close LOG;                                       # [22] save log data
    close IN;                                        # [23] close file
    sleep(10);                                       # [24] wait
}                                                    # [25] end main loop
```

The file services.txt needs to be created so that the script knows which servers and services to monitor. To obtain a list of services running on a server, issue the following command:

```
C:\>netsvc \\server /list
```

Next, create a text file and list each server and service, separated by a comma:

```
server01,spooler
server02,World Wide Web Publishing Service
server03,Simple Mail Transport Protocol (SMTP)
```

We start the loop by reading in each server and service [4]-[9], then executing netsvc [10]. Again, we assume there is a problem [11] until we encounter the message "Service is running." [13]. If there is a problem [17], log the incident [18] and notify [19]. There are a couple of functions worth noting. localtime() returns the date and time [2] (Wall, p. 738-739). split() divides a string into substrings (Wall, p. 794).

Explore the parameters of netsvc. Enhance this script to automatically start a service if it is stopped.

Connecting to Services

There are times when a service is running on a host, but the service port is does not respond to connection attempts, such as during a Denial of Service attacks. One can use Perl to telnet to a specific port and look for a banner to indicate that the service is indeed responding.

© SANS Institute 2003, Author retains full rights.

This would require the use of the Net::Telnet library. If you are unsure Net::Telnet is installed, run this at a command prompt:

```
C:\>perl -e "use Net::Telnet"
```

This command calls on Perl to execute the one-line command to look for the Net::Telnet library. If an error message appears similar to the following, then you may install Net::Telnet through ppm:

```
Can't locate Net/Telnet1.pm in @INC (@INC contains: C:/Perl/lib C:/Perl/site/lib
 .) at -e line 1.
BEGIN failed--compilation aborted at -e line 1.


C:\>ppm
PPM - Programmer's Package Manager version 3.0.1.
Copyright (c) 2001 ActiveState SRL. All Rights Reserved.

Entering interactive shell. Using Term::ReadLine::Stub as readline library.

Profile tracking is not enabled. If you save and restore profiles manually,
your profile may be out of sync with your computer. See 'help profile' for
more information.

Type 'help' to get started.
ppm> install Net::Telnet
====================
Install 'Net-Telnet' version 3.02 in ActivePerl 5.8.0.805.
====================
    .
    .
    .
Successfully installed Net-Telnet version 3.02 in ActivePerl 5.8.0.805.
ppm> quit
```

Once the library is installed, run the following code to test for the availability of the SMTP service on port 25:

© SANS Institute 2003, Author retains full rights.

```
use Net::Telnet;                                    # [1] use Net::Telnet lib

while (1) {                                         # [2] loop until terminated
    $dt = localtime;                                # [3] get date and time
    $flag = 1;                                      # [4] assume service is down
    eval{contact();};                               # [5] evaluate subroutine
    open (LOG, ">>smtp.log");                       # [6] append to log file
    if ($flag) {                                    # [7] if there is a problem,
        print LOG "SMTP server not responding at $dt\n";
                                                    # [8] log it
        $cmd = "net send janet \"SMTP_down\""; `$cmd`;
                                                    # [9] warn with a popup
    } else {                                        # [10] else
        print LOG "SMTP server responded OK at $dt\n";
                                                    # [11] log success
    }                                               # [12] end of if block
    close LOG;                                      # [13] save data
    sleep(120);                                     # [14] wait
}                                                   # [15] end main loop

sub contact                                         # [16] define subroutine
{                                                   # [17]
    $t = new Net::Telnet(                           # [18] create a telnet object
        Timeout => 30,                              # [19] that times out after 30s
        Port => 25);                                # [20] and connects to SMTP port
    $t->open("mail.company.com");                   # [21] attempt to connect
    @lines = $t->get();                             # [22] get banner
    $t->close;                                      # [23] close telnet session

    foreach $line (@lines) {                        # [24] cycle through banner
        if ($line =~ /SMTP MAIL Service/) {         # [25] if banner has expected string
            $flag = 0;                              # [26] then all is well
        }                                           # [27] end of if block
    }                                               # [28] end of foreach loop
}                                                   # [29] end of subroutine
```

We see several new scripting developments. Through the eval() function [5], a call is made to the subroutine contact, which tries to connect to port 25. Program execution should jump to [16], run through [29], and return to [6]. Why did I choose to branch into a subroutine when I could have listed the program lines in the order I want them to be executed? It turns out that as soon as an error condition occurs (such as when the service is not reachable by Telnet), the program will halt with a message like this:

```
problem connecting to "mail.company.com", port 25: Unknown error
at C:\smtp.pl line 34
```

By enclosing the subroutine call within eval(), this script will not be terminated abruptly (Wall, p. 705). In the subroutine, we see the object-oriented nature of the Net::Telnet library: [18]-[20] creates a telnet object that will attempt to connect to port 25 within thirty seconds. We open the connection [21], grab the banner [22], and close it [23]. We cycle through the output to look for an expected banner [25]. If found, the flag is cleared [26] before execution returns to [6]. Depending on the value of this flag, either the incident is logged and a warning is issued [8]-[9], or just a success message is logged [11].

Port Monitoring

Open only the ports you need, because each open port is another avenue for crackers to attempt to enter your system. When a system is configured,

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

document the ports that are supposed to be open. A great tool is fport (FoundStone). Not only does fport detect what ports are open, but it also identifies the responsible process ids, protocols, and executables.

First, we need to download from CPAN or ppm, Mark-Jason Dominus' Algorithm::Diff module, which compares two arrays and determines which elements have changed. Once installed, the following script calls on the diff() function to produce the differences.

```
use Algorithm::Diff qw(diff);                          # [1] need to access this module

while (1) {                                            # [2] loop until terminated
    `if exist before.txt del before.txt`;              # [3] preliminary
    `rename after.txt before.txt`;                     # [4] file shuffling
    open (IN, "<before.txt");                          # [5] read the baseline data
    @before = <IN>;                                    # [6] store file contents in array
    close IN;                                          # [7]

    @after = `fport /p`;                               # [8] run fport; output to array
    open (AFTER, ">after.txt");                        # [9] create new data file
    print AFTER @after;                                # [10] store data in file
    close AFTER;                                       # [11]

    $diffs = diff(\@before, \@after);                  # [12] compare baseline to new data

    if (@$diffs) {                                     # [13] if there are differences
        foreach $chunk (@$diffs) {                     # [14] this section was copied
            foreach $line (@$chunk) {                  # [15] from Mark-Jason Dominus'
                my ($sign, $lineno, $text) = @$line;   # [16] diff.pl script.
                if ($sign eq "+") {                    # [17] + means the 2nd array
                    print "Started: $text\n";          # [18] gained an item not in 1st
                    print LOG "Started: $text \n";     # [19] record to log file
                    `net send janet "$text started"`;  # [20]
                }                                      # [21]
                if ($sign eq "-") {                    # [22] - means the 2nd array does
                    print "Stopped: $text\n";          # [23] not contain an item in 1st
                    print LOG "Stopped: $text \n";     # [24] record to log file
                    `net send janet "$text stopped"`;  # [25]
                }                                      # [26]
            }                                          # [27]
        }                                              # [28]
    } else {                                           # [29] otherwise
        print "No change.\n";                          # [30] show that there is no change
    }                                                  # [31]
    sleep(30*60);                                      # [32] rest for 30 min
}                                                      # [33]
```

We begin by reading the baseline (contents of the previous run) and store it into the @before array [5]-[6]. Fport is run [8], and its contents are stored in @after. [12] compares the two arrays. If there are differences [13], we run the lines of code modified from Mark-Jason Dominus' diff.pl script to determine if an addition has been made [17] or a deletion [22]. Each instance is logged, and the administrator alerted. [19]-[20], [24]-[25]. If there were no differences, a message is displayed stating the fact. The process repeats every 30 minutes [32].

Periodically check for newly opened ports. Lock down your servers by disabling extraneous services. If a service port is no longer listening for connections, it is one less avenue for an attacker to a compromise your system.

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Microsoft Baseline Security Analyzer

Patching is a seemingly endless chore; it seems that as soon as you patch up your system, a new vulnerability is discovered. How does one keep up? One way is through a free download from Microsoft. The Microsoft Baseline Security Analyzer is a utility that assesses the security of your server. The script below uses the /hfnetchk command switch; it checks the patches on a server or range of servers and quickly informs you which hotfixes are missing.

This script takes the output to generate an HTML file that links directly to each Microsoft security bulletin, Common Vulnerabilities and Exposures names (see http://cve.mitre.org), and each server's list of patches to apply. Load this file through a web browser from a hardened computer to download and copy the hotfixes to the new computer. More information about MBSA is available at http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/tools/Tools/mbsaqa.asp

```
while (1) {                                                 # [1] loop until terminated
    $dt = localtime;                                        # [2] get date and time
    `c:\\progra~1\\micros~3\\mbsacli -hf -h 192.168.0.1 -f c:\\hf.out`;
                                                            # [3] run HF
    @qlist = ( );                                           # [4] clear Qarticles array
    open (IN, "<hf.out");                                   # [5] prepare to read output
    open (HTM, ">hotfixes2.htm");                           # [6] prepare to write
    print HTM "<HTML><TITLE>\n";#                           # [7] create title
    print HTM "Output of MBSACLI /HF on $dt (by Server)";# [8] title
    print HTM "</TITLE>\n";                                 # [9] end title
    print HTM "<PRE>\n";                                    # [10] preformat
    while ($line = <IN>) {                                  # [11] loop through each line
        if ($line =~ /(\S+)\s+\((\S+)\)/) {                 # [12] Look for Server and IP
            $server = $1;                                   # [13] server is first part
            $ip = $2;                                       # [14] IP address is second
            push @server_list, $server;                     # [15] store in server list
            print HTM "<A name=\"$server\">\n";             # [16] tag with server name
        }                                                   # [17] end of while loop
        print HTM $line;                                    # [18] write line to HTM file
        if ($line =~ /\* (.*)/) {                           # [19] Look for category name
            $category = $1;                                 # [20] when found, remember
        }                                                   # [21]
        if ($line =~ /Latest Service/i) {                   # [22] Look for old srv packs
            $service_pack_warning{$server} = "$category: $line";
                                                            # [23] store in hash
        }                                                   # [24]
        if ($line =~ /Patch NOT Found\s+(\S+)\s+(\d+)/) {#[25]Look for missing patches
            $bulletin = $1;                                 # [26] MS Bulletin is first
            $qarticle = $2;                                 # [27] Q Article is next
            $server_affected_by{$qarticle}.=" $server";     # [28] append server to hash
            $flag_found = 0;                                # [29] clear flag
            foreach $q (@qlist) {                           # [30] cycle through articles
                if ($q eq $qarticle) {                      # [31] if QArticle is in list
                    $flag_found = 1;                        # [32] raise the flag
                }                                           # [33]
            }                                               # [34]
            if ($flag_found == 0) {                         # [35] if flag is not raised
                $bullet{$qarticle} = $bulletin;             # [36] store new bulletin
                push @qlist, $qarticle;                     # [37] remember Q Article
            }                                               # [38]
        }                                                   # [39]
}                                                           # [40]
print HTM "</PRE>";                                         # [41] end of preformatting
close HTM;                                                  # [42] end of HTM file

open (HTM, ">hotfixes.htm") or die "Can not create HTM file";
                                                            # [43] create file
print HTM "<HTML><TITLE>Output of MBSACLI /HF on $dt</TITLE>\n";
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
                                                    # [44] title
    print HTM "<BODY><VLINK=\"#FFFFFF\"><H1> Patches</H1>\n";
                                                    # [45] label section
    print HTM "<TABLE BORDER>\n";                   # [46] create table
    print HTM "<TR><TH>Bulletin<TH>Reference\n";     # [47] show table header
    print HTM "<TH>Servers affected</TR>\n";        # [48]
    foreach $q (@qlist) {                           # [49] cycle through articles
        print HTM "<TR><TD><A
    HREF=http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/b
    ulletin/$bullet{$q}.asp>$bullet{$q} Q$q</A>";   # [50] generate MS link
        print HTM "<TD><A HREF=http://www.cve.mitre.org/cgi-
bin/cvekey.cgi?keyword=$bullet{$q}>CVE</A>";        # [51] generate CVE link
        @part = split(" ", $server_affected_by{$q}); # [52] split server string
        foreach $p (@part) {                        # [53] for every server
            print HTM "<TD><A HREF=\"c:\\hotfixes2.htm#$p\">$p</A>";
                                                    # [54] create link
        }                                           # [55]
        print HTM "</TR>\n";                        # [56] end of table row
    }                                               # [57]
    print HTM "</TABLE>\n";                         # [58] end of table
    print HTM "<HR><H1>Service Packs</H1>\n";       # [59] label section
    print HTM "<TABLE BORDER>\n";                   # [60] create another table
    print HTM "<TR><TH>Server<TH>Service Pack</TR>\n"; # [61] show table header
    foreach $s (@server_list) {                     # [62] cycle through servers
        if ($service_pack_warning{$s} ne "") {      # [63] Any missing srv packs?
            print HTM "<TR><TD>$s<TD> $service_pack_warning{$s}</TR>\n";
                                                    # [64]print row
        }                                           # [65]
    }                                               # [66]
    print HTM "</TABLE></BODY></HTML>\n";           # [67] end of table
    close HTM;                                      # [68] close the HTML file

    `start c:\\hotfixes.htm`;                       # [69] view the hotfixes
    sleep(60*60*24);                                # [70] repeat tomorrow
}                                                   # [71] end of while loop
```

This program starts by recording the date and time [2]. MBSACLI is run [3], and its output file is prepared to be read [4]. [5-10],[16],[18] write to a file that is essentially the same as the output file, except it includes internal links to make it easy to jump to the report of each server. The loop from [11]-[40] look for service packs and patches that have not been applied. [28] remembers each server that is missing a particular hotfix. [37] records each Q article. Another HTML file is generated in [43]-[68], containing the links to the relevant Microsoft Bulletin [50], CVE link [51], and hotfix report [54] from above. Missing service packs are also shown in [63]-[64]. At the end of the loop, this page is loaded for your review [69]. In summary, this script displays a web page of missing hotfixes and service packs, providing links for you to read about the vulnerabilities and download the hotfixes.

Event Logs

Event logs are one of the first places to turn to when investigating computer issues. The information may be seen through Event Viewer, but another tool is better suited if you wish to automate the event collection process. Dumpel is a powerful utility from the Windows 2000 Resource Kit. We will use it in our script to pull log information from not just one server, but from an entire list of servers. The data is sorted by date and time to give you a bigger picture of what is happening to your Windows NT/2000 network.

© SANS Institute 2003, Author retains full rights.

```
open (IN, "<servers.txt");                    # [1] open server list
while ($server=<IN>) {                        # [2] cycle through each server
    chomp $server;                            # [3] remove CRLF
    @results = `dumpel -s \\\\$server -l security -c`;
                                              # [4] capture its security log
    foreach $r (@results) {                   # [5] cycle through results
        push @everything, $r;                 # [6] save to master array
    }                                         # [7]
}                                             # [8]
close IN;                                     # [9]

@sorted = sort @everything;                   # [10] sort the master array
open (OUT, ">sorted_logs.csv");               # [11] open file to save array
print OUT @sorted;                            # [12] write to file
close OUT;                                    # [13]
```

First, we read the names of the servers in [1],[2]. Dumpel is then run on each server [4]. Notice the parameters: -s specifies the server, -l indicates which log, and –c means to store return the data in a comma-delimited format. Every entry for the security log is captured into the @everything array [6]. This array is sorted in [10], and written to a file in [12]. By default, dumpel returns the date first, followed by the time, event type, category, ID, source, user, computer, and string data. If you do not wish to sort the information by date and time, modify the sort order through dumpel's –format switch:

```
-format <fmt>    Specify output format.  Default format is
                 dtTCISucs
                 where
                   t - time
                   d - date
                   T - event type
                   C - event category
                   I - event ID
                   S - event source
                   u - user
                   c - computer
                   s - strings
```

Try to modify the code to save all three logs for each server. Take it one step further to pull the logs from all servers every hour, use Algorithm::Diff to find the changes, and alert you when events you define occur. One can see how this powerful utility, combined with your Perl script can make log reviews much more tolerable.

File Integrity

Your company's most valuable asset is its data; protect the integrity of your data by monitoring your files. If a Network Intrusion Detector fails to detect an intruder penetrating your network, you need to make sure critical files are secured on your hosts.

```
use Digest::MD5;                                        # [1] may already be present
$numfiles=0;                                            # [2] initialize counter
open(LIST, "<filelist.txt") || die "Can't open file list";
                                                        # [3] open file list
while ($line = <LIST>) {                                # [4] read each line
    chomp $line;                                        # [5] remove CRLF
    $filename[$num_files] = $line;                      # [6] record filenames
    $orig_hash{$filename[$num_files]} = get_hash($filename[$num_files]);
                                                        # [7] record hashes
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

```
    $num_files++;                                           # [8] increment counter
}                                                           # [9]
close LIST;                                                 # [10] close file

while (1) {                                                 # [11] main loop
    for ($i=0; $i<$num_files; $i++) {                       # [12] cycle through files
        print "origval of $filename[$i] = ", $orig_hash{$filename[$i]}, "\n";
                                                            # [13] before
        print "get_hash of $filename[$i] = ", get_hash($filename[$i]), "\n";
                                                            # [14] after
        if (get_hash($filename[$i]) == $orig_hash{$filename[$i]}) {
                                                            # [15] if same,
            print "Hashes compare ok\n";                    # [16] all's well
        } else {                                            # [17] else
            `net send Janet "$filename[$i] has changed"`;   # [18] warning
        }                                                   # [19]
    }                                                       # [20]
    sleep (60*60);                                          # [21] check every hour
}                                                           # [22] end while loop

sub get_hash                                                # [23] define sub
{                                                           # [24]
    $filename = shift;                                      # [25] get parameter
    open (FILE, $filename) || die "Can't open file to get hash";
                                                            # [26] open file
    binmode(FILE);                                          # [27] set to binary mode
    $hashval{$filename} = Digest::MD5->new->addfile(*FILE)->hexdigest, " $filename\n";
                                                            # [28] record MD5 hash
    return $hashval{$filename};                             # [29] return value
}                                                           # [30]
```

This Perl script uses the Digest::MD5 module [1] to read a list of files to watch and produces an MD5 hash for each one [7]. Periodically, hashes are generated and compared to the original hashes [15]. If the hashes match, then all is well; otherwise, an alert is generated.

Stay vigilant

As computer security professionals, we need to keep up to date with very current information. The SANS web site posts a wealth of knowledge, and links to the Internet Storm Center at http://incidents.org. We may see at a glance the current condition of Internet Security.

With the aid of Wget, available at http://www.interlog.com/~tcharron/wgetwin.html, the following script determines the Internet Threat Level.

```
`wget -O sans.htm http://www.sans.org`;                     # [1] grab SANS web page
open (IN, "<sans.htm");                                     # [2] open web page locally
while ($line=<IN>) {                                        # [3] read each line
    if ($line =~ /threat level is (\S+)\"/i) {              # [4] look for threat level
        $level = $1;                                        # [5] capture threat level
        print "Threat level is $level\n";                   # [6] show level
        if ($level !~ /green/i) {                           # [7] if not green
            `net send administrator "Elevated threat level"`;
                                                            # [8] notify administrator
        }                                                   # [9]
    }                                                       # [10]
}                                                           # [11]
close IN;                                                   # [12] close local web page
```

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

This concludes the paper, but hopefully not your zeal for programming security scripts in Perl. There is a large community of Perl programmers. Join the newsgroups. Do a Google search for Perl tutorials. Armed with the ability to automate the usage of powerful tools, we may do our jobs more effectively.

© SANS Institute 2003, Author retains full rights.

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Resources

Charron, Tim. "BLAT for Windows."
URL: http://www.interlog.com/~tcharron/blat.html (4 April 2003).

Foundstone, Inc. "fport."
URL: http://www.foundstone.com/knowledge/proddesc/fport.html (4 April 2003).

Microsoft Corporation. "Windows 2000 Resource Kits." URL:
http://www.microsoft.com/windows2000/techinfo/reskit/default.asp (4 April 2003).

Microsoft Corporation. "The Ten Immutable Laws of Security." URL:
http://www.microsoft.com/technet/columns/security/essays/10imlaws.asp (4 April 2003).

Perldoc. "eval." URL: http://www.perldoc.com/perl5.8.0/pod/func/eval.html (4 April 2003).

Robert, Kirrily. "Introduction to Perl: The Perl Philosophy." URL:
http://www.maths.adelaide.edu.au/cmc/tutorials/perlintro/x175.html (4 April 2003).

SANS Institute. "Security Policy Project."
URL: http://www.sans.org/resources/policies (4 April 2003).

Scrambray, Joel, and Stuart McClure. Hacking Windows 2000 Exposed. Berkeley: Osborne/McGraw-Hill, 2001.

Wall, L., Christiansen, T., and Orwant, J. Programming Perl, 3rd Ed. Beijing; Cambridge, Mass.: O'Reilly & Associates, 2000.

© SANS Institute 2003, Author retains full rights.