LoadRunner Script Recording: Winsock Protocol

BY

Scott Moore & James Pulley

Version 1.2
04/22/2015

Table of Contents

Introduction

The purpose of this document is to review a basic understanding of the Winsock protocol, and how to record Winsock traffic through LoadRunner or from network sniffer trace (i.e. Wireshark). There are few documented resources readily available that offer any in-depth discussion around performance testing applications using the LoadRunner Winsock protocol.

Once a business process has been recorded and enhanced as a VuGen script, test execution is managed like any other application. For this reason, building LoadRunner scenarios and test execution is outside the scope of this document.

Originally, many of the simple TCP/IP based applications for Telnet, FTP, POP3, SMTP, etc were ALL Winsock protocol applications within LoadRunner. As LoadRunner has evolved as a product over the years, "patterns" emerged from these applications and the LoadRunner Research and Development team added higher level functions to do much of the low level work to make scripting easier. New functions were added to LoadRunner headers as engineers gained experience from Winsock-based testing engagements. One example was the Tuxedo recording protocol for older versions of PeopleSoft. The Winsock protocol can still be used, but it will be more time consuming when there are not higher level functions available.

Why Does Everyone Hate Winsock?

As a concept, Winsock is not that difficult. Data is passed back and forth inside of buffers (or memory containers). The complexity increases when there are multiple conversations (sessions) to keep track of. The concept is the still the same. It is the tracking of the conversations that makes VuGen Winsock scripting difficult for many performance engineers. Once a buffer is “captured”, the issue becomes what to do with it. Does it need to be saved, modified, or simply ignored? If data needs to be manipulated inside the buffer, this is where knowledge of string manipulation using ANSI C code becomes important. Advanced C scripting skills will be needed in order to be proficient at modifying the data properly.

How Winsock Works

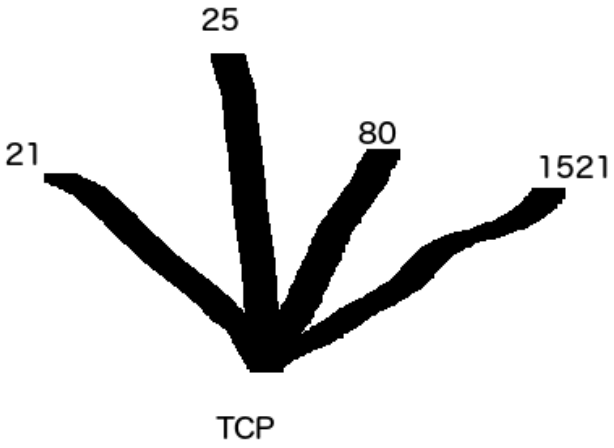
There are 3 ways an application can work with sockets:

- TCP (connection oriented)
- UDP (connectionless)
- RAW

Although RAW sockets are both powerful and dangerous, most of the time applications will not be using this method. If they do, the chances of capturing traffic within LoadRunner by recording (hooking into the application) are pretty low. An example of this would be the PING

command line executable. This uses the ICMP protocol, which is a RAW sockets call. Up until Windows XP, there was no RAW sockets interface for the Windows operating system.

Since WinSock is based on the standard TCP/IP stack, it is assumed an application will use standard ports. For example, FTP would be on port 21, while SMTP would be on port 25, HTTP would be on port 80, or 1521 for a database service. Sockets represent the foundation layer upon which all "well known services," that listen on a standard port, rely.



A great set of example documentation which illustrates this foundation layer of sockets in relation of a well known service is the documentation set associated with the SMTP (Simple Mail Transfer Protocol), RFC 2821 (https://www.ietf.org/rfc/rfc2821.txt) or the FTP protocol (http://tools.ietf.org/html/rfc959)

Because there is so much detail to the script, and it is usually hand coded Winsock conversations, it can take a lot longer to create a WinSock script than a web script. As a rough estimate, a standard Winsock script takes about three times as long to complete than a web script of the same business process complexity for a mid-level LoadRunner resource. This assumes that the user is already very proficient with LoadRunner and proficient in C.

Winsock Utilities and Resources

There are several utilities and/or online resources that will assist the performance engineer when working with the Winsock protocol.

- **LCC compiler for MS Windows:**

  http://www.cs.virginia.edu/~lcc-win32/

The LCC Compiler was instrumental in the creation of LoadRunner. Understanding how this works will give insight into the details of LoadRunner.

The help files can be downloaded and installed separately. Many times it is possible to demonstrate code that will work in LCC and will not work in LoadRunner. This will prove to Mercury support that a bug in LoadRunner exists. Understanding LCC will help in understanding LoadRunner better overall.

- **Wireshark Network Sniffer**

  https://www.wireshark.org/

  Wireshark is a free network sniffer that will show decoded traffic and understand the various "Conversations" by protocol. Use Wireshark to capture traffic and determine if an application really is a Winsock application. Any comparable network sniffer software can be substituted in place of this product, as long as it has roughly the same features.

- **Microsoft Winsock Information**

  https://msdn.microsoft.com/en-us/library/windows/desktop/ms740632%28v=vs.85%29.aspx

- Standard TCP / UDP Port Information. Listing Of Standard TCP / UDP Ports Updated Frequently:

  http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

- **OTHER Resources:**

  Wikipedia: http://wikipedia.org/wiki/Berkeley_sockets

Recording Winsock Protocol

To Determine If Application Can Be Recorded With Winsock:

Record sample traffic of the application with Wireshark. Look in the 'Protocol' column. If TCP/UDP is listed, it can most likely be recorded.

If using Vugen to record Winsock with a raw sockets interface, the chances are very low that LoadRunner will capture it.

If the application has encryption enabled it should be turned off, or the developer will spend a significant amount of time implementing the encryption and decryption algorithms within the sockets virtual user. This will greatly increase scripting development time.

TIP: *Get familiar with how different protocols look at socket level. For TCP applications, use Wireshark to determine what ports are sending data.*

Assuming the application can be recorded into LoadRunner, the next step is to ensure that the data can be correlated.

Best Practice for Correlation:

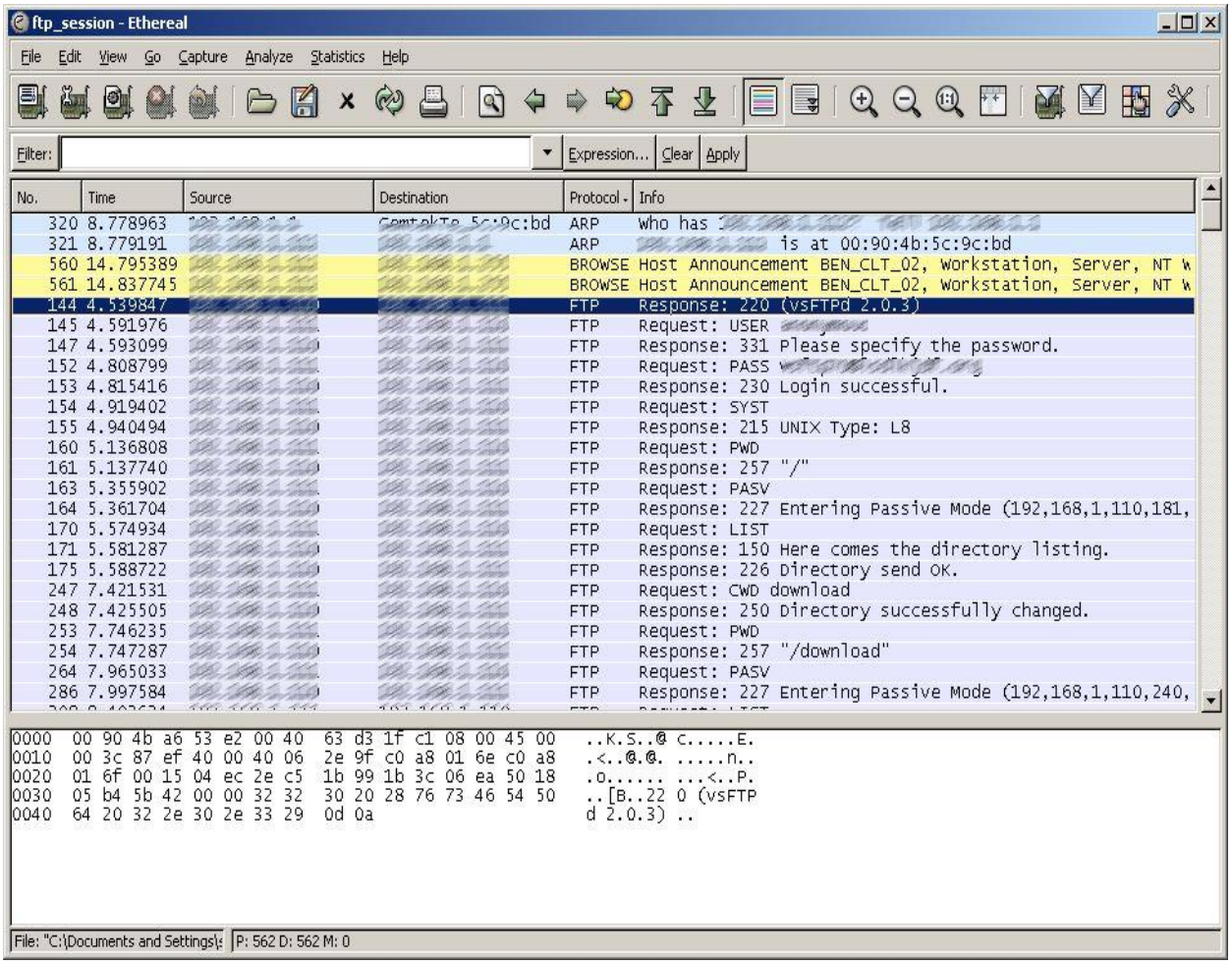
- Record a business process twice exactly the same way and use a comparison tool (Windiff is built into LoadRunner) for items that need correlation. Other options for tools include WinMerge or Beyond Compare.
- The data.ws section is basically a parameter file of what buffers were captured. They can be reused, or the buffer can be created dynamically. Get familiar with manually creating buffers.
- Make sure when sending a buffer back to the server that it is formatted correctly. On the end of the command there may be hidden characters like /r/n (carriage return or Enter). If these characters are missed, it will never send the final "enter" key for the command and the script will fail.

Basic LoadRunner commands for Winsock Scripting:

- lrs_create_socket – created a connection to the TCP application.
- lrs_set_send_buffer – used to create the contents of a buffer manually. This is used to avoid the common (and dreaded) "mismatched buffer" warning.
- lrs_send – send the contents of the buffer
- lrs_set_receieve_option – Set the receive options to control what happens if receive buffer is different.
- lrs_recieve – receive the buffer back
- lrs_get_received_buffer – Gets the buffer that was returned
- lrs_get_last_received_buffer_size – Get the size of the last buffer received
- lrs_close_socket – close the connection to the TCP application

Winsock Pattern Recognition Workflow

In this section, we will walk through the typical process of recording a script. First, capture the application in Wireshark. Launch the application and go through a simple business process using the application user interface. Try to keep other network traffic to a minimum by shutting down any other applications if possible, especially those that continually make connections to the Internet or some other remote server. Look in the column called "Protocol". For most applications, Wireshark should properly DCODE the application protocol and match some protocol in LoadRunner. The one notable exception would be DCOM, which looks like TCP/UDP. If the application communicates through TCP or UDP, most likely it can be captured with the Winsock protocol in Vugen.



In the above example, the FTP protocol is making a remote connection to an FTP server.

Second, after recording the business process in Wireshark, repeat it in LoadRunner and take a look at the two. Compare them and search out patterns. Pattern recognition is something that

each individual does and one can only get good at this by continually getting more experience with Winsock applications.

For anything outside rule #1, LoadRunner probably will not record it. Examples of this include SMB (server message block), VINES, etc...In those cases a C template user with hand coded C would have to launch a shell to complete actions in a business process.

Custom Monitors and Vusers

There is an API library from HP in the extendable LoadRunner Protocol SDK which allows developers to build custom virtual users by coding custom written DLL's. They can use this to build a custom interface on top of sockets. This is how many of the additional protocols built after Winsock were developed (POP3, FTP, etc). This is how protocols move up the stack.

HEX Information

Most people see all of the hexadecimal data being passed in a Winsock conversation and they are immediately overwhelmed at how to make sense of it. The HEX that is displayed in the data.ws file is not something to be concerned about. It is normally used for data separation in the sockets universe. To see if hex information is really useful, compare two recordings of a business process using only one piece of changed data and see what the differences are using Windiff.

HEX may show up when there are EBCDIC to ASCII conversions. This is found mostly in IBM mainframe or Minicomputer environments (AS400). These systems are well known for their scalability. To get away from having to test mainframes, simply ask if the mainframe has ever been tested (by the vendor or the client) for scalability. If so, use stubs instead of the actual mainframe to accomplish testing. Get a signature from the project owner that they believe their mainframe scales.

HEX can show up when there are STRUCTURED RECORDS with a consistent length. When buffers are always the same size no matter what, go to the developer and get the structure. This can be seen by recording twice while changing the data and then using Windiff.

Transaction Timings

Because it is an expensive use of memory, when manipulating buffers and strings, the best practice is to handle string manipulation outside of the context of the open transaction:. For example:

1. Start Transaction
2. lrs_send and receive
3. Pause transaction
4. Manipulate buffer and do string manipulation
5. Free buffer
6. Resume transaction
7. End transaction when the rest of the lrs_send/receive code is complete.

To pause a transaction, use the lr_stop_transaction method:

```
lr_start_transaction("foo");
...code...
lr_stop_transaction("foo");

/* do a bunch of heavy lifting code that consumes CPU cycles and adds
time */

lr_resume_transaction("foo");
...code...
lr_end_transaction("foo",mymarker);
```

If you have a transaction stopped and you end it, the transaction is automatically resumed and then ended. For more information about these additional functions, view the lrun.h file.

You may also simply store data that requires manipulation, end the transaction and manipulate the string if necessary. Processing the data by pausing the transaction may be required for data validation to appropriately set the status of the end_transaction() marker to LR_PASS or LR_FAIL.

Generic LoadRunner Winsock Recording

Below is a sample LoadRunner script in the generic form. The action section is where the LoadRunner code is sending and receiving the buffers. The data.ws file is simply a data file. It contains the information that was in the buffers during the time it was originally recorded. It can be overridden with new information and sent back modified. This would happen if changes need to be made to the SEND buffer to properly correlate the dynamic data being sent to the application server.

ACTION()

```
Action()
{
lrs_create_socket("socket0", "TCP", "LocalHost=0", "RemoteHost=192.168.453.973:23",  LrsLastArg);
lrs_set_receive_option(EndMarker, EndMarker_None);
lrs_receive("socket0", "buf0", LrsLastArg);
lrs_send("socket0", "buf6", LrsLastArg);
lrs_receive("socket0", "buf7", LrsLastArg);
lrs_send("socket0", "buf8", LrsLastArg);
lrs_receive("socket0", "buf9", LrsLastArg);
lrs_disable_socket("socket0", DISABLE_SEND_RECV);
lrs_close_socket("socket0");
return 0;
}
```

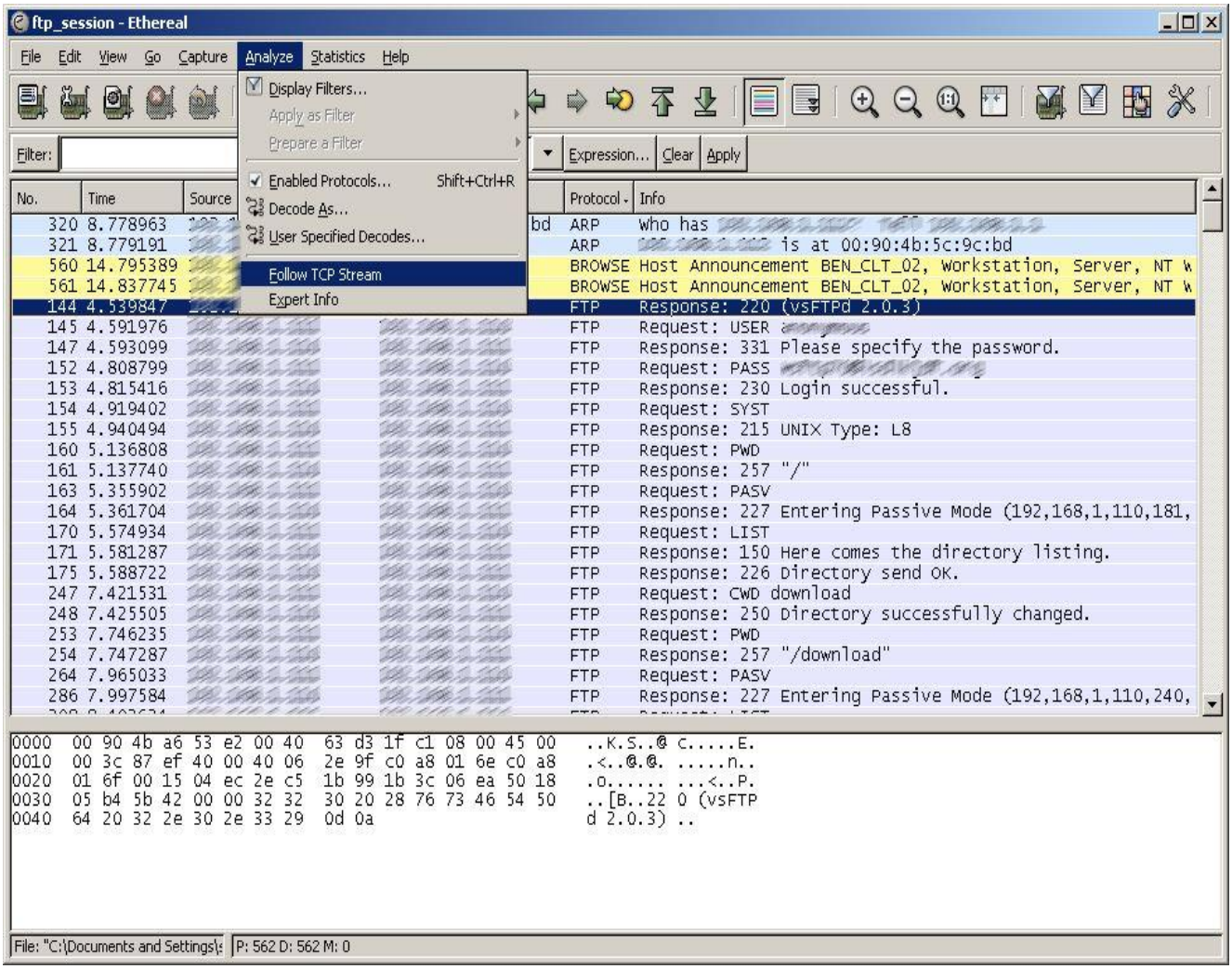
DATA.WS

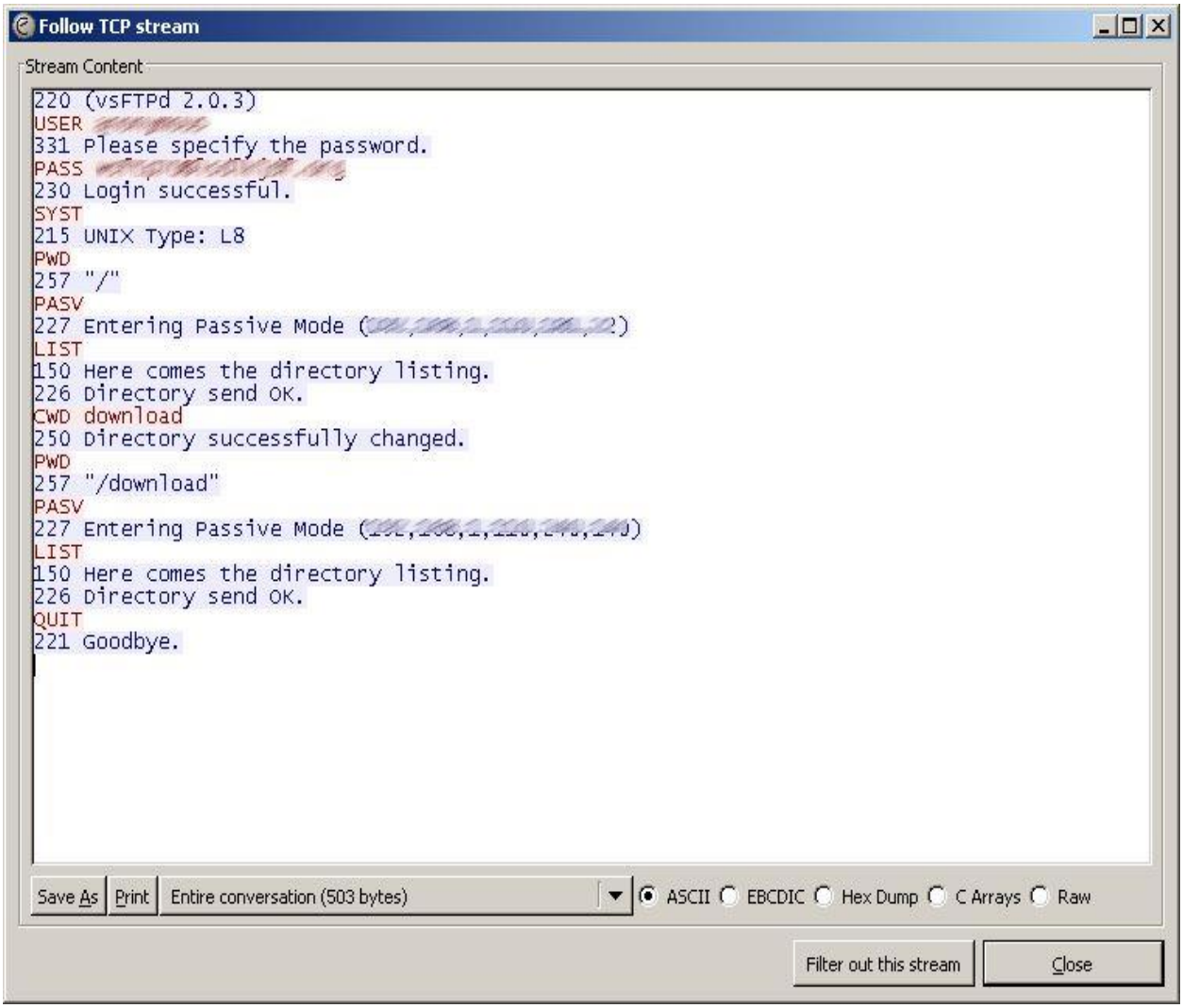
```
recv  buf6 75
        "\xff\xfe\x01\xff\xfb\x01"
        "Welcome to SUSE LINUX 10.0 (i586) - Kernel 2.6.13-15.8-default (2).\r\n"

send  buf7 6
        "\xff\xfc\x01\xff\xfd\x01"

recv  buf8 18
        "\r\n"
        "powerUX1 login: "

//Username
send  buf9 1
        "lxuser\r"

//Username
recv  buf10 1
        "lxuser"
```

Wireshark Trace Of A Business Process

There may be times when information gathered from Wireshark is used to hand code a Winsock script (without recording in Vugen). To do this:

1. Identify Business Process
2. Access a sniffer port and watch the business process from a network point of view
3. Reassemble the streams (using Wireshark) and get rid of extraneous network conversation
4. Convert the streams into buffers being sent by LoadRunner using send/recv buffers in the data.ws and custom LoadRunner code.



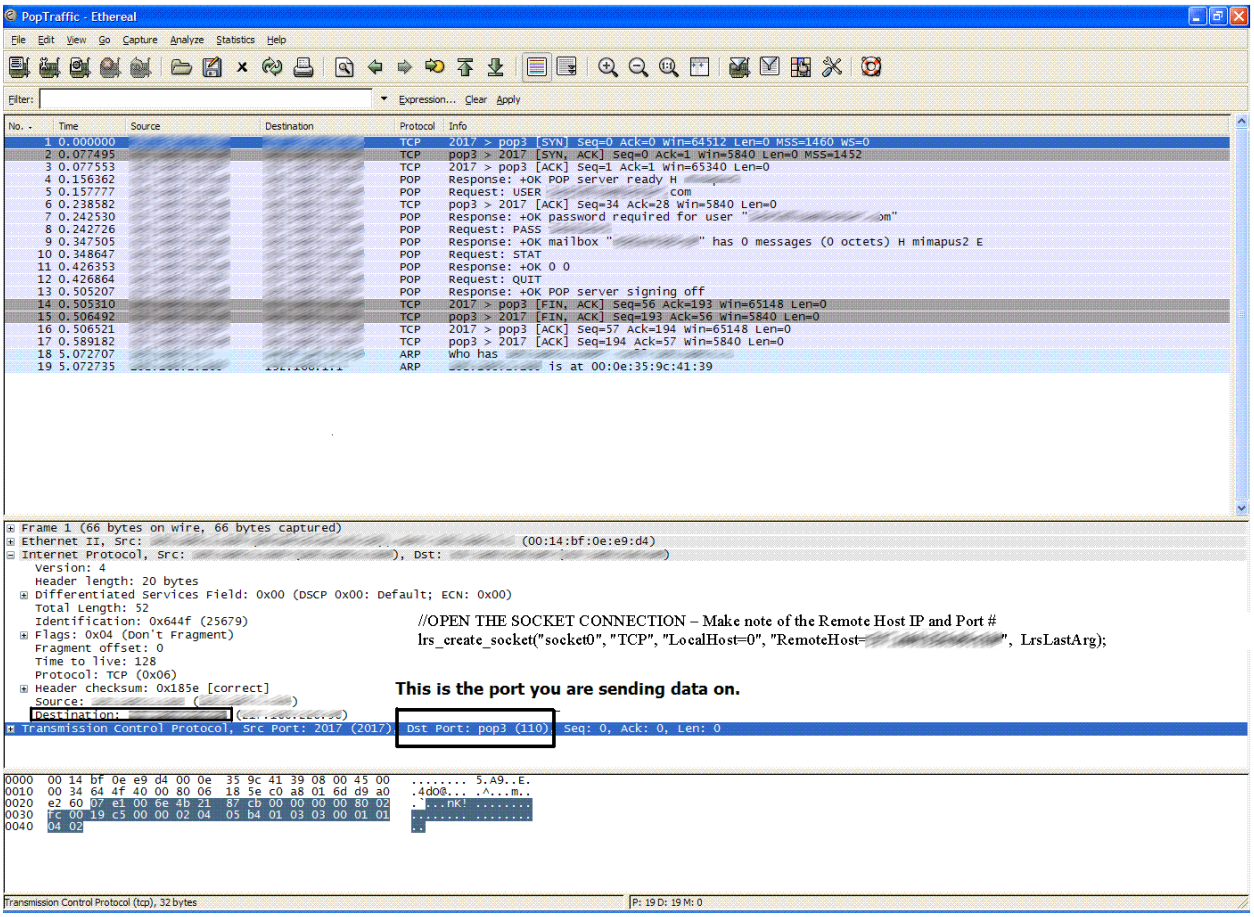


In the TCP stream shown above, after the initial connection "220 (vftp 2.0.3) was RECEIVED, he text "USER anonymous" was SENT to the ftp server. This is what would be sent in the lrs_send function. The response from the server is in the next line below. By following along in the conversation, it will reveal what needs to be sent next to the server. This is how to convert a sniffer trace to a LoadRunner script.

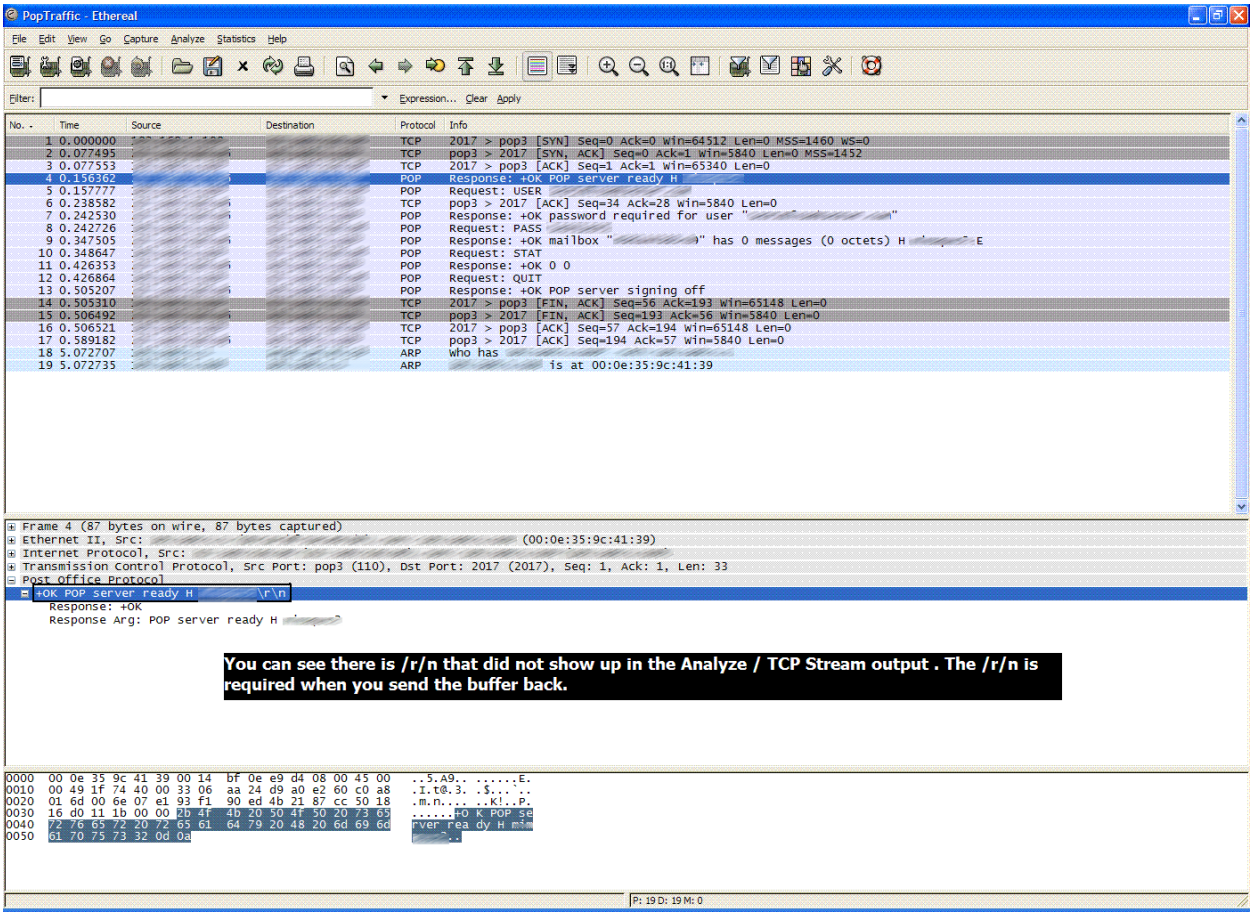
The Send buffer would be set by using lrs_set_send_buffer() manually from the information in the decoded TCP stream. The “follow stream” function in Wireshark is TCP protocol specific. UDP protocol streams will not have this functionality. Fortunately, most client-server applications ride on top of TCP connections

How To Find Port Number In Wireshark The Protocol Is Listening On

//OPEN THE SOCKET CONNECTION – Make note of the Remote Host IP and Port #
lrs_create_socket("socket0", "TCP", "LocalHost=0", "RemoteHost=**123.456.789.123:110**", LrsLastArg);
lrs_receive("socket0","buf2",LrsLastArg);

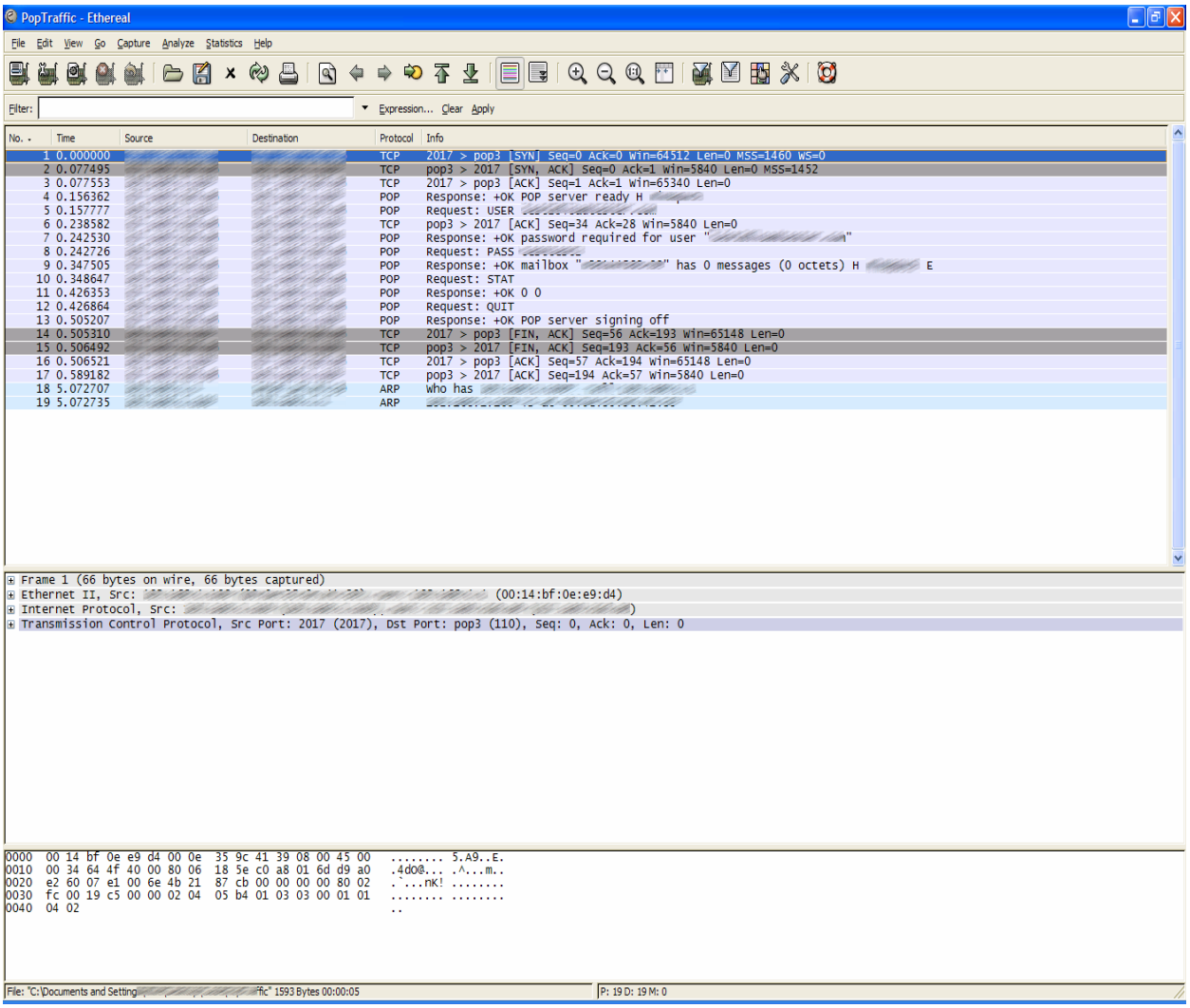


Identifying Hidden Characters At The End Of The Buffer

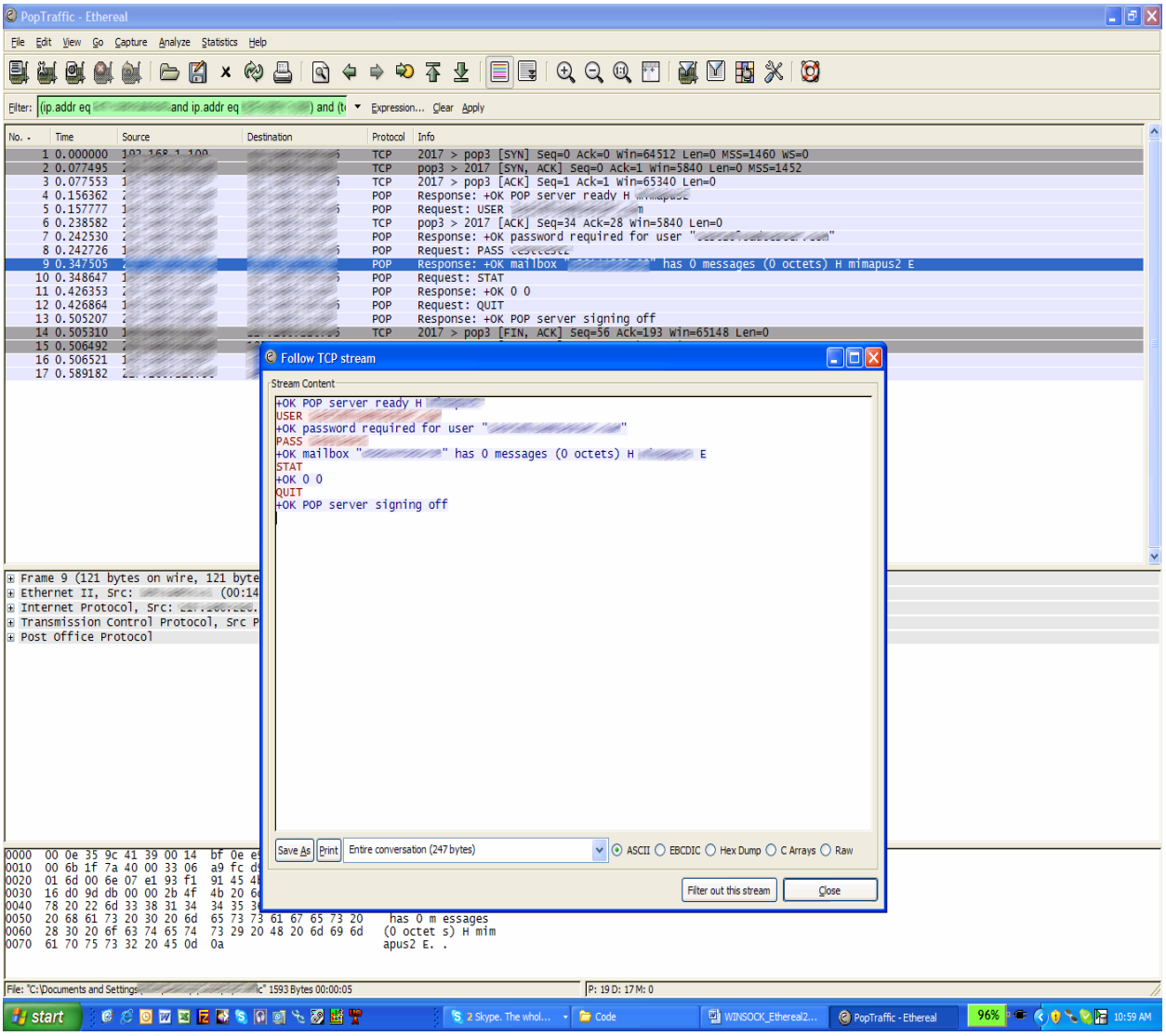


LoadRunner Winsock Script Created From A Wireshark Trace

Screen Shot of APOP 3 Session Checking Email



To See What Session Looked Like From Start to Finish (Analyze, Follow TCP Stream)



Script Example: Winsock Script That Checks Email and Displays How Many Messages There Are

SEND MESSAGES	RECEIVE MESSAGES
USER sdkjsdk@akjkajs.com	+OK password required for user "sdkjsdk@akjkajs.com"
PASS secret123	+OK mailbox "XXXXXX" has 0 messages (0 octets) H server2
STAT	+OK 0 0
QUIT	+OK POP server signing off
EOF	EOF

Action()

```
{
char tempbuffer[100];
char workingbuffer[100];
char messagetext[100];
char *ReceivedBuffer; // Store Contents Of Buffer
int count=1; //Count loop
int RecievedBufferSize;  // Store Size Of Buffer Being Returned
char *search_str = "message";
int offset;
char *position;
int Startoffset, Endoffset;
int x; // For The For LOOP

//Open Connection To Mail Server
lrs_create_socket("socket0", "TCP", "LocalHost=0", "RemoteHost=123.456.789.10:110",  LrsLastArg);

lrs_receive("socket0","buf2",LrsLastArg);


// The Logic Will Read The Paramater File Until It Finds "EOF".
while (strcmp(lr_eval_string("<SendMessages>"), "EOF"))

{
//Grab The Response From The Paramter File being sent And Append The \r\n To The Very End
sprintf(tempbuffer,"%s\r\n",lr_eval_string("<SendMessages>"));

//Set The Send Buffer To Value Of TEMPBUFFER And The Length
lrs_set_send_buffer("socket0",tempbuffer, strlen(tempbuffer));

//Send The Buffer, Buf0 Is Ingnored When lrs_set_send_buffer Is Used
lrs_send ("socket0", "buf0", "TargetSocket=123.456.789.10:110", LrsLastArg );

//Recieve The Response Back
lrs_receive("socket0","buf1",LrsLastArg);


// Get The Complete Buffer That Is Being Sent Back Starting at Offset 0
// -1 Tells The Command To Grab The Whole Buffer
// By Setting NULL It Will Automically Determine The Data Is ASCII or EBCIDIC
```

```
ReceivedBuffer = lrs_get_received_buffer("socket0",0,-1,"NULL");
//Get The Total Buffer Size Being Returned
RecievedBufferSize = lrs_get_last_received_buffer_size("socket0");
lr_output_message("INFO: Loop %d - Recievied Buffer: %s (Size = %d) ",count,ReceivedBuffer,RecievedBufferSize);


//*********************************************************************************************
//Search For # Of Message's In the Inbox. We will search for the word mesages in the buffer
//*********************************************************************************************

/*
This the data we are searching thru. We are looking for the number of emails recieved
+OK mailbox "XXXXXX" has 6 messages (10243 octets) H server2 N
*/

//Searches The String For Some Text And Sets The Pointer To 1st Position If Found,  Other Wise It Reports A Null
position = (char *)strstr(ReceivedBuffer, search_str);

//Determine If Match Was Found
if (position != NULL)

{

//Calculation To Get Offset Of Where Match Found. This Is Pointer Arthimetic.
//The Offset - 1 Moves The Pointer Passed The Initial Space And Onto The Number
offset = (int)((position - ReceivedBuffer)-1);

//Copy The String To A New Variable And Chop Of The Rest Of The String Not Needed.
//The New Buffer We Will Search Thru Will be: +OK mailbox "XXXXXX" has 6
strncpy(tempbuffer,ReceivedBuffer,offset); //Copy Just The Amount Of The Buffer I Need To Work, Starting At 0
lr_output_message("INFO: New Tempbuffer is %s",tempbuffer);

//lr_output_message("INFO: The end of the string \"%s\" was found at position %d", search_str, offset);

//Search For The Word "HAS" And Then Move 4 Spaces, The Number Of Emails Sent Has Been Catured
//When The Strcpy Was Performed It Saved The Buffer From Position We Moved To, To The Very End Of The Buffer
strcpy(workingbuffer,(char*)strstr(tempbuffer,"has")+4);
lr_output_message("The Number Of Messages Captured Is %s",workingbuffer);

}

//*********************************************************************************************

/Advnance The Parameter File
lr_advance_param("SendMessages");
lr_advance_param("RecieveMessages");

//Increment Counter Instance Of Loop
count++;

}


lrs_close_socket("socket0");
return 0;

}
```

DATA.WS

```
;WSRData 2 1

send buf0 50

recv buf1 50

recv buf2 50

-1
```

Output of Execution Log:

```
Virtual User Script started
Starting action vuser_init.
vuser_init.c(6): lrs_startup(257)
Ending action vuser_init.
Running Vuser...
Starting iteration 1.
Starting action Action.
Action.c(21): lrs_create_socket(socket0, TCP, ...)
Action.c(21): callSocket: name: 10471240, socket type: 1
Action.c(21): callBind: socket: socket0 (212), local host: (null), local port: 0
Action.c(21): callConnect: socket: socket0 (212), remote host: 123.456.789.10  remote port: 110
Action.c(21): callConnect: getting host address from the host name (123.456.789.10).
Action.c(21): callConnect: Sat Apr 29 13:41:24.328: Waiting for writable socket 10 secs, 0 usecs
Action.c(23): lrs_receive(socket0, buf2)
Action.c(23): fiInitRecv: socket: socket0 (212), buffer: buf2, expected size: 50
Action.c(23): callRecv: Sat Apr 29 13:41:24.343: About to read 50 bytes from socket0 (212)
Action.c(23): fiPerformReceive: Sat Apr 29 13:41:24.343: Waiting for readable socket 10 secs, 0 usecs
Action.c(23): fiPerformReceive: Sat Apr 29 13:41:24.421: Trying to read 8192 bytes
Action.c(23): fiPerformReceive: Sat Apr 29 13:41:24.437: store received data , 57 bytes
Action.c(23): fiPerformReceive: Sat Apr 29 13:41:24.437: Waiting for readable socket 10 secs, 0 usecs
Action.c(23): fiPerformReceive: Sat Apr 29 13:41:34.437: Select timed out
Action.c(23): Mismatch (expected 50 bytes, 57 bytes actually received)
Action.c(23): fiCheckRecvMismatch: Sat Apr 29 13:41:34.437: reading buffer buf2
Action.c(23): fiCheckRecvMismatch: Sat Apr 29 13:41:34.437: calling parameterization routines
================================EXPECTED BUFFER================================
================================================================================
Action.c(23): getAsciiReceivedBuffer: Sat Apr 29 13:41:34.437: Translate data for printing
Action.c(23): getAsciiReceivedBuffer: Sat Apr 29 13:41:34.437: Binary to ascii
================================RECEIVED BUFFER================================
        "+OK AVG POP3 Proxy Server 7.1.371/7.1.385 [123.5.1/327]\r\n"
================================================================================
Action.c(23): callRecv:57 bytes were received
Action.c(27): Notify: Parameter Substitution: parameter "SendMessages" =  "USER sdkjsdk@akjkajs.com"
Action.c(31): Notify: Parameter Substitution: parameter "SendMessages" =  "USER sdkjsdk@akjkajs.com"
Action.c(34): lrs_set_send_buffer(socket0, buf, 27)
Action.c(34): callSetSendBuffer: setting next send buffer (socket: socket0)
Action.c(37): lrs_send(socket0, buf0)
Action.c(37): callSend: socket: socket0 (212), buffer: buf0
Action.c(37): callSend: Sat Apr 29 13:41:34.453: Overriding send buffer buf0
```

```
Action.c(37): callSend: Sat Apr 29 13:41:34.453: calling parameterization routines
Action.c(37): callSend: Sat Apr 29 13:41:34.453: translate buffer to binary
Action.c(37): callSend: Sat Apr 29 13:41:34.453: Translate data for printing
==================================SENT BUFFER==================================
        "USER sdkjsdk@akjkajs.com\r\n"
===============================================================================
Action.c(37): callSend: getting host address from the host name (123.456.789.10).
Action.c(37): callSend: Sat Apr 29 13:41:34.468: About to send 27 bytes to socket0 (212)
Action.c(37): callSend: Sat Apr 29 13:41:34.468: Waiting for writable socket 10 secs, 0 usecs
Action.c(37): callSend: Sat Apr 29 13:41:34.468: Sent in this iteration 27 bytes (total in all iterations 27 bytes)
Action.c(40): lrs_receive(socket0, buf1)
Action.c(40): fiInitRecv: socket: socket0 (212), buffer: buf1, expected size: 50
Action.c(40): callRecv: Sat Apr 29 13:41:34.468: About to read 50 bytes from socket0 (212)
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:34.468: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:34.515: Trying to read 8192 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:34.515: store received data , 55 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:34.515: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:44.515: Select timed out
Action.c(40): Mismatch (expected 50 bytes, 55 bytes actually received)
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:41:44.515: reading buffer buf1
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:41:44.531: calling parameterization routines
================================EXPECTED BUFFER================================
===============================================================================
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:41:44.531: Translate data for printing
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:41:44.531: Binary to ascii
==================================RECEIVED BUFFER==================================
        "+OK password required for user \"sdkjsdk@akjkajs.com\"\r\n"
===============================================================================
Action.c(40): callRecv:55 bytes were received
Action.c(46): lrs_get_received_buffer(socket0, buf_p, size_p)
Action.c(46): callGetReceivedBuffer: socket: socket0 (212), offset: 0, length: -1
Action.c(46): callGetReceivedBuffer: reading last received buffer
Action.c(46): callGetReceivedBuffer: translate buffer to ascii
Action.c(46): callSetUserBuffer: setting user buffer (socket: socket0)
Action.c(48): lrs_get_last_received_buffer_size(socket0)
Action.c(49): INFO: Loop 1 - Recievied Buffer: +OK password required for user "sdkjsdk@akjkajs.com"
 (Size = 55)
Action.c(89): Notify: Next row for parameter SendMessages = 2 [table  = SendMessages].
Action.c(89): Notify: Getting new value for parameter 'SendMessages': table = 'SendMessages.dat' column = '0' row = '2'.
Action.c(90): Notify: Next row for parameter RecieveMessages = 2 [table  = RecieveMessages].
Action.c(90): Notify: Getting new value for parameter 'RecieveMessages': table = 'RecieveMessages.dat' column = '0' row = '2'.
Action.c(27): Notify: Parameter Substitution: parameter "SendMessages" =  "PASS secret123"
Action.c(31): Notify: Parameter Substitution: parameter "SendMessages" =  "PASS secret123"
Action.c(34): lrs_set_send_buffer(socket0, buf, 16)
Action.c(34): callSetSendBuffer: setting next send buffer (socket: socket0)
Action.c(37): lrs_send(socket0, buf0)
Action.c(37): callSend: socket: socket0 (212), buffer: buf0
Action.c(37): callSend: Sat Apr 29 13:41:44.562: Overriding send buffer buf0
Action.c(37): callSend: Sat Apr 29 13:41:44.562: calling parameterization routines
Action.c(37): callSend: Sat Apr 29 13:41:44.562: translate buffer to binary
Action.c(37): callSend: Sat Apr 29 13:41:44.562: Translate data for printing
==================================SENT BUFFER==================================
        "PASS secret123\r\n"
===============================================================================
Action.c(37): callSend: getting host address from the host name (123.456.789.10).
Action.c(37): callSend: Sat Apr 29 13:41:44.578: About to send 16 bytes to socket0 (212)
Action.c(37): callSend: Sat Apr 29 13:41:44.578: Waiting for writable socket 10 secs, 0 usecs
Action.c(37): callSend: Sat Apr 29 13:41:44.578: Sent in this iteration 16 bytes (total in all iterations 16 bytes)
```

Action.c(40): lrs_receive(socket0, buf1)
Action.c(40): fiInitRecv: socket: socket0 (212), buffer: buf1, expected size: 50
Action.c(40): callRecv: Sat Apr 29 13:41:44.578: About to read 50 bytes from socket0 (212)
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:44.578: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:44.625: Trying to read 8192 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:44.625: store received data , 71 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:44.625: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:54.625: Select timed out
Action.c(40): Mismatch (expected 50 bytes, 71 bytes actually received)
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:41:54.640: reading buffer buf1
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:41:54.640: calling parameterization routines
==========EXPECTED BUFFER==========
==========
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:41:54.640: Translate data for printing
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:41:54.640: Binary to ascii
==========RECEIVED BUFFER==========
"+OK mailbox \"XXXXXX\" has 6 messages (10243 octets) H server2 N\r\n"
==========
Action.c(40): callRecv:71 bytes were received
Action.c(46): lrs_get_received_buffer(socket0, buf_p, size_p)
Action.c(46): callGetReceivedBuffer: socket: socket0 (212), offset: 0, length: -1
Action.c(46): callGetReceivedBuffer: reading last received buffer
Action.c(46): callGetReceivedBuffer: translate buffer to ascii
Action.c(46): callSetUserBuffer: setting user buffer (socket: socket0)
Action.c(48): lrs_get_last_received_buffer_size(socket0)
Action.c(49): INFO: Loop 2 - Recievied Buffer: +OK mailbox "XXXXXX" has 6 messages (10243 octets) H mimapus1 N (Size = 71)
Action.c(76): INFO: New Tempbuffer is +OK mailbox "XXXXXX" has 6
Action.c(82): The Number Of Messages Captured Is 6
Action.c(89): Notify: Next row for parameter SendMessages = 3 [table = SendMessages].
Action.c(89): Notify: Getting new value for parameter 'SendMessages': table = 'SendMessages.dat' column = '0' row = '3'.
Action.c(90): Notify: Next row for parameter RecieveMessages = 3 [table = RecieveMessages].
Action.c(90): Notify: Getting new value for parameter 'RecieveMessages': table = 'RecieveMessages.dat' column = '0' row = '3'.
Action.c(27): Notify: Parameter Substitution: parameter "SendMessages" = "STAT"
Action.c(31): Notify: Parameter Substitution: parameter "SendMessages" = "STAT"
Action.c(34): lrs_set_send_buffer(socket0, buf, 6)
Action.c(34): callSetSendBuffer: setting next send buffer (socket: socket0)
Action.c(37): lrs_send(socket0, buf0)
Action.c(37): callSend: socket: socket0 (212), buffer: buf0
Action.c(37): callSend: Sat Apr 29 13:41:54.687: Overriding send buffer buf0
Action.c(37): callSend: Sat Apr 29 13:41:54.687: calling parameterization routines
Action.c(37): callSend: Sat Apr 29 13:41:54.687: translate buffer to binary
Action.c(37): callSend: Sat Apr 29 13:41:54.687: Translate data for printing
==========SENT BUFFER==========
"STAT\r\n"
==========
Action.c(37): callSend: getting host address from the host name (123.456.789.10).
Action.c(37): callSend: Sat Apr 29 13:41:54.687: About to send 6 bytes to socket0 (212)
Action.c(37): callSend: Sat Apr 29 13:41:54.687: Waiting for writable socket 10 secs, 0 usecs
Action.c(37): callSend: Sat Apr 29 13:41:54.687: Sent in this iteration 6 bytes (total in all iterations 6 bytes)
Action.c(40): lrs_receive(socket0, buf1)
Action.c(40): fiInitRecv: socket: socket0 (212), buffer: buf1, expected size: 50
Action.c(40): callRecv: Sat Apr 29 13:41:54.687: About to read 50 bytes from socket0 (212)
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:54.687: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:54.734: Trying to read 8192 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:54.734: store received data , 13 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:41:54.734: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:42:04.750: Select timed out

Action.c(40): Mismatch (expected 50 bytes, 13 bytes actually received)
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:42:04.750: reading buffer buf1
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:42:04.750: calling parameterization routines
==========EXPECTED BUFFER==========
==========
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:42:04.750: Translate data for printing
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:42:04.750: Binary to ascii
==========RECEIVED BUFFER==========
"+OK 6 10243\r\n"
==========
Action.c(40): callRecv:13 bytes were received
Action.c(46): lrs_get_received_buffer(socket0, buf_p, size_p)
Action.c(46): callGetReceivedBuffer: socket: socket0 (212), offset: 0, length: -1
Action.c(46): callGetReceivedBuffer: reading last received buffer
Action.c(46): callGetReceivedBuffer: translate buffer to ascii
Action.c(46): callSetUserBuffer: setting user buffer (socket: socket0)
Action.c(48): lrs_get_last_received_buffer_size(socket0)
Action.c(49): INFO: Loop 3 - Recievied Buffer: +OK 6 10243
(Size = 13)
Action.c(89): Notify: Next row for parameter SendMessages = 4 [table = SendMessages].
Action.c(89): Notify: Getting new value for parameter 'SendMessages': table = 'SendMessages.dat' column = '0' row = '4'.
Action.c(90): Notify: Next row for parameter RecieveMessages = 4 [table = RecieveMessages].
Action.c(90): Notify: Getting new value for parameter 'RecieveMessages': table = 'RecieveMessages.dat' column = '0' row = '4'.
Action.c(27): Notify: Parameter Substitution: parameter "SendMessages" = "QUIT"
Action.c(31): Notify: Parameter Substitution: parameter "SendMessages" = "QUIT"
Action.c(34): lrs_set_send_buffer(socket0, buf, 6)
Action.c(34): callSetSendBuffer: setting next send buffer (socket: socket0)
Action.c(37): lrs_send(socket0, buf0)
Action.c(37): callSend: socket: socket0 (212), buffer: buf0
Action.c(37): callSend: Sat Apr 29 13:42:04.781: Overriding send buffer buf0
Action.c(37): callSend: Sat Apr 29 13:42:04.781: calling parameterization routines
Action.c(37): callSend: Sat Apr 29 13:42:04.781: translate buffer to binary
Action.c(37): callSend: Sat Apr 29 13:42:04.781: Translate data for printing
==========SENT BUFFER==========
"QUIT\r\n"
==========
Action.c(37): callSend: getting host address from the host name (123.456.789.10).
Action.c(37): callSend: Sat Apr 29 13:42:04.796: About to send 6 bytes to socket0 (212)
Action.c(37): callSend: Sat Apr 29 13:42:04.796: Waiting for writable socket 10 secs, 0 usecs
Action.c(37): callSend: Sat Apr 29 13:42:04.796: Sent in this iteration 6 bytes (total in all iterations 6 bytes)
Action.c(40): lrs_receive(socket0, buf1)
Action.c(40): fiInitRecv: socket: socket0 (212), buffer: buf1, expected size: 50
Action.c(40): callRecv: Sat Apr 29 13:42:04.796: About to read 50 bytes from socket0 (212)
Action.c(40): fiPerformReceive: Sat Apr 29 13:42:04.796: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:42:04.843: Trying to read 8192 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:42:04.843: store received data , 28 bytes
Action.c(40): fiPerformReceive: Sat Apr 29 13:42:04.843: Waiting for readable socket 10 secs, 0 usecs
Action.c(40): fiPerformReceive: Sat Apr 29 13:42:04.843: Trying to read 8192 bytes
Action.c(40): Mismatch (expected 50 bytes, 28 bytes actually received)
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:42:04.859: reading buffer buf1
Action.c(40): fiCheckRecvMismatch: Sat Apr 29 13:42:04.859: calling parameterization routines
==========EXPECTED BUFFER==========
==========
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:42:04.859: Translate data for printing
Action.c(40): getAsciiReceivedBuffer: Sat Apr 29 13:42:04.859: Binary to ascii
==========RECEIVED BUFFER==========
"+OK POP server signing off\r\n"
==========

```
Action.c(40): callRecv:28 bytes were received
Action.c(46): lrs_get_received_buffer(socket0, buf_p, size_p)
Action.c(46): callGetReceivedBuffer: socket: socket0 (212), offset: 0, length: -1
Action.c(46): callGetReceivedBuffer: reading last received buffer
Action.c(46): callGetReceivedBuffer: translate buffer to ascii
Action.c(46): callSetUserBuffer: setting user buffer (socket: socket0)
Action.c(48): lrs_get_last_received_buffer_size(socket0)
Action.c(49): INFO: Loop 4 - Recievied Buffer: +OK POP server signing off
 (Size = 28)
Action.c(89): Notify: Next row for parameter SendMessages = 5 [table  = SendMessages].
Action.c(89): Notify: Getting new value for parameter 'SendMessages': table = 'SendMessages.dat' column = '0' row = '5'.
Action.c(90): Notify: Next row for parameter RecieveMessages = 5 [table  = RecieveMessages].
Action.c(90): Notify: Getting new value for parameter 'RecieveMessages': table = 'RecieveMessages.dat' column = '0' row = '5'.
Action.c(27): Notify: Parameter Substitution: parameter "SendMessages" =  "EOF"
Action.c(98): lrs_close_socket(socket0)
Action.c(98): callCloseSocket: socket: socket0 (212)
Ending action Action.
Ending iteration 1.
Ending Vuser...
Starting action vuser_end.
vuser_end.c(6): lrs_cleanup()
Ending action vuser_end.
Vuser Terminated.
```

String Manipulation With C

To capture an entire buffer, C code will be needed to search for what is needed and to save it as a LoadRunner parameter. This does the same thing as a web_reg_save_param() in the web/http recording protocol, but it is more of a "brute force" method. Each engagement and application will be different. Use the Function Reference Guide in LoadRunner. The following list of functions is a starting point.

Function Name	Description
strcat	Concatenates two strings.
strchr	Returns the pointer to the first occurrence of a character in a string.
strcmp	Compares two strings to determine the alphabetic order.
strcpy	Copies one string to another.
strdup	Duplicates a string.
stricmp	Performs a case-insensitive comparison of two strings.
strlen	Returns the length of a string.
strlwr	Converts a string to lower case.
strncat	Concatenates *n* characters from one string to another.
strncmp	Compares the first *n* characters of two strings.
strncpy	Copies the first n characters of one string to another.
strnicmp	Performs a case-insensitive comparison of *n* strings.
strrchr	Finds the last occurrence of a character in a string.
strset	Fills a string with a specific character.
strspn	Returns the length of the leading characters in a string that are contained in a specified string.
strstr	Returns the first occurrence of one string in another.
strtok	Returns a token from a string delimited by specified characters.
strupr	Converts a string to upper case.

Scripting Exercises

The following exercises can be used to learn how to work with Winsock effectively.

- **Telnet/POP3** - These protocols use a single session (or conversation) and are usually the best thing to start with when first learning how to script in Vugen.
    1. **Challenge #1** - Record a telnet session. Log on and type in a UNIX command line executable, such as"ls - l".
    2. **Challenge #2** - Enhance the script. Change the command typed in by paramaterizing it so that different UNIX command executables are sent to the console each time.
    3. **Challenge #3** - For POP3 applications, have the script check to see if there are any emails.
    4. **Challenge #4** - Add logic to the POP3 script, so that if there are no emails, disconnect. If emails are on the server, download them and then delete them from the POP3 server before disconnecting.

- **FTP** - This opens a minimum of 2 sessions (one for signal and the other for data). Some Windows-based FTP clients can open as many as 4 sessions. To ensure the right information gets sent to the server, keep track of all of the session conversations.
    1. **Challenge #5** - Record an FTP script where a single file is downloaded after doing a listing.
    2. **Challenge #6** - Enhance the script by adding logic to download a random file from the directory each time.

- **FlightSock Sample Application -** This is a Sockets version of the standard flight reservation sample application which ships with LoadRunner. The sockets are not encrypted
    1. **Challenge #7 –** Record a sample flight reservation. Parameterize the source city. Correlate for the destination cities.
    2. **Challenge #8 –** Correlate for both the source and destination cities, picking random cities for flights

- **Playing "GO" -** Go is an ancient strategy game played with black and white stones on boards of various dimensions. For a detailed description of GO turn to http://en.wikipedia.org/wiki/Go_%28game%29 . For this challenge you will need to install an open source GO server and clients to play the game, https://www.gnu.org/software/gnugo/ .
    1. **Challenge #9** – Develop two GO virtual users which will play against each other. This will require both programming logic as well as a synchronization mechanism to allow your virtual users to wait on each other to play before continuing. You may watch the game unfold with a third connection to the GO server, viewing the game being played by the virtual users.

In Closing…

Although the Winsock recording protocol in LoadRunner is not an overall difficult concept in theory, it is easy to get mired in the details of all the data being passed back and forth. The best way to learn is to practice on known Winsock applications and keep up C coding skills as it applies to string manipulation.

If this document was helpful, you would like to access additional resources, or you would like to connect and provide feedback use one of the following ways:

http://scottmoore.consulting

LinkedIn: https://www.linkedin.com/company/scott-moore-consulting-llc

Twitter: @loadtester

Google Plus: https://plus.google.com/u/0/+ScottMooreConsultingLLC/posts

YouTube Videos: https://www.youtube.com/channel/UCMndKhkWZp9EOWPrRI4V6bA