**1B4 5 lvl Exercise**

**OBJECTIVE:**

The purpose of this exercise is to familiarize students with a basic overview of a TCP connection from a programmatic and/or software viewpoint. (i.e. How a computer actually expects and handles incoming connections.) The entire class together should step through the whole exercise line by line and the instructor will explain what is happening, students may then answer questions at the bottom for the remainder of class time.

**INSTRUCTIONS:**

1. Open three PowerShell windows.

Start > PowerShell > Enter (Do not choose the ISE)

1. Change one title to be SERVER/LISTENER and the other to be CLIENT/SENDER.

[System.Console]::Title = "SERVER/LISTENER"

[System.Console]::Title = "CLIENT/SENDER"

1. Place the server one on the left and the client on the right, place the last window below the server side, they should be oriented like so.

|----------|----------|

|SERVER |CLIENT |

|----------| |

|POSH | |

|----------|----------|

1. This step is optional. Change background color of server side to have black background and leave the client to be the default blue. The color differences are only to help keep track of which side of the transaction we are working in. The unnamed POSH window can be any color. This window is only for examining the netstat output and viewing the connections to and from the machine.

Right Click Title Bar > Properties > Colors

1. **POSH** - Get a baseline netstat.

netstat -ano

1. **SERVER** - Create a TCP listener.

$Listener = (New-Object System.Net.Sockets.TCPListener -ArgumentList @("0.0.0.0",55555))

1. **POSH** - See that there is no listener in the output. (i.e. there is no line showing the TCP listener on 0.0.0.0:55555)

netstat -ano

1. **SERVER** - Start the listener.

$Listener.Start()

1. **POSH** - See that there is now a listener. (0.0.0.0:55555)

netstat -ano

1. **CLIENT** – Create a client connection and point it at the listener.

$Client = (New-Object System.Net.Sockets.TCPClient -ArgumentList @("127.0.0.1",55555))

1. **POSH** - See that the connections are there (the server side and client side) in addition to the listener. (i.e. There should now be a listener line for 0.0.0.0:55555 and a connection to the same port on the IP 127.0.0.1)

netstat -ano

1. **CLIENT** – Check the members of the connection variable. Note the ReceiveBufferSize property.

$Client | Get-Member

1. **CLIENT** - Save a reference to the stream to another variable.

$CliStream = $Client.GetStream()

1. **CLIENT** - Check the members of the stream variable.

$CliStream | Get-Member

1. **SERVER** - Check the members of the connection variable.

$Listener | Get-Member

1. **SERVER** – Verify that there is a connection pending receipt.

$Listener.Pending()

1. **SERVER** – Utilize the AcceptTCPClient() method, which returns a client connection reference, and save it to a variable.

$Server = $Listener.AcceptTCPClient()

1. **SERVER** - Check the members of the connection variable.

$Server | Get-Member

1. **SERVER** - Save the stream reference to another variable.

$SrvStream = $Server.GetStream()

1. **SERVER** - Stop the listener.

$Listener.Stop()

1. **POSH** - See that the listener is gone but the connections are still there.

netstat -ano

1. **SERVER/CLIENT** - View properties of stream objects.

$CliStream | Get-Member

$SrvStream | Get-Member

1. **SERVER/CLIENT** - Create buffer variables in each window.

$SrvBuff = (New-Object byte[] 1024)

$CliBuff = (New-Object byte[] 1024)

1. **CLIENT** - Set the buffer variable to test string bytes.

$CliBuff = [System.Text.Encoding]::UTF8.GetBytes("TEST STRING")

1. **CLIENT** - View the bytes.

$CliBuff

1. **CLIENT** - Write those bytes to the client stream.

$CliStream.Write($CliBuff,0,$CliBuff.Length)

1. **SERVER** - Check properties and note the DataAvailable property.

$SrvStream

1. **SERVER** - Read the full size of bytes into the server buffer.

$SrvStream.Read($SrvBuff,0,1024)

1. **SERVER** - View the bytes.

$SrvBuff

1. **SERVER** - Convert those bytes to a string.

[System.Text.Encoding]::UTF8.GetString($SrvBuff)

1. Perform the same last 8 steps and reverse the SERVER and CLIENT roles.
2. **SERVER** – Create a loop to check when data comes in and to print that data out to the console.

While($True){

$ReceivedData = ""

While($SrvStream.DataAvailable){

$SrvBuff = (New-Object byte[] 1024)

$SrvStream.Read($SrvBuff,0,1024)

$ReceivedData+=[System.Text.Encoding]::UTF8.GetString($SrvBuff)

Sleep -Milliseconds 100

}

If($ReceivedData){Write-Host $ReceivedData}

}

1. **CLIENT** - Create a function to send data that only takes a string as an argument.

Function SendData{

Param([String]$TempString)

$CliBuff = [System.Text.Encoding]::UTF8.GetBytes($TempString)

$CliStream.Write($CliBuff,0,$CliBuff.Length)

}

1. **CLIENT** - Send lots of test strings to the other end using the function.

SendData "TEST"

1. **SERVER** - View and verify the output, then ctrl+c to stop the loop.
2. **SERVER** - Change the loop behavior to send the data it receives back to the client, but with a space between each letter.

While($True){

$ReceivedData = ""

While($SrvStream.DataAvailable){

$SrvBuff = (New-Object byte[] 1024)

[Void]$SrvStream.Read($SrvBuff,0,1024)

$ReceivedData +=[System.Text.Encoding]::UTF8.GetString($SrvBuff)

Sleep -Milliseconds 100

}

If($ReceivedData){

$ReceivedData = ($ReceivedData.ToCharArray() -join " ")

$SrvBuff = [System.Text.Encoding]::UTF8.GetBytes($ReceivedData)

$SrvStream.Write($SrvBuff,0,$SrvBuff.Length)

}

}

1. **CLIENT** - Change the function to wait for a response after sending and to stop when data is no longer available.

Function SendData{

Param([String]$TempString)

$CliBuff = [System.Text.Encoding]::UTF8.GetBytes($TempString)

$CliStream.Write($CliBuff,0,$CliBuff.Length)

$ReceivedData = ""

While(!$ReceivedData){

While($CliStream.DataAvailable){

$CliBuff = (New-Object byte[] 1024)

[Void]$CliStream.Read($CliBuff,0,1024)

$ReceivedData+=[System.Text.Encoding]::UTF8.GetString($CliBuff)

Sleep -Milliseconds 100

}

}

If($ReceivedData){Write-Host $ReceivedData}

}

1. **CLIENT** - Send lots of test strings to the other end using the function and view the responses.

SendData "TEST"

1. **SERVER/CLIENT** - Close the connections

$CliStream.Close()

$Client.Close()

$Client.Dispose()

$SrvStream.Close()

$Server.Close()

$Server.Dispose()

**QUESTIONS:**

* What other ways can we use/abuse this functionality?
  + Could we connect to a legitimate running service and pull data out of it?
    - Things to consider:
      * Does this service or program on this port use encryption? Is it plaintext?
      * What kind of data would be in here?
      * How would the data be formatted?
      * Is the connection TCP?
  + Would we be able to serve data to others?
  + Could we use this to cheat in video games or other online services?
* HTTP or Hyper Text Transfer Protocol utilizes a giant text stream and does not use encryption. Knowing this, how would we make a fake webpage? How would people be able to connect to it if we didn’t have it running on the traditional port 80 or 443?

**BONUS:**

* Create a PowerShell loop that will serve up a webpage to whomever connects to it and read the data coming from the people who connect and display it to the console.
  + Are people on your local subnet (i.e. anyone in the class around you) able to connect?
  + What kind of information do people send when they connect to a webpage?

**1B4 5 lvl Exercise (Instructor/Answers)**

**OBJECTIVE:**

The purpose of this exercise is to familiarize students with a basic overview of a TCP connection from a programmatic and/or software viewpoint. (i.e. How a computer actually expects and handles incoming connections.) The entire class together should step through the whole exercise line by line and the instructor will explain what is happening, students may then answer questions at the bottom for the remainder of class time.

**INSTRUCTIONS:**

Open three PowerShell windows.

Start > PowerShell > Enter (Do not choose the ISE)

Change one title to be SERVER/LISTENER and the other to be CLIENT/SENDER. This is just how we can access properties of the host console itself.

[System.Console]::Title = "SERVER/LISTENER"

[System.Console]::Title = "CLIENT/SENDER"

Place the server one on the left and the client on the right, place the last window below the server side, they should be oriented like so.

|----------|----------|

|SERVER |CLIENT |

|----------| |

|POSH | |

|----------|----------|

This step is optional. Change background color of server side to have black background and leave the client to be the default blue. The color differences are only to help keep track of which side of the transaction we are working in. The unnamed POSH window can be any color. This window is only for examining the netstat output and viewing the connections to and from the machine.

Right Click Title Bar > Properties > Colors

**POSH** - Get a baseline netstat. Just note that there isn’t anything listening on port 55555 and there are no connections using that port either.

netstat -ano

**SERVER** - Create a TCP listener. Here we are creating an object of the listener type and setting the arguments to listen on 0.0.0.0:55555 (i.e. respond to all packets received on that port, not just ones addressed to the host machine.) We chose this port because it falls in the “Random High Range” which is technically anything above 1024, but Windows locks down certain ports above that range. This way even without admin credentials, we can start listening for incoming connections. Technically, we could have created this object with ‘$Listener = [System.Net.Sockets.TCPListener]::New(“0.0.0.0”,55555)’, however, the method shown is backwards compatible all the way to PowerShell Version 1.

$Listener = (New-Object System.Net.Sockets.TCPListener -ArgumentList @("0.0.0.0",55555))

**POSH** - See that there is no listener in the output. (i.e. there is no line showing the TCP listener on 0.0.0.0:55555) Explain how this is because we haven’t actually STARTED listening. We have created the object, but we need to actually begin handling incoming connections. It’s broken out like this because in certain applications we may be doing a lot of things asynchronously and we may need to make sure we are good to go before accepting new incoming connections.

netstat -ano

**SERVER** - Start the listener. This method could be found beforehand with ‘$Listener | Get-Member’.

$Listener.Start()

**POSH** - See that there is now a listener. (0.0.0.0:55555)

netstat -ano

**CLIENT** – Create a client connection and point it at the listener. Unlike the listener, the TCPClient object immediately tries to establish the connection. If we were to view this in WireShark, the three way TCP handshake would occur upon execution of this command.

$Client = (New-Object System.Net.Sockets.TCPClient -ArgumentList @("127.0.0.1",55555))

**POSH** - See that the connections are there (the server side and client side) in addition to the listener. (i.e. There should now be a listener line for 0.0.0.0:55555 and a connection to the same port on the IP 127.0.0.1) You may have to scroll down a bit to see this, but there should be both the listener ‘0.0.0.0:55555’ under ‘Local Address’ with the state ‘LISTENING’ and two connections to ‘127.0.0.1:55555’ under both ‘Local Address’ and ‘Foreign Address’ with the state ‘ESTABLISHED’. If you so choose, you can run ‘ps’ and verify that the PID number matches to one of the ‘powershell.exe’ processes in the process list for both the listener and the connection.

netstat -ano

**CLIENT** – Check the members of the connection variable. Note the ReceiveBufferSize property. The ‘$Obj | Get-Member’ is a common and simple way to view properties (attributes of the object) and methods (actions that object can perform). We want to note the receive buffer size because that is the maximum size we can send at a time in later steps. We checked the members here because we want to send data (The whole point of a network connection, right?) and we should note in here that there is no ‘send’ or ‘write’ method (action) we can use. This is because the $Client variable represents the actions we can take and the properties we can view on the CONNECTION not the DATA STREAM. To interact with the data stream we need a reference to that. That’s where we can see the ‘GetStream()’ method which returns a stream reference.

$Client | Get-Member

**CLIENT** - Save a reference to the stream to another variable.

$CliStream = $Client.GetStream()

**CLIENT** - Save a reference to the stream to another variable.

$CliStream = $Client.GetStream()

**SERVER** - Check the members of the connection variable. Here we would note the object is not the same type as the client. It’s a listener and we need a reference to the client connection that just connected to it. Note the ‘Pending()’ method. It returns a boolean (true/false) value regarding whether or not there is a connection awaiting handling. We should also note a method ‘AcceptTCPClient()’ that returns a ‘TCPClient’ object. That’s probably what we want in order to interact with the connection and the data stream for that connection.

$Listener | Get-Member

**SERVER** – Verify that there is a connection pending receipt.

$Listener.Pending()

**SERVER** – Utilize the AcceptTCPClient() method, which returns a client connection reference, and save it to a variable.

$Server = $Listener.AcceptTCPClient()

**SERVER** - Check the members of the connection variable. Here we see that this is the same object we referenced on the client side.

$Server | Get-Member

**SERVER** - Save the stream reference to another variable.

$SrvStream = $Server.GetStream()

**SERVER** - Stop the listener.

$Listener.Stop()

**POSH** - See that the listener is gone but the connections are still there. This is really just to help drive home the point that the listener and the connection are not the same thing.

netstat -ano

**SERVER/CLIENT** - View properties of stream objects. Again, we can see these are indeed the same type of object and we can see that they have the ‘write’ and ‘read’ methods we want. If we look at the definition of the method for ‘Write()’ and ‘Read()‘, we can see we will need a buffer and some integers for offset and size. Fortunately, these integers are easy to find or don’t need to be known. The buffers will have to be created though. They will contain data to be sent and they will fill with data that is received.

$CliStream | Get-Member

$SrvStream | Get-Member

**SERVER/CLIENT** - Create buffer variables in each window.

$SrvBuff = (New-Object byte[] 1024)

$CliBuff = (New-Object byte[] 1024)

**CLIENT** - Set the buffer variable to test string bytes. We could have skipped the previous ‘$CliBuff’ line because this line will overwrite the buffer completely with bytes created from the ‘TEST STRING’, but it was included for completeness when repeating these steps with the roles reversed. Also, note that the encoding type chosen was UTF8. Any encoding could have been used, but UTF8 is fairly universal.

$CliBuff = [System.Text.Encoding]::UTF8.GetBytes("TEST STRING")

**CLIENT** - View the bytes.

$CliBuff

**CLIENT** - Write those bytes to the client stream. This method will write the bytes from our variable ‘$CliBuff’, starting at the offset 0 (the beginning) and it will write all bytes contained in the array. The ‘$CliBuff.Length’ property serves us well here, because we do not need to know the length of the byte array. The computer will calculate it for us and pass that for the size.

$CliStream.Write($CliBuff,0,$CliBuff.Length)

**SERVER** - Check properties and note the DataAvailable property. It should be ‘True’ signifying there is data in the receive buffer. We noted the size earlier because the computer will only store 65KB of unhandled data.

$SrvStream

**SERVER** - Read the full size of bytes into the server buffer. In a real application we won’t know how many bytes we were just sent beforehand or how many we should read until we know what we are looking at so what we can do instead is read the full size of the buffer and the ‘Read()’ method will simply place zeros where there is no more data. This is why we created large empty byte arrays before, so we had a place to capture data to for later reading. 1024 would equate to 1024 characters or 256 emojis if we were using UTF32. The command below will return the number of bytes read from the buffer, starting at 0 and continuing until 1024 (the size of the byte array).

$SrvStream.Read($SrvBuff,0,1024)

**SERVER** - View the bytes.

$SrvBuff

**SERVER** - Convert those bytes to a string. This is the reverse of the conversion of a string to bytes command from earlier.

[System.Text.Encoding]::UTF8.GetString($SrvBuff)

Perform the same last 8 steps and reverse the SERVER and CLIENT roles. Let the students do this part on their own (10-15mins).

**SERVER** – Create a loop to check when data comes in and to print that data out to the console.

While($True){

Run this loop forever.

$ReceivedData = ""

Create and/or empty the ‘$ReceivedData’ string variable

While($SrvStream.DataAvailable){

While there is data available to read (i.e. Data in the receive buffer) keep looping.

$SrvBuff = (New-Object byte[] 1024)

Create and/or empty the ‘$SrvBuff’ byte array

$SrvStream.Read($SrvBuff,0,1024)

Read 1024 bytes out of the stream onto the buffer. Remember that if there are not enough bytes that the method will just put zeros. If there is still data, even after the 1024 are read, the loop will start over and pull the rest out. (The offset does not need to change because when bytes are read out of the stream the remaining bytes moves down towards 0, so 0 will always be the start of any data that is still in the stream.)

$ReceivedData+=[System.Text.Encoding]::UTF8.GetString($SrvBuff)

Append the 1024 bytes that were read in to the end of the ‘$ReceivedData’ variable. (We do this so that if the data in the stream is longer that 1024 bytes, the output will still be correct.)

Sleep -Milliseconds 100

This is to allow some time for more bytes to get into the stream from the remote end before we check to see if there is more data. An exceptionally slow connection may accidentally leave bytes off the end and sending extremely large amounts of data (>65KB) on a very fast connection could cause bytes to get dropped as the buffer would fill faster than we could empty it. Modern software application will handle this appropriately, but we aren’t expecting excessively large/fast traffic nor are we expecting exceptionally slow traffic.

}

If($ReceivedData){Write-Host $ReceivedData}

Check if there is data to print and print it if there is. (PowerShell interprets a non-empty string as ‘True’ and validates the ‘If()’ statement.)

}

**CLIENT** - Create a function to send data that only takes a string as an argument. Functions in PowerShell are ways to alias a short or oft repeated set of commands to a simple name and arguments.

Function SendData{

Create a function called ‘SendData’.

Param([String]$TempString)

Take only 1 string input and save it to ‘$TempString’

$CliBuff = [System.Text.Encoding]::UTF8.GetBytes($TempString)

Convert the input to bytes.

$CliStream.Write($CliBuff,0,$CliBuff.Length)

Write the bytes to the ‘$CliStream’ starting at 0 and continuing for the length of the byte array. (Note that this function assumes that ‘$CliStream’ exists already, so if we haven’t saved a ‘NetworkStream’ object to a variable ‘$CliStream’, then this will fail.)

}

**CLIENT** - Send lots of test strings to the other end using the function. We simply call the function by name and include the arguments, since there’s only one we just put a string on the end.

SendData "TEST"

**SERVER** - View and verify the output, then ctrl+c to stop the loop.

**SERVER** - Change the loop behavior to send the data it receives back to the client, but with a space between each letter.

While($True){

$ReceivedData = ""

While($SrvStream.DataAvailable){

$SrvBuff = (New-Object byte[] 1024)

[Void]$SrvStream.Read($SrvBuff,0,1024)

$ReceivedData +=[System.Text.Encoding]::UTF8.GetString($SrvBuff)

Sleep -Milliseconds 100

}

If($ReceivedData){

The inside of the ‘If()’ statement is the only different portion from the last loop. Instead of printing out the data, we create new data with spaces in it and send it back on the same stream it came in on. This is why we have the students do those 8 reversed steps, so that they can see that the ‘NetworkStream’ object allow data flow in both directions.

$ReceivedData = ($ReceivedData.ToCharArray() -join " ")

Take the string we created from the bytes we received and convert it to a character array. PowerShell has a handy ‘-join’ operator that will join string or character arrays with strings in between them. Our prompt said to return them with spaces, so we join the character array with spaces and save it to the same variable we originally used. At this point the data is the same, just expanded and saved to the same variable.

$SrvBuff = [System.Text.Encoding]::UTF8.GetBytes($ReceivedData)

Get the bytes of the string.

$SrvStream.Write($SrvBuff,0,$SrvBuff.Length)

Write them to the stream.

}

}

**CLIENT** - Change the function to wait for a response after sending and to stop when data is no longer available. This part is the same as the first loop for the server, just with the names switched to ‘Cli’ and the logic to run while ‘$ReceivedData‘ is empty has been changed from what was originally just ‘While($True)’.

Function SendData{

Param([String]$TempString)

$CliBuff = [System.Text.Encoding]::UTF8.GetBytes($TempString)

$CliStream.Write($CliBuff,0,$CliBuff.Length)

$ReceivedData = ""

While(!$ReceivedData){

The ! operator means not and recall that empty strings are ‘False’, we are saying run while ‘$ReceivedData’ is empty. (i.e. Not ‘False’)

While($CliStream.DataAvailable){

$CliBuff = (New-Object byte[] 1024)

[Void]$CliStream.Read($CliBuff,0,1024)

$ReceivedData+=[System.Text.Encoding]::UTF8.GetString($CliBuff)

Sleep -Milliseconds 100

}

}

If($ReceivedData){Write-Host $ReceivedData}

}

**CLIENT** - Send lots of test strings to the other end using the function and view the responses.

SendData "TEST"

**SERVER/CLIENT** - Close the connections. This step is not strictly necessary, but it’s good practice to clean up connections. In the ‘netstat –ano’ output it may be up to a minute before the connections completely go away.

$CliStream.Close()

$Client.Close()

$Client.Dispose()

$SrvStream.Close()

$Server.Close()

$Server.Dispose()

**QUESTIONS:**

* What other ways can we use/abuse this functionality? Games, online services, ssh listeners, could mess with nc and pull info from unix systems, really any answer is acceptable as long as they name any TCP service.
  + Could we connect to a legitimate running service and pull data out of it? Absolutely, there is no requirement, other than just the data is in plain text and even it if WAS encrypted, if we had the keys we could then decrypt the data. Exfiltration via this method is very plausible especially since ‘FileStream’ and ‘NetworkStream’ are almost the same type of object. You could use the ‘CopyTo()’ method from a file stream to copy bytes from a file directly into a network stream without even saving them somewhere intermittently. File streams can be opened with ‘$FileObj = (New-Object System.IO.FileStream -ArgumentList @(“C:\FULL\_PATH\_TO\_FILE”,[System.IO.FileMode]::Open))’
    - Things to consider:
      * Does this service or program on this port use encryption? Is it plaintext?
      * What kind of data would be in here?
      * How would the data be formatted?
      * Is the connection TCP?
  + Would we be able to serve data to others? Yes, the point of the reversed instructions and the custom loop adding spaces and returning data is meant to highlight the bi-directionality.
  + Could we use this to cheat in video games or other online services? Yes, many people do this. Not with PowerShell, but they do it. It may actually be beneficial to do it with PowerShell because anti-cheat software will not be able to block it as it is a legitimate built-in Windows utility.
* HTTP or Hyper Text Transfer Protocol utilizes a giant text stream and does not use encryption.
  + Knowing this, how would we make a fake webpage? Yes, the key here is to respond with the correct kind of HTTP header. An HTTP response from a server has a plaintext header that contains the server response code and the response length as well as the text encoding (it’s almost always UTF8). Then there is a blank line separating the web page body (the actual HTML) from the header. The browser like FireFox of IE will know what to do with this and display the page accordingly.

It should be something like:

HTTP/1.1 200 OK

Content-Length: 123

Connection: close

Content-Type: text/html; charset=utf-8

<link rel="icon" href="data:,">

<html>

<body>…

* + How would people be able to connect to it if we didn’t have it running on the traditional port 80 or 443? In the URL bar of a browser the ‘HTTP://’ actually specifies the protocol to use. You are telling the browser how to interpret the data it is going to get. It doesn’t have to be just HTTP, it could be ‘FTP://’ or ‘FILE://’ as you may have seen. As for the port, it is implied with the protocol, but it could be anything. When you connect to ‘HTTPS://www.google.com/’, what you are actually doing is telling the browser to utilize the HTTPS protocol and make a TCP connection to Google’s ip on the port 443 (the implied port for that protocol). So the actual connection looks like ‘HTTPS://<Google’s ip>:443/’. The 443 could be anything and if you want it to be something different, you just specify like so ‘HTTPS://www.google.com:8888/’. For our purposes since we want to do this in plain text, we would just do in the URL bar ‘HTTP://<ip>:55555’ for the unencrypted traffic (HTTP) and the custom port number 55555.

**BONUS:**

* Create a PowerShell loop that will serve up a webpage to whomever connects to it and read the data coming from the people who connect and display it to the console.

$Body = @'

<link rel="icon" href="data:,">

<html>

<body>

Hey guys, whatup!

</body>

</html>

'@

$Header = 'HTTP/1.1 200 OK

Content-Length: '+$Body.Length+'

Connection: close

Content-Type: text/html; charset=utf-8

'

$SrvResp = [System.Text.Encoding]::UTF8.GetBytes(($Header+$Body))

$Listener = (New-Object System.Net.Sockets.TCPListener -ArgumentList @("0.0.0.0",55555))

$Listener.Start()

While($True){

Write-Host "Listening for connection..." -ForeGround Yellow -BackGround Black

$Connection = $Listener.AcceptTcpClient()

Write-Host "Got connection!" -ForeGround Yellow -BackGround Black

$Stream = $Connection.GetStream()

$CliReqStr = ""

While($Stream.DataAvailable){

Write-Host "Reading in 1024 bytes!" -ForeGround Yellow -BackGround Black

$CliReq = (New-Object Byte[] 1024)

[Void]$Stream.Read($CliReq,0,1024)

$CliReqStr+=[System.Text.Encoding]::UTF8.GetString($CliReq)

Sleep -Milliseconds 100

}

Write-Host "Received bytes converted to a string:" -ForeGround Yellow -BackGround Black

Write-Host $CliReqStr

Write-Host "Sending webpage..." -ForeGround Yellow -BackGround Black

$Stream.Write($SrvResp,0,$SrvResp.Length)

Write-Host "Done!" -ForeGround Yellow -BackGround Black

}

The received bytes may occasionally be empty, just have them refresh the page a few times.

* + Are people on your local subnet (i.e. anyone in the class around you) able to connect? They should be, but it’s hard to predict how VLANs will be broken up. If two students are on the same subnet, they SHOULD be able to.
  + What kind of information do people send when they connect to a webpage? A GET request, complete with UA string, describing the bitness of the computer, the browser they are using, and even the OS!