



Scripting Essentials: A Guide to Using VBScript in SecureCRT®











VANDYKE®
SOFTWARE

4848 tramway ridge dr. ne
suite 101
albuquerque, nm 87111

505 - 332 - 5700

www.vandyke.com

A Guide to Using VBScript in SecureCRT

Introduction	1
How This Document is Organized	1
Conventions Used in This Document	1
SecureCRT's Scripting Objects/API Reference	2
Chapter 1: Creating Scripts	4
1.1 Starting from Scratch with a New Script	4
1.2 Recording a Script	5
1.3 Modifying an Existing Example Script	9
Chapter 2: Launching Scripts	11
2.1 Manual Script Execution	11
2.2 Automated Script Execution	20
Chapter 3: Connecting to Remote Machines	24
3.1 Connecting with a Pre-configured Session	24
3.2 Connecting in an "Ad Hoc" Fashion	25
3.3 Connecting in Tabs	26
 Solution: Open a Group of Sessions in Tabs	28
3.4 Disconnecting Active Connections	29
 Solution: Automatically Close an Inactive Connection (Auto Disconnect)	30
3.5 Connecting to a List of Remote Machines Within a Loop	31
3.6 Handling Connection Failures Within a Script	31
Chapter 4: Reading Data from Remote Machines	35
4.1 Accessing Selected Text on the Screen	35
 Solution: Performing a Web Search with Selected Text	35
4.2 Waiting for Specific Data to Arrive	36
 Solution: Receive Notification when "Error-Indicating" Text Appears	38
4.3 Capturing Data from a Remote Machine	46
Chapter 5: Sending Data to Remote Machines	54
5.1 Sending Plain Text	54
 Solution: Repeat a Command with Variable Input from User	54
5.2 Sending Control Codes	56
5.3 Simulating Keyboard Events	57
 Solution: Add "no" to Each Selected Line and Send to Remote	59
Chapter 6: Getting Information from the End User	62
6.1 Prompting for Simple Responses: Yes, No, OK, Cancel, etc.	62
6.2 Prompting for Arbitrary Text Input	69
 Solution: Clone Current Tab Multiple Times	72
6.3 Building Custom Dialogs or Forms	77
 Solution: Create a Custom Username and Password Input Dialog	81
Chapter 7: Logging, Reading, and Writing Files	83
7.1 Logging with SecureCRT's Session Object	83
7.2 Reading Data from Files Using the FileSystemObject	87
 Solution: Read Data from Separate Files: Hosts, Commands	90
7.3 Writing Data to Files Using the FileSystemObject	95
 Solution: Save Selected Text to a CSV File	97
 Solution: Import Data from File to SecureCRT Sessions	100
Chapter 8: Working with the Windows Clipboard	113
8.1 Retrieving Data Stored in the Clipboard	113
8.2 Storing Data to the Clipboard	113
 Solution: Auto-Save Command Results to the Clipboard	113
8.3 Changing Paste "Speed"	115
 Solution: "Slow Paste" (Line Delay)	115
 Solution: "Slow Paste" (Echo Delay)	116
 Solution: Vary Paste Speed Based on Clipboard Length	117
8.4 Setting the Clipboard Text Format (Encoding)	118

Introduction

A SecureCRT user once wrote to the VanDyke Software technical support department in reply to some answers to scripting questions saying:


“This will alleviate a lot my daily boring and annoying job. My dream is to be able to use scripts even at a higher level, ...activate macros pertaining to the various windows to automate almost completely my job.”

While your job may not be boring or annoying, freeing up time from the repetitive widget-cranking tasks you need to do on a regular basis may very well be a target worth shooting for.

The main objective of this document is to provide concepts and examples of scripting within SecureCRT – information that might even help you accomplish more work in less time.

How This Document is Organized

The first two chapters of this document provide introductory information about how to create or edit scripts and explore different ways of launching scripts.

The remaining chapters focus on goals/tasks and provide actual solutions in the form of example scripts. If you’re looking for example code right away, skim through the chapters and pay special attention to the solution sections indicated by the  **Solution** indicators within the table of contents and throughout this document.

If you’re looking to learn a few tips and tricks which may help stock your toolbox with time-saving techniques that you might find useful in various automation scenarios, consider reading through this document chapter by chapter.

Conventions Used in This Document

Language Used

SecureCRT’s use of ActiveX technology allows a script author to use any ActiveX scripting language for which a scripting engine is available to interface with SecureCRT as the script host. Examples include VBScript and JScript, which are both native to Windows. Another example of an ActiveX scripting language that can be used is PerlScript, which requires a third-party Perl ActiveX scripting engine be installed.

VBScript, also known as Visual Basic Scripting Edition, is the language of choice for this document. All example script code herein is written using the VBScript language. If you are not familiar with VBScript, reference documentation is available both online and for download from Microsoft.

Online:

<http://msdn.microsoft.com/en-us/library/t0aew7h6.aspx>

If the above link doesn’t work, perform a web search on the terms “Windows Script Technologies Documentation” and you should be provided with an up-to-date link. Of particular note would be the *VBScript User’s Guide* which gives information specific to using VBScript in general, and also the *VBScript Language Reference* which provides explanations of core components of the VBScript language.

Download:

<http://www.microsoft.com/downloads/details.aspx?FamilyId=01592C48-207D-4BE1-8A76-1C4099D7BBB9>

If the above link doesn't work, performing a web search on the terms "Windows Script Technologies Download" should provide you with an up-to-date link for downloading the documentation in Compiled HTML Help (.chm) format directly from Microsoft.

Coding Style and Conventions

Throughout this document you'll see the use of specific font styles that indicate either actual VBScript code or reflect user interface labels. The following table outlines these font styles and their associated meanings.

Style	Reflects	Example
Fixed-width font	VBScript Code, file names and extensions.	<code>MsgBox "Hello World!"</code>
Bolded Text	User interface labels or window titles.	Open up the main Script menu and choose Start Recording Script .

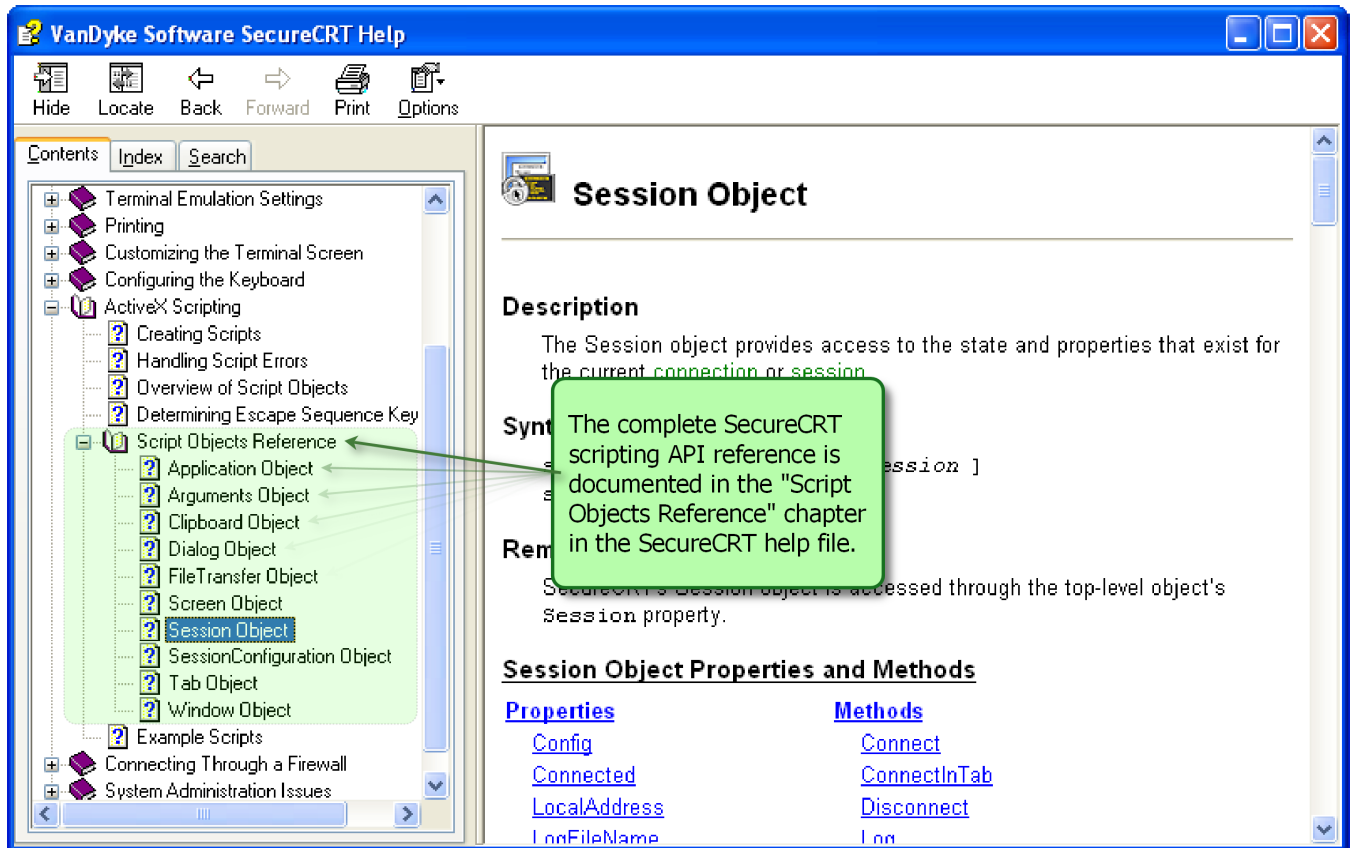
Examples of VBScript code throughout this document attempt to adhere to a common variable naming convention where variables are named loosely according to their purpose. Since VBScript is not a "typed" language, this naming convention helps remind the reader (and the author) of the intended use of each variable. This naming convention is presented in the table below.

Prefix	Intended Use and Meaning	Example
b	Boolean variable whose value will typically be either <code>True</code> or <code>False</code> .	<code>bConnected = crt.Session.Connected</code>
n	Numeric variable whose value will typically be numeric.	<code>nCounter = nCounter + 1</code>
str	String variable for holding generic text.	<code>strLogFile = crt.Session.LogFileName</code>
obj	Object variable referencing an object.	<code>Set objShell = CreateObject("WScript.Shell")</code>
v	Array (vector) variable that references an array of elements.	<code>vElements = Split(strLine, ",")</code>
g_	Indicates that a variable is intended for use in a global scope. Helps keep track of variables that might be used both inside subroutines and functions as well as outside the scope of methods and functions.	<code>Set g_strMyDocs = _ objShell.SpecialFolders("My Documents")</code>

SecureCRT's Scripting Objects/API Reference

This scripting guide focuses on scripting techniques specifically related to solving a variety of problems by providing actual solution code that you can take and modify to meet your needs.

If you are looking for documentation detailing all of the SecureCRT scripting objects, methods, properties, and related parameters (the API reference), you'll find it in the SecureCRT Help file. The SecureCRT scripting objects/API reference is found by opening the SecureCRT Help, and browsing to the **ActiveX Scripting / Script Objects Reference** chapter, as indicated in the graphic below:



Chapter 1: Creating Scripts

Have you ever heard the saying, “*You have to spend money in order to save money*”? The same concept can be applied to scripting in SecureCRT:

You have to take time to save time.

Fortunately, taking time to automate routine tasks with SecureCRT can pay big dividends – even without a huge initial investment.

So where do you start? A few options are available for generating a script that may save you oodles of time in the long run:

- [Starting from scratch with a new script.](#)
- [Recording a script using the SecureCRT Script Recorder.](#)
- [Modifying an existing example script.](#)

Let’s explore each of these options in more detail.

1.1 Starting from Scratch with a New Script

Whether you are a seasoned VBScript guru or are looking at using VBScript for the first time, you may want to know the following guidelines for successfully writing a script for use within SecureCRT.

Include a SecureCRT Script Header

The first requirement is to ensure that your script code identifies the ActiveX scripting language engine that needs to be instantiated in order to run your script code. This identification is done through the use of a SecureCRT *script header*. For all scripts written in the VBScript language, this script header should be written as follows:

```
#$Language="VBScript"  
#$Interface="1.0"
```

Note that a script header is not required as long as the file extension is registered in the registry to be handled by a corresponding ActiveX script engine. For example, on most Windows systems that have Internet Explorer 4.0 or later installed, the “.vbs” file extension is already registered for interpretation by the VBScript engine available by default.

However, you might want to save all of your script files without a .vbs extension. For example, if you want to store all your script files as .txt files, you will need to include a script header as indicated above. In this document, focus is centered on the use of the VBScript language. For additional information about script headers applicable to other ActiveX languages, visit the “ActiveX Scripting” chapter of the SecureCRT help, specifically the topic titled “Creating Scripts”.

Organize Code Into Subroutines and Functions

To `Main()` or not to `Main()`? That is the question. Native VBScript support for subroutines, functions, classes, and other statements remain alive and well when interpreted within SecureCRT. Therefore, statements like `Sub` and `Function` and their corresponding `End Sub` and

`End Function` counterparts are available to you for the purpose of organizing your script code into logical units of separation.

If a `Main()` subroutine is found by SecureCRT, `Main()` will be called automatically. Note that this is in contrast to any native VBScript code where subroutines and functions must be explicitly called in order to execute.

While it is not a requirement that you place your code within a `Main()` subroutine, there are some good reasons why you might want to do so.

- The VBScript engine will parse and execute *global*¹ script code before your `Main()` subroutine is executed. This allows you to set up global variables so they are initialized and available prior to your `Main()` subroutine being called.
- Including the core components of your script within a `Main()` subroutine provides a way of aborting script execution in the event that you detect a problem or otherwise need to abort execution (user cancels an `InputBox`, for example). Since the `WScript.Quit` method is not available for use within a SecureCRT script, if you want to stop execution of your script, you can do so with the use of the `Exit Sub` statement. For example:

```
Sub Main()
    ' Prompt the end user for data
    strAnswer = InputBox("Please enter your name:")

    ' Check to see if the user provided any data, or canceled.
    If strAnswer = "" Then
        MsgBox "User Canceled."
        Exit Sub
    End If

    ' If the user didn't cancel, move forward with
    ' the remainder of the script code...
    crt.Session.Connect "/S MySession"
    .
    .
    .
End Sub
```

1.2 Recording a Script

Perhaps the quickest way to create a script that will do something useful is to use the Script Recorder feature introduced in SecureCRT 5.5. One nice thing about using the Script Recorder is that relevant code reflecting exactly what you recorded is included in the resulting file. Code resulting from a script recording will likely need to be modified in order to be launched in succession on other machines or targeting additional devices, but the script recorder provides a great way for novices and experts alike to get started creating a SecureCRT script.

A demonstration of recording a script that performs a series of commands while connected to a remote UNIX machine will be shown. Then some of the more rudimentary edits that should be made to enhance the effectiveness of a script for use on remote machines other than the one on which the script was recorded will be exemplified.

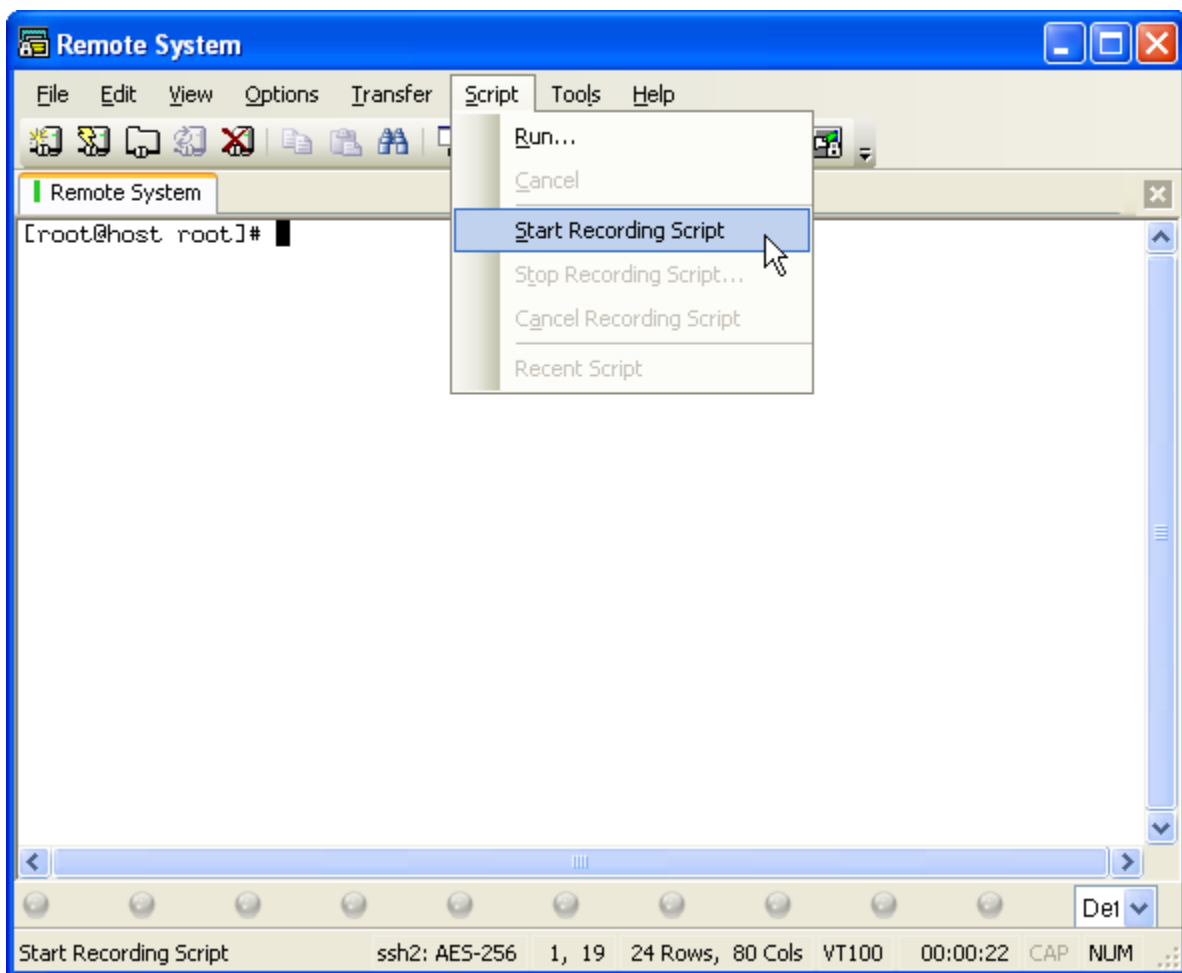
Here are the general steps to follow when using the SecureCRT Script Recorder:

¹ *Global* script code is code that exists outside of any subroutine, function, or class within your file.

- 1) Connect to the remote machine.
- 2) Start script recording.
- 3) Perform commands to be recorded.
- 4) Stop script recording.
- 5) Specify a filename in which to save the recorded script code.
- 6) Review and edit the recorded script file.

This walk-through will show the process of connecting to a remote UNIX system, enabling the script recorder, and then issuing commands that will update the existing VShell installation with the new version of the service. Once the script has been recorded, the script code will be reviewed and cleaned up for general use on other machines.

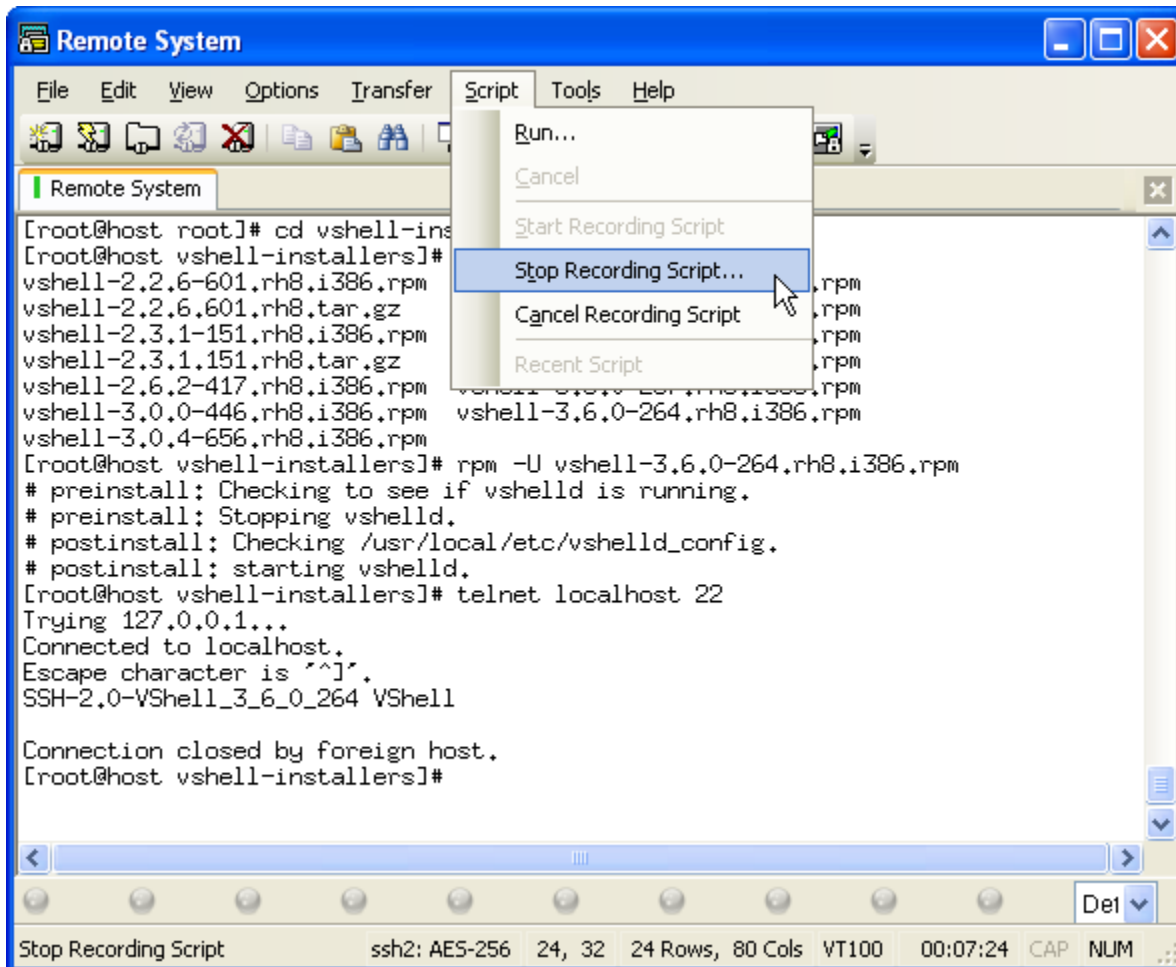
The first step is to connect to the remote UNIX system. Once a shell prompt on the remote system has been obtained in SecureCRT, choose **Start Recording Script** from the main **Script** menu as indicated in the graphic below.



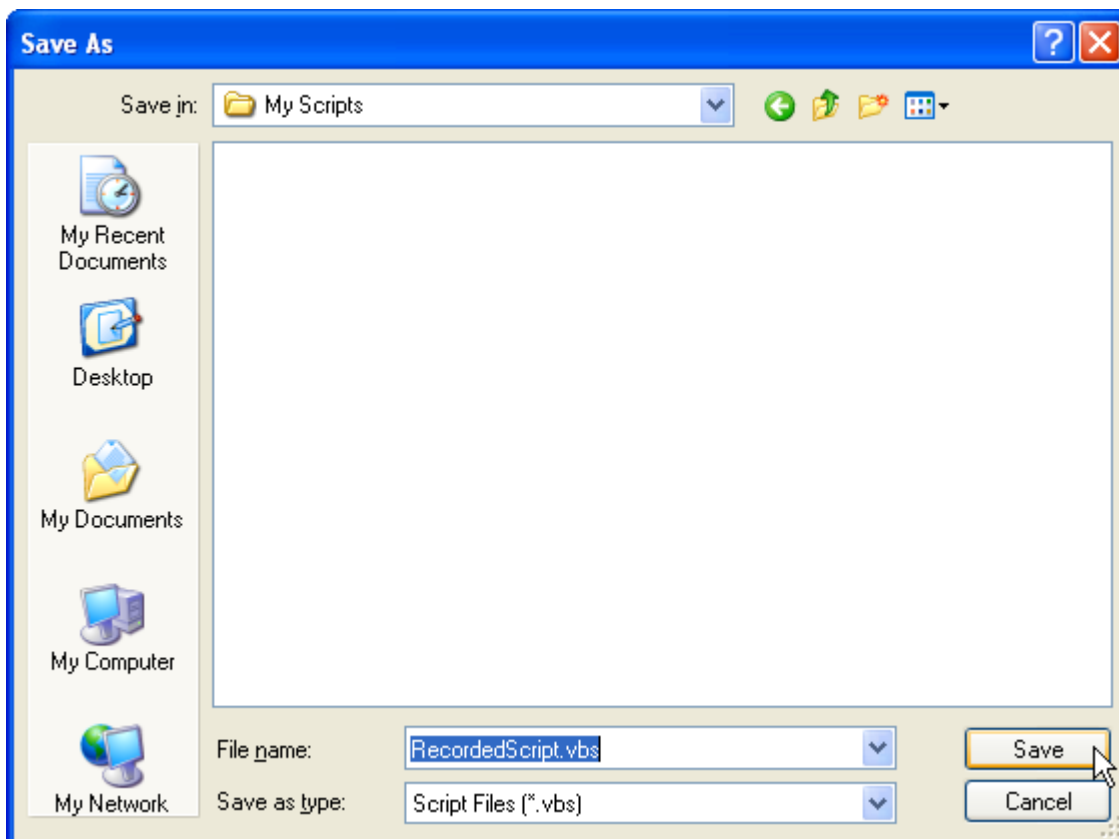
Now that the Script Recorder is active, commands specific to the task at hand are issued:

```
cd vshell-installers
ls
rpm -U vshell-3.5.0-306.rh8.i386.rpm
telnet localhost 22
```

When the work has been completed, open the main **Script** menu and choose **Stop Recording Script** as indicated in the graphic below.



Once the script recording operation has been stopped, the user is prompted for the name of the file in which to save the script code associated with the data that has been sent and received from the remote system, as indicated in the graphic below.



After bringing up the "RecordedScript.vbs" file in an editor, the VBScript code which reflects the send/receive activity that was just completed can be reviewed:

```
#$Language="VBScript"
#$Interface="1.0"

crt.Screen.Synchronous = True

' This automatically generated script may need to be
' edited in order to work correctly.

Sub Main()
    crt.Screen.Send "cd vshell" & chr(9) & chr(13)
    crt.Screen.WaitForString "$ "
    crt.Screen.Send "ls" & chr(13)
    crt.Screen.WaitForString "$ "
    crt.Screen.Send "rpm -U vshell-3.5.0-3" & chr(9) & chr(13)
    crt.Screen.WaitForString "$ "
    crt.Screen.Send "telnet localhost 22" & chr(13)
    crt.Screen.Send chr(13)
End Sub
```

Note that the script header is automatically included by the SecureCRT Script Recorder at the top of the file.

Note also that exactly what was typed is reflected in the script code (not necessarily what was displayed when it was typed). As an example, consider the first line of the `Sub Main()` code above, where pressing the **Tab** key resulted in a `chr(9)` (ASCII Tab character) being sent to the remote machine as part of the `Send()` operation:

```
crt.Screen.Send "cd vshell" & chr(9) & chr(13)
```

When typing this command in at the shell prompt within SecureCRT, the **Tab** key was pressed after typing `cd vshell`, kicking in tab-line completion and the remote shell filled in the full path as displayed on the screen: `cd vshell-installers`.

However, to be exactly sure of the command that should be sent when running the script on a different machine, the script code will need to be modified to reflect the full command to send, rather than reflect the exact keys that were typed while the script was being recorded:

```
crt.Screen.Send "cd vshell-installers" & chr(13)
```

The same approach would need to be taken to adjust for the use of tab-line completion when sending the command `rpm -U vshell-3.5.0-306.rh8.i386.rpm` to ensure that the exact file desired is in fact used (rather than relying on tab-line completion to always do the “right” thing). Hence, the following line of code as recorded by the script recorder...

```
crt.Screen.Send "rpm -U vshell-3.5.0-3" & chr(9) & chr(13)
```

...would be modified as follows to ensure reliable execution:

```
crt.Screen.Send "rpm -U vshell-3.5.0-306.rh8.i386.rpm" & chr(13)
```

1.3 Modifying an Existing Example Script

If at first you don't succeed, borrow from those who have already succeeded. In the event that creating a script from scratch or recording a script doesn't get you close enough to achieving your automation goals, you can always leverage existing solutions and examples to get you closer.

Example scripts exist in various forms:

- The “Example Scripts for SecureCRT for Windows” section of the VanDyke Support website: https://www.vandyke.com/support/securecrt/scripting_examples.html.
- The “Scripting Examples” section of the VanDyke Support website: <https://www.vandyke.com/support/scripting/scripting-examples/index.html>.
- The SecureCRT installer (version 6.1 or later) includes a variety of example scripts that perform tasks reflecting the inquiries of many SecureCRT end users. If you decide to use one of these examples as a starting place for your own script, it is strongly suggested that you make a copy of the example script file and modify the copy, rather than modifying the example's original script file itself. If the original script files are modified, updating SecureCRT to a newer version will result in restoring the example script files to their original contents (since the example script files themselves are included as part of the SecureCRT 6.1 and later installers), resulting in the loss of all your modifications made to the original example script files.

Here are some tips on what to look for when modifying existing code examples to meet your needs:

Review Example Code Completely Before Running It

You don't want to run a script that potentially does something catastrophic to your system (local or remote) without first knowing what you're getting into. Be aware that your system might be different from the system used by the author in creating the example script and take steps to adjust the example to work for your system.

Pay Attention to the Comments

Good examples provide comments as to the overall actions taken by the script, requirements, and setup. Read through these carefully before launching an example script.

Use Only What You Need

There may be times where an example script does much more than you need or want, but it contains that all-important clue as to how to do that one specific action you were looking for. Sometimes it's more effective to use only the components of the example you need to aid in understanding so you can employ similar techniques in your own script.

Chapter 2: Launching Scripts

At some point, regardless of whether you recorded a script, created a script from scratch, or modified an example script you acquired, you'll likely want to manually run the script and see if it does what you need. After you've had a chance to perfect the script so that it accomplishes what you need it to do, you may want to set up SecureCRT to run the script in an automated way so as to save yourself even more time. In this section, both manual and automated script execution within SecureCRT will be introduced. For automated script execution, ways to pass arguments to scripts to change the behavior of the script based on the script arguments provided will also be described.

2.1 Manual Script Execution

Manual script execution in SecureCRT is facilitated by means of the following mechanisms:

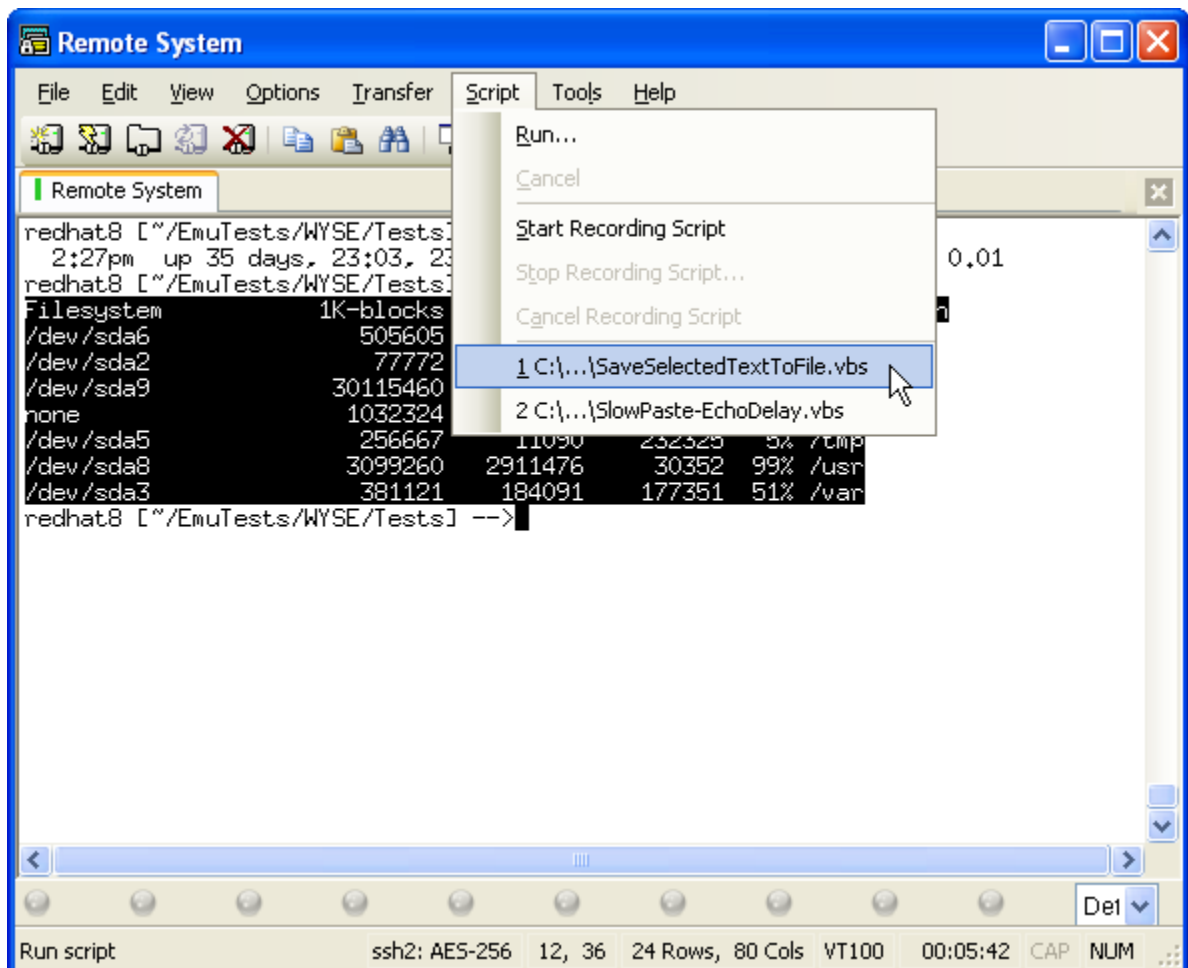
[Using **Script / Run...** from SecureCRT's main menu](#)

[Mapping a key to run a script](#)

[Launching a script from a button on SecureCRT's button bar](#)

Using Script / Run... from SecureCRT's Main Menu

The first time you run a script, it will likely be by manually choosing **Run...** from the main **Script** menu in SecureCRT. Once you have launched a script in this fashion, it will appear within the *Recent Scripts* section of SecureCRT's **Script** menu.

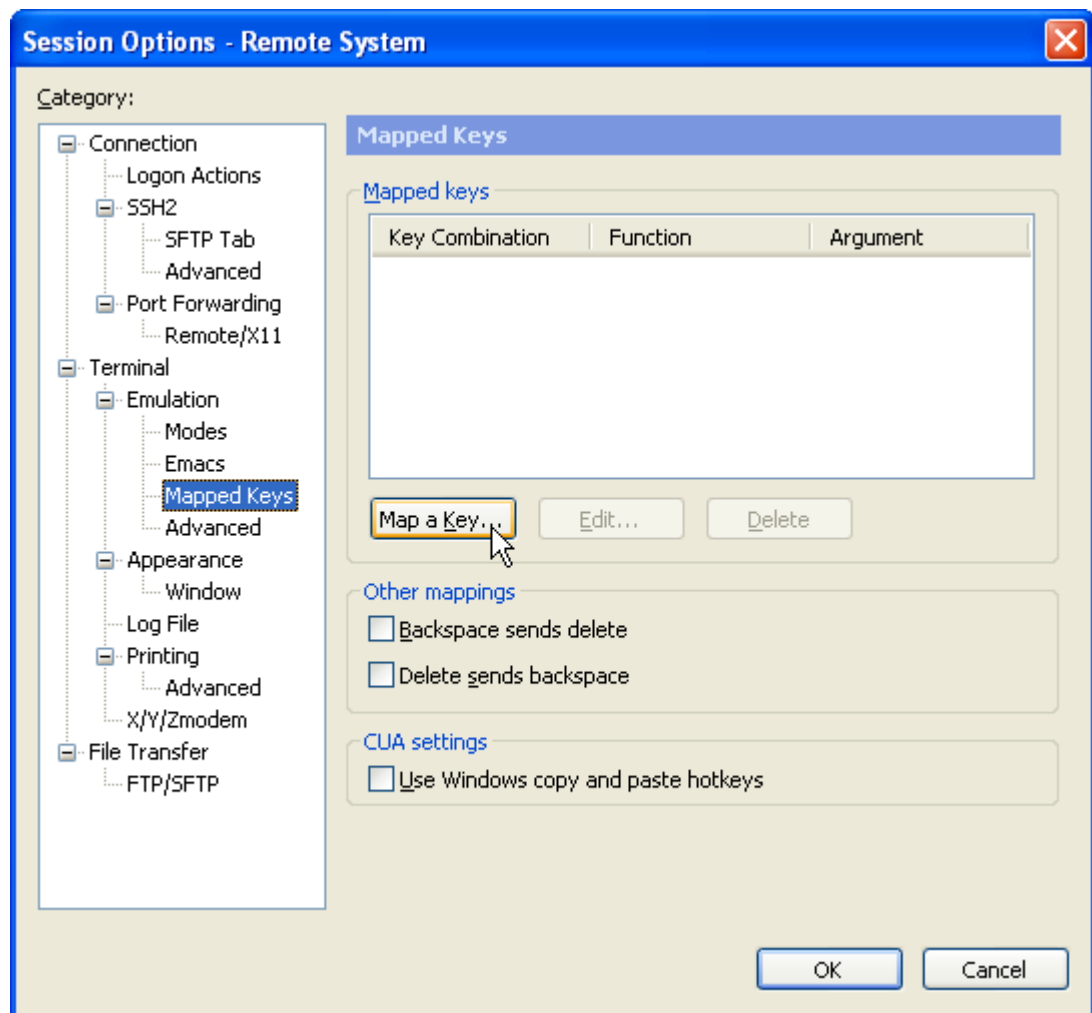


Tip: If you wish to run the same script over and over again, but you aren't ready to map a key or set up a button bar to run the script, you can use the following keyboard shortcut combination within SecureCRT to launch the most recently used script: **Alt + S, 1**.

Mapping a Key to Run a Script

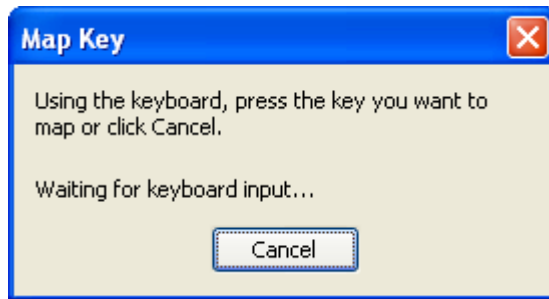
If you are anything like “Johnny Hotkeys”, and you have scripts that perform different functions, each of which you would like to launch with the press of a keyboard combination of their own, mapping a key to run a script might work well for you. Here's the basic approach to mapping a key to run a script:

- 1) Open the **Session Options** dialog and select the **Terminal / Emulation / Mapped Keys** category:

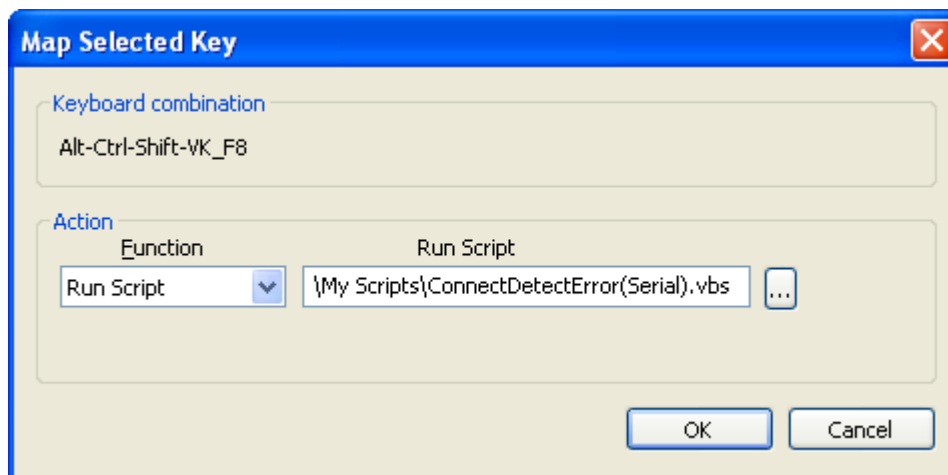


Accessing the Session Options, “Mapped Keys” Category

- 2) Press the **Map a Key...** button and the **Map Key** window will appear.



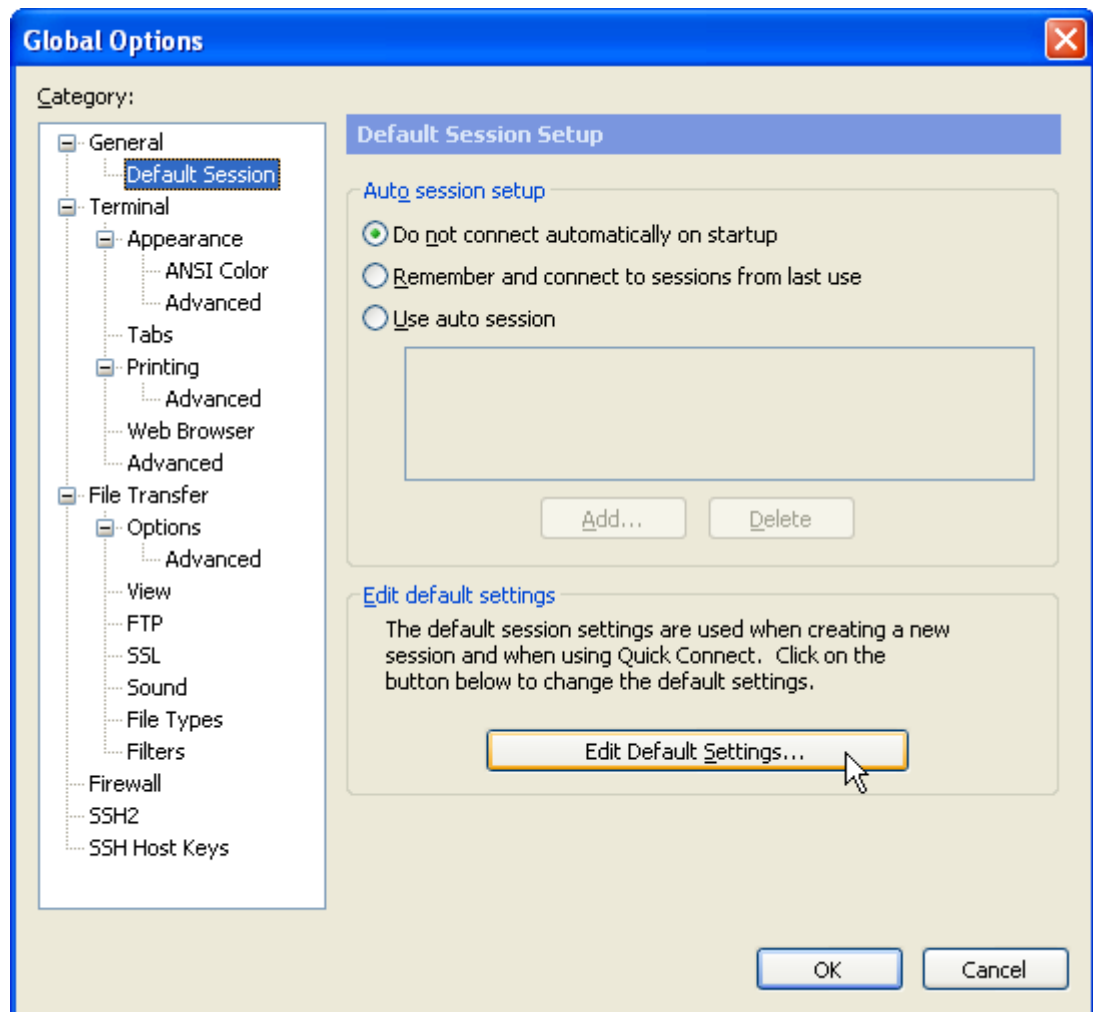
- 3) With the **Map Key** window active, press the keyboard combination you wish to use when launching the script (specifying which script to run is done in the subsequent step).
- 4) Select the **Run Script** action from the **Function** drop-down and then specify the path to the script you wish to launch in the **Run Script** field.



Mapping a Key to Run a Script

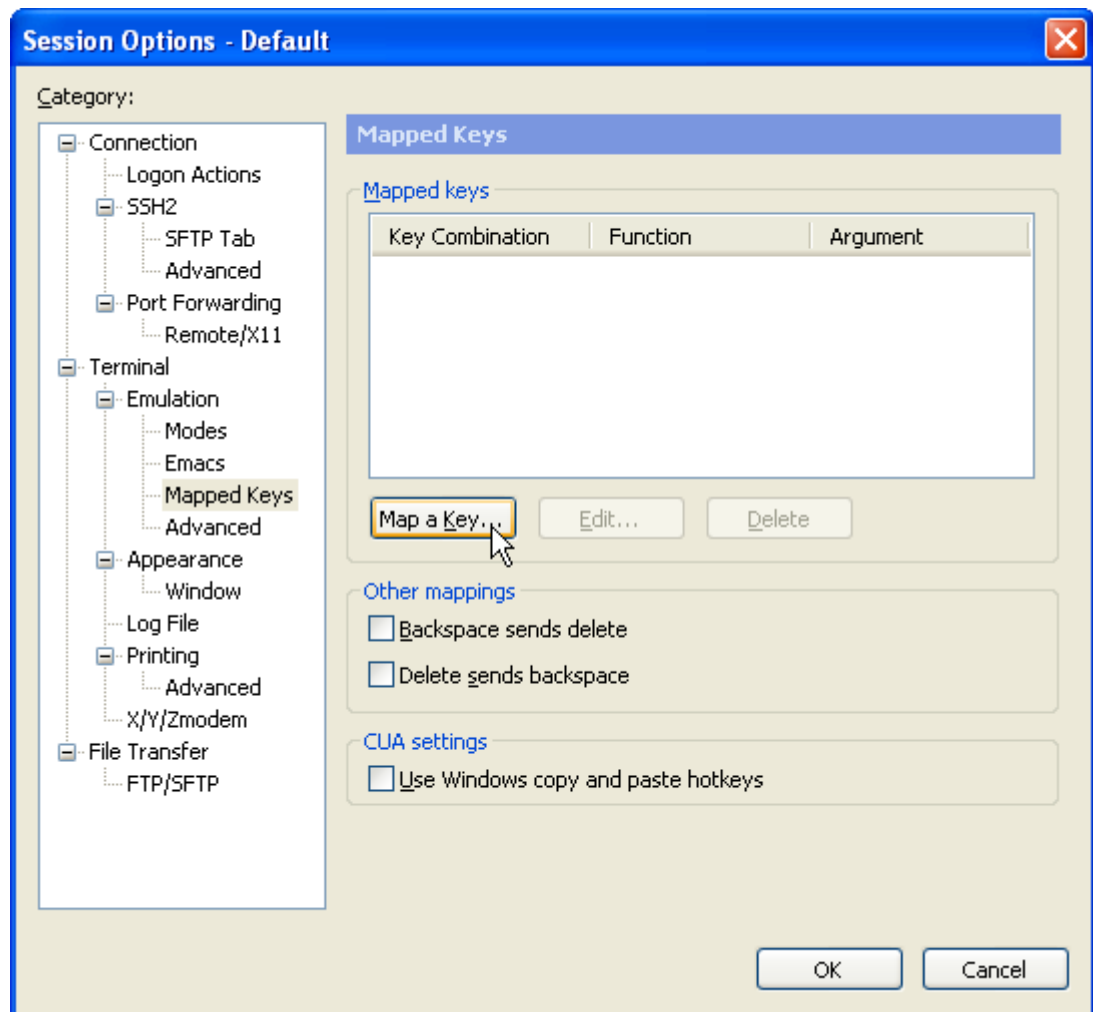
If you want to be able to launch a script using the same mapped key in any session you are connected to in SecureCRT, you'll likely want to define the new mapped keys in your "Default" session, and then apply the change to all your sessions. This task can be accomplished in much the same way as above, with the following differences:

- 1) Bring up the **Global Options** dialog and choose the **General / Default Session** category. Then press the **Edit Default Settings** button and the "Session Options – Default" window will appear, allowing you to select the **Mapped Keys** category for the Default session.



Accessing the “Default” Session

Pressing the **Edit Default Settings...** button will bring up the window depicted in the following graphic.

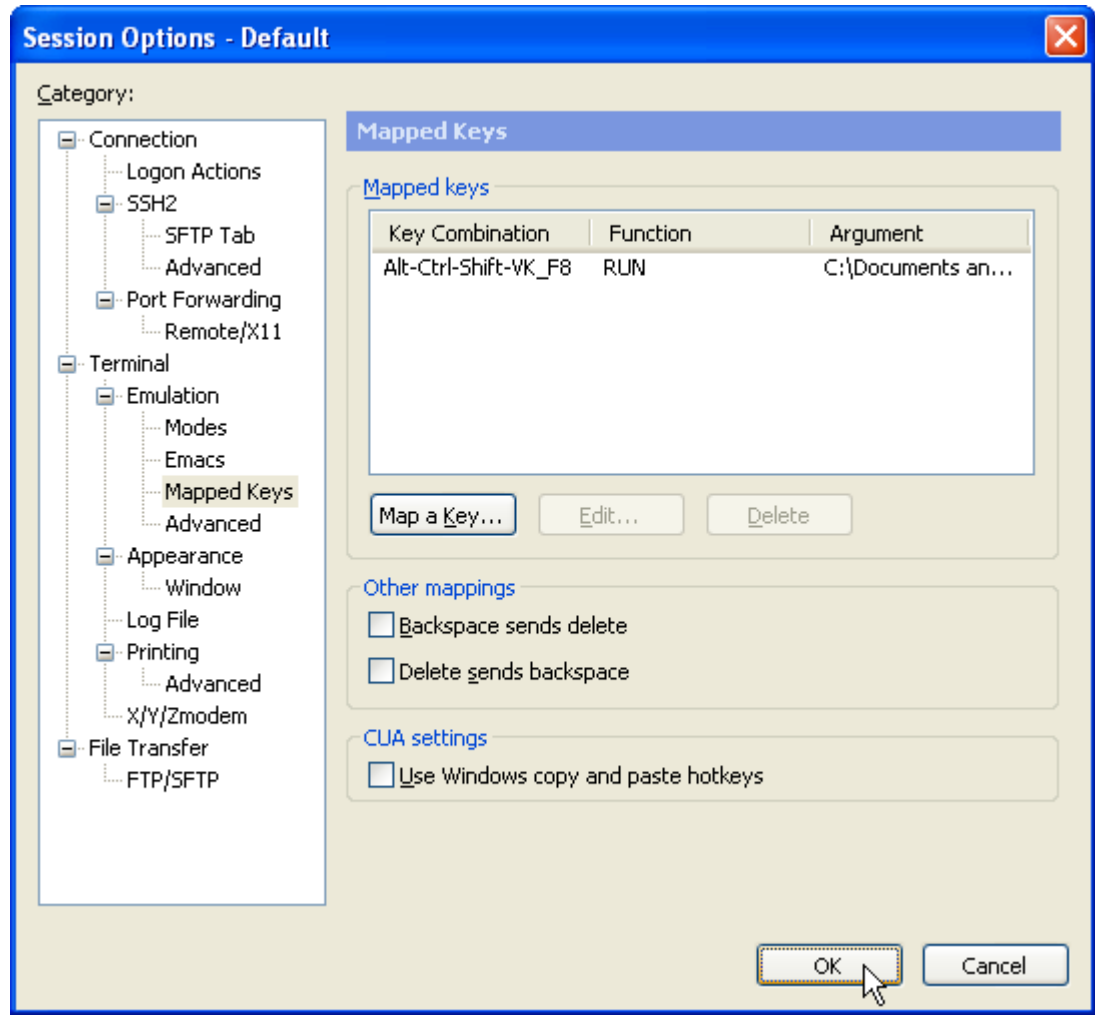


Mapping Keys for the “Default” Session

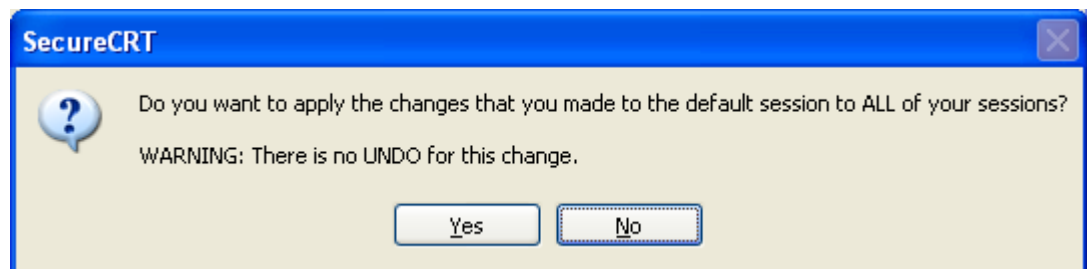
- 2) Make the same configuration settings for mapping the key combination to run the script as indicated above in the prior sequence of steps (1-4 as listed in the previous sequence at the beginning of this section).
- 3) When you close the **Session Options - Default** dialog, you’ll be prompted if you would like to make the change to all sessions.

WARNING: If you choose **Yes** when this prompt is presented, any existing mapped keys already defined in individual saved sessions (excluding `.key` key map files) will be replaced by the mapped key definitions now present in the Default session.

Choose **Yes** here *only* if you have made mapped key definitions you desire to be applied to all existing and future sessions – otherwise, you risk overwriting other session settings with the additional changes that you have made in the Default Session.



Press the OK button to save key map changes to the “Default” session



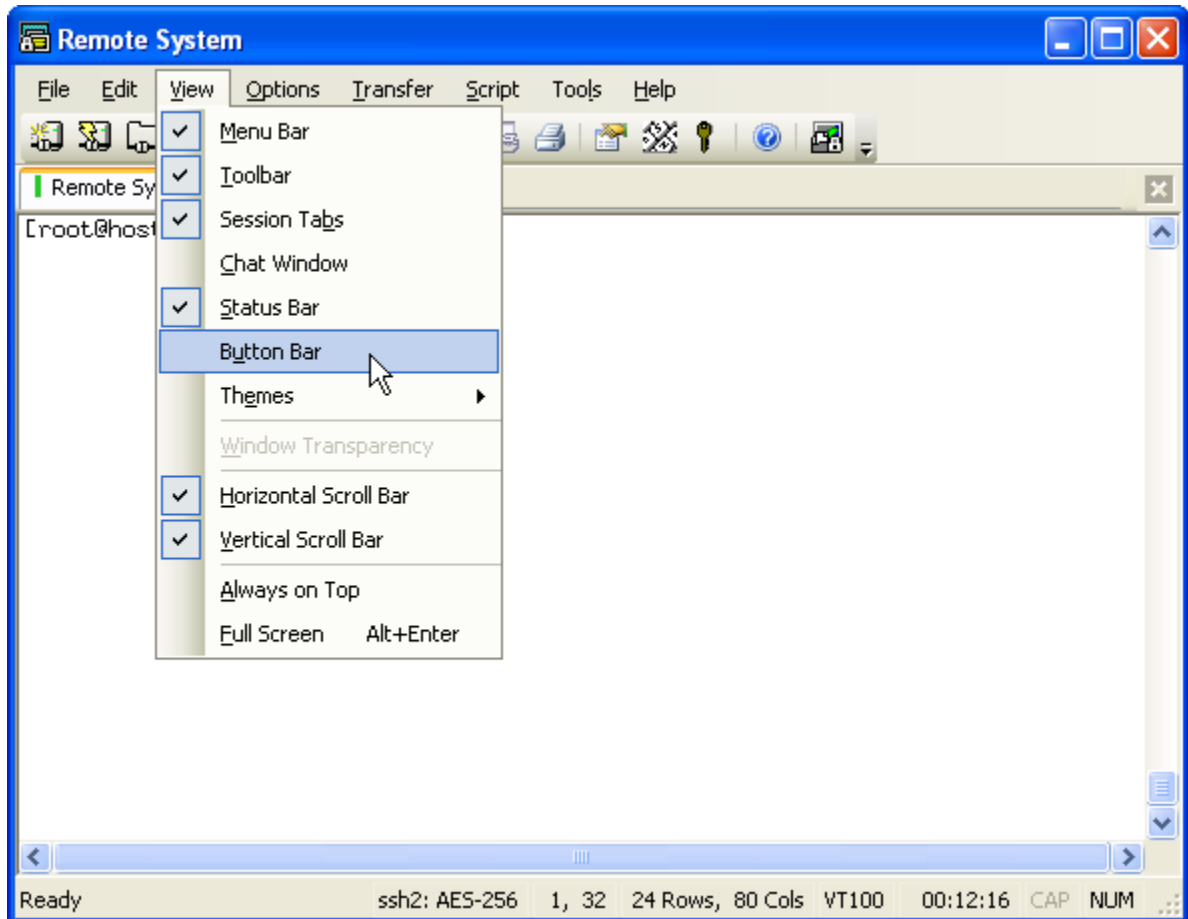
Choose “Yes” *only* if you want the changes made to all existing sessions

Launching a Script from a Button on SecureCRT’s Button Bar

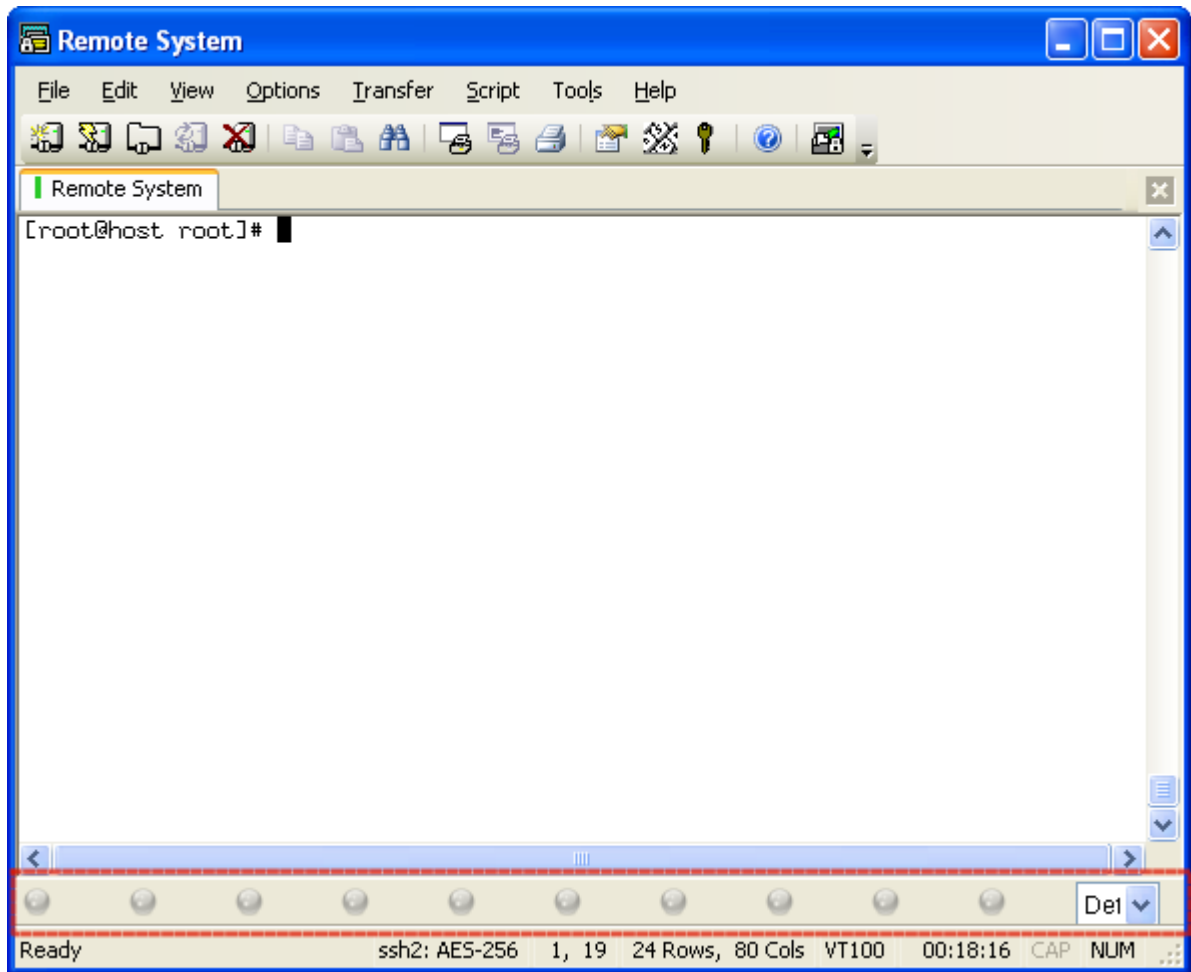
SecureCRT 6.0 and later versions include a customizable button bar feature that a user can employ to create buttons that perform specific actions. Running a script is one of the actions

available for assignment to a customized button on the button bar. This section describes the process of setting up a customized button to run a script.

Before getting started with configuring the button itself, the button bar will need to be enabled if it is not already visible. To enable the button bar in SecureCRT, open the main **View** menu and choose the **Button Bar** menu item:



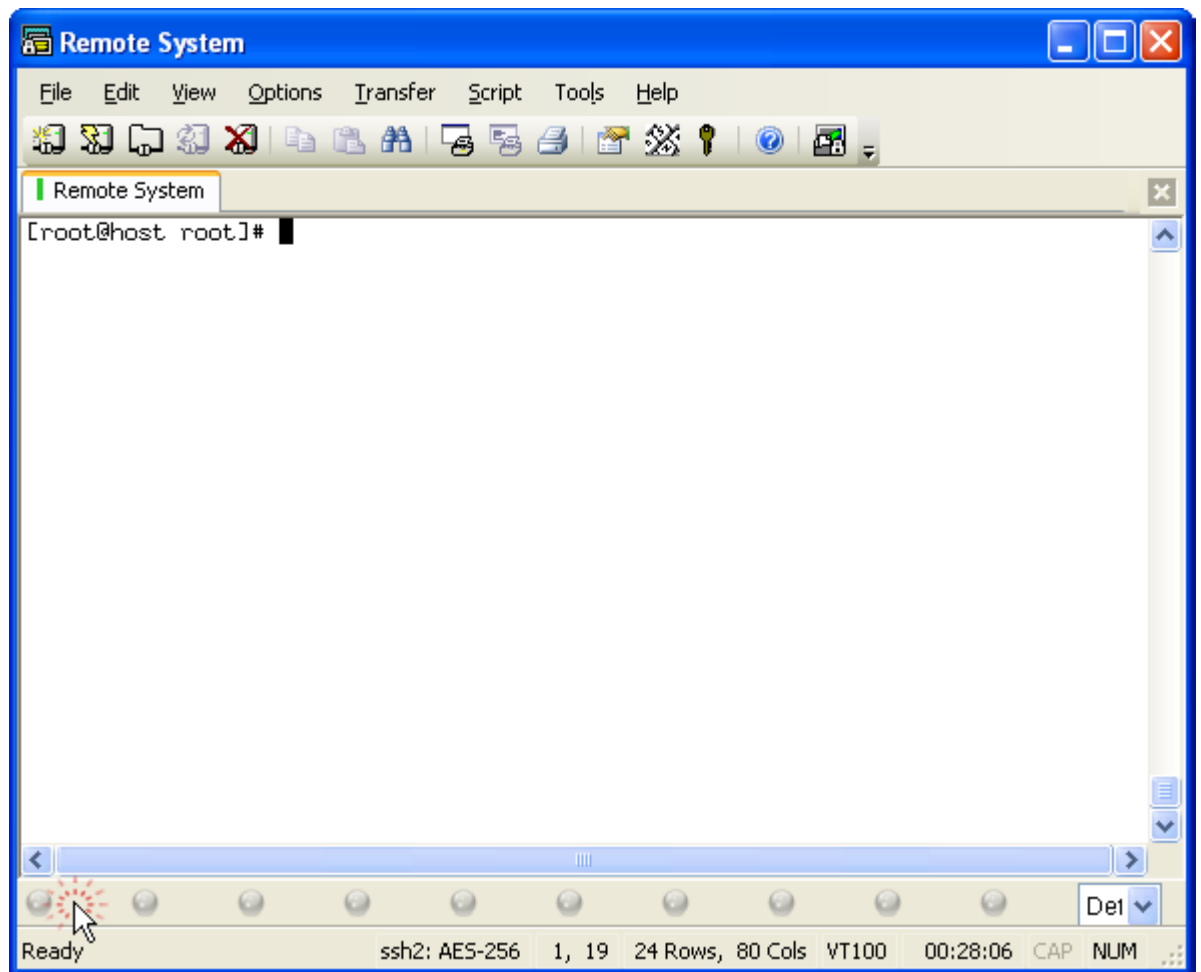
Enabling the Button Bar in SecureCRT



SecureCRT Window with the Button Bar Made Visible

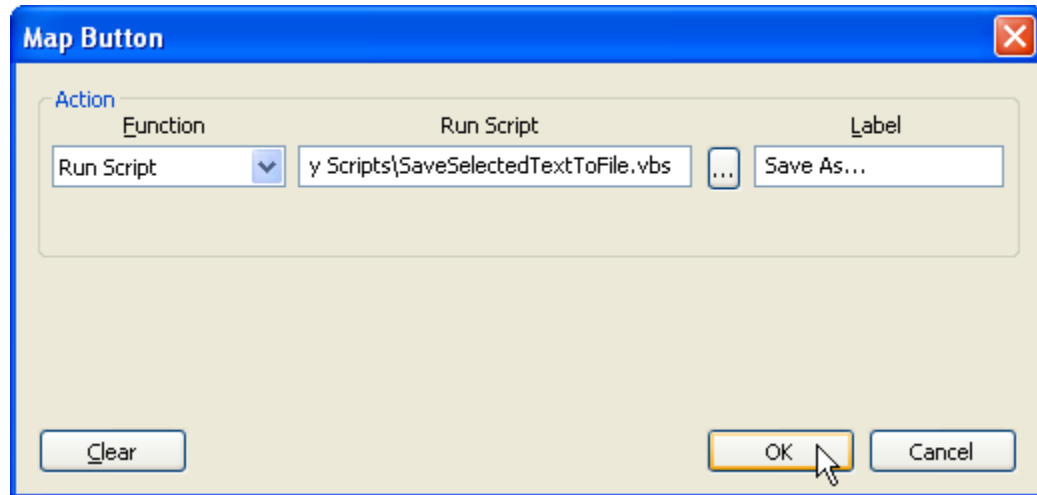
Once the button bar is enabled, any currently undefined button can have an action defined by clicking following these steps:

- 1) Press the undefined button.



Press an Undefined Button to Define an Action

- 2) When the **Map Button** window appears, choose **Run Script** from the **Function** drop-down list.
- 3) In the **Run Script** field, specify the path to the script file you would like to run.
- 4) Provide a name for the button in the **Label** field and press the **OK** button.



Mapping a Button to Run a Script

To modify an existing button's configuration, right-click on the button and choose **Configure...** from the context menu.

Tip: If you run out of buttons for all your scripts or other time-saving functions on the “Default” button bar, SecureCRT allows users to define more than one button bar, and switching between existing button bars is done by using the button bar selection drop-down control on the far right side of the button bar control.

Additional button bars can be created by right-clicking on any existing button, choosing **New** from the context menu, and providing a name for the new button bar.

If in your organization and reorganization of buttons and button bars, you develop stagnant button bars which you want to remove, you can do so by right-clicking on the targeted button bar and choosing **Delete** from the context menu.

2.2 Automated Script Execution

When you've tested your script and are ready to set it up for automated execution, you'll use one of the following methods depending on the goal you are trying to achieve. There are a couple of different ways to automatically launch a script that will run as part of the connection process to automate the logon process and even run a series of commands, if desired. The configuration process for launching a logon script is as simple as making a change to the session configuration or specifying a command line option.

Configuring a Logon Script in a Saved Session

SecureCRT allows a script to be configured as a saved session's “Logon Script”. If from within SecureCRT's **Connect** dialog a saved session with a logon script is used to establish a connection, the specified script will run as part of the connection process.

To set up a script to run as soon as an existing session is used to connect to a remote machine, follow the steps outlined below:

1. If the **Connect** dialog is not already displayed, open the **Connect** dialog by choosing **File / Connect** from SecureCRT's main menu.

2. Right-click on the session you would like to target and choose **Properties** from the context menu.
3. When the **Session Options** window appears, select the **Connection / Logon Actions** category (Note: in versions earlier than SecureCRT 6.1, the category is named **Logon Scripts**).
4. Enable the **Logon script** option, specify the path to the script file you would like to run during the connection process, and press the **OK** button.

If you are launching SecureCRT from the command line or from within a batch file, you can instruct SecureCRT to automatically connect with a saved session. To automatically connect with a saved session, use the `/S <session_path>` command line argument syntax, as in the following example:

```
SecureCRT.exe /S My_Pre-configured_Session_Name
```

Using the /SCRIPT Command Line Option

As previously mentioned, SecureCRT supports command line options for automatically connecting to a remote host using a saved session that you've pre-configured. In addition to using saved sessions, SecureCRT supports making connections in an ad hoc fashion by specifying connection information "on the fly" using appropriate session-specific command line arguments as described in the SecureCRT help.

Automating Script Execution with Saved Sessions

If you haven't already associated a logon script with your pre-configured session as described above (or if you want to use a different logon script specific to the task you will be performing each time you connect with the same session), you can specify a logon script by combining the `/SCRIPT` and `/S` command line arguments. Here's an example that will connect using the pre-configured session named "Redhat #1", and run the script found at "C:\Scripts\BackupCfg.vbs":

```
SecureCRT.exe /SCRIPT "C:\Scripts\BackupCfg.vbs" /S "Redhat #1"
```

Tip: If you use the `/SCRIPT` command line option in conjunction with `/S` for a session that already has a logon script defined within its session options, the script specified on the command line with the `/SCRIPT` option will be launched, rather than the script specified in the saved session options.

Automating Script Execution with Ad Hoc Connections

Ad hoc connections are created with SecureCRT using command line options such as `/SERIAL`, `/TELNET`, `/SSH1`, `/SSH2`, etc. with a command line typically resembling the following pattern:

```
SecureCRT.exe /SSH2 /L myUser /PASSWORD myPassword 192.168.0.1
```

To run a script as part of an ad hoc connection, first use the `/SCRIPT` command line option and provide the path to the script you want to run. Then append all of the usual ad hoc session command line parameters. For example:

```
SecureCRT.exe /SCRIPT C:\Login.vbs /SSH2 /L user /PASSWORD pwd 192.168.0.1
```

Tip: Any options not specified on the command line will be defaulted to values defined in the Default session (**Global Options, General / Default Session** category).

With a session configuration set up to run a Logon Script or `/SCRIPT` command line option as indicated above, the script code specified will automatically be launched as part of the connection process.

Script Execution Timing

The exact timing of the script code execution varies depending on the connection protocol specified for the session or ad hoc connection according to the following table.

Protocol	Script Execution Begins...				
Serial	After opening serial COM port.				
SSH1 & SSH2	<p><i>SecureCRT 6.0.x and earlier:</i> After successful authentication.</p> <p><i>SecureCRT 6.1.x and later:</i> Script execution timing depends on the value of the Display logon prompts in terminal window option (Session Options, Connection / Logon Actions category):</p> <table border="1"> <tr> <td>Disabled:</td> <td>After successful authentication.</td> </tr> <tr> <td>Enabled:</td> <td>After TCP connection is made to SSH server. This allows a script author to automate the authentication process by waiting for and responding to username and password prompts.</td> </tr> </table>	Disabled:	After successful authentication.	Enabled:	After TCP connection is made to SSH server. This allows a script author to automate the authentication process by waiting for and responding to username and password prompts.
Disabled:	After successful authentication.				
Enabled:	After TCP connection is made to SSH server. This allows a script author to automate the authentication process by waiting for and responding to username and password prompts.				
TAPI*	After dialed number answers and a connection is established.				
Telnet	After TCP connection is made and Telnet options are negotiated.				

- * **Tip:** If you intend on scripting a connection that involves dialing to a remote system using the TAPI protocol, please note that the dialing process is outsourced to the Windows operating system where you have no control over progress dialogs and error messages being displayed. If you intend to have a script dialing remote systems and you desire that no user interaction be required to handle error cases, you might want to consider using the Serial protocol to connect to the COM port of your mode and issue the dialing sequence manually using ATDT commands specific to your modem, rather than using TAPI to do the dialing. Dialing with the combination of the Serial protocol connecting to the COM port of the modem and issuing ATDT commands provides the capability to detect dialing results like `NO CARRIER`, and `BUSY`, as well as success cases such as `CONNECTED`.

Passing Arguments to Scripts

When the `/SCRIPT` command line argument is used on the command line to launch SecureCRT itself, script arguments can be specified using one or more `/ARG` command line arguments. Keep in mind the following details when using `/ARG` to pass command line arguments to your script:

- `/SCRIPT` and `/ARG` command line parameters are “standard” command line options (as opposed to “session-specific” command line options) and must be placed on the command line before any session-specific command line arguments when launching SecureCRT.

Consider the example of using a pre-configured session to a jump host and using a script to control the “jumping”. If, after connecting to your jump host using a saved session named “MyJumpHost”, you desire to have the script issue a command to connect to another host based on the arguments provided to your script for “IP address” and “port”. The command line labeled **Incorrect** below might cause some confusing error messages to be displayed. Since `/s` is a “session-specific” command line option, it

must be placed after (to the right of) all “standard” command line options as indicated in the command line labeled **Correct** below:

Incorrect:

```
SecureCRT.exe /S "MyJumpHost" /SCRIPT "C:\ConnectToHost.vbs" /ARG 192.168.0.123 /ARG 23
```

Correct:

```
SecureCRT.exe /SCRIPT "C:\ConnectToHost.vbs" /ARG 192.168.0.123 /ARG 23 /S "MyJumpHost"
```

With the command line options specified in the **Correct** example above, SecureCRT would first connect to the remote host specified in the saved session named “MyJumpHost”, and then launch the “ConnectToHost.vbs” script, passing two arguments to the script.

- Use the `crt.Arguments` script object in SecureCRT to access arguments passed to the script. The `crt.Arguments.Count` property will indicate the number of arguments that were passed to the script. Note in the “ConnectToHost.vbs” sample code below, `crt.Arguments.Count` property is used to determine if sufficient arguments have been passed to the script in order to successfully continue with script execution.

`crt.Arguments(0)` represents the 1st command line argument,
`crt.Arguments(1)` the 2nd,
`crt.Arguments(2)` the 3rd, and so on.

- Order is important when specifying more than one argument. Each `/ARG` option and corresponding value will be assigned an argument index in order from left to right. This means that the arguments must be specified in the same order as expected within the script file. Consider the following section of code from a hypothetical sample “ConnectToHost.vbs” implementation:

```
' First, ensure that sufficient arguments are provided as expected.
If crt.Arguments.Count < 2 Then
    MsgBox "Error: <hostname> and <port> arguments are required."
    Exit Sub
End If

strHost = crt.Arguments(0)
strIP = crt.Arguments(1)
MsgBox "Host arg: " & strHost & vbCrLf & "IP arg: " & strIP
```

Note the code emphasized in **bold** typeface above – the IP address is expected to be the first (left-most) argument – `crt.Arguments(0)`, and the port is expected to be the second argument – `crt.Arguments(1)`.

Chapter 3: Connecting to Remote Machines

Whether the machine you need to access is on the other side of the world, or on the other side of your office, at some point you'll likely need to connect to a remote machine in order to get your job done. This chapter will begin by introducing two of the most common methods used in SecureCRT scripting to connect to one or more devices from within a script: [Connecting with a pre-configured session](#), or [Connecting in an "ad hoc" fashion](#).

Later in this chapter, the process of establishing multiple connections within tabs will be detailed. This discussion will include an example script solution exemplifying how to open a group of sessions in separate tabs within the same instance of SecureCRT.

Although the topic is introduced as "how to *connect* to a remote machine", you might also want to know how to *disconnect* from a remote machine. An example script solution will be included that shows how to implement a client-side automatic "inactivity" disconnect if a certain amount of time has elapsed without any user keyboard activity within SecureCRT.

Within a SecureCRT script, connections to devices or remote machines are established using the **Session** object reference documented within the *ActiveX Scripting / Script Objects Reference* chapter within the SecureCRT Help. There are two methods available with the Session object that can be used to establish these connections: `Connect()` and `ConnectInTab()`. These methods both accept similar parameters containing information SecureCRT needs in order to successfully establish the desired connection. The `ConnectInTab()` method returns a reference to a new Tab object that can be used to further manipulate the newly-created tab in which the connection is made.

The parameters you will need to pass to the `Connect()` and `ConnectInTab()` methods will differ depending on whether you will be using a pre-configured session, or ad hoc connection information.

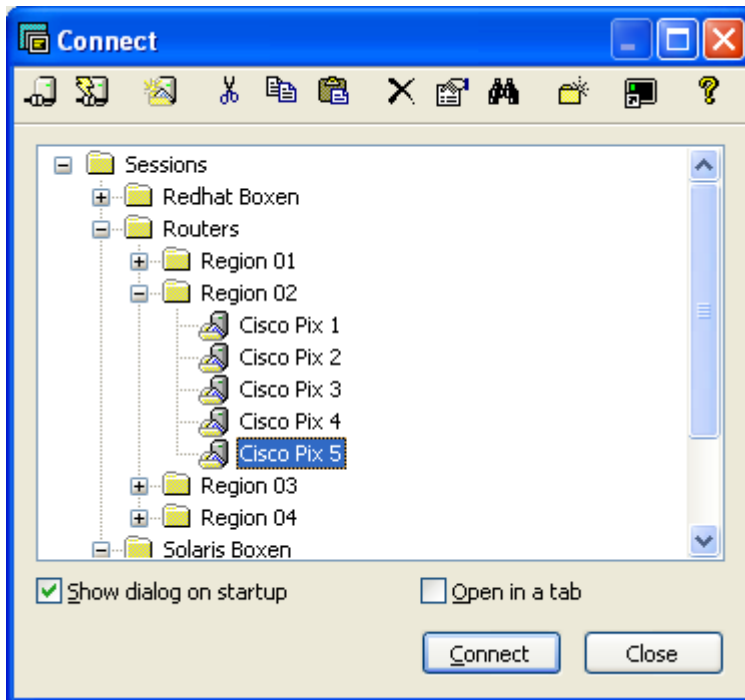
3.1 Connecting with a Pre-configured Session

If you've been using SecureCRT for a while now and have already built up a repository of session configurations you normally use for manual connections, you can use your existing sessions within a script to establish a connection. The syntax for connecting using a pre-configured session within a script follows the pattern SecureCRT provides for making connections when launching SecureCRT from the command line: `/S <session_path>`. For example, the following statement will instruct SecureCRT to establish a connection using a pre-configured session named "Redhat8" located directly beneath the "Sessions" folder in the SecureCRT **Connect** dialog:

```
crt.Session.Connect "/S Redhat8"
```

When using a pre-configured session, the hostname and connection protocol are specified within the session configuration data, as well as any other information needed in order to successfully connect and authenticate to the remote system (such as username and password, for example).

If your session exists within a nested folder one or more levels deep, as it appears within the SecureCRT **Connect** dialog folder tree, the entire "session path" will need to be specified. Consider the session named "Cisco Pix 5" shown selected in the graphic below:



The “Cisco Pix 5” session shown in the graphic above is located within several subfolders, and the name of the session includes spaces. In order to account for these characteristics, the following VBScript statement would be needed in order to properly specify the session path:

```
crt.Session.Connect "/S ""Routers\Region 02\Cisco Pix 5"""
```

TIP: Literal double quote characters need to be *escaped* within strings in VBScript code; one literal double quote within a string is specified with two double quotes: "" (That's two double quote characters, not 4 single quotes or "ticks"). If you find this technique or syntax confusing, you can use the VBScript built-in `chr()` function in combination with VBScript string concatenation to achieve an easier-to-read statement. The ASCII double quote character value is decimal 34, so `chr(34)` is another way of representing the literal double quote character. For example, the two statements below are identical in terms of what each statement accomplishes:

```
crt.Session.Connect "/S " & chr(34) & "Routers\Region 02\Cisco Pix 5" & chr(34)
crt.Session.Connect "/S ""Routers\Region 02\Cisco Pix 5"""
```

3.2 Connecting in an “Ad Hoc” Fashion

Earlier in this document, the process of establishing a connection using SecureCRT's command line options was explained in connection with running a script from the command line. Connections can be established within a SecureCRT script in an ad hoc manner similar to what SecureCRT expects when ad hoc connections are established with sufficient SecureCRT command line parameters. Here is an example of a VBScript statement that will result in using the SSH2 protocol to connect to a machine named "redhat8", and authenticate with a username of "root" and a password of "br8km3N0w!". Note the use of the familiar command line options `/SSH2`, `/L` and `/PASSWORD`:

```
crt.Session.Connect "/SSH2 /L root /PASSWORD br8km3N0w! redhat8"
```

Here is an example of a VBScript statement that will result in using the Serial protocol to connect to COM2 with a baud rate of 38400:

```
crt.Session.Connect "/Serial COM2 /BAUD 38400"
```

Tip: With ad hoc connections, session settings from the Default session are used for any options not specified as parameters to the `Connect()` method. Any options not specified as parameters to the `Connect()` method will be defaulted to values defined in the Default session (**Global Options, General / Default Session** category).

The following table from the SecureCRT Help file shows command line options that are specific to protocols supported by SecureCRT's `Connect()` method. The table also shows common options corresponding to each individual protocol. You can find a more complete table in the *Table of Protocol Specific Command Line Options* topic within the SecureCRT Help file.

Protocol Option (with <i>mandatory</i> and [optional] arguments as indicated)	Related Optional Arguments	Description
<code>/S session_name</code>	N/A	Opens a connection with a pre-existing session found in <i>session_path</i> .
<code>/SSH2 [options] hostname</code>	<code>/I IdentityFile</code> <code>/L Username</code> <code>/PASSWORD password</code> <code>/P port</code>	Opens a connection to <i>hostname</i> using the SSH2 protocol and any specified <i>options</i> . Location of the private key file for publickey authentication. Account name to use for authentication. Password to use for authentication. Port on which SSH2 server is listening for incoming connections.
<code>/Telnet hostname [port]</code>	N/A	Opens a connection to <i>hostname</i> using the Telnet protocol to connect to specified <i>port</i> (<i>port</i> 23 is used if <i>port</i> is not specified).
<code>/Serial port [options]</code>	<code>/BAUD baudrate</code>	Opens a connection using the Serial protocol to the COM <i>port</i> specified. Sets the baud rate to use when connecting to the serial port. The default baud rate is 38400.

3.3 Connecting in Tabs

While some scripting applications may only warrant connecting and disconnecting to various machines in a loop within the same tab, other use cases may involve the need to open up multiple connections, each in separate tabs.

While this document focuses on the scripting aspect, if you simply want to accomplish starting SecureCRT with several sessions in tabs, you can use the `/T` command line option in combination

with multiple `/S <session_path>` specifications to achieve your goal. For example, the following command line will launch SecureCRT and connect to the “Redhat 8”, “Cisco Pix 5”, and “SuseServer” sessions, each session within its own tab in the same SecureCRT window:

```
SecureCRT.exe /T /S "Redhat 8" /S "Cisco Pix" /S "SuseServer"
```

Opening multiple tabs can be accomplished in a variety of ways with a SecureCRT script:

- The **Session** object’s `ConnectInTab()` method
- The **Tab** object’s `Clone()` and `ConnectSFTP()` methods
- The **SessionConfiguration** object’s `ConnectInTab()` method

Opening a Connection in a New Tab with `session.ConnectInTab()`

The `Session.ConnectInTab()` and `SessionConfiguration.ConnectInTab()` methods require the same parameters required by the `Session.Connect()` method described earlier in this chapter.

In contrast to the `Session.Connect()` method, both the `Session.ConnectInTab()` and `SessionConfiguration.ConnectInTab()` methods result in the creation of a new tab. In addition, a reference to a tab object is returned by each method, allowing further control of the tab in which the connection attempt is made.

Here is an example of using the `Session.Connect()` method to establish connections to three different servers, each in a separate tab.

```
' Connect to three separate sessions each in its own tab.
Dim objTab1, objTab2, objTab3
Set objTab1 = crt.Session.ConnectInTab("/S "Redhat Boxen\Redhat 8")
objTab1.Caption = "Redhat 8"

Set objTab2 = crt.Session.ConnectInTab("/S "Routers\Cisco Pix 506E")
objTab2.Caption = "Cisco Pix"

Set objTab3 = crt.Session.ConnectInTab("/S "Solaris Boxen\Solaris 10")
objTab3.Caption = "Solaris 10"
```

Cloning an Existing Tab with `Tab.Clone()`

When working within the SecureCRT GUI, a tab can be "cloned" by right-clicking on the session tab and selecting the **Clone Session** option or by opening the **File** menu and selecting the **Clone Session** option. This will create a session connection that is identical to the current session. If the original session is using the SSH2 protocol, the transport is shared and re-authentication is not needed. For other protocols (except Serial and TAPI, which cannot be cloned), authentication is required.

To clone a tab within a SecureCRT script, you perform the following steps:

- 1) Get a reference to the tab object you wish to clone.
- 2) Call the `Clone()` method using the tab object reference acquired in step 1.

Here is an example snippet of VBScript code that shows how tab cloning is done:

```
Dim objTab, objTabClone1, objTabClone2
Set objTab = crt.Session.ConnectInTab("/S "Redhat Boxen\Redhat 8")
objTab.Clone objTabClone1
objTab.Clone objTabClone2
```

```

' Clone the SSH2 main tab once
Set objTabClone1 = objTab.Clone
objTabClone1.Caption = "Redhat 8 - Cloned Tab"

' Now clone the cloned tab
Set objTabClone2 = objTabClone1.Clone
objTabClone2.Caption = "Redhat 8 - Clone of a Clone"

```

Opening an SFTP Tab with `Tab.ConnectSFTP()`

When working within the SecureCRT GUI, if you have an SSH2 session connected, you can open an SSH file transfer protocol (SFTP) command line file transfer session using the same transport as the existing SSH2 session either by opening the SecureCRT **File** menu and selecting the **Connect SFTP Tab** option, or by right-clicking on the SSH2 session's tab and choosing **Connect SFTP Tab** from the tab context menu.

An SFTP tab can be opened within SecureCRT script using the `ConnectSFTP()` method associated with a reference to a tab object. Here is an example script showing a plausible file transfer automation using an SFTP tab:

```

Dim objTab, objSFTPTab
Set objTab = crt.Session.ConnectInTab("/S " & "Redhat Boxen\Redhat 8")
objTab.Caption = "Redhat 8 - Main"

' Create an SFTP tab associated with the existing tab
Set objSFTPTab = objTab.ConnectSFTP

' Wait for the SFTP tab to be ready for input.
objSFTPTab.Screen.Synchronous = True
crt.Sleep 1000
objSFTPTab.Screen.Send vbcr
objSFTPTab.Screen.WaitForString "sftp>"

' Upload all .txt files in the current local working directory to
' the current remote working directory.
objSFTPTab.Screen.Send "put *.txt" & vbcr
objSFTPTab.Screen.WaitForString "sftp>"

' Close the SFTP tab
objSFTPTab.Close

' Close the connection on the originating SSH2 connection's tab
objTab.Session.Disconnect

```

Note: The SFTP tab is only available for SSH2 connections. Attempting to open an SFTP tab with the `ConnectSFTP()` method using a tab connected via a non-SSH2 protocol will result in a scripting error.

Solution: Open a Group of Sessions in Tabs

This code solution shows how to open a group of tabs connected to different sessions each within the same SecureCRT window.

```

' 1) Connect to three separate sessions:
'    - Production machine: "Redhat 8 - Production"

```

```

' - Testing machine: "Redhat 8 - Testing"
' - Development machine: "Redhat 8 - Development"
'
' 2) Change working directory to match target machine first step after connection
'
' 3) Name each tab appropriately

Dim objProdTab, objTestTab, objDevTab

Set objProdTab = crt.Session.ConnectInTab("/S "Redhat Boxen\Redhat 8 - Production")
objProdTab.Screen.Synchronous = True
objProdTab.Screen.WaitForString "]"$, 5
objProdTab.Screen.Send "cd production" & vbcr
objProdTab.Caption = "Production"

Set objTestTab = crt.Session.ConnectInTab("/S "Redhat Boxen\Redhat 8 - Testing")
objTestTab.Screen.Synchronous = True
objTestTab.Screen.WaitForString "]"$, 5
objTestTab.Screen.Send "cd testing" & vbcr
objTestTab.Caption = "Testing"

Set objDevTab = crt.Session.ConnectInTab("/S "Redhat Boxen\Redhat 8 - Development")
objDevTab.Screen.Synchronous = True
objDevTab.Screen.WaitForString "]"$, 5
objDevTab.Screen.Send "cd development" & vbcr
objDevTab.Caption = "DEV"

```

3.4 Disconnecting Active Connections

Disconnecting an active connection is done through the `Session.Disconnect()` method. Example situations in which it is either desired or necessary to perform a disconnect operation include:

- When connecting to a series of target hosts within a single tab and performing a set of operations on each host, you must first disconnect any existing connection before you can successfully connect to another host within the same tab. Consider the following code fragment showing how to disconnect any currently established connection prior to connecting to the next machine:

```

' Make sure we are disconnected before attempting a connection
If crt.Session.Connected Then crt.Session.Disconnect

' ~~~~~
' Connect to host 192.168.0.1
crt.Session.Connect "/SSH2 /L user /PASSWORD p4$$w0rd 192.168.0.1"
' Do work on the remote machine.
...
' Disconnect before moving on to the next host
crt.Session.Disconnect

' ~~~~~
' Connect to host 192.168.0.2
crt.Session.Connect "/SSH2 /L user /PASSWORD vuln3r4ble 192.168.0.2"
' Do work on the remote machine.
...
' Disconnect before moving on to the next host
crt.Session.Disconnect

' ~~~~~
' Connect to host 10.0.225.11
crt.Session.Connect "/SSH2 /L user /PASSWORD g0ldpl8ted 10.0.225.11"

```



```

' Do work on the remote machine.
...
' Disconnect before moving on to the next host
crt.Session.Disconnect

.
.
.

```

- The user may want to automatically disconnect if no activity has occurred within a specified period of time (idle disconnect). The **Solution** section below details a fully functional auto disconnect script example.

Solution: Automatically Close an Inactive Connection (Auto Disconnect)

Many users want to ensure that they stay connected forever (or as long as possible) to a remote machine. However, if you are in an environment that incurs cost by the number of minutes connected, or if being connected to a remote machine precludes other connections from being made successfully to the server, you may desire to automatically close a connection if it hasn't been used for a specified period of time. The following script code exemplifies an auto logout solution using the `crt.Screen.WaitForKey()` method to detect when user input occurs, resetting a timeout loop until a key press isn't detected for a specified period of time.

```

#$Language="VBScript"
#$Interface="1.0"
' AutoTimeout.vbs
'
' Description: This example script checks activity (screen
'              output) and closes SecureCRT if "idle" for
'              more than the specified number of minutes.

Dim g_nTimeoutMinutes

' For testing purposes, this value is initially set to 15
' seconds (1/4 minute) Once you've tested that the script
' works successfully to disconnect your inactive session
' after 15 seconds, set the g_nTimeoutMinutes value to the
' desired number of idle minutes that would occur prior to
' disconnecting the session.
g_nTimeoutMinutes = .25

' Using GetScriptTab() will make this script 'tab safe' in
' that all of the script's functionality will be carried out
' on the correct tab. From here on, the objTab object will
' be used instead of the 'crt' object.
Dim g_objTab
Set g_objTab = crt.GetScriptTab
g_objTab.Screen.Synchronous = True

'~~~~~
Sub Main()
' Detecting key presses is likely the best way to determine
' if user activity is occurring. Each time the user presses
' a key, the "timer" will be reset, waiting for the timeout
' period to elapse.
Do
' WaitForKey takes seconds, so convert from g_nTimeoutMinutes to
' seconds

```



```

Loop While g_objTab.Screen.WaitForKey(g_nTimeoutMinutes * 60)

' We will not get to this line unless the loop above is
' terminated (which will happen as soon as we timeout
' waiting for any key to be pressed)
g_objTab.Session.Disconnect
End Sub

```

3.5 Connecting to a List of Remote Machines Within a Loop

In the example code provided in the [Disconnecting Active Connections](#) section above, connections were done in a sequential method, repeating the same code (with slight modifications) for each target machine. Even if there are only two or three machines being targeted for connections within a script, you will likely want to create a code loop in which to iterate through each target machine, connect, issue commands, and then disconnect. The example code below shows this pattern of *connect* → *do_work* → *disconnect* using an array (`vHosts`) to store the remote devices to which connections are established.

```

Dim vHosts(100)
vHosts(0) = "192.168.0.1"
vHosts(1) = "192.168.0.2"
vHosts(2) = "10.0.100.50"
.
.
.

For Each strHost In vHosts
  If strHost = "" Then Exit For

  ' Make sure we are disconnected before attempting a connection
  If crt.Session.Connected Then crt.Session.Disconnect

  ' Connect to the next host
  crt.Session.Connect "/SSH2 /L user /PASSWORD p4$$w0rd " & strHost

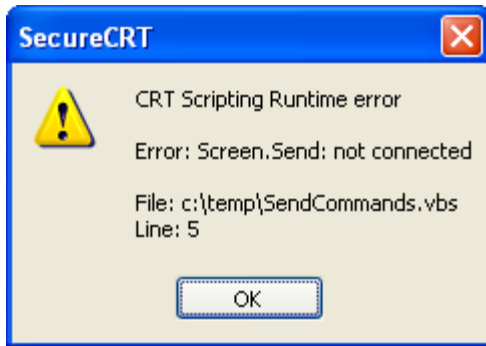
  ' Do work on the remote machine.
  .
  .
  .

  ' The Disconnect() action is done at the top of the loop (in the event
  ' that the tab in which this script is launched already has an
  ' active connection).
Next

```

3.6 Handling Connection Failures Within a Script

If there's a problem successfully connecting to a remote machine from within a script, by default, SecureCRT will halt script execution and provide a notification of the failure using a message box similar to what you see below.



This notification is great if you are debugging your script, or when a script is running within an interactive SecureCRT window, because it halts script execution and provides you with specific information about the script error – including the line of code on which the problem occurred.

However, if your goal is to automate a series of tasks to multiple remote machines, you likely do not want such connection errors to result in pop-up messages halting your script. You may want to handle the error in some other way (log it to an error file or some log file) that will allow your script to move on to the remaining machines.

The mechanism for allowing errors to occur without halting script execution in VBScript code is the `On Error Resume Next` statement which instructs the script host to allow any run-time errors to be handled within the script.

It is important to recognize that the `On Error Resume Next` statement enables error handling for the procedure in which the statement is made. If the `On Error Resume Next` statement is made globally (outside of any function or subroutine) all run-time errors will be suppressed, potentially masking errors other than those you intend to handle yourself.

You can resume normal run-time error handling by using the `On Error Goto 0` statement, which is VBScript's unique way of allowing your script to tell the host, "OK. I'd like you, VBScript host engine, to start handling errors as you normally do from here on."

If you are connecting to multiple machines from within a loop, the following pattern can be used to handle connection errors:

```
' Instruct the script host we want to handle errors ourselves, now:
On Error Resume Next

' Attempt to connect to the remote machine:
crt.Session.Connect "/SSH2 /L user /PASSWORD vuln3r4ble 192.168.0.2"

' Capture error code and description (if any)
nError = Err.Number
strErr = Err.Description

' Now, tell the script host that it should handle errors as usual now:
On Error Goto 0

If nError <> 0 Then
    ' Handle the error (log to a file, etc.)
    .
    .
    .
```

```

Else
    ' Do work on the remote machine.
    .
    .
End If

```

The pattern outlined above can be applied to a function which can be called within a script, resulting in a true or false return value from a newly-created `Connect()` function as demonstrated in the code below:

```

#$Language="VBScript"
#$Interface="1.0"
' Connect-DetectErrorConnecting.vbs

Dim g_strError

Sub Main()
    Dim nResult, strConnectInfo

    ' ~~~~~
    ' Example of host not found
    strConnectInfo = "/TELNET host_not_found"
    nResult = Connect(strConnectInfo)
    If nResult <> 0 Then
        ' Instead of displaying a message to the user, you would maybe
        ' write an error code to a file... MsgBox is just for sample
        MsgBox "Error connecting with info: " & strConnectInfo & _
            vbCrLf & vbCrLf & g_strError
    Else
        ' Connection was successful.
        ' Do the required work
        ' . . .

        ' Now that the work is done, disconnect from the remote...
        crt.Session.Disconnect
    End If
End Sub

' ~~~~~
Function Connect(strConnectInfo)
    ' Workaround that uses "On Error Resume Next" VBScript directive to detect
    ' Errors that might occur from the crt.Session.Connect call and instead of
    ' stopping the script with an unrecoverable error, allow for error handling
    ' within the script as the script author desires.

    g_strError = ""

    ' Turn off error handling before attempting the connection
    On Error Resume Next
        crt.Session.Connect strConnectInfo
        ' Capture the error code and description (if any)
        nError = Err.Number
        strErr = Err.Description
    ' Restore normal error handling so that any other errors in our
    ' script are not masked/ignored
    On Error Goto 0

```

```
Connect = nError
If nError <> 0 Then
    g_strError = strErr
End If
End Function
```

Chapter 4: Reading Data from Remote Machines

This chapter details techniques that relate to waiting for and capturing specific data upon arrival from a remote machine. Some scenarios call for waiting for a command prompt to make sure the remote machine is “ready” for commands to be sent. Other scenarios might require not only waiting for specific data to appear, but also capturing and storing data from the remote machine.

4.1 Accessing Selected Text on the Screen

Perhaps the most straightforward way to “capture” data received from the remote machine is by selecting the desired text as it appears within the terminal screen in SecureCRT. The selected text can be accessed by using the `crt.Screen.Selection` object, a read-only object that returns the text selected on the screen. See the example solution section below for a script that accesses the selected text and performs a web search using the selection.

Solution: Performing a Web Search with Selected Text

This example scripting solution shows how one can quickly do a web search using text selected in the terminal window using the popular Google search engine.

```
# $language = "VBScript"
# $interface = "1.0"
' GoogleSelectedText.vbs
'
' Description:
'   When this script is launched, the text selected within the terminal
'   window is used as the search term for a web search using google.com.
'   This script demonstrates capabilities only available in SecureCRT 6.1
'   and later (Screen.Selection property).
'
' Demonstrates:
'   - How to use the Screen.Selection property in SecureCRT 6.1 and later
'     to get access to the text selected in the terminal window.
'   - How to use the WScript.Shell object to launch an external application.
'   - How to branch code based on the version of SecureCRT in which this script
'     is being run.

Sub Main()
    ' Extract SecureCRT's version components to determine how to go about
    ' getting the current selection (version 6.1 provides a scripting API
    ' for accessing the screen's selection, but earlier versions do not)
    strVersionPart = Split(crt.Version, " ")(0)
    vVersionElements = Split(strVersionPart, ".")
    nMajor = vVersionElements(0)
    nMinor = vVersionElements(1)
    nMaintenance = vVersionElements(2)

    If nMajor >= 6 And nMinor > 0 Then
        ' Use available API to get the selected text:
        strSelection = Trim(crt.Screen.Selection)
    Else
        MsgBox "The Screen.Selection object is available" & vbCrLf & _
            "in SecureCRT version 6.1 and later." & vbCrLf & _
            vbCrLf & _
            "Exiting script."
        Exit Sub
    End If

    ' Now search on Google for the information.
    g_strSearchBase = "http://www.google.com/search?hl=en&q="
```

```

Set g_shell = CreateObject("WScript.Shell")

' Instead of launching Internet Explorer, we'll run the URL, so that the
' default browser gets used :).
If strSelection = "" Then
    g_shell.Run chr(34) & "http://www.google.com/" & chr(34)
Else
    g_shell.Run chr(34) & g_strSearchBase & strSelection & chr(34)
End If
End Sub

```

4.2 Waiting for Specific Data to Arrive

“Seek first to understand, and then to be understood.”

- Steven Covey, 7 Habits of Highly Effective People

If the task you are attempting to automate involves sending a series of commands to the remote machine, keep the following “best practice” in mind: First make sure the remote machine is ready to receive commands before you actually send them.

In the [Creating Scripts](#) chapter presented earlier, the concept of [recording a script](#) was introduced. You may notice that [the resulting recorded script code](#) included in this earlier chapter exemplifies this best practice, and almost every instance of sending a carriage return (`chr(13)`) is followed up with a call to `WaitForString()` prior to issuing a subsequent call to `Send()`.

SecureCRT provides a few methods that are designed to provide a way to halt script execution until specified text arrives from the remote machine. Two of the most commonly-used functions of this nature are `WaitForString()`, and `WaitForStrings()`.

WaitForString()

The `WaitForString()` method's syntax adheres to the following pattern:

```
[ result = ] object.WaitForString string [, timeout]
```

Object is always a reference to the `Screen` object associated with the connection to the remote machine from which the data is being received. Here are a few examples with explanatory comments:

```

' Example #1
' Wait for text using the screen associated with the
' connection within the tab in which the script started:
crt.Screen.WaitForString "pixfirewall>"

' Example #2
' Wait for text in the screen associated with the
' connection in the 2nd tab from the left:
Set objTab = crt.GetTab(2)
objTab.Screen.WaitForString "pixfirewall>"

' Example #3
' Wait for text in the screen associated with the
' connection that was just established in a new
' tab using crt.Session.ConnectInTab:
Set objTab = crt.Session.ConnectInTab("/Serial COM2 /BAUD 9600")
objTab.Screen.WaitForString "pixfirewall>"

```

If a *timeout* parameter is not specified, `WaitForString()` will wait indefinitely until the text specified in the *string* parameter is detected in the output coming from the remote machine. This means that if the text is never found, the script will run forever (or until SecureCRT is closed, or **Cancel** is chosen from SecureCRT's main **Script** menu).

If a *timeout* parameter is specified, `WaitForString()` will wait until either the text specified in the *string* parameter is found coming from the remote, or until the number of seconds specified in the *timeout* parameter has elapsed. If the text in *string* is not found before the time expires, the result of the `WaitForString()` method will be 0 (or `False`). Otherwise, it will be -1 (or `True`, since any non-zero value in VBScript is treated as `True`).

Using a *timeout* parameter is useful in situations where you aren't certain that the string specified will be sent from the remote machine (or you want to guard against a coding error in your script), and you don't want the script to run indefinitely. Here is a brief example of using `WaitForString()` in combination with a *timeout* parameter to detect when the specified time has elapsed without seeing the specified *string*.

```
Dim nResult
nResult = crt.Screen.WaitForString("pixfirewall>", 10)
If nResult = 0 Then
    MsgBox ""pixfirewall>" prompt not found in 10 second " & _
        "timeout period specified."
Else
    MsgBox ""pixfirewall>" prompt was found."
End If
```

WaitForStrings()

The `WaitForStrings()` method functions very similar to the `WaitForString()` method, with the exception that `WaitForStrings()` supports passing in 1 or more string arguments to detect.

The `WaitForStrings()` method's syntax is as follows:

```
[ result = ] object.WaitForStrings(string1, [string2, ..., stringn] [, timeout])
```

Instead of passing in a list of strings as arguments, an array of strings can be passed in as the first argument, followed by an optional timeout argument. For example:

```
Dim vWaitFors
vWaitFors = Array("File not found", _
    "Access denied", _
    "Permission denied")

' Start a loop, waiting for any one of the target strings
' within 10 seconds. If a timeout occurs, exit the loop
Do
    Dim nResult
    nResult = crt.Screen.WaitForStrings(vWaitFors, 10)

    Select Case nResult
        Case 0
            MsgBox "Timed out waiting for strings!"
            Exit Do
```

```

    Case 1
        MsgBox "Found ""File not found"" string!"

    Case 2
        MsgBox "Found ""Access denied"" string!"

    Case 3
        MsgBox "Found ""Permission denied"" string!"
End Select
Loop

```

An expanded sample script based on the code above is presented below.

Solution: Receive Notification when “Error-Indicating” Text Appears

```

#language = "VBScript"
#interface = "1.0"
' NotifyOnOutput.vbs
'
' Description:
' This example script shows how to use WaitForStrings()
' in combination with a Case..Select statement in
' VBScript to provide various forms of notification when
' specified text from the remote is detected.

Sub Main()
' This script is always running in the background until
' "Cancel" is selected from the Script menu.
Do
    Dim nIndex
    ' Some example trigger strings... modify these, add
    ' more, etc... to match your use pattern
    nIndex = crt.screen.WaitForStrings( _
        "File not found", _
        "Access denied", _
        "Permission denied")

    ' The WaitForStrings() function returns the index of
    ' the string it found, so we could even use
    ' different notifications based on the output.
    Select Case nIndex
        Case 1
            ' Found "File not found"
            ' Activate the tab on which the script is
            ' running before displaying the message box.
            ' Otherwise, the end user will be missing
            ' some context :).
            crt.GetScriptTab.Activate
            crt.Dialog.MessageBox("Detected ""File not found"" error")

        Case 2
            ' Found "Access denied"
            ' In this case, we'll hide the window for a
            ' second, then show it again.
            crt.Window.Show 0
            crt.Sleep 1000
            crt.Window.Show 1

        Case 3

```



```

        ' Found "Permission denied"
        ' Let's be tricky/cheesy and play a sound
        ' file for this notification
Set myShell = CreateObject("WScript.Shell")
Set myFso = CreateObject("Scripting.FileSystemObject")
myShell.Run "RunDll32.exe " & _
        myFso.GetSpecialFolder(1) & "\msdxm.ocx,RunDll " & _
        "/play /close C:\Windows\Media\Notify.wav"

Case Else
    ' This is where you might consider handling
    ' for example, an unexpected index, or a
    ' timeout if it occurs (a time-out case
    ' would of course require that such a
    ' parameter be added to the WaitForStrings()
    ' call above).

End Select
Loop
End Sub

```

Avoid "Missing" Data with `Screen.Synchronous = True`

In order to avoid the potential for `WaitForString()` and its related methods to miss data that is sent from the remote while other code within your script is being executed, it's important to know about the `Synchronous` property associated with the `Screen` object. You may have seen `Screen.Synchronous = True` statements appear within some of the sample code you've seen earlier in this document; this section attempts to explain when it would be important to use such statements within your code.

In order to demonstrate how data might be "missed" if the `Synchronous` property is set to `False`, an example code segment is given below, followed by a series of screen-shots walking through the code's execution. The code is initially crafted so that the `Synchronous` property is explicitly set to `True`, and then later modified to `False` to demonstrate data being "missed".

```

' Set Synchronous to True so that we don't miss any data
crt.Screen.Synchronous = True

Sub Main()

    ' Drive the initial SSH handshake process by connecting
    ' to an SSH server, port 22 using the telnet protocol
crt.Session.Connect "/telnet localhost 22"

    If crt.Screen.WaitForString("SSH", 5) Then
        MsgBox "Found " & "SSH" & ""
    Else
        MsgBox "Timed out waiting for " & "SSH" & ""
        Exit Sub
    End If

    ' Send a legitimate client IDENT string as per SecSSH RFC
crt.Screen.Send "SSH-2.0-" & vbcr

    ' Now, let's explore waiting for various strings as they
    ' would appear from the server
    If crt.Screen.WaitForString("diffie-hellman-group1", 5) Then
        MsgBox "Found " & "diffie-hellman-group1" & ""
    Else
        MsgBox "Timed out waiting for " & "diffie-hellman-group1" & ""
    End If
End Sub

```

```

        Exit Sub
    End If

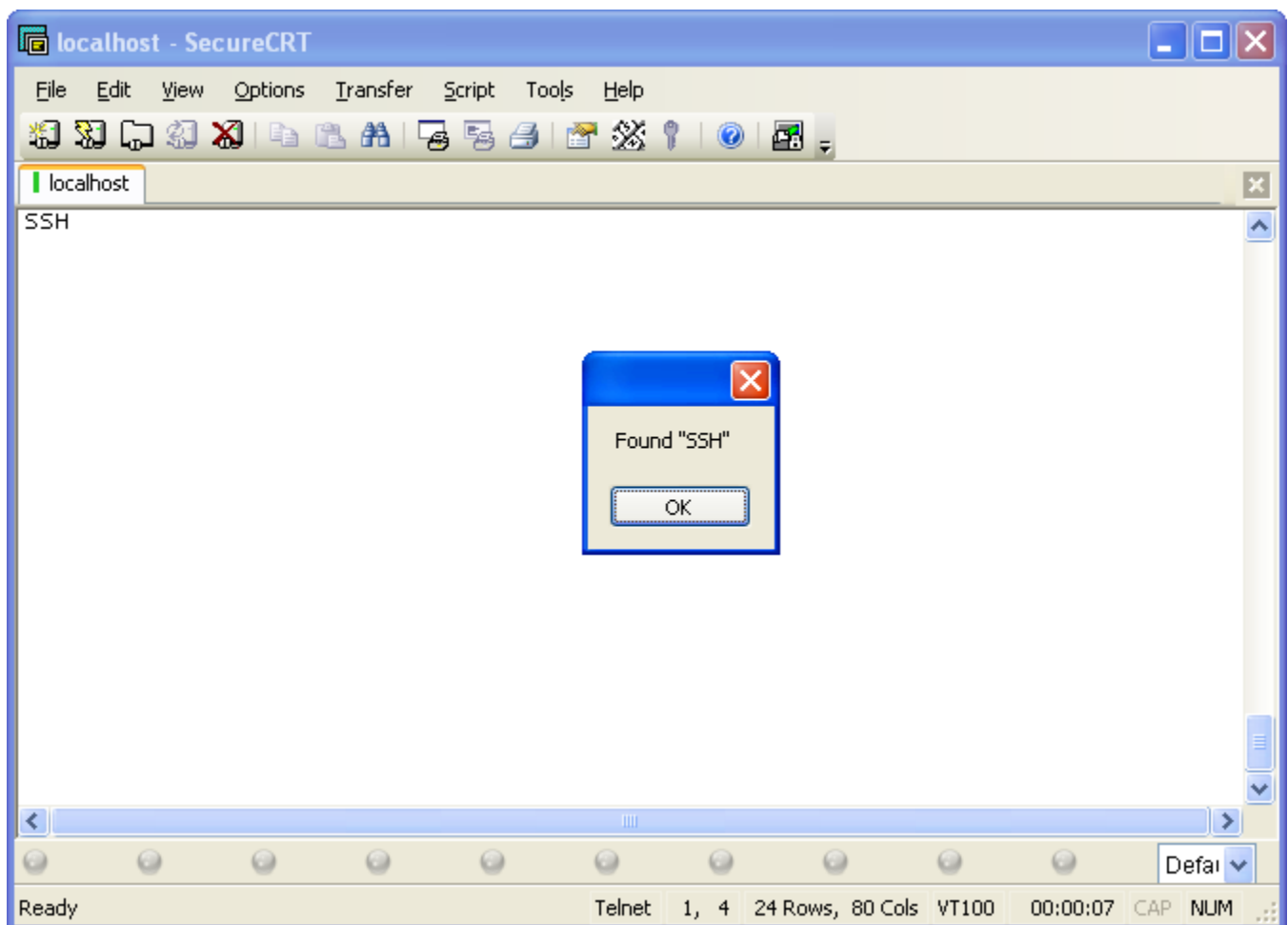
    If crt.Screen.WaitForString("diffie-hellman-group-exchange-", 5) Then
        MsgBox "Found "diffie-hellman-group-exchange-""
    Else
        MsgBox "Timed out waiting for "diffie-hellman-group-exchange-""
        Exit Sub
    End If

End Sub

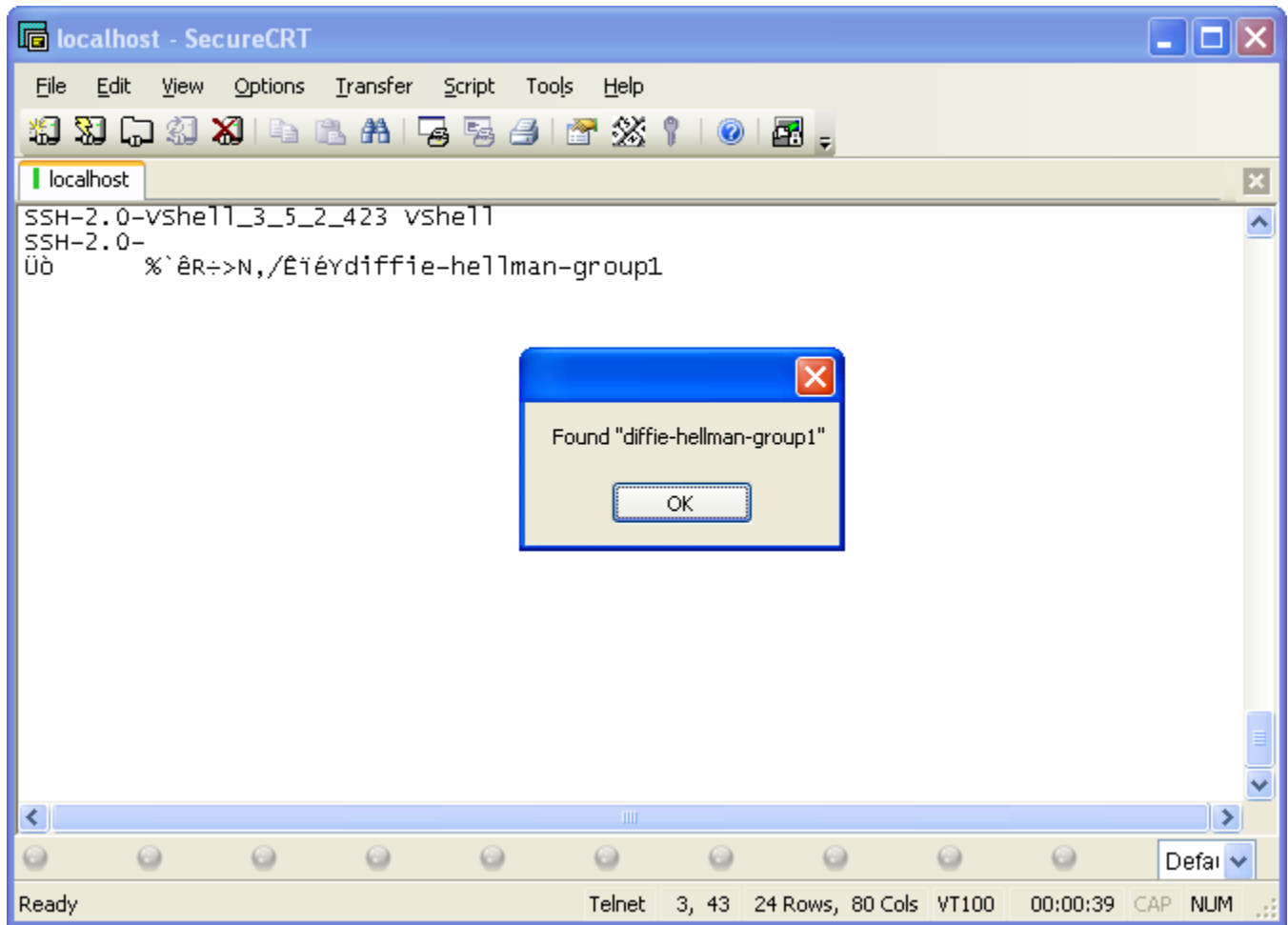
```

Here is a sequence of screen-shots depicting the code above in action (Synchronous is set to True).

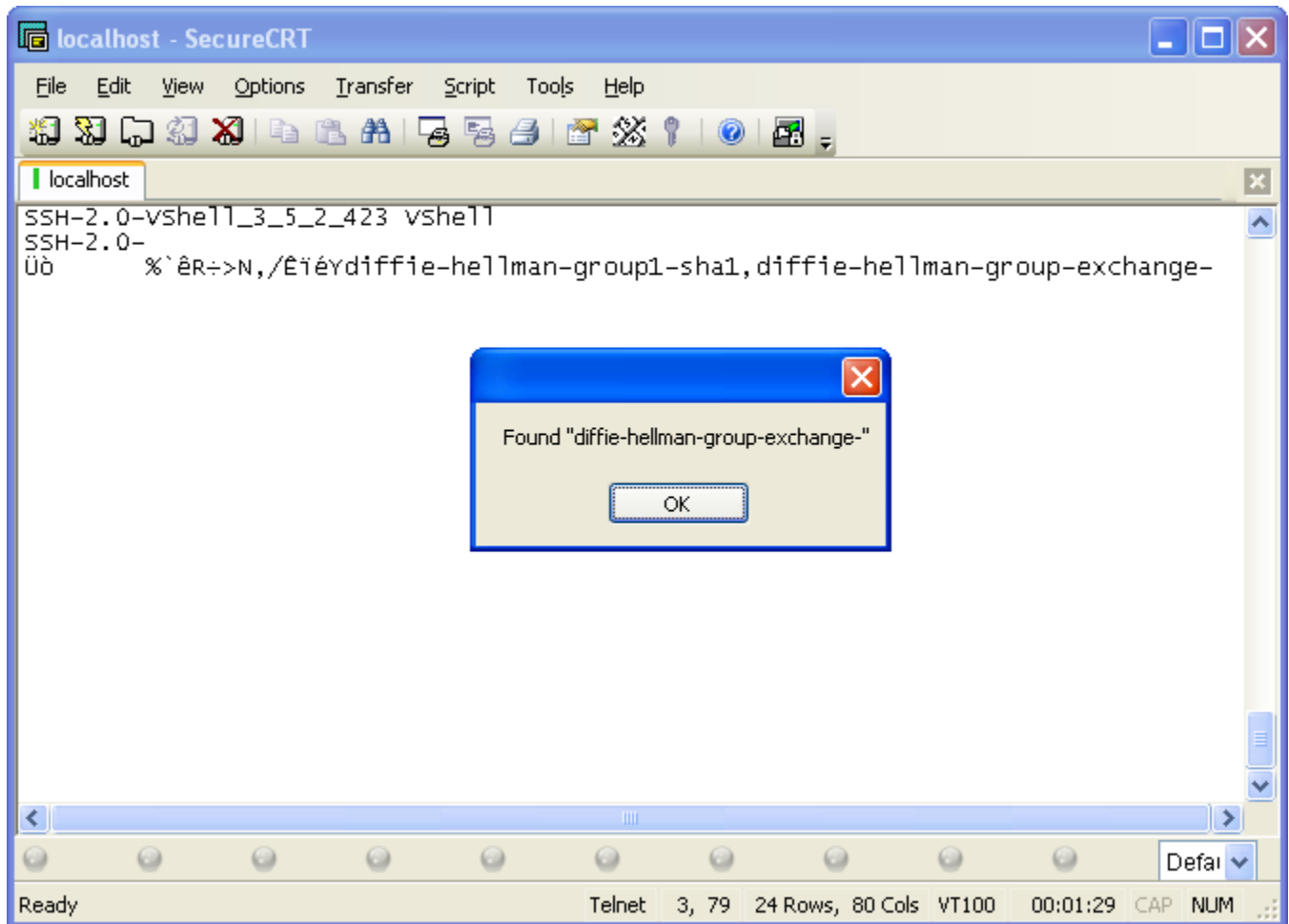
- 1) As expected, the first string the code is waiting for, "SSH", is found. Note that the screen only shows the data received up to the "SSH" string:



- 2) The string "SSH-2.0-" is then sent to the remote system and the code then waits for the next string to appear, "diffie-hellman-group1"; it appears as expected:

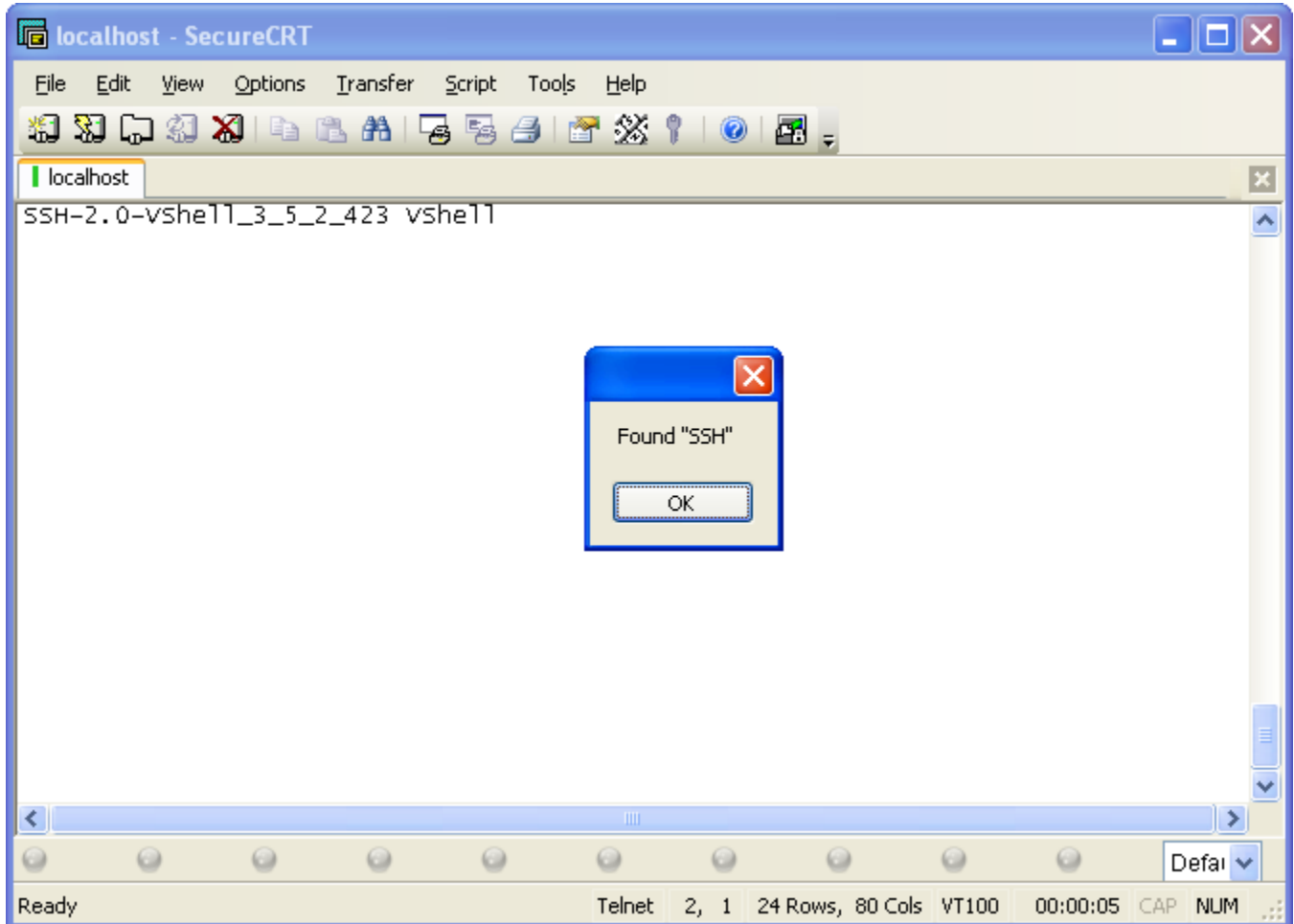


- 3) And finally, the last string our code waits for, "diffie-hellman-group-exchange-", is also found as expected:

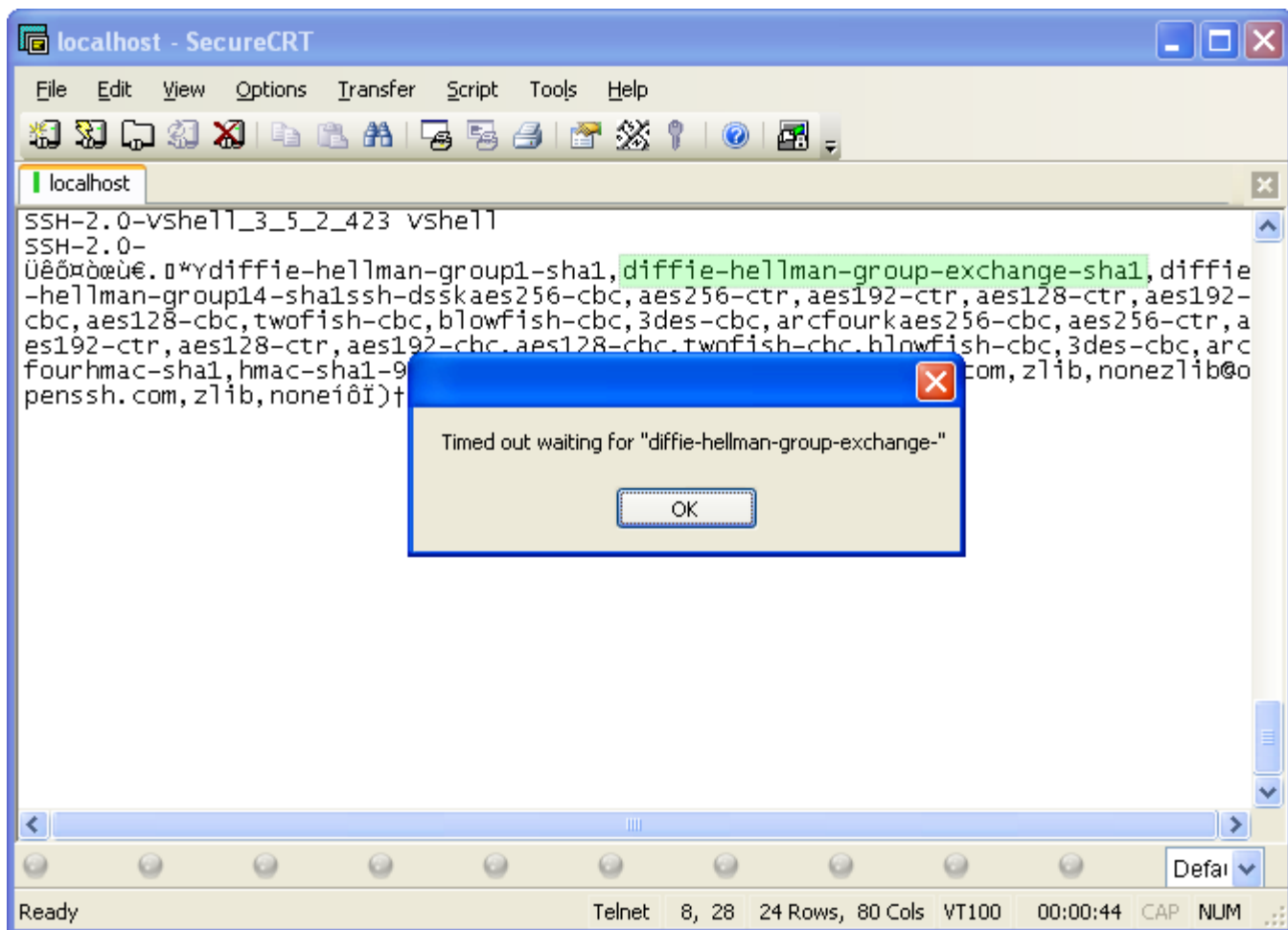


With only one modification to the original code above to set `Synchronous` to `False`, the behavior and timing of `WaitForString()` changes drastically. Here's a walk-through of the same code, but with `Synchronous` set to `False`:

- 1) "SSH" is found just fine. Note, however, that with `Synchronous` set to `False`, the screen displays data that has been received past the point at which the "SSH" string was found:



- 2) The "diffie-hellman-group1" string may also be found depending on the script and network timing. However a timeout may occur, as happens in this particular case, when waiting for the "diffie-hellman-group-exchange-" string – even though it appears on the screen (as indicated in the marked section within the graphic below). This behavior is due to the nature of the script code executing independent of the data being received from the remote machine such that in the time it takes the next `WaitForString()` call to be made within the script, the text being waited on in the code has already arrived:



The `Synchronous` property has the same relation to `WaitForString()` as it does to the `WaitForStrings()`, and `ReadString()` methods.

If your script code seems to be "missing" data that appears on the screen, inspect your code and ensure that the `Synchronous` property of the `Screen` object with which you are working is set to `True`.

Be aware, however, that setting `Screen.Synchronous` to `True` can seem to have an impact on SecureCRT's performance because data will not be displayed to the screen until calls to `WaitForString`, `WaitForStrings`, `ReadString`, or `WaitForCursor` are made. For example, the following script code will result in a successful connection to the remote host, but nothing will be displayed to the terminal window since the script is looping forever (without making any calls to `ReadString`, `WaitForString`, `WaitForStrings`, or `WaitForCursor`).

```

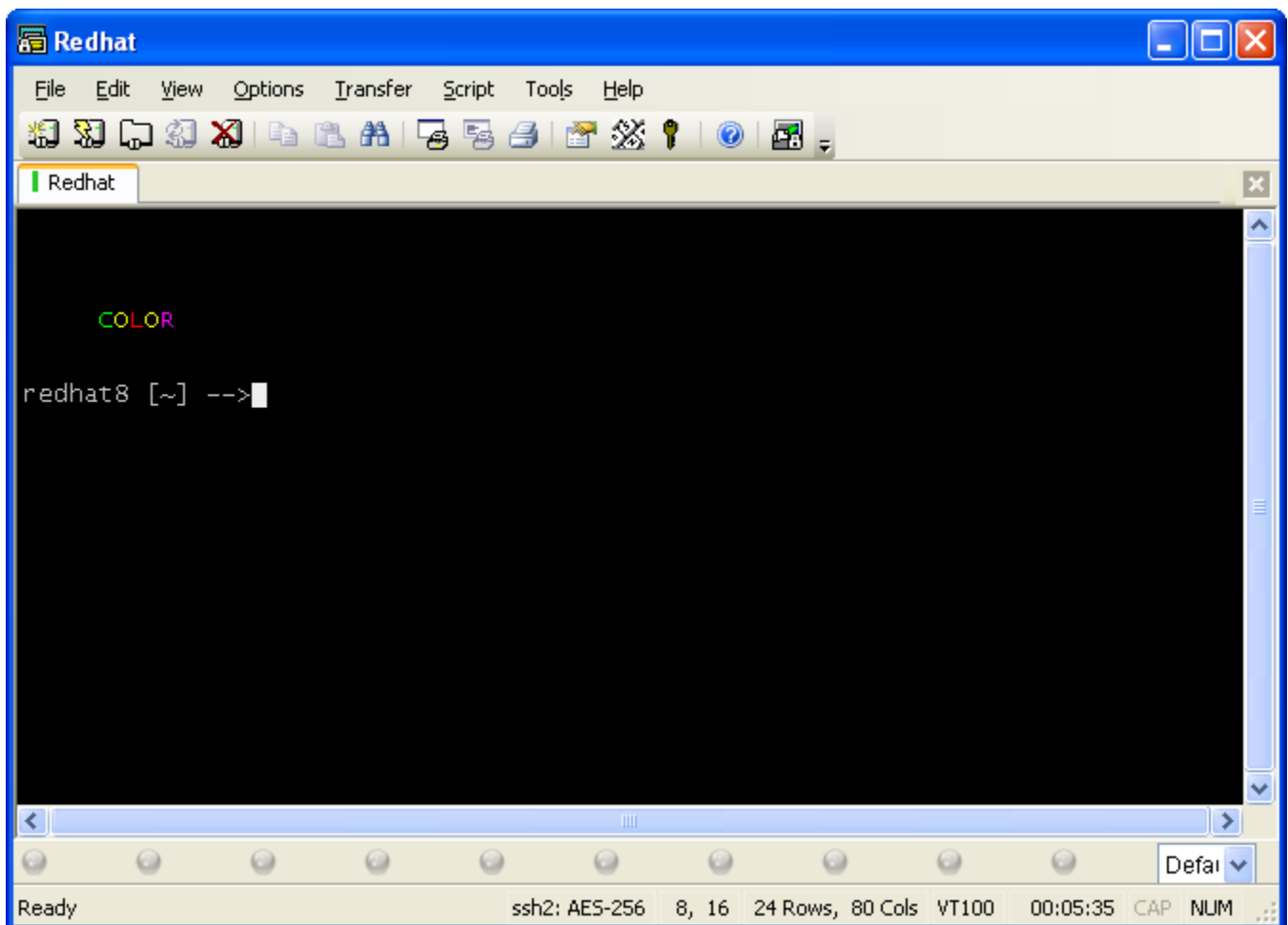
crt.Screen.Synchronous = True
crt.Session.Connect "/SSH2 localhost"

' Even though the connection above may be successful,
' nothing will ever be displayed to the terminal screen
' because a call to WaitForString, WaitForStrings,
' ReadString, or WaitForCursor is never made:
Do
    crt.Sleep 10
Loop

```

Ignoring Escape Codes (Or Not)

There might be situations in which the specific string you see on the screen isn't found by `ReadString()`, `WaitForString()` or `WaitForStrings()` because when it is sent from the remote system it has escape codes embedded within it, such as ANSI color codes. For the sake of example, assume that your task involved waiting for the string "COLOR" (as depicted in the graphic below) to occur before sending a command to the remote system:



Naturally, one would want to use the `WaitForString()` method. However, issuing a `WaitForString("COLOR")` call wouldn't work because the text came from the remote as an ANSI color escape sequence to change the color to green – followed by a 'C' character,

followed by an ANSI color escape sequence to change the color to yellow – followed by an 'O' character, followed by an ANSI color escape sequence to change the color to red – followed by an 'L' character, etc. If one looked at a raw log of the data SecureCRT received from the remote for the "COLOR" string indicated in the graphic above, it would be similar to the following:

```
ESC[ 32mCESC[ 33mOESC[ 31mLESC[ 33mOESC[ 35mR
```

Fortunately, SecureCRT's scripting API provides a mechanism to allow a script developer to ignore escape codes, if desired, by setting the `Screen.IgnoreEscape` property to `True`. By default, a `Screen` object's `IgnoreEscape` property is set to `False`. Setting `IgnoreEscape` to `True` will allow `WaitForString("COLOR")` to work in the example above. For example:

```
crt.Screen.IgnoreEscape = True
crt.Screen.WaitForString("COLOR")
```

4.3 Capturing Data from a Remote Machine

The task you perform over and over again might involve looking for specific information on the remote device or copying the data to another application. This section will describe different methods that can be used to capture data from a remote machine.

The following methods associated with the `Screen` object can be used to capture data via a connection to a remote machine: `ReadString()`, `Get()`, and `Get2()`. Although SecureCRT's logging capabilities can also be used to capture data from a remote device, the logging API is referenced and discussed in a later chapter ([Writing Data to Files Using the FileSystemObject](#)). If you're looking for ways to access data that is currently selected within the SecureCRT terminal screen, please see the earlier section, [Accessing Selected Text on the Screen](#).

Capturing Text Displayed in a Specific Location on the Screen

If your task involves running an application that displays important data in specific locations within the terminal screen, you can use the `Screen` object's `Get()` or `Get2()` methods to capture data. Depending on your goal, you may find `Get2()` more useful than `Get()`. The following screen-shot of a UNIX 'top' command in action is displayed in reference to the explanation of how to use these two functions.

4:06pm up 52 days, 4:47, 16 users, Load average: 0.02, 0.03, 0.00
 99 processes: 98 sleeping, 1 running, 0 zombie, 0 stopped
 CPU0 states: 0.0% user, 0.0% system, 0.0% nice, 100.0% idle
 CPU1 states: 0.0% user, 2.1% system, 0.0% nice, 97.4% idle
 Mem: 2064648K av, 1810652K used, 253996K free, 0K shrd, 463876K buff
 Swap: 522072K av, 18136K used, 503936K free 1161484K cached

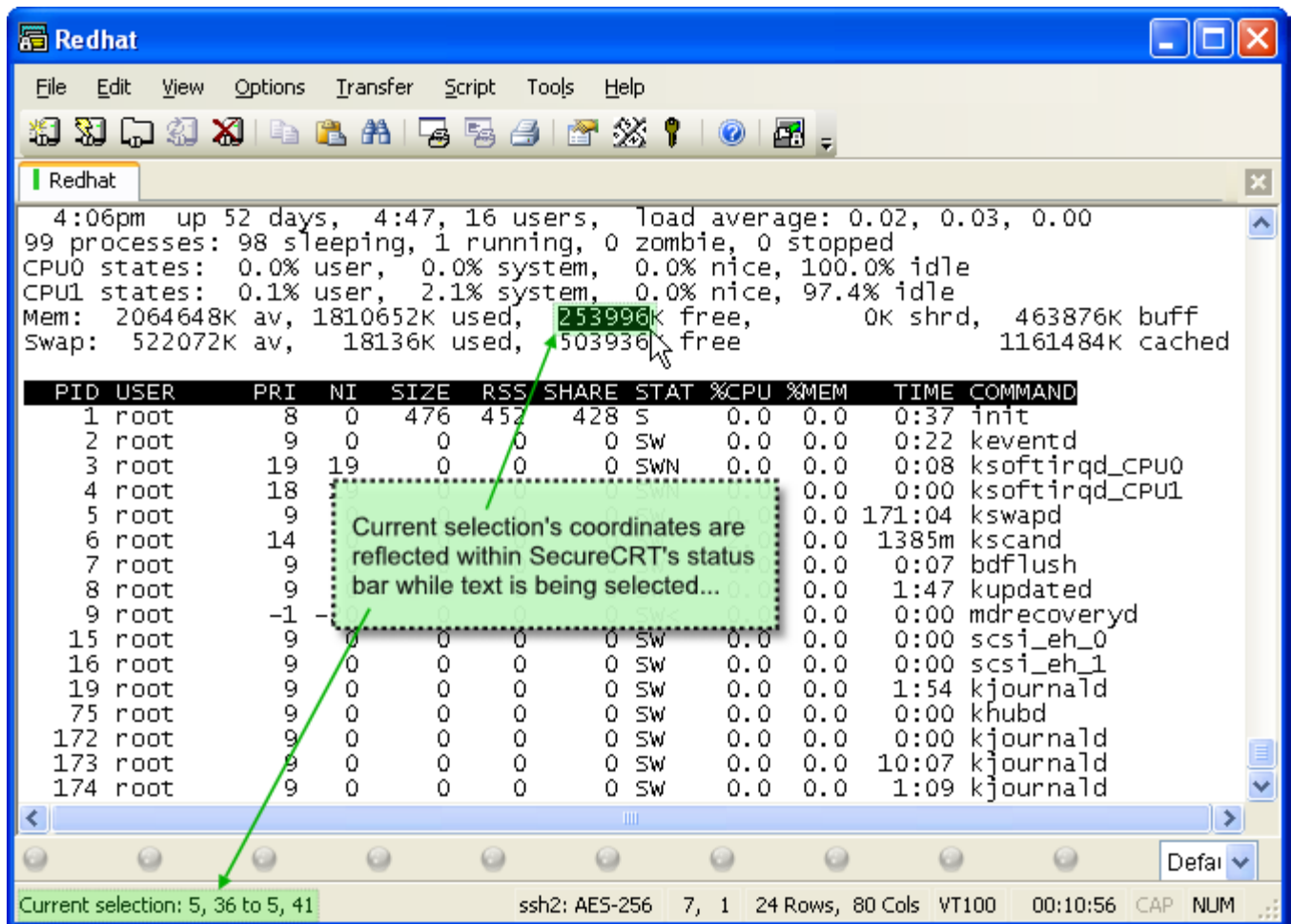
PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	COMMAND
1	root	8	0	476	452	428	S	0.0	0.0	0:37	init
2	root	9	0	0	0	0	SW	0.0	0.0	0:22	keventd
3	root	19	19	0	0	0	SWN	0.0	0.0	0:08	ksoftirqd_CPU0
4	root	18	19	0	0	0	SWN	0.0	0.0	0:00	ksoftirqd_CPU1
5	root	9	0	0	0	0	SW	0.0	0.0	171:04	kswapd
6	root	14	0	0	0	0	SW	2.0	0.0	1385m	kscand
7	root	9	0	0	0	0	SW	0.0	0.0	0:07	bdflush
8	root	9	0	0	0	0	SW	0.0	0.0	1:47	kupdated
9	root	-1	-20	0	0	0	SW<	0.0	0.0	0:00	mdrecoveryd
15	root	9	0	0	0	0	SW	0.0	0.0	0:00	scsi_eh_0
16	root	9	0	0	0	0	SW	0.0	0.0	0:00	scsi_eh_1
19	root	9	0	0	0	0	SW	0.0	0.0	1:54	kjournald
75	root	9	0	0	0	0	SW	0.0	0.0	0:00	khubd
172	root	9	0	0	0	0	SW	0.0	0.0	0:00	kjournald
173	root	9	0	0	0	0	SW	0.0	0.0	10:07	kjournald
174	root	9	0	0	0	0	SW	0.0	0.0	1:09	kjournald

Ready ssh2: AES-256 7, 1 24 Rows, 80 Cols VT100 00:03:12 CAP NUM

For this demonstration, assume your task involves capturing the number of total processes running on the machine, as well as the amount of free memory. Looking at the screen, it would be clear by way of visual inspection that the number of processes is 99, and the amount of free memory is currently 253996K. But how would you automate the extraction of this information within a script?

For such a task, the Screen object's `Get()` method provides a quick way to extract this information from the screen. The `Get()` method expects four parameters that define the starting `row,col` and ending `row,col` values associated with the location on the screen that contains the desired text.

Rather than manually counting rows and columns, you can select the area of the screen you want to capture with your mouse and pay attention to the starting and ending rows and columns associated with the text you are selecting. For example, in the graphic below, the selected text starts on row 5, column 36. The selection ends on row 5, column 41:



The example code below extracts the process count and free memory data values from the screen as indicated above into two separate script variables:

```

' Extract data displayed on row 2, between columns 1 and 2
nProcesses = crt.Screen.Get(2,1,2,2)

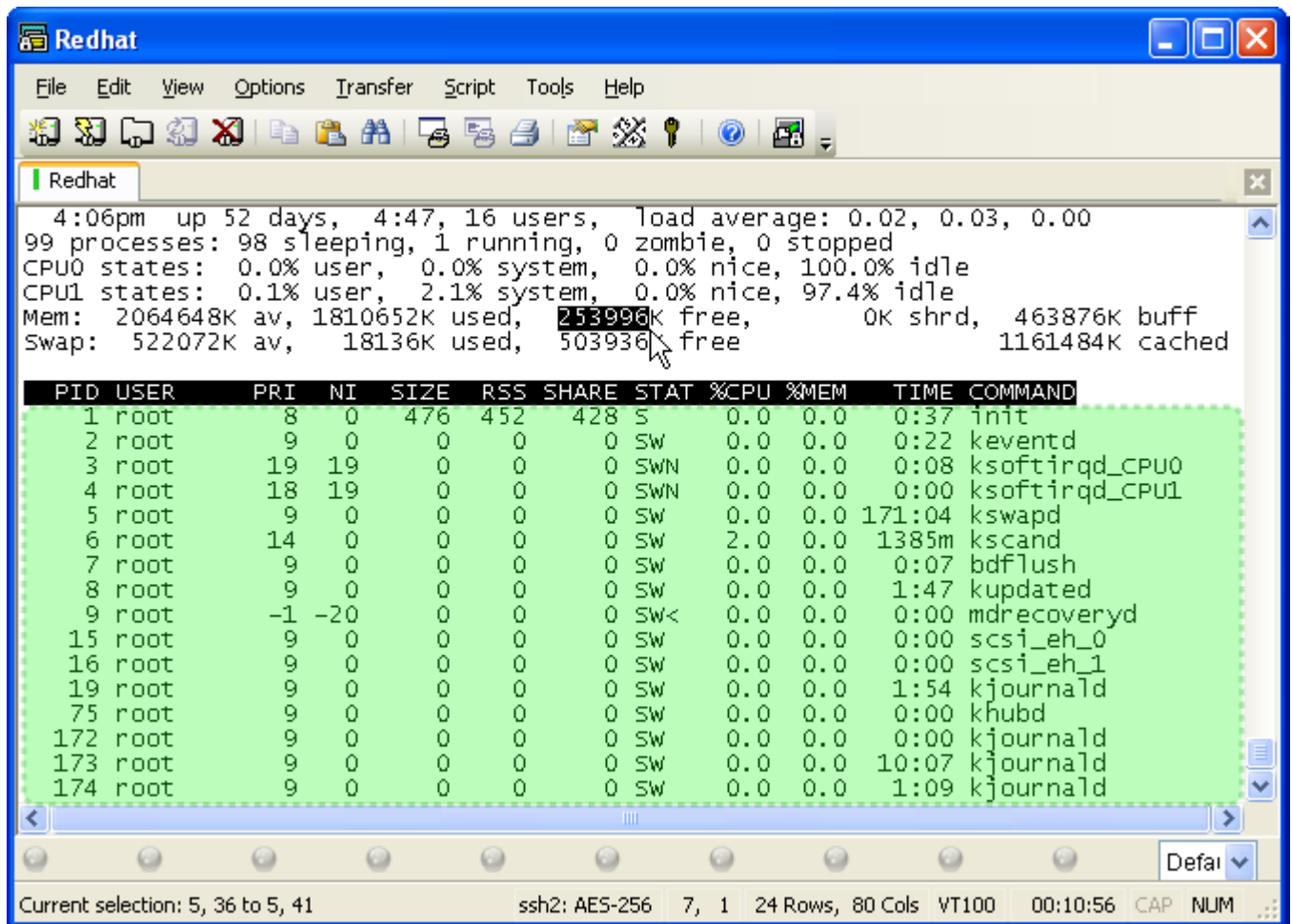
' Extract data displayed on row 5, between columns 36 and 41
nFreeMem = crt.Screen.Get(5,36,5,41)

MsgBox "Number of Processes: " & nProcesses & vbcrLf & _
      "Free Memory: " & nFreeMem

```

Capturing Multiple Lines of Data Displayed on the Screen

Perhaps instead of capturing text from a specific location on the screen as described earlier, your task involves capturing multiple lines of text as displayed on the screen. While the `Get()` method can provide you with all of the lines you want, the `Get2()` method returns each line separated by a carriage return plus line feed so that the lines can easily be split into an array for processing. Using the previous example of 'top' running on a remote UNIX machine, if your task involves capturing the block of lines indicated in the graphic below, you can use `Get2()` to assist you.



The following code segment shows how to use `Get2()` to capture the block of lines indicated in the above graphic.

```
' Extract lines of data from row 9 to row 24. Note that we
' start at column 1, and end with the number of columns
' currently associated with the screen; as the number of
' columns can be dynamic, we use the "Columns" property of the
' Screen object to tell us how many columns there currently
' are.
strLines = crt.Screen.Get2(9,1, 24,crt.Screen.Columns)

vLines = Split(strLines, vbCrLf)
For nIndex = 1 To UBound(vLines)
    MsgBox "Line #" & nIndex & ": " & vLines(nIndex - 1)
Next
```

For the sake of demonstration, the example above only displays each line individually. Later on, tips will be given in the [Extracting Specific Information](#) section on how to extract specific information from each line using a regular expression.

Capturing Data Streamed from the Remote

If the types of commands you are running on the remote system don't provide information in nice tabular format, extracting the data you need would be tedious if all you had available were `Get()` and `Get2()`. Instead of running a curses application like 'top', perhaps your task involves running a command and capturing the output of that command. The `ReadString()` method can be used to capture all data received from the remote until specific data is found. For example, if you are attempting to connect to a Cisco device and retrieve its serial number, the following code might work for you:

```
crt.Screen.Synchronous = True

' Send a command to a Cisco device to get the serial number
' of the device.
crt.Screen.Send "sh tech-support | in ([sS]erial)" & vbcr

' Wait for the CR to be echoed back to us so that what is
' read from the remote is only the output of the command
' we issued (rather than including the command issued along
' with the output).
crt.Screen.WaitForString vbcr

' Capture the result into a script variable
strResult = crt.Screen.ReadString("pixfirewall#")

' strResult will contain something like:
'   Serial Number: 1850889413810201 (0x6935FC6075819)
MsgBox strResult
```

The example above simply captures the output that is received by SecureCRT between the time that the command was sent and when the Cisco device's prompt appeared. Later in the [Extracting Specific Information](#) section, tips will be given as to how to extract data more precisely.

Using `MatchIndex` to Determine Which String `ReadString()` Detected

What if the command you issue returns more data that can fit on the SecureCRT screen? For example, some commands issued on a Cisco device will display as much information as will fit on the screen, followed by a "<--- More --->" prompt at the bottom. One way around this would be to configure the device so that the device will send information without prompting the user for input to display additional data. For example, on a Cisco Pix device, the 'pager lines 0' command will cause commands like "sh config" to display all information without prompting the user each time the screen fills up with data. However, if your device doesn't support such a command, there is still a way to use `ReadString()` to capture all of the output and simulate pressing the space bar to encourage the remote system to send another screen full of data.

Consider the following example code which takes advantage of `ReadString()`'s ability to wait for multiple strings and uses the `MatchIndex` property to determine which string was found in the data from the remote. If a "More" prompt was found, the script code simply accumulates the data read from the remote and presses the space bar so that more data will be sent. Once the

shell prompt is found, the script determines that all the data from the command has been received from the remote.

```
crt.Screen.Synchronous = True

' Send a 'sh config' command to the Cisco PIX firewall.
crt.Screen.Send "sh config" & vbcr

' Wait for the CR to be echoed back to us so that what is
' read from the remote is only the output of the command
' we issued (rather than including the command issued along
' with the output).
crt.Screen.WaitForString vbcr

strCompleteOutput = ""
Do
    ' Capture the result into a script variable.
    ' Have ReadString() return after it sees one of the
    ' supplied strings "pixfirewall#" or the
    ' "...More..." tag.
    strResult = crt.Screen.ReadString("<--- More --->", "pixfirewall#")

    ' Append the results we've got so far to the complete
    ' results that are being accumulated.
    strCompleteOutput = strCompleteOutput & strResult

    ' If a "More" was found, send a space to get the next
    ' page of output from the remote
    If crt.Screen.MatchIndex = 1 Then crt.Screen.Send " "

    ' If the shell prompt was found, then we've received
    ' all of the output expected to have been sent from
    ' the remote.
    If crt.Screen.MatchIndex = 2 Then Exit Do
Loop

' A Cisco PIX sends each line followed by a line feed (LF)
' and a Carriage return (CR). Take care of each line that
' would have had "<--- More --->" on it. Depending on the way
' your device tries to "erase" the "<--- More --->" prompt, you
' may need to change the replace string below.
strCompleteOutput = Replace(strCompleteOutput, _
    vbLf & vbCr & vbCr & " " & vbCr, vbLf & vbCr)

MsgBox strCompleteOutput
```

Extracting Specific Information

Earlier, an example was provided describing how to capture the line of text returned by a Cisco router indicating the serial number of the device. In this example, the output from the device was an entire line that reads: Serial Number: 1850889413810201 (0x6935FC6075819)

Now, you'd like to get just the serial number itself (the part that reads, "1850889413810201"). This section will exemplify some of the VBScript built-in mechanisms that can be employed to get at exactly what it is you need from the output that has been received from the remote system. Not wanting to "re-invent the wheel", no attempt will be made to provide a completely exhaustive documentation of all functions that could be possibly be used in such scenarios. For complete documentation of the more common methods that can be used, consult Microsoft's online

“VBScript Language Reference” mentioned in the introduction: <http://msdn.microsoft.com/en-us/library/t0aew7h6.aspx>. More specifically, look in the “Functions” section for the following gems:

- InStr
- InStrRev
- Mid
- Left
- Right
- Split

The discussion of the above will be limited to an example of the `Split()` function. In addition, an example of using a regular expression to extract data will also be provided.

Using `split()` to Parse Information

So you’ve extracted the following line from output from the remote:

```
Serial Number: 1850889413810201 (0x6935FC6075819)
```

One easy way to extract the serial number (1850889413810201), would be to use the VBScript `Split()` function since the serial number is separated by a single space character on each side. From the vantage point used for this example, the line above is composed of 4 “tokens” with each token being separated by a “delimiter”. The delimiter in this case is the space character and the tokens are “Serial”, “Number:”, “1850889413810201”, and “(0x6935FC6075819)”. The `Split()` function takes a string along with a delimiter and “splits” the string up into an array of tokens, where the tokens were separated from each other by the delimiter supplied to the function. Consider the following example code that demonstrates that the decimal value version of the serial number is the 3rd token (at index #2, since arrays in VBScript are “zero-based” – in other words, the first index of an array is zero):

```
strResults = "Serial Number: 1850889413810201 (0x6935FC6075819)"
vTokens = Split(strResults, " ")

For nIndex = 0 To UBound(vTokens) - 1
    MsgBox "Token at index " & nIndex & " is: " & vTokens(nIndex)
Next

MsgBox "The serial number is: " & vTokens(2)
```

Using Regular Expressions to Parse Information

Once again, you’ve extracted the following line of information as part of the output from the remote machine:

```
Serial Number: 1850889413810201 (0x6935FC6075819)
```

This time, however, you are faced with the following challenges:

- You know that some devices provide serial number information surrounded by more than one space.
- You know that other devices only provide the output in terms of “Serial: 1850889413810201 (0x6935FC6075819)”, so using `Split()` and getting the token at index #2 will not always provide you with the data you want.

In this situation, using a regular expression pattern could correctly extract the serial number from either of the following strings that might be returned from the different remote devices:

```
Serial Number: 1850889413810201 (0x6935FC6075819)
Serial: 1850889413810201 (0x6935FC6075819)
```

```
Serial No.: 1850889413810201 (0x6935FC6075819)
Serial Number: 1850889413810201 (0x6935FC6075819)
```

Here's a pattern that would work in all of the above cases to extract the serial number: "\s+(\d+)\s+". Here's a breakdown of this pattern:

- \s+ matches one or more whitespace characters
- (\d+) matches one or more numeric digit characters (0-9), and the surrounding ()s indicate that the regular expression execution should capture the text matching the pattern of one or more digits for retrieval.

Here's a section of code exemplifying one way to use a regular expression to extract the serial number from any of the output variations described earlier:

```
strResults = "Serial Number: 1850889413810201 (0x6935FC6075819)"

' Create a new regular expression object
Set re = New RegExp

' Set the pattern to reflect the data we would like to extract;
' data between the ()s will be extracted if the pattern is found
re.Pattern = "\s+(\d+)\s+"

' First, test to see if the text we have contains the pattern...
If re.Test(strResults) <> True Then
    MsgBox "Pattern "" & re.Pattern & _
    "" wasn't found within the following text: " & _
    vbCrLf & vbCrLf & vtab & """" & strResults & """"
Else
    ' Now that we know the text we have contains the pattern
    ' we're looking for, let's execute the regular
    ' expression so that we can iterate through the pattern
    ' matches that are available within the text.
    Set matches = re.Execute(strResults)
    For Each match In matches
        strSerial = match.SubMatches(0)
        MsgBox "Serial number extracted as: " & strSerial
    Next
End If
```

More detailed documentation of regular expression syntax can be found within Microsoft's online "VBScript Language Reference" mentioned in the introduction: <http://msdn.microsoft.com/en-us/library/t0aew7h6.aspx>. Specifically, **VBScript Language Reference → Objects and Collections → Regular Expression (RegExp) Object → Regular Expression Object Properties and Methods → Pattern Property**.

Chapter 5: Sending Data to Remote Machines

Sending data to a remote device is a fairly straightforward task once you have an assurance that the remote machine is ready to receive data. In this chapter, you'll learn how to send plain text as well as non-printing control codes. You'll also learn how to simulate keyboard events to simplify automating tasks that involve function keys or mapped keyboard combinations defined within a key map file or session options.

5.1 Sending Plain Text

Sending plain text to a remote device is as simple as using the `Screen` object's `Send()` method. You can take advantage of the VBScript language's built-in concatenate operator (`&`) to combine multiple string segments together. A full-featured example of sending plain-text commands to a remote device is presented below.

Solution: Repeat a Command with Variable Input from User

The following script solution shows how you can send the same command repeatedly to a remote machine, each time with different parameters based on input from the user. This example uses the following command as an example: `ping -c 2 -w 2 192.168.0.[1-5,32,45-40]`. The script will iterate through all of the ranges provided within the `[]` characters, sending a modified version of the same command for each number identified within the range specified by the user. Each line that actually causes data to be sent to the remote is indicated in **bold** typeface.

```
' SendCommandWithVariableInputFromUser.vbs
'
' Prompts user for a command that may include a variable range (defaults to
' "ping -c 2 -w 2 192.168.0.[1-5,32,45-40]", as an example).
'
' Once a command is entered, a different instance of each command for the range
' provided is sent to the remote machine. For example, if the command specified
' by the user was "ping -c 2 -w 2 192.168.0.[1-5]", then the following commands
' would be sent to the remote machine:
'   ping -c 2 -w 2 192.168.0.1
'   ping -c 2 -w 2 192.168.0.2
'   ping -c 2 -w 2 192.168.0.3
'   ping -c 2 -w 2 192.168.0.4
'   ping -c 2 -w 2 192.168.0.5
'
' Although this example requires an active connection, one could modify this
' example script to first connect to a session and then prompt the user for the
' command "template".
'
' ~~~~~
Sub Main()

    If Not crt.Session.Connected Then
        crt.Dialog.MessageBox _
            "This script was designed to be launched after" & vbcrLf & _
            "a connection has already been made. Please" & vbcrLf & _
            "establish a connection prior to running this" & vbcrLf & _
            "script."
        Exit Sub
    End If

    ' Change the strPrompt variable to match the prompt you expect to see on the
    ' screen indicating that each command has been completed and the remote is
    ' ready to receive the next cmd.
    strPrompt = "]" -->
```



```

strCmd = "ping -c 2 -w 2 192.168.0.[1-5,32,45-40]"
strCmd = crt.Dialog.Prompt( _
    "Enter the IP range to target for ping testing:", _
    "Specify Ping Target IP Address/Range", _
    strCmd)
If strCmd = "" Then Exit Sub

' Check to see if a range was ever included in the input from the user:
If Instr(strCmd, "[") = 0 Then
    ' Range was not specified, simply issue the whole command provided
    crt.Screen.Send strCmd & vbcr
Else
    ' A range was specified. Iterate through the range specified, sending
    ' a separate command for each interval in the range:

    ' First, detect the range specified in the command instructions provided
    ' by the end user:
    nPosRangeStart = Instr(strCmd, "[")
    nPosRangeClose = Instr(strCmd, "]")
    If nPosRangeClose = 0 Then
        crt.Dialog.MessageBox _
            "Didn't find expected close bracket for range specification."
        Exit Sub
    End If
    strRange = Mid(strCmd, _
        nPosRangeStart + 1,
        Len(strCmd) - nPosRangeStart - 1)

    ' Next, separate out the command part from the range
    strCmdLeft = Left(strCmd, Instr(strCmd, "[") - 1)
    strCmdRight = Mid(strCmd, Instr(strCmd, "]") + 1)

    ' Next, iterate through all comma-separated sub-ranges
    vSubRanges = Split(strRange, ",")
    For Each subRange In vSubRanges
        ' Detect whether or not we have just a number or a <start-end> pattern
        If Instr(subRange, "-") = 0 Then
            ' Just send the number as-is for this command
            crt.Screen.Send strCmdLeft & subRange & strCmdRight & vbcr
            crt.Screen.WaitForString strPrompt
        Else
            ' Handle the range as specified, for example: 1-5
            ' Get the start and end range numbers
            vElements = Split(subRange, "-")
            nStart = vElements(0)
            nEnd = vElements(1)

            ' Handle cases where the range might be specified in reverse order,
            ' such as 50-48, compared to 48-50...
            If nStart <= nEnd Then
                ' Increasing range (e.g., 1-5)
                For nIndex = nStart To nEnd
                    ' Send the current command to the remote
                    crt.Screen.Send strCmdLeft & nIndex & strCmdRight & vbcr
                    ' Wait for the command to complete (the prompt should appear)
                    ' before continuing on with the next command
                    crt.Screen.WaitForString strPrompt
                Next
            Else
                ' Decreasing range (e.g., 5-1)
                For nIndex = nStart To nEnd Step -1
                    ' Send the current command to the remote

```

```

        crt.Screen.Send strCmdLeft & nIndex & strCmdRight & vbcr
        ' Wait for the command to complete (the prompt should appear)
        ' before continuing on with the next command
        crt.Screen.WaitForString strPrompt
    Next
    End If 'subRange decreasing or increasing?
End If 'subRange contains a "-"?
Next
End If 'Command contains a range [x-y]?
End Sub

```

5.2 Sending Control Codes

When you find it necessary to send control codes or escape sequences to the remote, you can use the VBScript built-in `Chr()` method to achieve the desired results. For example, if you need to send an ASCII Ctrl+A control character to the remote, you would use `crt.Screen.Send Chr(1)`.

Although there is an ASCII table in the SecureCRT help that will assist you in determining the codes required to send non-printing control sequences to a remote device, you might consider using the following "Control()" function example, or the `SendKeys()` method described later on in this chapter, as a way of facilitating the process more efficiently.

```

' ~~~~~
Function Control(character)
    ' The following formula is useful for this task:
    '   ASCIIControlCode = chr(asc(<UPPERCASE character>) - 64)
    '
    ' For example, Ctrl+@ is really an ASCII 0 (NUL), Ctrl+A is really
    ' an ASCII 1 (SOH), and so on...
    '
    ' Since our formula only works if upper-case characters are used,
    ' we'll up-case any lower-case characters that are provided:
    character = UCase(character)
    '
    ' In VBScript, the return value of a function is set by
    ' assigning the name of the function to the value that
    ' should be returned.
    Control = chr(asc(character) - 64)
End Function

' ~~~~~
Sub Main()
    ' Send all non-printing control characters to the remote:
    crt.Screen.Send Control("@")   ' chr(0)
    crt.Screen.Send Control("A")   ' chr(1)
    crt.Screen.Send Control("B")   ' chr(2)
    crt.Screen.Send Control("C")   ' chr(3)
    crt.Screen.Send Control("D")   ' chr(4)
    crt.Screen.Send Control("E")   ' chr(5)
    crt.Screen.Send Control("F")   ' chr(6)
    crt.Screen.Send Control("G")   ' chr(7)
    crt.Screen.Send Control("H")   ' chr(8)
    crt.Screen.Send Control("I")   ' chr(9)
    crt.Screen.Send Control("J")   ' chr(10)
    crt.Screen.Send Control("K")   ' chr(11)
    crt.Screen.Send Control("L")   ' chr(12)
    crt.Screen.Send Control("M")   ' chr(13)
    crt.Screen.Send Control("N")   ' chr(14)
    crt.Screen.Send Control("O")   ' chr(15)

```

```

crt.Screen.Send Control("P") ' chr(16)
crt.Screen.Send Control("Q") ' chr(17)
crt.Screen.Send Control("R") ' chr(18)
crt.Screen.Send Control("S") ' chr(19)
crt.Screen.Send Control("T") ' chr(20)
crt.Screen.Send Control("U") ' chr(21)
crt.Screen.Send Control("V") ' chr(22)
crt.Screen.Send Control("W") ' chr(23)
crt.Screen.Send Control("X") ' chr(24)
crt.Screen.Send Control("Y") ' chr(25)
crt.Screen.Send Control("Z") ' chr(26)
crt.Screen.Send Control("[") ' chr(27)
crt.Screen.Send Control("\") ' chr(28)
crt.Screen.Send Control("]") ' chr(29)
crt.Screen.Send Control("^") ' chr(30)
crt.Screen.Send Control("_") ' chr(31)
End Sub

```

5.3 Simulating Keyboard Events

As you may have seen throughout many of the examples in this document, simulating an **Enter** keyboard event is done by using a `crt.Screen.Send vbcr` command. While this works in many cases, you might need to simulate pressing a function key that works to do what you need on the remote system. For example, you may not know the low-level control codes that are actually sent to the remote system when the **F12** key is pressed, but you know it's what the remote requires and you want to simulate the **F12** key being pressed from within your script. Perhaps in a different situation, you want to take advantage of a key map setup that already contains a specific sequence of characters to send to the remote.

In either case, you can take advantage of the `SendKeys()` method available as part of the `Screen` object in SecureCRT versions 6.1 and newer to simulate actual keyboard events that you would normally press manually within SecureCRT. For example, if you have mapped the **Ctrl+Shift+Alt+F7** keyboard combination to send a particular string of commands, you can invoke this pre-defined keyboard "macro" using the following line of code:

```
crt.Screen.SendKeys "^+#{F7}"
```

SecureCRT's `SendKeys()` method follows the patterns established by the WScript built-in `WshShell.SendKeys()` method in terms of the string values required to indicate the keystroke(s) you want to simulate. However, there are some differences between SecureCRT's `SendKeys()` method and WshShell's `SendKeys()` method:

- SecureCRT allows you to simulate pressing the **Prt Scn** (Print Screen) button (the `WshShell.SendKeys()` method does not allow this). Note that unless mapped otherwise, pressing the **Prt Scn** key will toggle the auto print functionality.
- SecureCRT allows you to distinguish between pressing keys on the normal keypad vs. the number keypad (e.g., "0" vs. "{NUM_0}", and "{+}" vs. "{NUM_+}").
- SecureCRT allows you to include a numeric repeating indicator for keyboard combinations, not just single characters. For example: `crt.Screen.SendKeys("^f 50")` will result in 50 **Ctrl+F** keyboard events.

As per the pattern established by WScript's `SendKeys()` method, the following characters have special meaning when included within a `SendKeys` string:

- + The plus sign indicates a **Shift** modifier (e.g., `Shift+F1` is represented as

- `{F1}`). To simulate a literal '+' key, use `{+}` or `{NUM_+}` to specify a normal keypad "+" or a number keypad "+" key press event, respectively.
- `^` The caret indicates a **Ctrl** modifier (e.g., `Ctrl+C` is represented as `^C`). To simulate a literal "^" key, you could use `{^}` or `+6`.
- `%` The percent sign indicates an **Alt** modifier (e.g., `Alt+X` is represented as `%X`). To simulate a literal "%" key, you could use `{%}` or `+5`.
- `~` The tilde character represents pressing the **Enter** key. To simulate a literal "~" key, you would use `{~}`.

A table describing the special keys and the corresponding string representations `SendKeys()` expects is found in the SecureCRT help within the `Screen` object's reference. A version of that same table is included in this document as a convenience:

Key	String Argument for <code>SendKeys()</code>
BACKSPACE	<code>{BACKSPACE}</code> , <code>{BS}</code> , or <code>{BKSP}</code>
BREAK	<code>{BREAK}</code>
CAPS LOCK	<code>{CAPSLOCK}</code>
DEL or DELETE	<code>{DELETE}</code> or <code>{DEL}</code>
DOWN ARROW	<code>{DOWN}</code>
END	<code>{END}</code>
ENTER	<code>{ENTER}</code> or <code>~</code>
ESC	<code>{ESC}</code>
HELP	<code>{HELP}</code>
HOME	<code>{HOME}</code>
INS or INSERT	<code>{INSERT}</code> or <code>{INS}</code>
LEFT ARROW	<code>{LEFT}</code>
NUM LOCK	<code>{NUMLOCK}</code>
PAGE DOWN	<code>{PGDN}</code>
PAGE UP	<code>{PGUP}</code>
PRINT SCREEN	<code>{PRTSC}</code>
RIGHT ARROW	<code>{RIGHT}</code>
SCROLL LOCK	<code>{SCROLLLOCK}</code>
TAB	<code>{TAB}</code>
UP ARROW	<code>{UP}</code>
F1, F2, ... F16	<code>{F1}</code> , <code>{F2}</code> , ... <code>{F16}</code>
0, 1, ... 9 on number pad	<code>{NUM_0}</code> , <code>{NUM_1}</code> , ... <code>{NUM_9}</code>
. (dot) on number pad	<code>{NUM_.</code>
/ (slash) on number pad	<code>{NUM_/}</code>
* (asterisk) on number pad	<code>{NUM_*}</code>
- (minus) on number pad	<code>{NUM_-}</code>
+ (plus) on number pad	<code>{NUM_+}</code>
ENTER on number pad	<code>{NUM_ENTER}</code>
HOME on number pad	<code>{NUM_HOME}</code>
PAGE UP on number pad	<code>{NUM_PGUP}</code>
END on number pad	<code>{NUM_END}</code>
PAGE DOWN on number pad	<code>{NUM_PGDN}</code>
UP ARROW on number pad	<code>{NUM_UP}</code>
DOWN ARROW on number pad	<code>{NUM_DOWN}</code>

LEFT ARROW on number pad	{NUM_LEFT}
RIGHT ARROW on number pad	{NUM_RIGHT}

Solution: Add "no" to Each Selected Line and Send to Remote

For network administrators who spend a lot of time configuring devices, this solution could result in significant time savings. For example, consider the following lines selected within the SecureCRT terminal window:

```
http server enable
http 192.168.1.0 255.255.255.0 inside
floodguard enable
telnet 0.0.0.0 0.0.0.0 inside
```

Running this example solution script (code provided below) with the above lines selected within the SecureCRT terminal window would cause the following lines to be sent to the remote system:

```
no http server enable
no http 192.168.1.0 255.255.255.0 inside
no floodguard enable
no telnet 0.0.0.0 0.0.0.0 inside
```

Note also that the script code "toggles" the value of the "no"; if it already exists as the first part of the line, it will be removed. As another example, consider the following lines selected within the SecureCRT terminal window:

```
no http server enable
no http 192.168.1.0 255.255.255.0 inside
floodguard enable
telnet 0.0.0.0 0.0.0.0 inside
```

Running the script with the above lines selected would cause the following lines to be sent to the remote:

```
http server enable
http 192.168.1.0 255.255.255.0 inside
no floodguard enable
no telnet 0.0.0.0 0.0.0.0 inside
```

This example script code also doesn't require the text be selected within the SecureCRT terminal window. The script code will ask the user if the clipboard should be used in the event that there isn't any text selected within the current SecureCRT terminal window. This allows you to copy lines from another SecureCRT tab/window or even another application and then send the modified versions of these lines to the remote when the script is launched.

```
' ToggleNoForEachLineInSelectionAndPaste.vbs
'
' Description:
' This script demonstrates how to prepend or remove "no " to or from the
' beginning of lines selected within the SecureCRT screen, then paste the "no"
' versions of the lines back into SecureCRT. Useful for Network
' Administrators who configure Cisco routers and switches and find the need to
' do this "no" toggling frequently.
'
```

```

' This script will only work in SecureCRT 6.1 and later, as the
' Screen.Selection object isn't available in earlier versions of SecureCRT.
'
' Usage:
' 1) Select lines within the SecureCRT Window or copy them to the clipboard
'    from another application.
' 2) Run this script. Lines that already have "no " at the beginning of them
'    will be sent to the remote with the "no" removed. Lines that do not
'    have a "no " at the beginning will be sent to the remote with a "no "
'    added to the front of the line.
'
' ~~~~~
' Constants
' ~~~~~
' Button parameter options
Const BUTTON_YESNO = 4          ' Yes and No buttons

' Handled MessageBox() return values
Const IDYES = 6                ' Yes button clicked

' ~~~~~
Sub Main()
' Find out which tab we're running in...
Set objTab = crt.GetScriptTab

' Pull the selected text into a variable
strLines = Trim(objTab.Screen.Selection)

' Make sure we actually have some text to work with... if not, exit the
' script with a message to that effect.
If strLines = "" Then
    If crt.dialog.MessageBox(_
        "There isn't any text selected within the SecureCRT screen." & _
        vbcrLf & vbcrLf & _
        "Would you like to use the text within the clipboard instead?", _
        ""No"-ify Text Script", _
        BUTTON_YESNO) <> IDYES Then Exit Sub

' If there wasn't selected text and the user answers yes to the above
' prompt, pull the text from the clipboard, and let's work with it
' instead.
strLines = crt.Clipboard.Text
End If

' Split up the selected text into separate lines.
' First, let's standardize the text by making sure
' each line is separated by only a carriage return.
strLines = Replace(strLines, vbcrLf, vbcr)
strLines = Replace(strLines, vblf, vbcr)
' Now, use Split() to create an array out of each of the lines.
vLines = Split(strLines, vbcr)

' Prepend (or remove) "no " to (or from) each line & send to the remote.
For Each strLine In vLines
' Remove leading and trailing spaces
strLine = Trim(strLine)

' Ignore empty lines
If strLine <> "" Then

' If the line already has a "no " at the front, remove it
If Left(strLine, 3) = "no " Then
    objTab.Screen.Send Trim(Mid(strLine, 3)) & vbcr

```

```
        Else
            ' Otherwise, go ahead and prepend "no " to the line
            objTab.Screen.Send "no " & strLine & vbcr
        End If
    End If
Next
End Sub
```

Chapter 6: Getting Information from the End User

Earlier in this document you might have noticed a few examples that show how to prompt for information: `InputBox` and `MsgBox`. This chapter will present these functions in more detail, and address a few additional mechanisms that can be employed to prompt end users for information. Examples of how to do simple input validation will also be provided.

The most readily-available means of prompting the end user for information are as follows:

- `MsgBox` (VBScript built-in)
- `WshShell.Popup` (VBScript built-in)
- `crt.Dialog.MessageBox` (SecureCRT built-in)
- `InputBox` (VBScript built-in)
- `crt.Dialog.Prompt` (SecureCRT built-in)

Although a *tiny bit* more complicated (notice the sarcasm?), another way of prompting for information involves using the `InternetExplorer.Application` ActiveX object to build up a web-page form on the fly. This is currently the more flexible and powerful method, but may be overkill if your prompting needs are not that complicated.

This chapter will begin by providing examples of the more simple cases of prompting for input, and conclude with showing an example of prompting for information by means of an on-the-fly custom dialog created using the `InternetExplorer.Application` ActiveX object.

6.1 Prompting for Simple Responses: Yes, No, OK, Cancel, etc.

For simple program logic control such as presenting basic yes/no options to the user, the following methods do the job well:

- `MsgBox` (VBScript built-in)
- `crt.Dialog.MessageBox` (SecureCRT built-in)
- `WshShell.Popup` (VBScript built-in)

MsgBox

`MsgBox` is a native VBScript function that provides the foundation upon which the basic "Hello World" example VBScript code is written:

```
MsgBox "Hello World!"
```



The default when calling `MsgBox` is to display only an **OK** button within a window having a blank title. Different variations of behavior can be achieved by changing the parameters supplied to the function. The general syntax for the `MsgBox` function is as follows:

```
MsgBox(strPrompt[ , nButtons][ , strTitle])
```


The VBScript language interpreter provides built-in constants such as `vbYes`, `vbNo`, `vbYesNoCancel`, etc., that help control the appearance and behavior of the `MsgBox` window. These constants are documented within the [VScript Help](#) referenced earlier. Here is an example of a function that can be used to display information to the end user and ask if they would like to continue with script operation or cancel the script:

```
Sub Main()
    ' ...

    If Not Continue("Do you wish to continue?", "Continue?") Then Exit Sub

    ' ...

    If Not Continue("The time is now: " & _
                   Time & _
                   ", do you wish to continue?", _
                   "How about now?") Then Exit Sub

    ' ...

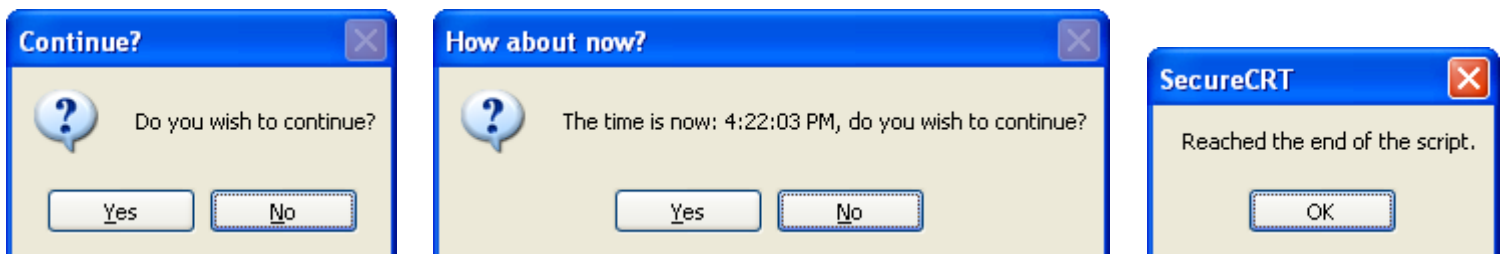
    MsgBox "Reached the end of the script."
End Sub

'~~~~~
Function Continue(strMsg, strTitle)
    ' Set the buttons as Yes and No, with the default button
    ' to the second button ("No", in this example)
    nButtons = vbYesNo + vbDefaultButton2

    ' Set the icon of the dialog to be a question mark
    nIcon = vbQuestion

    ' Display the dialog and set the return value of our
    ' function accordingly
    If MsgBox(strMsg, nButtons + nIcon, strTitle) <> vbYes Then
        Continue = False
    Else
        Continue = True
    End If
End Function
```

Running the script above presents windows displayed in the following order, assuming you've chosen "Yes" at each prompt:



If you need to present a user with three separate options, and perform a different action for each option, you could use `MsgBox` to present these options to the end user and take action based on the response provided. For example, consider the following code:

```

Sub Main()

    ' Display a message to the end user that presents 3 different options.
    ' Try to explain that each button maps to a different action.
    strMsg = "Please choose from one of the options below." & vbCrLf & _
        vbCrLf & _
        "Press" & vbtab & "Abort" & vbtab & "... to send 'ls -al'" & _
        vbCrLf & _
        "Press" & vbtab & "Retry" & vbtab & "... to send 'ps -eaf'" & _
        vbCrLf & _
        "Press" & vbtab & "Ignore" & vbtab & "... when you are done."

    Do

        ' Set the buttons as Abort, Retry, and Ignore, with the default button
        ' to the second button ("Retry", in this example)
        nButtons = vbAbortRetryIgnore + vbDefaultButton2

        ' Set the icon of the dialog to be a question mark
        nIcon = vbQuestion

        ' Display the dialog and capture the return value so that we can
        ' act upon it.
        nResult = MsgBox(strMsg, nButtons + nIcon, "Please Choose One")
        Select Case nResult
            Case vbAbort
                crt.Screen.Send "ls -al" & vbcr

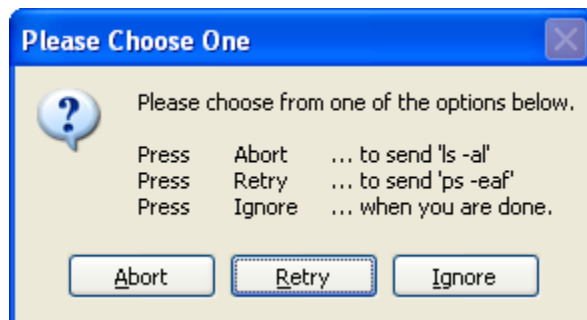
            Case vbRetry
                crt.Screen.Send "ps -eaf" & vbcr

            Case vbIgnore
                MsgBox "User Exited Script."
                Exit Do
        End Select
    Loop

End Sub

```

Running the script code above would produce a window that looks like the following:



It isn't the most user-friendly interface, and confusion can be generated by presenting a dialog with **Abort**, **Retry**, and **Ignore** buttons that really mean something different depending on how your script wishes to use them. Depending on your user base, you may want to present various options in a more user-friendly manner. In such cases where a more complete set of GUI options are needed, consider creating a custom dialog using the InternetExplorer.Application ActiveX object [as described below](#).

crt.Dialog.MessageBox

`crt.Dialog.MessageBox` functions in a manner very similar to the VBScript `MsgBox` built-in function. Arguments are provided in a slightly different order, and with the exception of one noticeable difference, the behavior is nearly identical to `MsgBox`. SecureCRT provides this function for the benefit of users that aren't using VBScript as their scripting language. JScript, for example, doesn't have a `MsgBox` equivalent, so those who find themselves using JScript can take advantage of SecureCRT's `crt.Dialog.MessageBox` function to present the user with simple prompts.

The syntax for `crt.Dialog.MessageBox` is not the same as `MsgBox`. Note that the order of the last two function arguments is reversed in comparison to VBScript's built-in `MsgBox` function:

```
crt.Dialog.MessageBox(strMessage [, strTitle [, nButtons]])
```

The earlier code listings for `MsgBox` can be rewritten to use `crt.Dialog.MessageBox` by simply replacing `MsgBox` with `crt.Dialog.MessageBox` and rearranging the order of the last two arguments. For example, the earlier "Continue" example using `crt.Dialog.MessageBox` is displayed below:

```
Sub Main()
    ' ...

    If Not Continue("Do you wish to continue?", "Continue?") Then Exit Sub

    ' ...

    If Not Continue("The time is now: " & _
                   Time & _
                   ", do you wish to continue?", _
                   "How about now?") Then Exit Sub

    ' ...

    crt.Dialog.MessageBox "Reached the end of the script."
End Sub

'~~~~~
Function Continue(strMsg, strTitle)
    ' Set the buttons as Yes and No, with the default button
    ' to the second button ("No", in this example)
    nButtons = vbYesNo + vbDefaultButton2

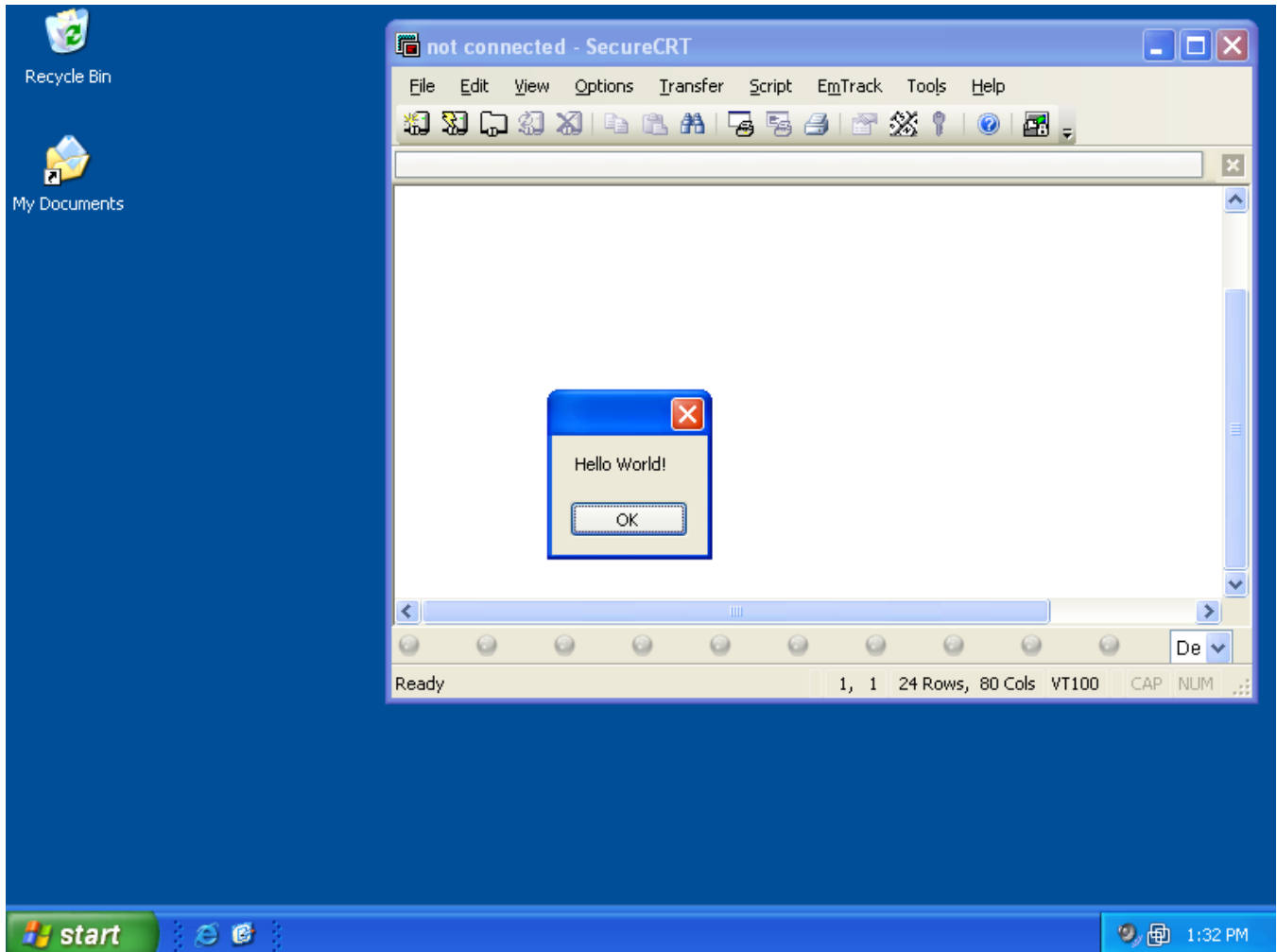
    ' Set the icon of the dialog to be a question mark
    nIcon = vbQuestion

    ' Display the dialog and set the return value of our
    ' function accordingly
    If crt.Dialog.MessageBox(strMsg, strTitle, nButtons + nIcon) <> vbYes Then
        Continue = False
    Else
        Continue = True
    End If
End Function
```

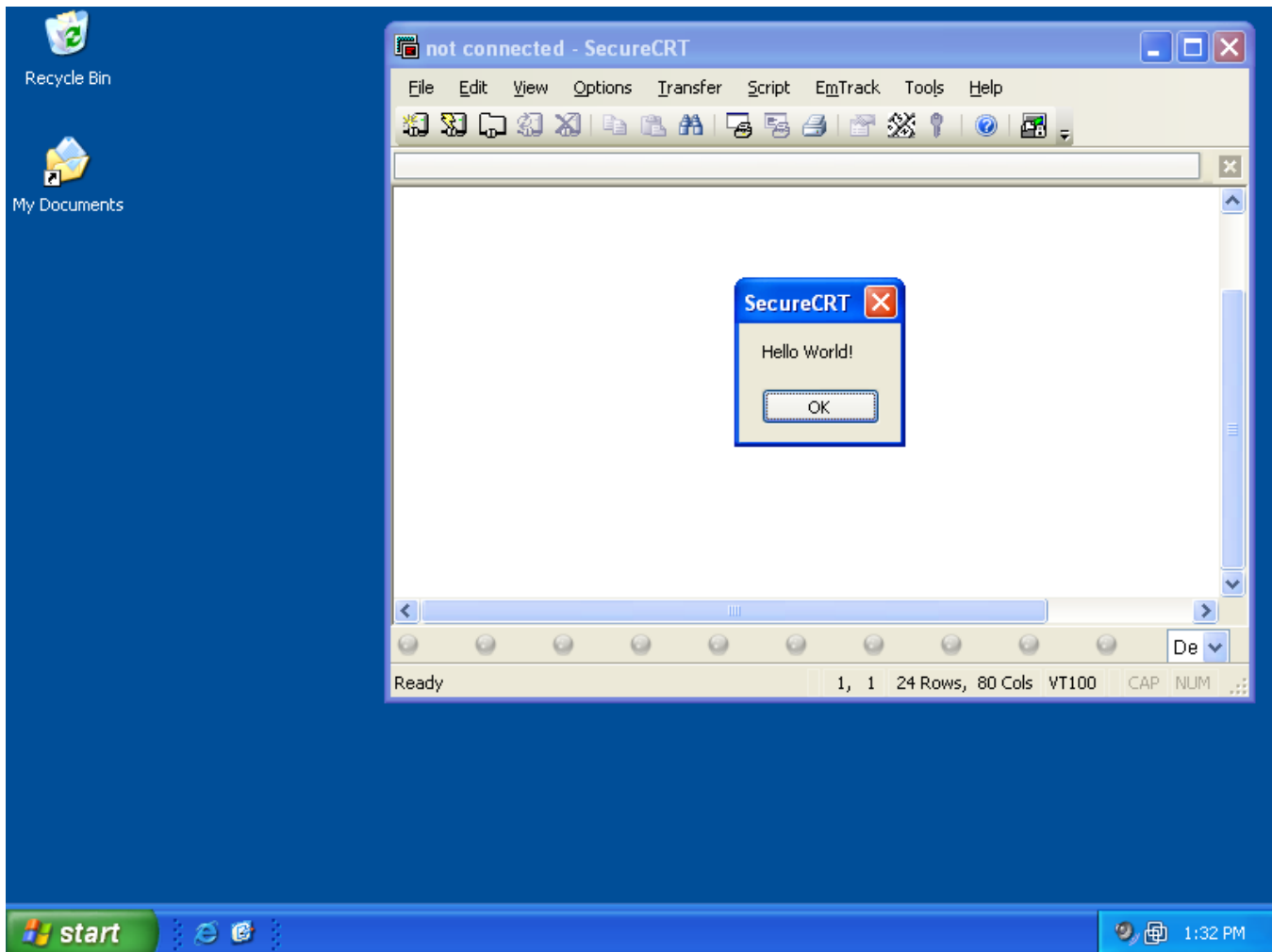
Notice that all of the built-in constants (`vbYes`, `vbNo`, `vbQuestion`, etc.) still work for `crt.Dialog.MessageBox` just as they did for VBScript's `MsgBox` function. Although the

order of the arguments is different, the meaning of the arguments still remains the same between `crt.Dialog.MessageBox` and VBScript's `MsgBox`. The main difference (other than syntax) between `MsgBox` and `crt.Dialog.MessageBox` is that the window spawned by a call to `MsgBox` always appears in the center of the desktop associated with the primary monitor, whereas the window spawned by a call to SecureCRT's `crt.Dialog.MessageBox` always appears centered within the SecureCRT application window.

Also of note in terms of difference in behavior compared to `MsgBox`, `crt.Dialog.MessageBox` provides a default window title of "SecureCRT", rather than leaving the window title blank.



VBScript's `MsgBox` (no title; centered on desktop, rather than on SecureCRT's application window)



`crt.Dialog.MessageBox` (default title; centered on SecureCRT's main application window)

WshShell.Popup

`WshShell.Popup` is a built-in VBScript function that is similar to `MsgBox`, but it allows for specification of a timeout parameter that causes the dialog to be automatically dismissed after a specified number of seconds if there is no user action taken. This function can be useful in situations where you may want to prompt for a choice, but take the default action if no response is given within the timeframe specified. One such example is a scenario in which you have an automated script that performs the same command repeatedly, providing an opportunity to interactively cancel the script every tenth time the command is sent. A prompt is displayed asking if execution of the script should continue. After five seconds, if a response is not provided by the end user, the default action is taken to continue with the script.

```
' Number of times the command should be sent to the remote before prompting if
' the script should continue:
g_nMaxRepetitions = 10

' The number of seconds the continuation prompt should appear before being
' dismissed automatically.
g_nTimeoutSeconds = 5

' The number of seconds to wait between each time the command is sent
```

```

g_nCommandDelaySeconds = 2

' The command we'll be sending to the remote every g_nCommandDelaySeconds.
g_strCommand = "top -n 1"

' Reference to the WshShell object that will provide the Popup() function
Set g_objShell = CreateObject("WScript.Shell")

'~~~~~
Sub Main()
    nCounter = g_nMaxRepetitions
    Do
        If nCounter = 0 Then
            If Not Continue("Do you wish to continue script execution?", _
                "Continue?") Then Exit Do

            ' Reset the repetition counter
            nCounter = g_nMaxRepetitions
        Else
            ' Before sending our command(s), let's check to ensure that we have
            ' an active connection
            If Not crt.Session.Connected Then
                Do
                    If Continue("Connection lost. Should we reconnect?", _
                        "Reconnect?") <> True Then Exit Sub

                    ' crt.Session.Connect will perform a reconnect (this script
                    ' assumes that it was launched with an active connection
                    ' already present).
                    On Error Resume Next
                    crt.Session.Connect
                    nError = Err.Number
                    strErr = Err.Description
                    On Error Goto 0
                    If nError = 0 Then Exit Do
                Loop
            End If

            crt.Screen.Send "top -n 1" & vbcr

            nCounter = nCounter - 1

            ' Delay for a specified number of seconds between commands.
            ' Since crt.Sleep takes MS, convert the variable from Seconds...
            crt.Sleep g_nCommandDelaySeconds * 1000
        End If
    Loop
End Sub

'~~~~~
Function Continue(strMsg, strTitle)
    ' Set the buttons as Yes and No, with the default button
    ' to the Second button ("Yes", in this example)
    nButtons = vbYesNo + vbDefaultButton

    ' Set the icon of the dialog to be a question mark
    nIcon = vbQuestion

    ' Display the dialog and set the return value of our
    ' function accordingly
    nResult = g_objShell.Popup(strMsg, _
        g_nTimeoutSeconds, _

```

```

                                strTitle, nButtons + nIcon)
    If nResult = vbNo Then
        Continue = False
    Else
        Continue = True
    End If
End Function

```

6.2 Prompting for Arbitrary Text Input

If prompting the end user for input via a simple button push isn't sufficient, and you need to present more of a "fill in the blank" prompt, this section will explain a couple of methods that can be used to accomplish your goal. Once you've received input from the end user, you might also want to validate it to ensure that it is the right type of data, within a certain range, or meets your other criteria.

InputBox

`InputBox` is a native VBScript function that presents a simple window allowing input of arbitrary text data. Since the first three function parameters for `InputBox` match the order and meaning of the first three function parameters for `crt.Dialog.Prompt`, you may want to consider always using `crt.Dialog.Prompt` since it allows for hiding user input – a feature that works very well when prompting for sensitive information such as a user's password. Although the functionality of `InputBox` is almost identical to `crt.Dialog.Prompt`, the behavior between the two functions differs in the following areas:

- SecureCRT's `crt.Dialog.Prompt` function allows you to hide the text that the end user types, thereby facilitating the entry of passwords and other sensitive information that needs to be hidden from the curiosity of those who might be looking over your shoulder.
- The fourth and fifth parameters allowed by `InputBox` allow you to control the coordinates of the upper left-hand corner of the window that is displayed, so you can position it where you want on the screen. SecureCRT's `crt.Dialog.Prompt` always displays the input window centered within the corresponding SecureCRT application window.

Here's an example of using VBScript's `InputBox` function to prompt for user text input that is then used to attempt a connection to a specified host:

```

Sub Main()

    strHost = InputBox(_
        "Please specify the host to connect:", _
        "Target Host?", _
        "192.168.0.1")
    If strHost = "" Then Exit Sub

    strUsername = InputBox(_
        "Please enter your username:", _
        "Username?")
    If strUsername = "" Then Exit Sub

    nRetryAttempts = InputBox(_
        "How many connection attempts should be made?", _
        "Retry Count?", _
        3)
    If nRetryAttempts = "" Then Exit Sub

```

```

nRetryDelaySeconds = InputBox(_
    "How many seconds between retry attempts?", _
    "Retry Delay?", _
    1)
If nRetryDelaySeconds = "" Then Exit Sub

strConnectString = "/SSH2" & _
    " /L " & strUsername & _
    " " & strHost

If Connect(strConnectString, _
    nRetryAttempts, _
    nRetryDelaySeconds) Then
    crt.Dialog.MessageBox "Successfully connected to " & strHost
Else
    crt.Dialog.MessageBox "Failed to connect to " & strHost
Exit Sub
End If

' Do work, now that we know we're connected to the specified host...
' .
' .
' .

End Sub

'~~~~~
Function Connect(strConnectString, nRetryAttempts, nRetryDelaySeconds)
    nRetryCount = nRetryAttempts
    Do
        On Error Resume Next
        crt.Session.Connect strConnectString
        nError = Err.Number
        strErr = Err.Description
        On Error Goto 0

        If nError = 0 Then
            Connect = True
            Exit Function
        End If

        ' Convert nRetryTimeoutSeconds to MS, which is what crt.Sleep
        ' requires; delay script execution for the amount of time specified.
        crt.Sleep nRetryDelaySeconds * 1000

        nRetryCount = nRetryCount - 1
    Loop Until nRetryCount < 1
End Function

```

crt.Dialog.Prompt

The list below provides a summary of the benefits `crt.Dialog.Prompt` offers over the standard `InputBox` function provided by VBScript:

- The input window is automatically centered within SecureCRT's main application window.
- The title of the window is set automatically to "SecureCRT" if a title parameter isn't specified.
- Sensitive input (like passwords) can be hidden, preventing prying eyes from snooping sensitive information.

Since SecureCRT's `crt.Dialog.Prompt` function offers the ability to hide data being entered, the script code for `Sub Main()` presented in the previous section can be modified to include prompting the end user for a password, without putting the end user at risk of over-the-shoulder onlookers.

```
Sub Main()

    strHost = crt.Dialog.Prompt(_
        "Please specify the host to connect:", _
        "Target Host?", _
        "192.168.0.1")
    If strHost = "" Then Exit Sub

    strUsername = crt.Dialog.Prompt(_
        "Please enter your username:", _
        "Username?")
    If strUsername = "" Then Exit Sub

    strPassword = crt.Dialog.Prompt(_
        "Please enter your password:", _
        "Password?", _
        "", _
        True)
    If strPassword = "" Then Exit Sub

    nRetryAttempts = crt.Dialog.Prompt(_
        "How many connection attempts should be made?", _
        "Retry Count?", _
        3)
    If nRetryAttempts = "" Then Exit Sub

    nRetryDelaySeconds = crt.Dialog.Prompt(_
        "How many seconds between retry attempts?", _
        "Retry Delay?", _
        1)
    If nRetryDelaySeconds = "" Then Exit Sub

    strConnectionString = "/SSH2" & _
        " /L " & strUsername & _
        " /PASSWORD " & strPassword & _
        " " & strHost

    If Connect(strConnectionString, _
        nRetryAttempts, _
        nRetryDelaySeconds) Then
        crt.Dialog.MessageBox "Successfully connected to " & strHost
    Else
        crt.Dialog.MessageBox "Failed to connect to " & strHost
        Exit Sub
    End If

    ' Do work, now that we know we're connected to the specified host...
    ' .
    ' .
    ' .

End Sub
```

Validating User Input

Validating user input can save time and potentially reduce frustration in the long run. For example, in the sample script provided in the previous section, the end user is prompted for the host to which a connection is to be made. If the end user specifies an IP address that isn't valid, for example, the connection attempt will fail... but not until a timeout occurs after trying to contact the remote host.

There are a seemingly infinite variety of ways in which user-supplied information can be validated. This section will present a couple of different approaches:

- `IsNumeric()` built-in VBScript function
- Regular Expressions and VBScript code

IsNumeric

If you're prompting the end user for a number, the VBScript built-in `IsNumeric` function can be used to verify that the data provided is actually a number. Consider the following example that demonstrates how to clone the active tab a specified number of times. Note how the `IsNumeric` function is used to validate the input provided by the end user:

Solution: Clone Current Tab Multiple Times

```
' CloneCurrentTabMultipleTimes.vbs

' Description:
'   This example script demonstrates how to create multiple cloned tabs based
'   on user input.
'
' Demonstrates:
'   - How to get the script tab.
'   - How to get the protocol used for connection by querying session options.
'   - One way of prompting for and validating numerical input is between a
'     range of numbers.
'   - How to clone an active tab.

Sub Main()

    Set objTab = crt.GetScriptTab

    If objTab.Session.Connected <> True Then
        crt.Dialog.MessageBox "Not connected. Current tab must be connected" & _
            " with Telnet or SSH2."
        Exit Sub
    Else
        strProtocol = objTab.Session.Config.GetOption("Protocol Name")
        If strProtocol <> "Telnet" And strProtocol <> "SSH2" Then
            crt.Dialog.MessageBox "Invalid protocol. Current tab must be " & _
                "connected with Telnet or SSH2."
            Exit Sub
        End If
    End If

    Dim nMaxClones
    nMaxClones = 20

    Dim nCloneCount
    nCloneCount = 8

    Do
        nCloneCount = crt.Dialog.Prompt(_
```

```

        "Current tab is: " & objTab.Caption & vbCrLf & _
        vbCrLf & _
        "How many clones would you like to make?", _
        "Clone Current Tab", _
        nCloneCount)
    If nCloneCount = "" Then Exit Sub
    If Not IsNumeric(nCloneCount) Then
        crt.Dialog.MessageBox "Invalid number. Please enter a number " & _
            "between 1 and " & nMaxClones
    Else
        ' Convert user-supplied data to a number so we can safely perform
        ' comparisons.
        nCloneCount = CInt(nCloneCount)
        If (nCloneCount >= 1) And (nCloneCount <= nMaxClones) Then
            Exit Do
        Else
            crt.Dialog.MessageBox "Number out of range. Please enter a " & _
                "number between 1 and " & nMaxClones
        End If
    End If
End If

Loop

For nIndex = 1 To nCloneCount
    objTab.Clone
    crt.Sleep 50
Next
End Sub

```

Regular Expressions

You may recognize regular expressions from earlier reading in section 4.3, where the topic of parsing data received from the remote was addressed. Because regular expressions are pattern-matching instructions, they can also be used as a mechanism for some types of end user input validation.

Regular expression syntax for use with the `RegExp` object native to VBScript is documented as part of the [WScript56.chm file available for download from Microsoft](http://www.microsoft.com/MSDN/EN-US/library/1400241x(VS.85).aspx), as well as in online format on the MSDN web site:

[http://msdn.microsoft.com/en-us/library/1400241x\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/1400241x(VS.85).aspx)

One example of using a regular expression to validate user input is checking to see if a user-supplied IP address is valid. Here's an example that demonstrates how you can use a regular expression to determine whether or not end user input represents a valid IP address:

```

Sub Main()
    ' Set up a default value to use when we prompt the end user.
    strAddress = "192.168.0.1"

    Set re = New RegExp
    ' Set up a pattern that starts at the beginning ("^"), and looks for a
    ' digit that's between 1 and 3 characters long ("\d{1,3}"), followed by
    ' a dot character ("\.") (repeated again 3 times, but the last one
    ' doesn't have a trailing dot character), followed immediately by the
    ' end of the input ("$$"):
    re.Pattern = "^(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})$"

    ' This loop allows us to continue prompting for valid input until either

```

```

' the user cancels, or provides a valid IP address.
Do
    strAddress = crt.Dialog.Prompt(_
        "Please specify an IPv4 address", _
        "Enter IP Address", _
        strAddress)
    If strAddress = "" Then Exit Sub

    Set matches = re.Execute(strAddress)

    If matches.Count < 1 Then
        MsgBox "Invalid IPv4 address entered!"
    Else
        ' Now check to see if each octet is within a valid range
        bValid = True
        For Each nOctet In matches(0).Submatches
            If nOctet < 0 Or nOctet > 255 Then
                crt.Dialog.MessageBox _
                    "Invalid IPv4 address entered. " & _
                    "Each octet must be >= 0 and <= 255."
                bValid = False
            End If
        Next
        If bValid Then Exit Do
    End If
Loop

crt.Dialog.MessageBox _
    "Congratulations! You entered a valid IPv4 address: " & strAddress
End Sub

```

You may notice that in the example above, the validation isn't done 100% by means of a regular expression. The regular expression pattern simply validates that the pattern of 1-3 numbers ("`\d{1,3}`"), followed by a '.' character ("`\.`") is repeated 3 times, followed by another 1-3 numbers. If this pattern is found, then a further check is done via VBScript code to ensure that each octet is within a valid range for IPv4 addresses. If desired, the complete validation could be done by creating a more complex regular expression pattern to be matched. However, depending on the pattern you are attempting to match, it may be much more difficult to construct a pattern that correctly matches the desired expression. The remainder of this section might not only a good example of how to build up a regular expression pattern, but may also serve as an example of how difficult it can be to match certain patterns solely through the use of a regular expression.

To create a regular expression that matches any number between 0 and 255, you'll need to start with a simple pattern expression and build it up piece-by-piece until you arrive at the complex pattern that matches all possible valid data. Here's a step-by-step explanation:

1. First, start by creating a pattern that will match single digits, between 0 and 9 so that all octets within addresses 0.0.0.0 through 9.9.9.9 can be matched: `[0-9]`.
2. Second, the pattern for matching numbers 10 – 99 is as follows: `[1-9][0-9]`.

Combine this pattern with the previous pattern using the logical OR operator (specified using the ASCII *pipe* character, "|"; this is also known as *alternation*, in terms of regular expressions) to accommodate double digits 10-99 so that all octets within addresses 0.0.0.0 through 99.99.99.99 can be matched: `[1-9][0-9][0-9]`.

3. Third, the pattern for matching numbers 100-199 is as follows: `1[0-9][0-9]`.

Combine the 100-199 pattern with the previous pattern using the logical OR operator, since single, double, and triple digit combinations are allowed in IPv4 addresses: `1[0-9][0-9]|[1-9][0-9][0-9]`.

4. Fourth, the pattern for matching numbers 200-255 must be specified in separate chunks: one to match numbers 200-249, and another to match numbers 250-255:

Pattern for matching 200-249: `2[0-4][0-9]`

Pattern for matching 250-255: `25[0-5]`

Since either pattern is legitimate, they should be combined with a logical OR operator to match all numbers 200-255: `25[0-5]|2[0-4][0-9]`.

Combine this pattern with the previous patterns to match all available, legitimate values for an IPv4 octet: `25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9]`.

To group the logical OR statements in the above pattern so that the octet pattern can be combined with additional patterns, it needs to be wrapped in parentheses: `(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9])`. This pattern matches all numbers between 0 and 255.

5. The next step would be to combine the above pattern together to match 3 octets separated by a dot character, ".", followed by a fourth octet pattern: `(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9])`.

Since there are three instances of an octet followed by a dot character in a legitimate IPv4 address, the first three "duplicates" of the octet pattern followed by a dot character can be written in shorthand as follows:

`((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9])\.)\{3}`

Note that the sequence of logical OR statements must be wrapped in parentheses, otherwise, the regular expression pattern would match just a single "." character all by itself (whether or not there was a legitimate number in front of it). The entire octet plus trailing dot character sequence is wrapped in parentheses in order to specify that it should appear three times, as instructed by the trailing "{3}" component of the above pattern.

6.3 Building Custom Dialogs or Forms

If you are a subscriber to the VanDyke Software [News You Can Use](#) newsletter, this section may look somewhat familiar to you because portions of this chapter were published earlier in the "Scripting Corner" of the May, 2008 newsletter.

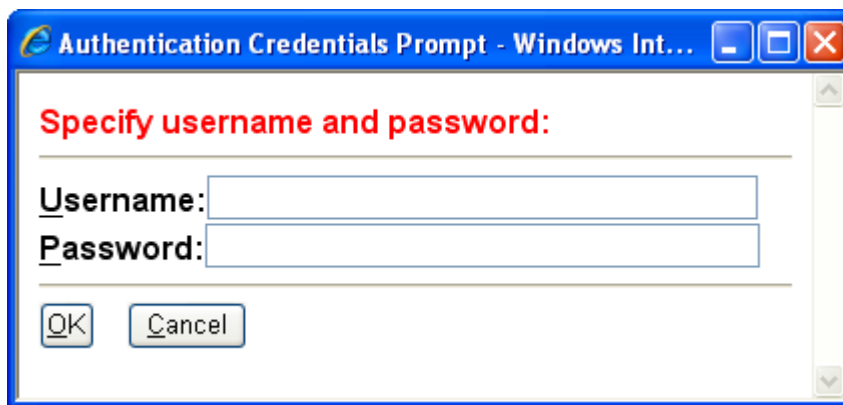
A common need in scripting is the ability to create GUI interfaces for user interaction, such as gathering username and password data, or displaying script output. While VBScript's `MsgBox` and `InputBox` functions let you control simple interactions, you can create more sophisticated dialogs using HTML code with the Internet Explorer (IE) object. This section of the scripting manual walks you through the process of building a dialog, displaying the dialog, and handling events that occur within the dialog.

The process of implementing a custom dialog for use with SecureCRT scripts involves the following steps:

1. [Building the GUI](#)
2. [Displaying the GUI](#)
3. [Handling UI Events](#)

Building the GUI

For the purpose of this example, the dialog that you will be creating is a simple dialog with two text fields (one of which is a password field, where input can be hidden), and two buttons. Here is a screenshot of the example custom dialog:



The GUI interface for this custom dialog will be built using HTML code displayed through Internet Explorer. There are two ways to store the HTML code.

- Creating an HTML file with the HTML code, or
- Dynamically specifying the HTML code within script code

The example in this section will use the second method: dynamically specifying the HTML code within the script code. Below is an example of what this code might look like within a SecureCRT script to build a dialog that displays fields for entering username and password, and displays **OK** and **Cancel** buttons:

```
strHTMLBody = _  
    "<font color=red><b>Specify username and password:</b></font>" & _
```



```

g_objIE.Toolbar = False

' Set the initial size of the IE window
g_objIE.height = 200
g_objIE.width = 425

' Set the title of the IE window
g_objIE.document.Title = "Authentication Credentials Prompt"
g_objIE.Visible = True

' This loop is required to allow the IE window to fully display
' before moving on
Do
    crt.Sleep 100
Loop While g_objIE.Busy

' This code brings the IE window to the foreground.
Set objShell = CreateObject("WScript.Shell")
objShell.AppActivate g_objIE.document.Title

' Once the dialog is displayed and has been brought to the
' foreground, set focus on the control of our choice...
g_objIE.Document.All("Username").Focus

```

If you run the script code above, you'll notice that a window is displayed, as expected, but nothing happens when pressing the OK and Cancel buttons. This is because the buttons aren't configured to allow the script code to detect when they have been pressed. The button definitions must be modified slightly, and script code will need to be added to detect and handle these button click events.

Handling UI Events

There are four main components of an event handler for our custom dialog using IE:

1. A hidden variable in the HTML code (`ButtonHandler` in the example below; this hidden input variable is used to store the name of the button that has been pressed)
2. `OnClick` action configurations for each button in the dialog to set the `ButtonHandler` value to the name of the button that was pressed
3. A `Do` loop that waits for any change in the `ButtonHandler` variable
4. A `Select Case` code block that detects and handles each button press event

The Hidden `ButtonHandler` Variable

This variable is included in the HTML code and is updated every time something important happens, such as the user clicking **OK** or **Cancel**. The following HTML code for the **OK** and **Cancel** buttons sets the value of our hidden `ButtonHandler` variable appropriately:

OK button definition:

```

"<button name=OK AccessKey=O " & _
    "OnClick=document.all("ButtonHandler").value="OK";" & _
"><u>O</u>K </button>" & _

```

Cancel button definition:

```

"<button name=Cancel AccessKey=C " & _
    "OnClick=document.all("ButtonHandler").value="Cancel";" & _
"><u>C</u>ancel</button>" & _

```

ButtonHandler hidden input definition:

The following code creates the `ButtonHandler` variable and sets its initial value to "Nothing Clicked Yet" as an explicit indication that no buttons have been pressed yet:

```
"<input name=ButtonHandler value="Nothing Clicked Yet" " & _
    "type=hidden >"
```

The Do Loop

The `Do` loop is used in the SecureCRT script to continue looping while checking the value of the `ButtonHandler` variable. In the event that the IE window is closed by means other than the **Cancel** button, this will be handled as well.

The Select Case

The `Select Case`, contained in the `Do` loop, is used to act on the different inputs received from the user inside a custom dialog. Since the `Do` loop does not include a `While` statement that would inherently exit the loop should a certain clause be met, code to exit the loop will need to be included in the `Select Case`.

An example of the complete `Do` loop with the embedded `Select Case` is below:

```
Do
    ' If the user closes the IE window by Alt+F4 or clicking on the 'X'
    ' button, we'll detect that here, and exit the script if necessary.
    On Error Resume Next
        Err.Clear
        strNothing = g_objIE.Document.All("ButtonHandler").Value
        if Err.Number <> 0 then exit do
    On Error Goto 0

    ' Check to see which buttons have been clicked, and address each one
    ' as it's clicked.
    Select Case g_objIE.Document.All("ButtonHandler").Value
        Case "Cancel"
            ' The user clicked Cancel. Exit the loop
            g_objIE.quit
            Exit Do

        Case "OK"
            ' The user clicked OK. Act on the information in the
            ' Username and Password fields.
            ' Capture data from each field in the dialog...
            strUsername = g_objIE.Document.All("Username").Value
            strPassword = g_objIE.Document.All("Password").Value
            g_objIE.quit
            ' Now that we have closed the IE dialog, we can act on our data.
            ' For this example, we will just display the information entered
            ' in a text box.
            If crt.Dialog.MessageBox("Username provided: " & strUsername & _
                vbCrLf & vbCrLf & _
                "Would you also like to see the " & _
                "password that was entered?", _
                "Data Display", _
                vbYesNo) <> vbYes Then Exit Do

            crt.Dialog.MessageBox "Here is the password that was entered:" & _
                vbCrLf & vbTab & _
                "Password: " & strPassword
```



```

g_objIE.Visible = True

' This loop is required to allow the IE window to fully display
' before moving on
Do
    crt.Sleep 100
Loop While g_objIE.Busy

' This code brings the IE window to the foreground (otherwise, the window
' would appear, but it would likely be behind the SecureCRT window, not
' easily visible to the user).
Set objShell = CreateObject("WScript.Shell")
objShell.AppActivate g_objIE.document.Title

' Once the dialog is displayed and has been brought to the
' foreground, set focus on the control of our choice...
g_objIE.Document.All("Username").Focus

Do
    ' If the user closes the IE window by Alt+F4 or clicking on the 'X'
    ' button, we'll detect that here, and exit the script if necessary.
On Error Resume Next
    Err.Clear
    strNothing = g_objIE.Document.All("ButtonHandler").Value
    if Err.Number <> 0 then exit do
On Error Goto 0

    ' Check to see which buttons have been clicked, and address each one
    ' as it's clicked.
Select Case g_objIE.Document.All("ButtonHandler").Value
    Case "Cancel"
        ' The user clicked Cancel. Exit the loop
        g_objIE.quit
        Exit Do

    Case "OK"
        ' The user clicked OK. Act on the information in the
        ' Username and Password fields.
        ' Capture data from each field in the dialog...
        strUsername = g_objIE.Document.All("Username").Value
        strPassword = g_objIE.Document.All("Password").Value
        g_objIE.quit
        ' Now that we have closed the IE dialog, we can act on our data.
        ' For this example, we will just display the information entered
        ' in a text box.
        If crt.Dialog.MessageBox("Username provided: " & strUsername & _
            vbCrLf & vbCrLf & _
            "Would you also like to display the " & _
            "password that was entered?", _
            "Data Display", _
            vbYesNo) <> vbYes Then Exit Do

        crt.Dialog.MessageBox "Here is the password that was entered:" & _
            vbCrLf & vbTab & _
            "Password: " & strPassword
        Exit Do
End Select

    ' Wait for user interaction with the dialog...
    crt.Sleep 200
Loop

```

Chapter 7: Logging, Reading, and Writing Files

Whether you are saving a history of commands entered for auditing purposes, reading in fields from a CSV file, or backing up a Cisco router, you may at some point need to write data to or read data from a file. This chapter introduces the logging methods provided by SecureCRT's **Session** object.

Techniques for reading and writing data from and to files using VBScript's built-in **FileSystemObject** and its related objects and methods are also discussed.

7.1 Logging with SecureCRT's Session Object

SecureCRT provides for session logging either on the fly by choosing **File / Log Session** or **File / Raw Log Session** from SecureCRT's main menu, or by setting up logging to occur automatically with a pre-configured session in the **Terminal / Log File** category of the Session Options dialog. The method required for controlling log files within a script depends on whether you are connecting in an [ad hoc](#) manner versus connecting with an [pre-configured session](#).

Logging while Connected with an Ad Hoc Connection

If you are connected to a remote device with an ad hoc connection (see [section 3.2](#) in Chapter 3), you can enable logging using the `Session.Log` method, which has the following general syntax:

```
crt.Session.Log bStart[, bAppend[, bRaw]]
```

bStart: Boolean value indicating whether or not logging should be started (**True**) or stopped (**False**).

bAppend: Boolean value instructing SecureCRT to either append (**True**) to any existing log file, or overwrite (**False**, default) any existing log file.

bRaw: Boolean value instructing SecureCRT to either log in raw mode (**True**), or normal mode (**False**, default). A raw log in SecureCRT captures all data received from the remote system, including control codes and escape sequences. A normal log file captures only data received as displayed within the terminal window in SecureCRT.

Specifying a Log File Name

If you have not yet specified a log file name within your script code, the end user will be prompted for the log file name as soon as the `crt.Session.Log True` statement is executed. To specify a log file name, set the `crt.Session.LogFileName` property. For example:

```
' Specify a log file name that is time-stamped (with legal
' filename characters). This way, the files can be sorted
' easily within Windows Explorer.
crt.Session.LogFileName = _
    "C:\Temp\Logs\%Y-%M-%D--%h-%m-%s.%t-%S(%H).txt"
```

TIP: As indicated in the example above, you can take advantage of the log file name substitutions supported by SecureCRT when specifying a log file name. A table of the substitutions supported in SecureCRT 6.2 is included here for your

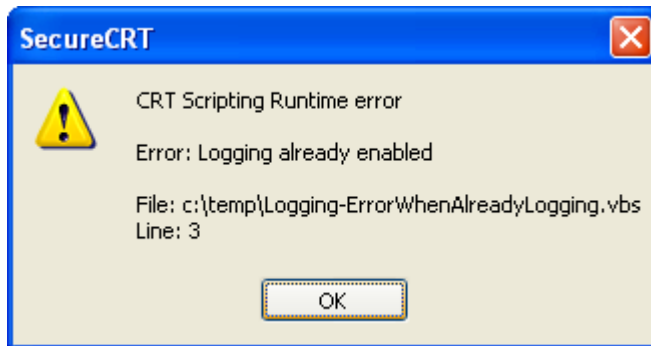
convenience. Note that these SecureCRT log file name substitution variables are case sensitive in contrast to Windows environment variables.

Variable	Substitution Value
%H	Value of the Hostname field as specified in Session Options. If connection is ad hoc, the value for this substitution is always the hostname or IP address of the remote device when using a protocol other than Serial or TAPI. If using the Serial protocol, %H is substituted as "Serial-COM#", where "#" indicates the actual COM port to which the connection was made. When connecting with TAPI, %H is substituted with the phone number that was dialed.
	<p>Example #1:</p> <pre>crt.Session.Connect "/SERIAL COM1" crt.Session.LogFileName = "C:\%S--%H.log" ' Above line results in a log file named: ' "Default--Serial-COM1.log"</pre>
	<p>Example #2:</p> <pre>crt.Session.Connect "/SSH2 192.168.0.1" crt.Session.LogFileName = "C:\%S--%H.log" ' Above line results in a log file named: ' "Default--192.168.0.1.log"</pre>
%S	The name of the Session, for a pre-configured session. If connection is ad hoc, the value for this substitution is always "Default", since with ad hoc connections the Default session configuration is used.
%Y	4-digit year (e.g., "2009")
%M	2-digit month (e.g., "05")
%D	2-digit day (e.g., "19")
%h	2-digit hour (24-hour, e.g., "17")
%m	2-digit minutes (e.g., "40")
%s	2-digit seconds (e.g., "32")
%t	3-digit milliseconds (e.g., "894")
%%	A literal "%" character (if you want one in your log file name)
%EnvVar%	Value of environment variable named "EnvVar" (e.g., %TEMP%). Note that environment variable substitution occurs first.

In such cases where an end user may have already started logging to a file manually (say, before your script has been launched), you should first check to ensure logging isn't already enabled before you attempt to start logging from within your script. If logging has already been started, calling `Session.Log` will result in an error. For example:

```
' Attempting to enable logging if it has already been
' enabled will result in an error:
crt.Session.Log True
```

The above code will result in the following error if logging has already been started:



If you are unsure whether or not logging has been enabled already, you can determine this programmatically with an `If` statement within your script code:

```
If crt.Session.Logging <> True Then
    crt.Session.Log True
End If
```

If at any point within your script code, you desire to turn off logging, simply use the `Session.Log` method, providing an argument of `False`. For example:

```
crt.Session.Log False
```

If logging is already enabled, and you want to ensure that specific log settings are active (using a specific log file name, append vs. overwrite, etc.), then you may want to stop the logging process, configure log settings within your script, and then enable logging as per your requirements. For example:

```
If crt.Session.Logging Then
    ' Turn off logging before setting up our script's
    ' logging...
    crt.Session.Log False
End If

' Set up the log file so that it resides in the path
' defined in the TEMP environment variable, and is
' named according to the remote IP address:
crt.Session.LogFileName = "%TEMP%" & _
    "LogFileForIP(" & crt.Session.RemoteAddress & ").txt"

crt.Session.LogUsingSessionOptions

' Send commands, etc.
' ...

' Stop logging
```

```

crt.Session.Log False

' Perform other script work, or setup on the remote for the
' next batch of logging...
' ...

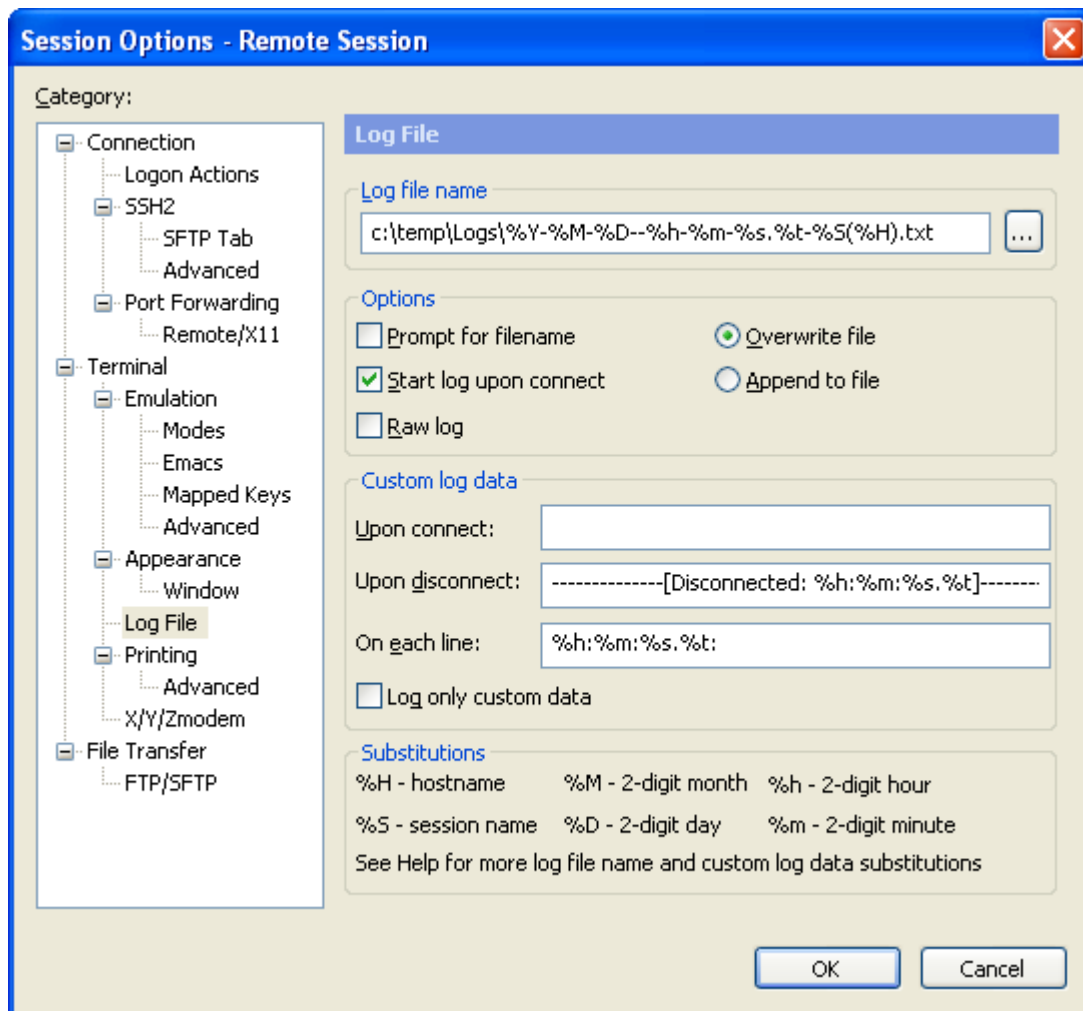
' Start logging again (be sure to append or else we'll
' overwrite the data we've logged so far).
crt.Session.Log True, True

' Send more commands, etc.
' ...
' ...

```

Logging while Connected with a Pre-configured Session

A log file configuration can also be set up within a session for ease of use with repetitive logging needs (Session Options dialog, **Terminal / Log File** category). Consider the following example session logging configuration:



If you are connecting with a [pre-configured session](#) for which you have already configured log file settings similar to what is displayed above, you can start logging with that session by

calling the `LogUsingSessionOptions` method associated with the session to which you are currently connected. Since all of the settings are defined within the `Session` configuration, there are no arguments provided to the `LogUsingSessionOptions` method. For example:

```
If crt.Session.Logging Then
    ' Turn off logging before setting up our script's logging...
    crt.Session.Log False
End If

' Turn on logging; use session option log file configuration
' (if used with an ad hoc session, the Default session's log
' file configuration is used).
crt.Session.LogUsingSessionOptions

' Send commands, etc.
' ...

' Stop logging
crt.Session.Log False
```

Note: The `LogUsingSessionOptions` method is also available with ad hoc connections. If the **Session** object is associated with an ad hoc connection, the Default session's log file settings will be used when `LogUsingSessionOptions` is called.

7.2 Reading Data from Files Using the `FileSystemObject`

In some situations, you may have connection data stored in a `.csv` file, or some other type of flat text file. You can make use of the `FileSystemObject` to read in data from a file for use within a SecureCRT script. Reading data from a file using the VBScript `FileSystemObject` can typically be accomplished using the following general steps:

- 1) Get a reference to the `FileSystemObject`:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

- 2) Use the `FileSystemObject` reference to retrieve a reference to a `TextStream` object associated with the text file from which the data is to be read:

```
Set objStream = objFSO.OpenTextFile("C:\Temp\MyDataFile.txt")
```

- 3) Read the data from the `TextStream` object using a loop to read in each line. Alternatively, if the data file isn't too large (less than 1-2 MB in size), read in all the data at once and use the `Split()` method to create an array representing each line of the file that was read:

```
Do While objStream.AtEndOfStream <> True
    strLine = objStream.ReadLine
    '...
Loop
```

Or:

```
strFileData = objStream.ReadAll
objStream.Close
vLines = Split(strFileData, vbCrLf)
For each strLine in vLines
```

Next

The risk of using `ReadAll()` to read a file that is larger than 1-2 MB in size is decreased performance and increased memory usage. You may find this threshold smaller or larger, depending on the capabilities of the computer on which you are running SecureCRT.

4) Close the file.

```
objStream.Close
```

In some scenarios, you may even find the need to read specific data from one file, and additional data from one or more different files. You may also desire to establish a convention you can use to indicate that a line should be ignored by introducing the concept of a "comment" character. The following example code shows how you can read in the contents of a file, ignoring blank lines as well as lines considered "comments" by the presence of a comment character found at the beginning of the line.

```
' ReadDataFromFile.vbs
' Reads in data line-by-line from a file and aggregates the information
' into an array. We then iterate over each element of the array and
' display the non-comment lines in a message box.
'
' This script assumes a data file named "MyDataFile.txt" lives in your
' "My Documents" folder.
'
' Example file contents that this script would expect (remove leading comment
' characters ("")s ) are provided in the comment lines below:
'
'     # This is a comment line
'     # This is a comment line, too.
'     pager 0
'
'     # Blank lines are ignored (like above), and counted as comment lines
'     # The following 2 lines will not be ignored:
'     sh version
'     sh config

Dim g_fso, g_shell
Set g_fso = CreateObject("Scripting.FileSystemObject")
Set g_shell = CreateObject("WScript.Shell")

g_strDataFile = g_shell.SpecialFolders("MyDocuments") & "\MyDataFile.txt"
g_strComment = "#"

MainSub

Sub MainSub()

    Dim vLines()
    Dim nLineCount, nCommentLines
    If Not ReadDataFromFile(g_strDataFile, _
                           g_strComment, _
                           vLines, _
                           nLineCount, _
                           nCommentLines) Then
        MsgBox "No lines were found in file: " & vbCrLf & vtab & g_strDataFile
        Exit Sub
    End If
```

```

' Iterate through each element of our vLines array and compose a
' message
For nIndex = 1 To nLineCount
    ' Arrays are indexed at zero...
    strFileText = strFileText & _
        "Line #" & nIndex & ": " & vLines(nIndex - 1) & vbCrLf
Next

' Note that the MsgBox built-in VBScript function will truncate the
' data displayed in the message box window at 256 characters. So,
' even though strFileText will contain all the data from the file,
' if the file is rather large, it may not all be displayed in the
' message box. However, at least you won't have a window that is too
' large to see the OK button at the bottom :).
MsgBox "Total number of non-blank, non-comment lines from file: " & _
    nLineCount & vbCrLf & _
    "Total number of comment & blank lines found: " & nCommentLines & _
    vbCrLf & vbCrLf & _
    g_strDataFile & ":" & _
    vbCrLf & strFileText
End Sub

'~~~~~
Function ReadDataFromFile(strFile, _
    strComment, _
    ByRef vLines, _
    ByRef nLineCount, _
    ByRef nCommentLines)
' Returns True if the file was found.
'   strFile: IN parameter specifying full path to data file.
'   strComment: IN parameter specifying string that preceded
'               by 0 or more space characters will indicate
'               that the line should be ignored.
'   vLines: OUT parameter (destructive) containing array
'           of lines read in from file.
'   nLineCount: OUT parameter (destructive) indicating number
'              of lines read in from file.
'   nCommentLines: OUT parameter (destructive) indicating number
'                 of comment/blank lines found
'
' Check to see if the file exists... if not, bail early.
If Not g_fso.FileExists(strFile) Then Exit Function

' Start off with a reasonable size for the array:
ReDim vLines(5)

' Open a TextStream Object to the file...
Set objTextStream = g_fso.OpenTextFile(strFile, 1, False)

' Used for detecting comment lines, a regular expression object
Set re = New RegExp
re.Pattern = "(^[ \t]*(?:" & strComment & ")+.*$)|(^[ \t]+$)|(^$)"
re.Multiline = False
re.IgnoreCase = False

' Now read in each line of the file and add an element to
' the array for each line that isn't just spaces...
nLineCount = 0
nCommentLines = 0

Do While Not objTextStream.AtEndOfStream

```

```

strLine = ""

' Find out if we need to make our array bigger yet
' to accommodate all the lines in the file. For large
' files, this can be very memory-intensive.
If UBound(vLines) >= nLineCount Then
    ReDim Preserve vLines(nLineCount + 5)
End If

strLine = Trim(objTextStream.ReadLine)

' Look for comment lines that match the pattern
' [whitespace][strComment]
If re.Test(strLine) Then
    ' Line matches our comment pattern... ignore it
    nCommentLines = nCommentLines + 1
Else
    vLines(nLineCount) = strLine
    nLineCount = nLineCount + 1
End If
Loop

objTextStream.Close

ReadDataFromFile = True
End Function

```

With a function like `ReadDataFromFile` in the above example, you can easily read different sets of data from multiple files. For example, consider a situation in which you might have two files, one file that lists the IP addresses or host names of machines to which you need to connect, and a second file that lists the commands that need to be sent to each of the remote machines.

Solution: Read Data from Separate Files: Hosts, Commands

The following example shows how you might have a nested loop that connects to each of the hosts stored in a "hosts.txt" file, individually sending commands read from a "commands.txt" file before moving on to the next host.

```

' ReadDataFromHostFile-SendCommandsFromCommandsFile.vbs
'
' Description:
' Reads in a "hosts.txt" file (located in your "My Documents" folder),
' ignoring comment/blank lines. Reads in a "commands.txt" file (located
' in your "My Documents" folder), ignoring comment/blank lines.
' For each legitimate line found in the "hosts.txt" file, a connection
' is made in series, and all legitimate lines read in from the "commands.txt"
' file are sent to the remote system.
'
' After connecting to each host, and prior to sending any commands, this
' script attempts to detect the command prompt, as command prompts on
' various hosts may be different. This auto-detect process involves
' using the crt.Screen.WaitForCursor method to determine when the cursor
' stops moving (one way to tell that the remote has sent all data after
' a login, including the prompt). The script then asks the Screen object
' for the current line on which the cursor is resting. The line text is
' what is used for the prompt when sending multiple commands (after all
' we want to make sure the remote machine is ready to receive commands
' before attempting to send any commands).
'
' Example "hosts.txt" file:
' # hosts file example

```

```

'
'   # Redhat Linux Hosts
'   192.168.0.1
'   192.168.1.1
'
'   # Solaris Hosts
'   192.168.0.2
'   192.168.1.9
'
'
' Example "commands.txt" file:
'   # Example Commands.txt file
'
'   # First, issue a hostname command so it's easy to see where we are in
'   # the output within the terminal window...
'   hostname
'
'   # Run 'uptime' to find out how long the machine has been up, and
'   # basic load average.
'   uptime
'
'   # Run 'w' command to see who's logged on
'   w
'
'   # Run 'df' to find out how much disk space is used.
'   df

Dim g_fso, g_shell
Set g_fso = CreateObject("Scripting.FileSystemObject")
Set g_shell = CreateObject("WScript.Shell")

g_strHostsFile = g_shell.SpecialFolders("MyDocuments") & "\hosts.txt"
g_strCommandsFile = g_shell.SpecialFolders("MyDocuments") & "\commands.txt"
g_strComment = "#"

' If you need to connect via a firewall, specify the necessary
' pre-defined firewall name right here (must match the name of the
' firewall as defined in the Global Options, Firewall category)...
' otherwise, set the g_strFirewall variable to an empty string (""):
g_strFirewall = " /FIREWALL=linux-dynpf "

' Define user credentials to be used when authenticating to all remote
' machines. For this example, we'll be using publickey authentication.
g_strUsername = "user"
g_strIdentity = "C:\MyKeys\MyKeyFile.pub"

Dim g_strError

MainSub

Sub MainSub()

    Dim vHosts(), vCommands()
    Dim nHostCount, nCommandCount, nCommentLines
    If Not ReadDataFromFile(g_strHostsFile, _
        g_strComment, _
        vHosts, _
        nHostCount, _
        nCommandCount, _
        nCommentLines) Then
        crt.Dialog.MessageBox g_strError
    Exit Sub
End If

```

```

If Not ReadDataFromFile(g_strCommandsFile, _
    g_strComment, _
    vCommands, _
    nCommandCount, _
    nCommentLines) Then
    crt.Dialog.MessageBox g_strError
Exit Sub
End If

Dim strErrors, strSuccesses
' Iterate through each element of our vHosts array...
For nIndex = 0 To nHostCount - 1
    ' Arrays are indexed starting at 0 (zero)
    strHost = vHosts(nIndex)

    ' Exit the loop if the host name is empty (this means we've
    ' reached the end of our array)
    If strHost = "" Then Exit For

    strConnectionString = g_strFirewall & _
        " /SSH2 /ACCEPTHOSTKEYS " & _
        " /L " & g_strUsername & _
        " /I " & g_strIdentity & " " & strHost

    If Connect(strConnectionString) <> 0 Then
        strErrors = strErrors & vbCrLf & _
            vtab & "Failed to connect to " & strHost & _
            ": " & g_strError
    Else
        crt.Screen.Synchronous = True

        Do
            ' Attempt to detect the command prompt heuristically...
            Do
                bCursorMoved = crt.Screen.WaitForCursor(1)
            Loop Until bCursorMoved = False
            ' Once the cursor has stopped moving for about a second, we'll
            ' assume it's safe to start interacting with the remote system.

            ' Get the shell prompt so that we can know what to look for when
            ' determining if the command is completed. Won't work if the prompt
            ' is dynamic (e.g., changes according to current working folder, etc.)
            nRow = crt.Screen.CurrentRow
            strPrompt = crt.screen.Get(nRow, _
                0, _
                nRow, _
                crt.Screen.CurrentColumn - 1)

            ' Loop until we actually see a line of text appear (the
            ' timeout for WaitForCursor above might not be enough
            ' for slower-responding hosts).
            strPrompt = Trim(strPrompt)
            If strPrompt <> "" Then Exit Do
        Loop

        ' Send each command to the remote system.
        For Each strCommand in vCommands
            ' Exit the For..Next loop if our current command is empty...
            ' otherwise, we'll just be pressing the Enter key pointlessly.
            If Trim(strCommand) = "" Then Exit For

            crt.Screen.Send strCommand & vbcr
        Next
    End If
Next

```

```

        ' Wait for the command to complete before moving on to
        ' the next command
        crt.Screen.WaitForString strPrompt
    Next

    ' Now disconnect from the current machine before connecting to the
    ' next machine
    crt.Session.Disconnect

    strSuccesses = strSuccesses & vbCrLf & _
        strHost

    End If

Next

strMsg = "Commands were sent to the following hosts: " & _
    vbCrLf & strSuccesses

If strErrors <> "" Then
    strMsg = strMsg & vbCrLf & vbCrLf & _
        "Errors were encountered connecting to these hosts:" & _
        vbCrLf & strErrors
End If

crt.Dialog.MessageBox strMsg

End Sub

'~~~~~
Function ReadDataFromFile(strFile, _
    strComment, _
    ByRef vLines, _
    ByRef nLineCount, _
    ByRef nCommentLines)
' Returns True if the file was found.
'   strFile: IN parameter specifying full path to data file.
'   strComment: IN parameter specifying string that preceded
'               by 0 or more space characters will indicate
'               that the line should be ignored.
'   vLines: OUT parameter (destructive) containing array
'           of lines read in from file.
'   nLineCount: OUT parameter (destructive) indicating number
'              of lines read in from file.
'   nCommentLines: OUT parameter (destructive) indicating number
'                 of comment/blank lines found
'
'
'   Check to see if the file exists... if not, bail early.
If Not g_fso.FileExists(strFile) Then
    g_strError = "File not found: " & strFile
    Exit Function
End If

' Start off with a reasonable size for the array:
ReDim vLines(5)

' Open a TextStream Object to the file...
Set objTextStream = g_fso.OpenTextFile(strFile, 1, False)

' A regular expression object used for detecting blank/comment lines,
Set re = New RegExp

```

```

re.Pattern = "(^[ \t]*(?:" & strComment & ")+.*$)|(^[ \t]+$)|(^$)"
re.Multiline = False
re.IgnoreCase = False

' Now read in each line of the file and add an element to
' the array for each line that isn't just spaces...
nLineCount = 0
nCommentLines = 0

Do While Not objTextStream.AtEndOfStream
    strLine = ""

    ' Find out if we need to make our array bigger yet
    ' to accommodate all the lines in the file. For large
    ' files, this can be very memory-intensive.
    If UBound(vLines) >= nLineCount Then
        ReDim Preserve vLines(nLineCount + 5)
    End If

    strLine = Trim(objTextStream.ReadLine)

    ' Look for comment lines that match the pattern
    ' [whitespace][strComment]
    If re.Test(strLine) Then
        ' Line matches our comment pattern... ignore it
        nCommentLines = nCommentLines + 1
    Else
        vLines(nLineCount) = strLine
        nLineCount = nLineCount + 1
    End If
Loop

objTextStream.Close

If nLineCount = 0 Then
    g_strError = "ReadDataFromFile() Error. " & _
        "No lines (other than blank/comment lines)" & _
        " were found in file: " & vbCrLf & vtab & strFile
    Exit Function
End If

g_strError = ""
ReadDataFromFile = True
End Function

'~~~~~
Function Connect(strConnectInfo)
' Workaround that uses "On Error Resume Next" VBScript directive to detect
' errors that might occur from the crt.Session.Connect call and instead of
' stopping the script with an unrecoverable error, allow for error handling
' within the script as the script author desires.

    g_strError = ""

    ' Turn off error handling before attempting the connection
    On Error Resume Next
        crt.Session.Connect strConnectInfo
        ' Capture the error code and description (if any)
        nError = Err.Number
        strErr = Err.Description
    ' Restore normal error handling so that any other errors in our
    ' script are not masked/ignored
    On Error Goto 0

```



```

Connect = nError
If nError <> 0 Then
    g_strError = strErr
End If
End Function

```

7.3 Writing Data to Files Using the FileSystemObject

If SecureCRT's logging capability discussed earlier doesn't provide you with the flexibility and control you need over what and how data is written, you can take advantage of VBScript's `FileSystemObject` (mentioned earlier) and **TextStreamObject**. Another example of when you might want to write to a file is if you needed to create a CSV file from the text selected within the terminal window.

Writing data to a file using the VBScript `FileSystemObject` can typically be accomplished using the following general steps:

- 1) Get a reference to the `FileSystemObject`:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

- 2) Use the `FileSystemObject` reference to retrieve a reference to a `TextStream` object associated with the text file to which the data is to be written.

```

Const ForWriting = 2
Const ForAppending = 8

' Create a new file (or overwrite an existing one)
Set objStream = objFSO.OpenTextFile( _
    "C:\Temp\MyDataFile.txt", _
    ForWriting, _
    True)

' Append to an existing file (create if it doesn't exist)
Set objStream = objFSO.OpenTextFile( _
    "C:\Temp\MyDataFile.txt", _
    ForAppending, _
    True)

```

- 3) Write the data to the file using the `TextStream` object's `Write` or `WriteLine` methods. The `WriteLine` method automatically appends EOL characters to the end of the data supplied to the method.

```

objStream.Write "This data is written "
objStream.Write "as one single line."

```

Or:

```

objStream.WriteLine "This data is written "
objStream.WriteLine "as two separate lines."

```

Note that you can write multiple lines using either method by manually inserting CRLF sequences as needed, for example:

```
objStream.Write "This is line one." & vbcrLf & _  
    "This is line two." & vbcrLf & _  
    "This is line three." & vbcrLf & _  
    "This is line four." & vbcrLf
```

Or:

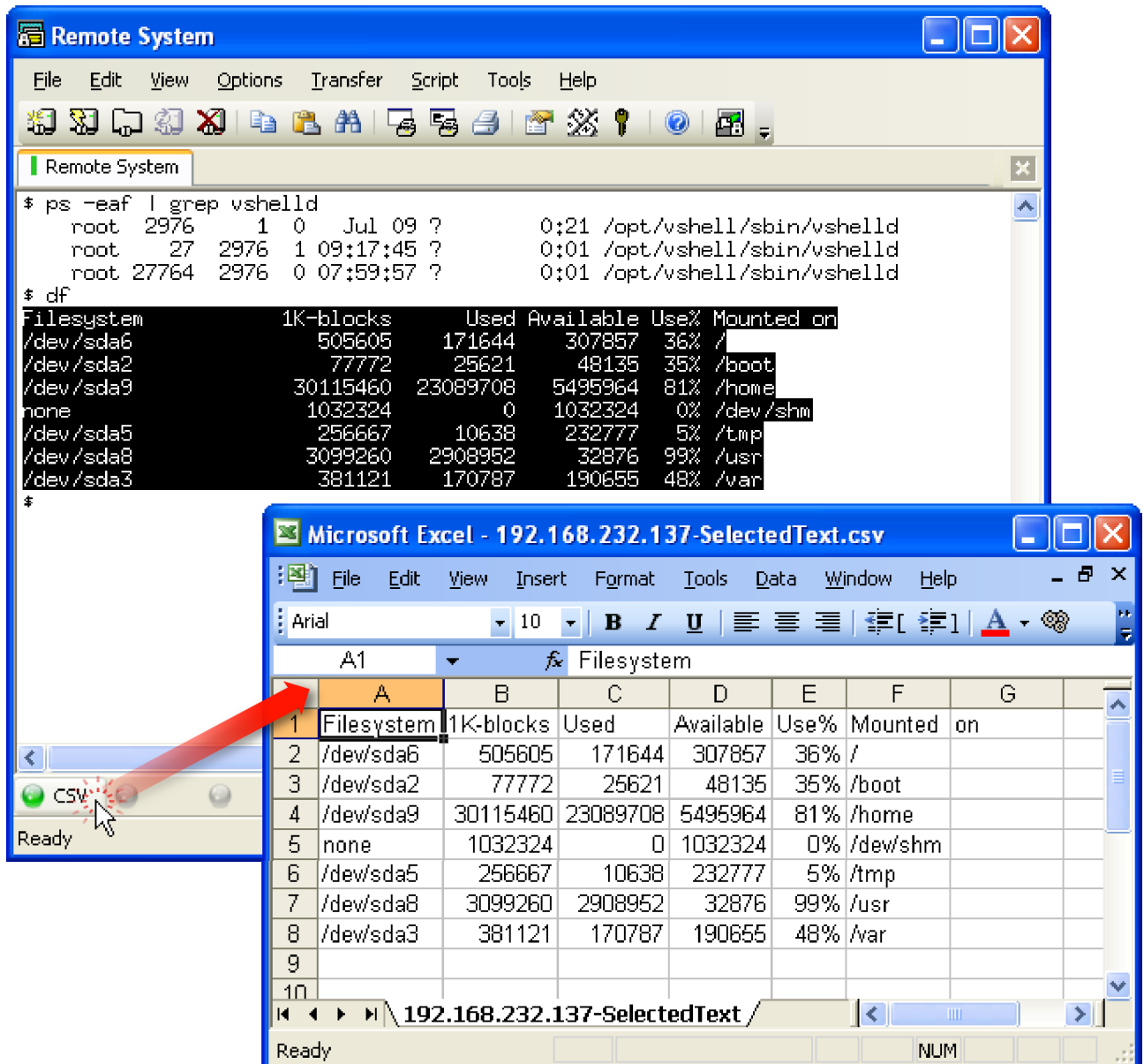
```
objStream.WriteLine "This is line one." & vbcrLf & _  
    "This is line two." & vbcrLf & _  
    "This is line three." & vbcrLf & _  
    "This is line four."
```

- 4) When you have completed writing data to the file, close the file:

```
objStream.Close
```

Solution: Save Selected Text to a CSV File

The following example solution shows how you might be able to select data from the terminal window, save the data to a CSV file, and then display the CSV file within the default CSV file application, as depicted in the graphic below.



Saving Selected Text to a CSV File

Here's the sample solution code for saving selected text to a CSV file:

```
#!/language = "VBScript"
#!/interface = "1.0"
' SaveSelectedTextToCSVFile.vbs
'
' Description:
'   If non-whitespace text is selected within the terminal screen, the user
'   will be prompted for a location + filename in which to store the selected
'   text as a CSV file. Multiple space characters within the data will be
```

```

' converted to a single comma character.
'
' Demonstrates:
' - How to use the Screen.Selection property new to SecureCRT 6.1 to access
'   text selected within the terminal window.
' - How to use the Scripting.FileSystemObject to write data to a
'   text file.
' - How to use RegExp's Replace() method to convert sequential space
'   characters into a single comma character.
' - One way of determining if the script is running on Windows XP or not.
' - One way of displaying a file browse/open dialog in Windows XP
'
'
' g_nMode is used only if the user specifies a file that already exists, in
' which case the user will be prompted to overwrite the existing file, append
' to the existing file, or cancel the operation.
Dim g_nMode
Const ForWriting = 2
Const ForAppending = 8

' Be "tab safe" by getting a reference to the tab for which this script
' has been launched:
Set objTab = crt.GetScriptTab

Set g_shell = CreateObject("WScript.Shell")
Set g_fso = CreateObject("Scripting.FileSystemObject")

SaveSelectedTextToCSVFile

' ~~~~~
Sub SaveSelectedTextToCSVFile()

' Capture the selected text into a variable. The 'Selection' property
' is available in SecureCRT 6.1 and later. This line of code will cause
' an error if launched in a version of SecureCRT earlier than 6.1.
strSelection = objTab.Screen.Selection

' Check to see if the selection really has any text to save... we don't
' usually want to write out nothing to a file.
If Trim(strSelection) = "" Then
    crt.Dialog.MessageBox "Nothing to save!"
    Exit Sub
End If

strFilename = g_shell.SpecialFolders("MyDocuments") & _
"\\" & crt.Session.RemoteAddress & "-SelectedText.csv"
Do
    strFilename = BrowseForFile(strFilename)

    If strFilename = "" Then Exit Sub

' Do some sanity checking if the file specified by the user already
' exists...
If g_fso.FileExists(strFilename) Then

    nResult = crt.Dialog.MessageBox( _
        "Do you want to replace the contents of "" & _
        strFilename & _
        "" with the selected text?" & vbCrLf & vbCrLf & _
        vbtab & "Yes = overwrite" & vbCrLf & vbCrLf & _
        vbtab & "No = append" & vbCrLf & vbCrLf & _
        vbtab & "Cancel = end script" & vbCrLf, _
        "Replace Existing File?", _

```

```

        vbYesNoCancel)
    Select Case (nResult)
        Case vbYes
            g_nMode = ForWriting
            Exit Do
        Case vbNo
            g_nMode = ForAppending
            Exit Do
        Case Else
            Exit Sub
    End Select
Else
    g_nMode = ForWriting
    Exit Do
End If

Loop

' Automatically append a .csv if the filename supplied doesn't include
' any extension.
If g_fso.GetExtensionName(strFilename) = "" Then
    strFilename = strFilename & ".csv"
End If

' Replace instances of one or more space characters with a comma. Use
' the VBScript built-in RegExp object's Replace method to perform this
' task
Set re = New RegExp
' The pattern below means "one or more sequential space characters"
re.Pattern = "[ ]+"
re.Global = True
re.Multiline = True
strCSVData = re.Replace(strSelection, ",")

Do
    On Error Resume Next
    Set objFile = g_fso.OpenTextFile(strFilename, g_nMode, True)
    nError = Err.Number
    strErr = Err.Description
    On Error Goto 0
    If nError = 0 Then Exit Do

    ' Display a message indicating there were problems opening
    ' the file.
    nResponse = crt.Dialog.MessageBox( _
        "Failed to open "" & strFilename & "" (" & nError & "): " & _
        vbCrLf & vtab & strErr & vbCrLf & vbCrLf & _
        "Check to see if the file is already open in another " & _
        "application and make sure you have permissions to " & _
        vbCrLf & "edit the file and create new files within the " & _
        "destination folder.", _
        "Save Operation Failed", _
        vbRetryCancel)
    If nResponse <> vbRetry Then Exit Sub
Loop

objFile.Write strCSVData & vbCrLf
objFile.Close

g_strMode = "Wrote"
If g_nMode = ForAppending Then g_strMode = "Appended"

crt.Dialog.MessageBox _

```

```

        g_strMode & " " & Len(strSelection) & " bytes to file:" & vbCrLf & _
        vbCrLf & strFilename

' Now open the CSV file in the default .csv file application handler...
g_shell.Run chr(34) & strFilename & chr(34)

End sub

' ~~~~~
Function BrowseForFile(strDefault)
' Determine if we're running on Windows XP or not...
Dim strOSName
Set objWMIService = GetObject("winmgmts:" & _
    "{impersonationLevel=impersonate}!\.\root\cimv2")
Set colSettings = _
    objWMIService.ExecQuery("SELECT * FROM Win32_OperatingSystem")
For Each objOS In colSettings
' Windows XP "Name" might look like this:
' "Microsoft Windows XP Professional|C:\WINDOWS\Device\Harddisk0"...
' Vista might appear as follows:
' "Microsoft® Windows Vista™ Business |C:\Windows\Device\Harddisk0"...
    strOsName = Split(objOS.Name, "|")(0)
Exit For
Next

If Instr(strOsName, "XP") > 0 Then
' Based on information obtained from
' http://blogs.msdn.com/gstemp/archive/2004/02/17/74868.aspx
' NOTE: Will only work with Windows XP or newer since other OS's
'       don't have a UserAccounts.CommonDialog ActiveX
'       object registered.
Set objDialog = CreateObject("UserAccounts.CommonDialog")
objDialog.FileName = strDefault
objDialog.Filter = "CSV Files|*.csv|Text Files|.txt|All Files|*.*"
objDialog.FilterIndex = 1
objDialog.InitialDir = g_shell.SpecialFolders("MyDocuments")
nResult = objDialog.ShowOpen

If nResult <> 0 Then
    BrowseForFile = objDialog.FileName
End If
Else
' On Windows other than XP, we'll just pop up an InputBox
BrowseForFile = crt.Dialog.Prompt(_
    "Save selected text to file:", _
    "SecureCRT - Save Selected Text To File", _
    strDefault)
End If
End Function

```

Solution: Import Data from File to SecureCRT Sessions

If you're new to SecureCRT and you have host information stored in an Excel spreadsheet or a flat text file, you might be interested in this example solution. This solution also provides an example of how to create both files as well as folders using the `FileSystemObject`.

```

' ImportArbitraryHostDataToCRTSessions.txt
' (Designed for use with SecureCRT 5.0 and later)
'
' This sample script is designed to create sessions from a

```

```

' text file (.csv format by default, but this can be edited
' to fit the format you have).
'
' The first line of your data file should contain a
' comma-separated (or whatever you define as the g_strDelimiter below)
' list of "fields" designated by the following keywords:
' session_name: The name that should be used for the session. If this field
'                 does not exist, the hostname field is used as the session_name.
'                 folder: Relative path for session as displayed in the Connect dialog.
'                 hostname: The hostname or IP for the remote server
'                 protocol: The protocol (SSH2, SSH1, telnet, rlogin)
'                 port: The port on which remote server is listening
'                 username: The username for the account on the remote server
'                 emulation: The emulation (vt100, xterm, etc.)
'                 description: The comment/description. Multiple lines are separated with '\r'
'
' The first line of the data file instructs this script as to the format of the
' fields in your data file and their meaning.
' 192.168.0.1,root
' 192.168.0.2,administrator
' 192.168.0.3,root
' 192.168.0.4,root
' 192.168.0.5,administrator
' ... and so on
' ... then you would need to insert the following line at
' the top of your data file in order to use this script
' (without any leading spaces -- the leading spaces are
' included here to improve readability):
'
'     hostname,username
'
' If your data file uses spaces or a character other than comma as the delimiter,
' you would also need to edit the g_strDelimiter value a few lines below to
' indicate that fields are separated by spaces, rather than by commas. For
' example:
'     g_strDelimiter = " "
'     g_strDelimiter = " "

Dim g_strSupportedFields
g_strSupportedFields = _
    "description,emulation,folder,hostname,port,protocol,session_name,username"

' If you wish to overwrite existing sessions, set this to
' True; for the default sample script, we're playing it safe
' and leaving any existing sessions in place :).
Dim g_bOverwriteExistingSessions
g_bOverwriteExistingSessions = False

Dim g_strProduct

Dim g_strHostsFile, g_strDelimiter, g_strExampleHostsFile
g_strHostsFile = "C:\Temp\MyDataFile.csv"
g_strExampleHostsFile = _
    vbtab & "hostname,protocol,username,folder,emulation" & vbcrlf & _
    vbtab & "192.168.0.1,SSH2,root,Linux Machines,XTerm" & vbcrlf & _
    vbtab & "192.168.0.2,SSH2,root,Linux Machines,XTerm" & vbcrlf & _
    vbtab & "..." & vbcrlf & _
    vbtab & "10.0.100.1,SSH1,admin,CISCO Routers,VT100" & vbcrlf & _
    vbtab & "10.0.101.1,SSH1,admin,CISCO Routers,VT100" & vbcrlf & _
    vbtab & "..." & vbcrlf & _
    vbtab & "myhost.domain.com,SSH2,administrator,Windows Servers,VShell" & _
    vbcrlf & _
    vbtab & "..."

```

```

g_strDelimiter = ","      ' comma
' g_strDelimiter = " "    ' space
' g_strDelimiter = ";"    ' semi-colon
' g_strDelimiter = chr(9) ' tab

g_strExampleHostsFile = Replace(g_strExampleHostsFile, ",", g_strDelimiter)

Dim g_fso, g_shell
Set g_fso = CreateObject("Scripting.FileSystemObject")
Set g_shell = CreateObject("WScript.Shell")

Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8

Dim g_strConfigFolder
Dim strFieldDesignations
Dim vFieldsArray
Dim vSessionInfo

Dim strSessionName, strHostName, strPort
Dim strUserName, strProtocol, strEmulation
Dim strPathForSessions, strLine, nFieldIndex
Dim strSessionFileName, strFolder, nDescriptionLineCount, strDescription

' This list must be in the same order as the entries in the lines of the
' text file. It is not a requirement that all the options be used. Just
' remove any option that is not in the text file.
strFieldDesignations = "session_name,protocol,hostname,port,username,emulation"

Dim g_strLastError, g_strErrors
Dim g_nSessionsCreated, g_nDataLines

Import

'~~~~~
Sub Import()
    g_strProduct = "SecureCRT"

    g_strHostsFile = BrowseForFile( _
        "Please select the host data file to be imported.", _
        g_fso.GetParentFolderName(g_strHostsFile))

    If g_strHostsFile = "" Then
        Exit Sub
    End If

    ' Open our data file for reading
    Dim objDataFile
    Set objDataFile = g_fso.OpenTextFile(g_strHostsFile, ForReading, False)

    ' Now read the first line of the data file to determine
    ' the field designations
    strFieldDesignations = lcase(objDataFile.ReadLine)

    ' Validate the data file
    If Not ValidateFieldDesignations(strFieldDesignations) Then
        objDataFile.Close
        Exit Sub
    End If

    ' Find out where the CRT/SecureCRT configuration lives.

```



```

g_strConfigFolder = ReadRegKey("HKCU\Software\VanDyke\" & _
                               g_strProduct & "\Config Path")

Do
    g_strConfigFolder = InputBox(_
        "Your current config folder is specified as seen below. " & _
        vbCrLf & vbCrLf & _
        "If you want to have the session files created in another " & _
        "folder, please specify the folder below.", _
        vbCrLf & vbCrLf & _
        "Please select/confirm Config folder", _
        g_strConfigFolder)

    If g_strConfigFolder = "" Then Exit Sub

    If Not g_fso.FolderExists(g_strConfigFolder & "\Sessions") Then
        Dim nAnswer
        nAnswer = MsgBox(_
            "This folder doesn't have a "Sessions" subfolder." & _
            vbCrLf & vbCrLf & _
            "Would you like to create one now?", vbYesNoCancel, _
            "Import Data To SecureCRT Sessions")
        Select Case nAnswer
            Case vbCancel
                Exit Sub
            Case vbYes
                If Not CreateFolderPath(_
                    g_strConfigFolder & "\Sessions") Then
                    MsgBox "Failed to create folder (" & _
                        g_strConfigFolder & "\Sessions" & "): " & _
                        vbCrLf & vbCrLf & g_strLastError, _
                        vbOkOnly, _
                        "Import Data To SecureCRT Sessions"
                Else
                    Exit Do
                End If
            End Select
        Else
            ' Folder already exists, so we know we can successfully continue
            Exit Do
        End If
    Loop

    ' Here we create an array of the items that will be used to create
    ' the new session, based on the fields separated by the delimiter
    ' specified in g_strDelimiter
    vFieldsArray = Split(strFieldDesignations, g_strDelimiter)

    ' Loop through reading each line in the data file and creating a session
    ' based on the information contained on each line.
    Do While Not objDataFile.AtEndOfStream
        strLine = ""
        strLine = objDataFile.ReadLine

        ' This sets v_File Data array elements to
        ' each section of strLine, separated by the delimiter
        vSessionInfo = Split(strLine, g_strDelimiter)
        If UBound(vSessionInfo) < UBound(vFieldsArray) Then
            g_strErrors = g_strErrors & vbCrLf & _
                "Insufficient data on line #" & _
                objDataFile.Line - 1 & ": " & strLine
        Else

```

```

' Variable used to determine if a session file should actually be
' created, or if there was an unrecoverable error (and the session
' should be skipped).
Dim bWriteFile

' Now we will match the items from the new file array to the correct
' variable for the session's ini file
bWriteFile = True
For nFieldIndex = 0 To UBound(vSessionInfo)

    Select Case vFieldsArray(nFieldIndex)
        Case "session_name"
            strSessionName = vSessionInfo(nFieldIndex)
            ' Check folder name for any invalid characters
            Dim re
            Set re = New RegExp
            re.Pattern = "[\\|\/\:\*\?\\\"<\>]"
            If re.Test(strSessionName) Then
                bWriteFile = False
                If g_strErrors <> "" Then g_strErrors = _
                    vbCrLf & g_strErrors

                g_strErrors = _
                    "Error: " & _
                    "Invalid characters found in SessionName "" & _
                    strSessionName & "" specified on line #" & _
                    objDataFile.Line - 1 & ": " & strLine & _
                    g_strErrors
            End If

        Case "port"
            strPort = Trim(vSessionInfo(nFieldIndex))
            If Not IsNumeric(strPort) Then
                bWriteFile = False
                If g_strErrors <> "" Then g_strErrors = _
                    vbCrLf & g_strErrors

                g_strErrors = _
                    "Error: Invalid port "" & strPort & _
                    "" specified on line #" & _
                    objDataFile.Line - 1 & ": " & strLine & _
                    g_strErrors
            End If
            strPort = Hex(strPort)
            strPort = NormalizeNumber(strPort)

        Case "protocol"
            strProtocol = Trim(LCase(vSessionInfo(nFieldIndex)))

            Select Case strProtocol
                Case "ssh2"
                    strProtocol = "SSH2"
                Case "ssh1"
                    strProtocol = "SSH1"
                Case "telnet"
                    strProtocol = "Telnet"
                Case "serial"
                    bWriteFile = False
                    g_strErrors = g_strErrors & vbCrLf & _
                        "Warning: This sample script does " & _
                        "not support creating sessions " & _
                        "with protocol "" & _
                        vSessionInfo(nFieldIndex) & _

```

```

        "" specified on line #" & _
        objDataFile.Line - 1 & ": " & strLine
Case "tapi"
    bWriteFile = False
    g_strErrors = g_strErrors & vbCrLf & _
        "Warning: This sample script does " & _
        "not support creating sessions " & _
        "with protocol "" " & _
        vSessionInfo(nFieldIndex) & _
        "" specified on line #" & _
        objDataFile.Line - 1 & ": " & strLine
Case "rlogin"
    strProtocol = "RLogin"
Case Else
    bWriteFile = False
    If g_strErrors <> "" Then g_strErrors = _
        vbCrLf & g_strErrors

    g_strErrors = _
        "Error: Invalid protocol "" " & _
        vSessionInfo(nFieldIndex) & _
        "" specified on line #" & _
        objDataFile.Line - 1 & ": " & strLine & _
        g_strErrors
End Select ' for protocols

Case "hostname"
    strHostName = Trim(vSessionInfo(nFieldIndex))
    If strHostName = "" Then
        bWriteFile = False
        g_strErrors = g_strErrors & vbCrLf & _
            "Warning: 'hostname' field on line #" & _
            objDataFile.Line - 1 & " is empty: " & strLine
    End If

Case "username"
    strUserName = Trim(vSessionInfo(nFieldIndex))

Case "emulation"
    strEmulation = LCase(Trim(vSessionInfo(nFieldIndex)))
    Select Case strEmulation
        Case "xterm"
            strEmulation = "Xterm"
        Case "vt100"
            strEmulation = "VT100"
        Case "vt102"
            strEmulation = "VT102"
        Case "vt220"
            strEmulation = "VT220"
        Case "ansi"
            strEmulation = "ANSI"
        Case "linux"
            strEmulation = "Linux"
        Case "scoansi"
            strEmulation = "SCOANSI"
        Case "vshell"
            strEmulation = "VShell"
        Case "wyse50"
            strEmulation = "WYSE50"
        Case "wyse60"
            strEmulation = "WYSE60"
        Case Else
            bWriteFile = False

```

```

        g_strErrors = g_strErrors & vbCrLf & _
            "Warning: Invalid emulation "" & _
            strEmulation & "" specified on line #" & _
            objDataFile.Line - 1 & ": " & strLine
    End Select

Case "folder"
    strFolder = Trim(vSessionInfo(nFieldIndex))

    ' Check folder name for any invalid characters
    Set re = New RegExp
    re.Pattern = "[\\|\\/\\:\\*\\?\\\"\\<\\>]"
    If re.Test(strFolder) Then
        bWriteFile = False
        If g_strErrors <> "" Then g_strErrors = _
            vbCrLf & g_strErrors

        g_strErrors = _
            "Error: Invalid characters in folder "" & _
            strFolder & "" specified on line #" & _
            objDataFile.Line - 1 & ": " & strLine & _
            g_strErrors
    End If

Case "description"
    strDescription = Trim(vSessionInfo(nFieldIndex))
    If strDescription <> "" Then
        Dim vDescriptionLines
        vDescriptionLines = Split(strDescription, "\r")
        nDescriptionLineCount = _
            UBound(vDescriptionLines) + 1
        strDescription = " " & _
            Replace(strDescription, "\r", vbCrLf & " ")
    Else
        g_strErrors = g_strErrors & vbCrLf & _
            "Warning: 'description' field on line #" & _
            objDataFile.Line - 1 & " is empty: " & strLine
    End If

Case Else
    ' If there is an entry that the script is not set to use
    ' in strFieldDesignations, stop the script and display a
    ' message
    Dim strMsg1
    strMsg1 = "Error: Unknown field designation: " & _
        vFieldsArray(nFieldIndex) & vbCrLf & vbCrLf & _
        "    Supported fields are as follows: " & _
        vbCrLf & vbCrLf & vtab & g_strSupportedFields & _
        vbCrLf & _
        vbCrLf & "    For a description of " & _
        "supported fields, please see the comments in " & _
        "the sample script file."

    If Trim(g_strErrors) <> "" Then
        strMsg1 = strMsg1 & vbCrLf & vbCrLf & _
            "Other errors found so far include: " & _
            g_strErrors
    End If

    MsgBox strMsg1, _
        vbOkOnly, _
        "Import Data To SecureCRT Sessions: Data File Error"
Exit Sub

```

```

    End Select
Next

If bWriteFile Then
    'Write the session file
    If strSessionName = "" Then
        strSessionName = strHostName
    End If
    strPathForSessions = g_strConfigFolder & "\Sessions"
    'Call function to check if a folder needs to be created
    If strFolder <> "" Then
        strPathForSessions = strPathForSessions & "\" & strFolder
    End If

    If Not CreateFolderPath(strPathForSessions) Then
        Dim strMsg
        If g_nSessionsCreated > 0 Then
            strMsg = "Error: We were able to create " & _
                g_nSessionsCreated & _
                ", but encountered the following fatal error:" & _
                vbCrLf & vbCrLf
        End If

        MsgBox strMsg & "Unable to create folder: " & _
            strPathForSessions & vbCrLf & vbCrLf & vtab & _
            g_strLastError & vbCrLf & vbCrLf & _
            "Other errors/warnings found so far include:" & _
            vbCrLf & g_strErrors, _
            vbOkOnly, _
            "Import Data To SecureCRT Sessions"
        Exit Sub
    End If

    strSessionFileName = strSessionName & ".ini"

    If g_fso.FileExists(strPathForSessions & "\" & _
        strSessionFileName) And _
        g_bOverwriteExistingSessions = False Then
        g_strErrors = g_strErrors & vbCrLf & _
            "Warning: Session already exists (and it was " & _
            "left in place) for name provided on line #" & _
            objDataFile.Line - 1 & ": "" & strLine &; """"
    Else

        Dim objSessionFile
        Set objSessionFile = g_Fso.OpenTextFile(_
            strPathForSessions & "\" & _
            strSessionFileName, _
            ForWriting, _
            True)
        objSessionFile.Write "S:" & "Hostname" & "=" & strHostName & _
            vbCrLf
        objSessionFile.Write "S:" & "Username" & "=" & strUserName & _
            vbCrLf

        If strProtocol = "SSH2" Then
            If strPort = "" Then strPort = NormalizeNumber("16")
            objSessionFile.Write "D:" & "[SSH2] Port" & "=" & strPort & _
                vbCrLf
        End If
        If strProtocol = "SSH1" Then
            If strPort = "" Then strPort = NormalizeNumber("16")
            objSessionFile.Write "D:" & "[SSH1] Port" & "=" & strPort & _

```

```

        vbcrLf
    End If
    If strProtocol = "Telnet" Then
        If strPort = "" Then strPort = NormalizeNumber("17")
        objSessionFile.Write "D:""Port"="" & strPort & vbcrLf
    End If

    objSessionFile.Write "S:""Protocol Name"="" & _
        strProtocol & vbcrLf
    objSessionFile.Write "S:""Emulation"="" & strEmulation & _
        vbcrLf

    If strDescription <> "" Then
        objSessionFile.Write "Z:""Description"="" & _
            NormalizeNumber(nDescriptionLineCount) & vbcrLf
        objSessionFile.Write strDescription & vbcrLf
    End If

    objSessionFile.Close
    g_nSessionsCreated = g_nSessionsCreated + 1
End If
End If

strEmulation = ""
strPort = ""
strHostName = ""
strFolder = ""
strUserName = ""
strSessionName = ""
strDescription = ""
nDescriptionLineCount = 0
End If

Loop

g_nDataLines = objDataFile.Line
objDataFile.Close

Dim strResults
strResults = "Operation completed."

If g_nSessionsCreated > 0 Then
    strResults = strResults & _
        vbcrLf & "--> Number of Sessions created: " & g_nSessionsCreated
Else
    strResults = strResults & vbcrLf & _
        "--> No sessions were created from " & g_nDataLines & _
        " lines of data."
End If

If g_strErrors <> "" Then
    strErrors = g_strErrors
    ' MsgBox can only handle a small amount of data. Let's truncate the
    ' data in a semi-sensible manner, making sure that we include a full
    ' last line if the data is beyond 500 characters in length.
    If Len(g_strErrors) > 500 Then
        strErrors = Left(g_strErrors, 500)
        ' Find the last CRLF in the data and take everything to
        ' the left of it (so that we don't have a partial line of
        ' an error message displayed)
        strErrors = Left(strErrors, InstrRev(strErrors, vbcrLf))
    End If

```

```

        strResults = strResults & _
        vbCrLf & vbCrLf & _
        "--> Errors/warnings from this operation include:" & _
        vbCrLf & strErrors & "...".
    End If

    MsgBox strResults, _
        vbOkOnly, _
        "Import Data To SecureCRT Sessions"

End Sub

'~~~~~
'      Helper Methods and Functions
'~~~~~

Function ValidateFieldDesignations(strFields)
    If Instr(strFieldDesignations, g_strDelimiter) = 0 Then
        Dim strErrorMsg, strDelimiterDisplay
        strErrorMsg = "Invalid header line in data file. " & _
            "Delimiter character not found: "
        If Len(g_strDelimiter) > 1 Then
            strDelimiterDisplay = g_strDelimiter
        Else
            If Asc(g_strDelimiter) < 33 or Asc(g_strDelimiter) > 126 Then
                strDelimiterDisplay = "ASCII[" & Asc(g_strDelimiter) & "]"
            Else
                strDelimiterDisplay = g_strDelimiter
            End If
        End If
        strErrorMsg = strErrorMsg & strDelimiterDisplay & vbCrLf & vbCrLf & _
            "The first line of the data file is a header line " & _
            "that must include" & vbCrLf & _
            "a '" & strDelimiterDisplay & _
            "' separated list of field keywords." & vbCrLf & _
            vbCrLf & "'hostname' and 'protocol' are required keywords." & _
            vbCrLf & vbCrLf & _
            "The remainder of the lines in the file should follow the " & _
            vbCrLf & _
            "pattern established by the header line " & _
            "(first line in the file)." & vbCrLf & "For example:" & vbCrLf & _
            g_strExampleHostsFile
        MsgBox strErrorMsg, _
            vbOkOnly, _
            "Import Data To SecureCRT Sessions"
    Exit Function
End If

If Instr(strFieldDesignations, "hostname") = 0 Then
    strErrorMsg = "Invalid header line in data file. " & _
        "'hostname' field is required."
    If Len(g_strDelimiter) > 1 Then
        strDelimiterDisplay = g_strDelimiter
    Else
        If Asc(g_strDelimiter) < 33 or Asc(g_strDelimiter) > 126 Then
            strDelimiterDisplay = "ASCII[" & Asc(g_strDelimiter) & "]"
        Else
            strDelimiterDisplay = g_strDelimiter
        End If
    End If
End If

```

```

MsgBox strErrorMsg & vbCrLf & _
    "The first line of the data file is a header line " & _
    "that must include" & vbCrLf & _
    "a '" & strDelimiterDisplay & _
    "' separated list of field keywords." & vbCrLf & _
    vbCrLf & "'hostname' and 'protocol' are required keywords." & _
    vbCrLf & vbCrLf & _
    "The remainder of the lines in the file should follow the " & _
    vbCrLf & _
    "pattern established by the header line " & _
    "(first line in the file)." & vbCrLf & "For example:" & vbCrLf & _
    g_strExampleHostsFile, _
    vbOkOnly, _
    "Import Data To SecureCRT Sessions"
Exit Function
End If

If Instr(strFieldDesignations, "protocol") = 0 Then
MsgBox "Invalid data file header line: " & vbCrLf & vbCrLf & _
    vtab & strFieldDesignations & vbCrLf & _
    vbCrLf & "--> 'protocol' field is required.", _
    vbOkOnly, _
    "Import Data To SecureCRT Sessions"
Exit Function
End If

ValidateFieldDesignations = True
End Function

'~~~~~
Function ReadRegKey(strKeyPath)
On Error Resume Next
Err.Clear
ReadRegKey = g_shell.RegRead(strKeyPath)
If Err.Number <> 0 Then
    ' Registry key must not have existed.
    ' ReadRegKey will already be empty, but
    ' for the sake of clarity, we'll set it
    ' to an empty string explicitly
    ReadRegKey = ""
End If
On Error Goto 0
End Function

'~~~~~
Function CreateFolderPath(strPath)
' Recursive function
If g_fso.FolderExists(strPath) Then
    CreateFolderPath = True
    Exit Function
End If

' Check to see if we've reached the drive root
If Right(strPath, 2) = ":\\" Then
    CreateFolderPath = True
    Exit Function
End If

' None of the other two cases were successful, so attempt to create the
' folder
On Error Resume Next
g_fso.CreateFolder strPath
nError = Err.Number

```



```

strErr = Err.Description
On Error Goto 0
If nError <> 0 Then
    ' Error 76 = Path not found, meaning that the full path doesn't exist.
    ' Call ourselves recursively until all the parent folders have been
    ' created:
    If nError = 76 Then _
        CreateFolderPath(g_fso.GetParentFolderName(strPath))

    On Error Resume Next
    g_fso.CreateFolder strPath
    nError = Err.Number
    strErr = Err.Description
    On Error Goto 0

    ' If the Error is not = 76, then we have to bail since we
    ' no longer have any hope of successfully creating each folder in the
    ' tree
    If nError <> 0 Then
        g_strLastError = strErr
        Exit Function
    End If
End If

CreateFolderPath = True
End Function

' ~~~~~
Function NormalizeNumber(nNumber)
' Normalizes a single digit number to have a 0 in front of it
Dim nIndex, nOffbyDigits, strResult, nDesiredDigits
nDesiredDigits = 8
nOffbyDigits = nDesiredDigits - Len(nNumber)

strResult = nNumber

For nIndex = 1 To nOffbyDigits
    strResult = "0" & strResult
Next
NormalizeNumber = strResult
End Function

' ~~~~~
Function BrowseForFile(strTitle, strInitialDir)
' Since Windows XP is the only OS on which the UserAccounts.CommonDialog
' object is available, find out if we are using Windows XP or a different
' operating system (Vista and newer Windows versions are not guaranteed to
' have this control available, so for these operating systems, we will need
' to present a much simpler interface for choosing a file: an input box).
Dim strOSName
Set objWMIService = GetObject("winmgmts:" & _
    "{impersonationLevel=impersonate}!\.\root\cimv2")
Set colSettings = _
    objWMIService.ExecQuery("SELECT * FROM Win32_OperatingSystem")
For Each objOS In colSettings
    ' Windows XP might look like this:
    ' Microsoft Windows XP Professional|C:\WINDOWS|\Device\Harddisk0\Partition1
    strOSName = Split(objOS.Name, "|")(0)
Exit For
Next

If Instr(strOSName, "Windows XP") = 0 Then
    ' If not using Windows XP, we are limited in our ability to present

```

```

' an adequate file browser dialog, so we'll just pop up an InputBox
' asking for the path to the file.
strFilePath = g_strHostsFile
Do
    strFilePath = InputBox(strTitle, _
        "SecureCRT Import Script", _
        strFilePath)
    If strFilePath = "" Then Exit Function
    If g_fso.FileExists(strFilePath) Then Exit Do
    MsgBox "Path not found: " & vbCrLf & vbCrLf & vtab & _
        strFilePath & vbCrLf & vbCrLf & _
        "Please specify a valid file path", _
        vbOkOnly, _
        "Import Data To SecureCRT Sessions"
Loop
BrowseForFile = strFilePath
Else
' Based on information obtained from
' http://blogs.msdn.com/gstemp/archive/2004/02/17/74868.aspx
' NOTE: Will only work with Windows XP since other OS's
'       don't have a UserAccounts.CommonDialog ActiveX
'       object registered.
Dim objDialog
Set objDialog = CreateObject("UserAccounts.CommonDialog")
'Set objDialog = CreateObject("MSComDlg.CommonDialog")
objDialog.Filter = "CSV Files|*.csv;Text Files|.txt;All Files|*.*"
objDialog.FilterIndex = 1
objDialog.InitialDir = strInitialDir
'objDialog.InitDir = g_strMyDocs
'objDialog.MaxFileSize = 512
If MsgBox(strTitle, _
    vbOkCancel, _
    "Import Data To SecureCRT Sessions") <> vbok Then
    Exit Function
End If
'objDialog.DialogTitle = strTitle
'objDialog.CancelError = False
objDialog.ShowOpen

    BrowseForFile = objDialog.FileName
End If
End Function

```

Chapter 8: Working with the Windows Clipboard

The ability to easily work with data from a variety of applications is often a "must have". If you store commands, code, or configuration data within another application and need to frequently transfer portions of this information to a remote device using SecureCRT, the `Clipboard` object can be used to access text that has been copied to the Windows Clipboard.

8.1 Retrieving Data Stored in the Clipboard

Retrieving data that another application has placed in the Windows Clipboard can be done within a SecureCRT script by using the `crt.Clipboard.Text` property. Multiple lines of text are stored in the Windows Clipboard with a CRLF indicator separating each line. Once you have a script variable containing the text found in the Windows Clipboard, you can modify it as needed to meet your needs and then send it to the remote system. For an example, see the earlier solution script in section 5.3: [Add "no" to Each Selected Line and Send to Remote](#).

8.2 Storing Data to the Clipboard

Sometimes you may want to transfer data from the SecureCRT terminal window to the Windows Clipboard for use in other applications. Manually, this can be done using a variety of commonly-known techniques:

- Select the desired text with the mouse, then choose **Copy** from either the **Edit** menu or the right-click context menu.
- Enable the **Copy on select** option in SecureCRT's global options. With this option enabled, each time text is selected within SecureCRT's terminal window, it is automatically copied to the Windows Clipboard.
- Select the desired text with the mouse, then use SecureCRT's `Ctrl+Shift+C` built-in shortcut key for copying data to the Windows Clipboard. Alternatively, use the `Ctrl+Insert` to copy.

Each of these manual methods of copying text from SecureCRT to the Windows Clipboard have their pros and cons. But what if you want to extract text from a specific location on the screen from within a script, and place the information on the clipboard? You can expand on the example earlier by placing the text found in a specific location on the screen directly within the Windows Clipboard. For example:

```
' Extract data displayed on row 2, between columns 1 and 2
nProcesses = crt.Screen.Get(2,1,2,2)

' Extract data displayed on row 5, between columns 36 and 41
nFreeMem = crt.Screen.Get(5,36,5,41)

strData = "Processes:" & vbtab & nProcesses & vbcrLf & _
         "Free Memory:" & vbtab & nFreeMem

' Place the data in the Windows Clipboard for use in another application:
crt.Clipboard.Text = strData
```

Solution: Auto-Save Command Results to the Clipboard

What if you wanted to automatically have the results of each command placed in the Windows Clipboard, ready to paste into another application without needing to select the text? Here's an

example script solution that runs in the background, placing the results of each remote command within the Windows Clipboard:

```
' StoreResultsOfLastCommandInWindowsClipboard.vbs
'
' Description:
'   - Determines what the shell prompt is.
'   - Enters a loop, continuously setting the Windows Clipboard to the
'     text of each command entered on the remote system.
'   - To stop the script, choose "Cancel" from SecureCRT's main Script menu.
'
' Try not to miss any data
crt.Screen.Synchronous = True
' Also, don't capture escape sequences
crt.Screen.IgnoreEscape = True

Sub Main()

  If Not crt.Session.Connected Then
    crt.Dialog.MessageBox _
      "This script requires an active connection to a remote machine."
    Exit Sub
  End If

  ' Determine what the shell prompt is...
  If crt.Dialog.MessageBox( _
    "Trying to determine what your shell prompt is..." & vbCrLf & _
    vbCrLf & _
    "Please press the "OK" button. Then press the Enter " & vbCrLf & _
    "key while at the shell prompt.", _
    "SecureCRT Script: Determine Shell Prompt", _
    vbOkCancel) <> vbOk Then Exit Sub

  crt.Screen.WaitForString vbcr
  Do
    ' Attempt to detect the command prompt heuristically...
    Do
      bCursorMoved = crt.Screen.WaitForCursor(1)
    Loop Until bCursorMoved = False
    ' Once the cursor has stopped moving for about a second, we'll
    ' assume it's safe to start interacting with the remote system.

    ' Get the shell prompt so that we can know what to look for when
    ' determining if the command is completed. Won't work if the prompt
    ' is dynamic (e.g., changes according to current working folder, etc.)
    nRow = crt.Screen.CurrentRow
    strPrompt = crt.screen.Get(nRow, _
      0, _
      nRow, _
      crt.Screen.CurrentColumn - 1)

    ' Loop until we actually see a line of text appear (the
    ' timeout for WaitForCursor above might not be enough
    ' for slower-responding hosts.
    strPrompt = Trim(strPrompt)
    If strPrompt <> "" Then Exit Do
  Loop

  ' Determine what the shell prompt is...
  If crt.Dialog.MessageBox( _
    "All set!" & vbCrLf & vbCrLf & _
    "Your shell prompt was detected as being: "" & strPrompt & """" & _
    vbCrLf & _
```

```

vbCrLf & _
"While this script continues to run, the results of each command " & _
"you run will be stored" & vbCrLf & _
"in the Windows Clipboard." & vbCrLf & _
vbCrLf & _
>Note, however, that if your prompt includes your current " & _
"working directory and is" & vbCrLf & _
"dynamically updated, the change to your prompt will cause " & _
"this script to capture data" & vbCrLf & _
"up until you return to the working directory you were in when " & _
"the script was initially" & vbCrLf & _
"launched.", _
"SecureCRT Script: Clipboard Command Result Store Activated", _
vbOkCancel) <> vbOk Then Exit Sub

' Now that we know what the prompt is, we'll loop waiting for the prompt
' to appear.
Do
' First, wait for a command to be issued (if you want the command
' you ran to be included above the results, comment out the
' following line of code).
crt.Screen.WaitForString vbCr

' Now that we know a command has been sent to the remote, let's
' use ReadString to capture everything that we receive from the
' remote machine up until we see the prompt.
strData = crt.Screen.ReadString(strPrompt)

' Some devices like Cisco PIX return lines terminated by LFCR,
' which is backwards from what Windows really needs in the
' clipboard. We'll look for these and simply replace them with
' the correct CRLF sequence:
strData = Replace(strData, vbCr, "")
strData = Replace(strData, vbLf, vbCrLf)

' Now store the rectified data into the Windows Clipboard...
crt.Clipboard.Text = strData
Loop

End Sub

```

8.3 Changing Paste "Speed"

If you are a network administrator responsible for configuration of routers or switches, you might sometimes find you need to "slow" down pasting so that it doesn't overwhelm the remote device to which you are connected – especially if you're pasting a larger amount of text.

Solution: “Slow Paste” (Line Delay)

One way to slow down the rate at which text is sent to a remote system is to introduce a delay after each line has been sent. Some SecureCRT users reading this would be right in thinking, "This sounds familiar..." because SecureCRT sports a session option named "Line send delay" that is intended specifically for this purpose (see the **Terminal / Emulation / Advanced** category of the Session Options dialog). Although this option is a welcome benefit when manually pasting large amounts of text, the **Line send delay** and **Character send delay** options do not apply when sending data via `crt.Screen.Send` within a SecureCRT script. You'll likely need to experiment with the amount of time required for "sleeping" between each line to match the needs of the remote device. Try this script out by [defining a button](#) to run a script file containing the following code:

```

' SlowPaste-LineDelay.vbs
'
' Description:
' Demonstrates how to send each line within the Windows Clipboard
' to a remote host after introducing a delay after each line
' is sent. This is one approach to prevent overwhelming a remote
' host that cannot accept data as fast as SecureCRT normally sends
' it.

' Define the amount of time (milliseconds) that should be introduced as
' a delay between each line being sent to the remote host. You'll
' probably want to experiment with different values for g_nDelay to find
' out which one provides the best balance between performance and over-
' whelming the remote host.
g_nDelay = 300

Sub Main()
' If there isn't an active connection, there's no point in continuing
' since we can't send data.
If Not crt.Session.Connected Then
    crt.Dialog.MessageBox "Sending data requires an active connection."
    Exit Sub
End If

' If there isn't anything in the Windows Clipboard, it doesn't make any
' sense to continue with the script. Note that we're checking for
' vbCrLf here, since an empty clipboard will only contain a trailing
' CRLF sequence.
If Trim(crt.Clipboard.Text) = vbCrLf Then
    crt.Dialog.MessageBox "No text found in the Windows Clipboard."
    Exit Sub
End If

' Multiple lines in the Windows Clipboard are stored with a CRLF separator.
' Break up lines into an array (each line as an element within the array):
vLines = Split(crt.Clipboard.Text, vbCrLf)

' Enter a loop, sending a line, and then sleeping for g_nDelay milliseconds
' after each line is sent.

For Each strLine In vLines
    ' The vbcr is needed to simulate pressing the Enter key since we
    ' lost the line ending vbcr during the above Split operation.
    crt.Screen.Send strLine & vbcr
    crt.Sleep g_nDelay
Next

' Inform that the data has all been sent.
crt.Dialog.MessageBox _
    "All data in the Windows Clipboard has been sent to the remote."
End Sub

```

Solution: “Slow Paste” (Echo Delay)

Another way to slow down the rate at which data is sent to the remote is to introduce a `WaitForString()` call right after sending a line, waiting for the remote to echo the line back (indication that the remote has received the data, and is ready for the next line). Depending on the latency of the connection between SecureCRT and the remote device, this may be slower than desired since it takes time for the remote to echo data back to SecureCRT. A modification

could be to only wait for each third line or fifth line to appear from the remote. Give this example a spin by [defining a button](#) to run a script file containing the following code:

```
' SlowPaste-EchoDelay.vbs
'
' Description:
' Demonstrates how to send each line within the Windows Clipboard and wait
' for it to be echoed back by the remote host before the next line is sent.
' This is one approach to prevent overwhelming a remote host that cannot
' accept data as fast as SecureCRT normally sends it by making sure the
' remote machine has received each line before moving on to the next line.

Sub Main()
' If there isn't an active connection, there's no point in continuing
' since we can't send data.
If Not crt.Session.Connected Then
    crt.Dialog.MessageBox "Sending data requires an active connection."
    Exit Sub
End If

' If there isn't anything in the Windows Clipboard, it doesn't make any
' sense to continue with the script. Note that we're checking for
' vbCrLf here, since an empty clipboard will only contain a trailing
' CRLF sequence.
If Trim(crt.Clipboard.Text) = vbCrLf Then
    crt.Dialog.MessageBox "No text found in the Windows Clipboard."
    Exit Sub
End If

' Multiple lines in the Windows Clipboard are stored with a CRLF separator.
' Break up lines into an array (each line as an element within the array).
vLines = Split(crt.Clipboard.Text, vbCrLf)

' Enter a loop, sending a line, and then waiting for the line to be
' echoed back to us from the remote, before the next line is sent.
For Each strLine In vLines
    ' The vbcr is needed to simulate pressing the Enter key since we
    ' lost the line ending vbcr during the above Split operation.
    crt.Screen.Send strLine & vbcr

    ' Now, wait for the remote to echo what we just sent
    crt.Screen.WaitForString strLine
Next

' Inform that the data has all been sent.
crt.Dialog.MessageBox _
    "All data in the Windows Clipboard has been sent to the remote."
End Sub
```

Solution: Vary Paste Speed Based on Clipboard Length

Another approach would be to send the data to the remote without delay if the amount of text detected in the Windows Clipboard is below a certain threshold, say, 500 characters. The following example solution demonstrates how to detect the number of characters currently stored in the Windows Clipboard before deciding whether or not to invoke a delayed send.

```
' DelayPasteIfClipboardLengthExceedsThreshold.vbs
'
' Description:
' Demonstrates how to send each line within the Windows Clipboard
' to a remote host after introducing a delay after each line
```

```

' is sent -- only if the number of characters in the clipboard exceeds
' a certain length. This is another approach to prevent overwhelming
' a remote host that cannot accept data as fast as SecureCRT normally
' sends it.
'
' Define the threshold: the length of the clipboard (number of characters)
' that, if exceeded, will cause the paste operation to be slowed down.
g_nClipboardLengthThreshold = 500

' Define the amount of time (milliseconds) that should be introduced as
' a delay between each line being sent to the remote host. You'll
' probably want to experiment with different values for g_nDelay to find
' out which one provides the best balance between performance and over-
' whelming the remote host:
g_nDelay = 300

Sub Main()
' If there isn't an active connection, there's no point in continuing
' since we can't send data.
If Not crt.Session.Connected Then
    crt.Dialog.MessageBox "Sending data requires an active connection."
    Exit Sub
End If

' If there isn't anything in the Windows Clipboard, it doesn't make any
' sense to continue with the script. Note that we're checking for
' vbCrLf here, since an empty clipboard will only contain a trailing
' CRLF sequence.
If Trim(crt.Clipboard.Text) = vbCrLf Then
    crt.Dialog.MessageBox "No text found in the Windows Clipboard."
    Exit Sub
End If

' Only delay if the length of the clipboard is greater than the established
' threshold
If Len(crt.Clipboard.Text) > g_nClipboardLengthThreshold Then
' Multiple lines in the Windows Clipboard are stored with a CRLF separator.
' Break up lines into an array (each line as an element within the array):
vLines = Split(crt.Clipboard.Text, vbCrLf)

' Enter a loop, sending a line, and then sleeping for g_nDelay milliseconds
' after each line is sent.

For Each strLine In vLines
' The vbcr is needed to simulate pressing the Enter key since we
' lost the line ending vbcr during the above Split operation.
crt.Screen.Send strLine & vbcr
crt.Sleep g_nDelay
Next

' Inform that the data has all been sent.
crt.Dialog.MessageBox _
    "All data in the Windows Clipboard has been sent to the remote."
Else
' The amount of text in the clipboard is small enough to send without a delay.
crt.Screen.SendSpecial "MENU_PASTE"
End If
End Sub

```

8.4 Setting the Clipboard Text Format (Encoding)

If you copy text from a Microsoft Office application or from a web page, you may notice that certain characters are not displayed correctly when the Clipboard contents are pasted to the remote

machine via SecureCRT. This problem is sometimes caused by a difference between the text encoding in the application from which it was copied and the encoding format accepted by the remote system. The remote system may recognize a hyphen, but not know what an *em* dash or an *en* dash is. It might recognize a standard quote character (ASCII 34 decimal), but complain about being sent "smart quotes". SecureCRT provides a private clipboard format named "VDS_TEXT" that will automatically convert upper ASCII smart quotes, *em* dashes, and *en* dashes to their lower ASCII equivalents in order to facilitate compatibility with remote applications/shells that do not know about these upper ASCII characters.

You might also find that you need to copy Unicode text containing foreign language characters, which need to be preserved when copying from or pasting to a capable remote system.

If you've changed the clipboard format within a SecureCRT script for a specific task and want to restore it to the default format (defined in SecureCRT's `Global.ini` file as the value of the `Clipboard Data Format .ini` file only option), you can reset the clipboard format by using the "DEFAULT" format constant. As an example of how changing the clipboard format can affect how certain characters appear after being copied and pasted, copy the following text to the Windows Clipboard:

```
These are "Smart Quotes"  
ls -l *.txt  
vshelld -restart
```

Then, map a button to run the following script code:

```
' ChangeClipboardFormat.vbs  
'  
' Description:  
' Demonstrates how to temporarily set the clipboard format to match the  
' type of text that is copied into the Windows Clipboard.  
  
crt.Dialog.MessageBox _  
  "Clipboard viewed in DEFAULT format (as defined in Global.ini): " & _  
  vbCrLf & vbCrLf & _  
  crt.Clipboard.Text  
  
crt.Clipboard.Format = crt.Clipboard.VDS_TEXT  
crt.Dialog.MessageBox _  
  "Clipboard viewed in VDS_TEXT format:" & _  
  vbCrLf & vbCrLf & _  
  crt.Clipboard.Text  
  
crt.Clipboard.Format = crt.Clipboard.CF_UNICODETEXT  
crt.Dialog.MessageBox _  
  "Clipboard viewed in CF_UNICODETEXT format:" & _  
  vbCrLf & vbCrLf & _  
  crt.Clipboard.Text  
  
crt.Clipboard.Format = nFormat  
crt.Dialog.MessageBox _  
  "Now, Back to DEFAULT format: " & crt.Clipboard.Format & _  
  vbCrLf & vbCrLf & _  
  crt.Clipboard.Text
```

If you have a default SecureCRT configuration, you'll see the following sequence of dialogs:



Sequence of Dialogs that Appear When Running [ChangeClipboardFormat.vbs](#)

You can see by the difference in appearance of the quotation marks and the dashes in the graphics above, that the VDS_TEXT format causes the text to be treated as lower 7-bit ASCII characters. Also, since the CF_UNICODETEXT format appears identical to the DEFAULT format, you can see that CF_UNICODETEXT is the default clipboard format in SecureCRT.