inside:

PROGRAMMING

Flynt: The Tclsh Spot

the tclsh spot

by Clif Flynt

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of *Tcl/Tk for Real Programmers* and the *TclTutor* instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

clif@cflynt.com

Monitor your system with expect

The previous few *Tclsh Spot* articles described generating IP packets to attack a firewall system being tested. Once the packets can be generated, the next step is to confirm that they are captured and logged correctly.

For someone familiar with the log files, this is a simple exercise. You can tail -f the appropriate file and watch for the log messages, or grep for the expected pattern.

For someone familiar with the log files, this is also a tedious exercise. Not too bad when you are looking for something tricky, but too much monkey-work if you are generating hundreds of attacks and need to confirm that all are caught correctly.

Automating tedious tasks that would otherwise be done by a skilled (or at least easily bored) human is what the expect extension was written for.

The expect extension to Tcl was developed by Don Libes of the NIST in 1990, very shortly after John Ousterhout's first Tcl paper was presented at the USENIX Annual Technical meeting.

Expect will interact with a text-oriented task, looking for patterns, and sending responses as appropriate.

For those with fond memories of UUCP chat scripts, you can think of expect as a chat script with much more power.

The expect extension can be dynamically loaded into a running Tcl shell with the package require command like this:

```
% package require expect
5.38.0
```

It's more common to use the standalone expect shell – invoked as expect. When invoked as expect a special prompt is provided showing the depth of the evaluation stack, and the number of commands that have been processed.

Defining a simple procedure within the expect shell looks like this:

```
$> expect
expect1.1> proc a {a b} {
+> puts "$a"
+> }
expect1.2>
```

The expect package is extensive with many commands and options to support the various interactions one might need to perform. There are only three essential commands, however.

spawn	Starts a new task for the expect script to interact with.
expect	Defines a set of patterns and actions to perform when a pattern is matched.
exp_send	Sends a stream of data to the child task.

The spawn command starts a child process to be controlled by the expect script. It starts a new process and connects the process's stdin and stdout to the expect interpreter.

Syntax: spawn *options commandName commandArgs*

options
>The spawn command supports several options, including
>*-noecho*
>>The spawn will echo the command line and arguments unless this option is set.
>*-open fileID*
>>The -open option lets you process the input from a file handle (returned by open) instead of executing a new program. This allows you to use an expect script to evaluate program output from a file as well as directly controlling a program.

commandName
>The name of the executable program to start.

commandArgs
>The command line arguments for the executable program being spawned.

Opening an ssh session would look like this:

```
expect1.1> spawn ssh -l clif 127.0.0.1
spawn ssh -l clif 127.0.0.1
1105
expect1.2>
```

Once a new child task has been spawned, your script needs to interact with it. The expect command listens to the child task, while the exp_send command talks to it.

The expect command will scan the input for one of several patterns. When a pattern is recognized the associated action will be evaluated.

Syntax: expect *?-option? pattern1 action1 ?-option? pattern2 action2 ...*

-option Options that will control the matching are:
>*-exact* Match the pattern exactly.
>*-re* Use regular expression rules to match this pattern.
>*-glob* Use glob rules to match this pattern.

pattern A pattern to match in the output from the spawned program.

action A script to be evaluated when a pattern is matched.

This code snippet will look for the password prompt and report that the prompt was seen:

```
expect1.3> expect {
+> word: {puts "The password: prompt was seen"}
+> }
```

The pattern/action format of the expect command resembles the switch command. The switch command uses the meta-pattern default to process unexpected conditions. The expect command also supports meta-patterns to process unexpected input.

The eof meta-pattern matches when the expect interpreter loses contact with a child task. Usually this happens when a child task dies.

The timeout pattern will match if no other pattern has been matched within a given period of time. This script checks for the password prompt, and complains if it doesn't appear:

```
expect {
    word: {puts "The password: prompt was seen"}
    timeout {puts "There was no password prompt!"}
}
```

The default timeout period is 10 seconds. This can be changed by setting a variable named timeout to the number of seconds to wait before matching the timeout pattern.

Note that the timeout value is a maximum value, not a minimum value. The timeout is implemented by watching the system's one-second timer. Each time this fires, the timeout value is decremented by one. When the value hits zero, the timeout pattern is matched, and the associated script is evaluated.

Thus, setting the timeout value to 1 will never wait longer than a single second, but could timeout in .0001 seconds.

This code snippet will watch for the password prompt and complain if either the child task has lost contact with the expect script or there is a timeout.

```
expect {
    word: {puts "The password: prompt was seen"}
    eof    {puts "The child task has died!"}
    timeout {puts "There was no password prompt!"}
}
```

The flip-side to listening to a child task is to send strings of data to the child. The exp_send sends string to the child process.

Syntax: exp_send *string*

string   The string to be transmitted to the slave process. Note that a newline is not appended to this text.

A simple script to look for the password prompt and send a password looks like this:

```
expect {
    word: {exp_send "PASSWORD\n"}
}
```

Notice the \n at the end of PASSWORD. In order to interact with applications that key off of single keystrokes, the expect command sends exactly what you tell it to, and does not add any new characters (like newlines) to the string.

A script to log into a remote system, look for the shell prompt, and change to the /var/log directory would look like this:

```
spawn ssh 127.0.0.1

expect {
    word:   {exp_send "$password\n"}
    timeout {error "Timeout waiting for password"}
    eof     {error "Eof waiting for password"}
}

expect {
    "%>"    {exp_send "cd /var/log\n"}
    timeout {error "Timeout waiting for prompt"}
    eof     {error "Eof waiting for prompt"}
}
```

As with the switch statement's default option, you don't need to use the timeout and eof patterns. But, as with the switch (or C case statement), if you ignore catching the exception conditions, you will live to regret it.

This can lead to a lot of repeated code if your application has a lot of simple challenge/response interactions.

One solution to this problem is to create a procedure to maintain the process flow and provide the customization with the procedure arguments.

The syntax for the proc command is:

Syntax: proc *name args body*

The args and body parameters are generally grouped with braces when you define a procedure within your application.

This procedure generalizes the challenge/response nature of many conversations.

```
##############################################
# proc challResp {pattern response info}
# Hold a single interchange challenge/response
# interaction.
# Arguments
#    pattern:   The pattern to wait for as a challenge
#    response:  The response to this pattern
#    info:      Identifying information about this interac-
#               tion for use with exception reporting
#
# Results
#
#
proc challResp {pattern response info} {
    expect {
        $pattern  {exp_send "$response\n"}
        timeout   {error "Timeout at $info"}
        eof       {error "Eof at $info"}
    }
}
```

Using the challResp procedure, automating this conversation:

```
%> ssh -l root 127.0.0.1
root@127.0.0.1's password:
Last login: Thu Jun 5 05:45:02 2003 from localhost
Have a lot of fun...
%> cd /var/log
%> tail -f messages
```

can be done with these four lines of code:

```
spawn ssh -l root 127.0.0.1

challResp "word:" $passWord "password prompt"
challResp "%>" "cd /var/log" "first shell prompt"
challResp "%>" "tail -f messages" "second shell prompt"
```

We can create a generic procedure for watching log files for a given pattern with this procedure.

```
#############################################
# proc watchFile {host passwd fileName pattern}—
#    Watch a file on a remote system for a given pattern
# Arguments
#    host:       IP Address of the host on which the file
#                resides
#    passwd:     Password for accessing the host
#    fileName:   Full path to the file to be watched
#    pattern:    A pattern to watch for in the file
# Results
#    Throws an error if the expected pattern is not seen
#
proc watchFile {host passwd fileName pattern} {
    global spawn_id
    set stty_init -echo
    spawn ssh -l root 127.0.0.1

    challResp "word:" $passwd "password prompt"
    challResp "%>" "cd /var/log" "first shell prompt"
    challResp "%>" "tail -f messages" "second prompt"

    expect {
      "$pattern" {puts "ROOT LOGIN SUCCESSFUL"}
      timeout {error "timeout" "Timeout  \"$pattern\""}
      eof    {error "eof" "Eof waiting for \"$pattern\""}
    }
}
```

Note the global spawn_id at the beginning of this procedure. When spawn creates a new task, it creates a unique identifier for that task and saves that value in the variable spawn_id in the current scope. If the new task is spawned in a procedure, spawn_id is created in the procedure scope, and it vanishes when the procedure returns. Keeping the spawn_id in the global scope is the simplest way to deal with this for applications that only have a single child task.

This discussion of expect barely scratches the surface of what the package can do, but it's enough to automate accessing a remote system and scanning a log file for a pattern.

The next Tclsh Spot article will start looking at ways to coordinate the packet generators described previously with this log file scanner to generate attacks on a test system and confirm that the test system responds to the attacks correctly.