SCRIPT: SSH Tunnels, Port Testing and X11/X-Windows VTS

|  |  |
| --- | --- |
| **Action on Screen** | **Narration** |
| Local Port Forwarding | |
| # Server  iptables -F -t filter  iptables -F -t nat  iptables -t filter -L -n  iptables -t nat -L -n | Dash F stands for flush and dash T specifies the table to act on. We’ll flush out the filter and nat tables and then print out the tables with dash L to make sure there are no firewall rules loaded. |
| # Client1  elinks --dump server.example.com  # Client2  elinks --dump server.example.com | We have apache loaded on **server** running a couple of very simple websites. Let’s use a program called **elinks** to test one of the website from our clients, **client1** and **client2** to make sure it works. Both clients are able to retrieve the website running on **server**. |
| # Server  iptables -A INPUT -p tcp --dport 80 -s client2 -j REJECT --reject-with icmp-net-prohibited  iptables -t filter -L -n | Now, let’s use the firewall on **server** to block traffic from **client2** on port 80. |
| # Client1  elinks --dump server.example.com  # Client2  elinks --dump server.example.com | Confirm that the website on **server** is still reachable from **client1** but is unreachable from **client2**. Typically, a router firewall will not respond with an ICMP message like we see here. In most cases, the firewall will drop the packets and you will never get a response but I’ve set this firewall to respond so we know the connection is being blocked. |
| # Client2  ssh -N -f -L 80:localhost:80 server | **client2** is unable to connect directly to port 80 on **server** but it still has access to port 22 so we can establish an SSH connection between the two servers and tunnel a connection from localhost on **client2** to localhost on **server**. |
| # Client2  lsof -nP -i 4tcp | grep -G "^ssh\s" | Using the lsof command, we can see that our SSH connection is not only established between the two servers, but it is also listening on localhost port 80. |
| # Client2  elinks --dump 127.0.0.1 | Now, instead of connecting to server on port 80 with elinks, we connect to our localhost forwarded port and we can see that the connection is successful. |
| # Server  iptables -A INPUT -p tcp --dport 22 -s client2 -j REJECT --reject-with icmp-net-prohibited  iptables -L -n | But, what if **server** is not only blocking port 80 for **client2**, it is also blocking port 22. Which means, we can’t establish the SSH connection, making the tunnel creation impossible. |
| # Client2  lsof -nP -i 4tcp | grep -G "^ssh\s" | awk "{print \$2}" | uniq | xargs kill | Let’s go back to **client2**. First we’ll kill the the SSH connection we just created. |
| # Client2  ssh –N –f –L 80:localhost:80 server | Now we’ll try to recreate that connection. We can see that the SSH connection cannot be made because of the firewall rule we added on **server**. |
| # Client2  elinks –dump server.example.com | And we know that we still cannot connect directly to the website. |
| # Client2  ssh -N -f -L 80:server:80 client1 | However, we know that **client1** does have access to not only port 22, but port 80 as well. Knowing this, we can use the same tunneling technology to get to port 80 on **server** from **client2** by running that tunnel through **client1**. |
| # Client2  elinks --dump 127.0.0.1 | The connection looks the same from **client2**’s perspective and we connect to it the same way as we did previously. |
| # Client2  ssh -N -f -L 12345:server:80 client1 | Up until now we’ve been connecting the tunnel to localhost on port 80, but what if there was a webserver or an already forwarded connection running on that IP and port, giving you an error when you try to bind to that port on that address? In this case, we could simply bind to a different port. The port on the local side does not have to be the same as that on the remote side. |
| # Client2  elinks --dump 127.0.0.1:12345 | The only difference now is that our tunnel starts on port 12345 instead of port 80 so we need to specify that port when making a connection to the website. |
| Remote Port Forwarding | |
| # Server  iptables -F  iptables -L -n | We’ll create another demonstration to show how remote tunnels work, but first we’ll clear out the firewall rules currently in place on **server**. |
| # Client1  cat /etc/hosts | Let’s say I have monitoring software running on **client1** that needs to establish a connection to an agent on **server**. The monitoring software uses DNS to resolve the hostname of **server** which we have mimicked using the “hosts” file on **client1**. |
| # Client1  elinks --dump server.example.com:7801 | The agent that the monitoring software communicates with is depicted here as another website running on **server**. We can test this by using elinks to view the website created for the agent connection. |
| # Server  iptables -A INPUT -p tcp --dport 7801 -s client1 -j REJECT --reject-with icmp-net-prohibited  iptables -A INPUT -p tcp --dport 22 -s client1 -j REJECT --reject-with icmp-net-prohibited  iptables -L -n | Now, we find out that the monitoring software on **client1** can no longer communicate with the agent on **server** because port 7801 is being blocked by the firewall. Let’s set up a firewall rule to block **client1** from reaching **server** on ports 7801 and 22. |
| # Client1  elinks --dump server.example.com:7801 | We can see on **client1** that the port is now blocked and the monitoring software can no longer reach the agent running on **server**. |
| # Client1  ssh -N -f -L 7801:localhost:7801 server | From the demonstration on local port forwarding, we know we can simply tunnel port 7801 locally through an SSH connection to **server**. However, port 22 is being blocked. |
| # Client1  ssh -N -f -L 7801:server:7801 client2 | No problem, we can simply tunnel the connection through another server we know has access to port 7801 on **server**. Unfortunately, that doesn’t work either because port 22 from **client1** is being blocked on **client2** as well. |
| # Client2  ssh -N -f -R 7801:server:7801 client1 | This is a good example of when remote tunnels are our only option. We need to get the traffic from **client1** to **server** but we can’t do it locally. So, we’ll use **client2** to establish a remote tunnel from port 7801 on **client1** to port 7801 on **server**. |
| # Client2  lsof -nP -i 4tcp | grep -G "^ssh\s" | If we try to look at this tunnel from **client2**, we don’t see anything because there is nothing being forwarded from this server. |
| # Client1  lsof -nP -i 4tcp | grep -- "->192.168.78.63\|:7801" | However, we can view this tunnel from **client1** using the lsof command and can see that there is a socket open on 127.0.0.1, port 7801. |
| # Client1  # Change the /etc/hosts file on **client1** so that the entry for **server** now points to 127.0.0.1  cat /etc/hosts | If we don’t have the ability to change the IP in the application communicating with the agent on **server**, we can override the DNS in most cases by using the hosts file. |
| # Client1  elinks --dump server.example.com:7801 | Now we can once again connect, this time through a remote tunnel. |
| Port Testing | |
| # Client1  elinks --dump server.example.com:7901 | To demonstrate this, we can simply try to connect to a port on a server that does not have anything listening on it with elinks. |
| # Client1  telnet server.example.com 7901  curl server.example.com:7901 | There are a number of other tools that can be used to test the port as well, including “telnet” and “curl”. |
| # Server  iptables -A INPUT -p tcp --dport 7901 -s client1 -j REJECT | Now let’s put a firewall rule in place that will return the exact same ICMP message I just received when trying to connect to a port without something listening on it. |
| # Client1  elinks --dump server.example.com:7901  telnet server.example.com 7901  curl server.example.com:7901 | If we run the same commands to test the connection, we get the exact same error messages. In this case, there is no way to differentiate between an empty port and a firewall. So we need to have something listening on the receiving end to accurately test the port communication. |
| # Server  iptables -F | Remove the firewall rule on **server**. |
| # Server  import BaseHTTPServer  HOST\_NAME = 'server.example.com'  PORT\_NUMBER = 7901  class MyHandler(BaseHTTPServer.BaseHTTPRequestHandler):  def do\_GET(s):  s.wfile.write(" Test Python Site is Working\n")  if \_\_name\_\_ == '\_\_main\_\_':  server\_class = BaseHTTPServer.HTTPServer  httpd = server\_class((HOST\_NAME, PORT\_NUMBER), MyHandler)  try:  httpd.serve\_forever()  except KeyboardInterrupt:  pass  httpd.server\_close() | The first way was to use a Python script to do this on the Linux and Unix servers. Every server we tested had Python installed on it and almost every server had the necessary Python modules installed as well. Creating this lightweight web server is as simple as logging into the Python command line and running these commands. |
| # Client1  elinks --dump server.example.com:7901 | We now have a web server running on the target server. We can now accurately test if the port is accessible. |
| # Windows  $listener = New-Object System.Net.HttpListener  $listener.Prefixes.Add('http://+:7901/')  $listener.Start()  'Listening ...'  while ($true) {  $context = $listener.GetContext()  $request = $context.Request  $response = $context.Response    if ($request.Url -match '/$') {  $response.ContentType = 'text/plain'  $message = " Test PowerShell Site is Working`n"  }  # This will terminate the script. Remove from production!  # if ($request.Url -match '/end$') { break }  [byte[]] $buffer = [System.Text.Encoding]::UTF8.GetBytes($message)  $response.ContentLength64 = $buffer.length  $output = $response.OutputStream  $output.Write($buffer, 0, $buffer.length)  $output.Close()  break  }  $listener.Stop() | PowerShell can be used to create a very simple web server as well with this script. There is no need to run it as a script. Simply open a PowerShell prompt with administrator privileges and paste in the code. |
| # Client1  elinks --dump windows.example.com:7901 | We can run the same elinks command against the Windows server to test the connection. |
| # Server  iptables -A INPUT -p tcp --dport 7901 -s client1 -j REJECT | Now we can add the same firewall we used previously to test a blocked connection. |
| # Client1  elinks --dump server.example.com:7901 | And run the test again to see that it is being blocked and we receive the “Connection refused” message. |
| # Server  iptables -F  iptables -A INPUT -p tcp --dport 7901 -s client1 -j DROP | Often times enterprise firewalls will drop packets instead of replying with ICMP messages. |
| # Client1  elinks --dump server.example.com:7901 | In this case, you won’t receive a reply. The connection will just hang until it times out. |
| X11/X-Windows | |
| # Client1  # Display client1 desktop | This desktop interface is Gnome running on Oracle Linux. The underlying desktop manager can display visual applications such as software installers. |
| # Client1  xeyes | A simple graphical application that I use often to test X is “xeyes”. |
| # Client1  echo $DISPLAY | By default, the application will display on the desktop I am running it from. The default setting for the DISPLAY variable is colon zero dot zero. This means, first desktop of the first display on this server. By default, access to the desktop is only allowed from a local socket, there is no network access. |
| # Client1  xhost | Access control to the desktop environment is controlled with the “xhost” command. Executing it alone shows the currently set allowed access to the display defined by the DISPLAY environment variable. This setting means that only the root user is allowed to connect and must be local to the desktop environment. |
| # Client1  xeyes | To demonstrate the client-server architecture we can start the “xeyes” application from an SSH session instead of inside the desktop to demonstrate that as long as I have access to the local socket and I have my DISPLAY variable set properly, I can display a graphical application. |
| # Client1  ps -ef | grep 'Xorg' | grep -v grep | The output of this “ps” command shows the “Xorg” process running on “:0”. |
| # Client1  ls -l /tmp/.X0-lock | We can also see that there is a lock file in the temp directory indicating that Xorg is currently running on the first socket. |
| # Client2  export DISPLAY=client1.example.com:0.0  xeyes | In a secure environment, xhost can be used to allow additional access to the desktop environment. This method should be used with caution and never used across the internet since it is easily exploitable. If we try to display applications from any other servers on **client1**’s desktop, we’ll get an error. |
| # Client1  xhost +client2.example.com | Let’s add network access for **client2** using the “xhost” command. |
| # Client2  echo $DISPLAY  xeyes | Now we can display applications from **client2** to the desktop on **client1**. |
| # Client1  vncserver | We can use a different type of desktop environment if a full desktop environment like Gnome is not available. We can start an instance of vncserver. |
| # Client1  ps -ef | grep 'Xorg\|Xvnc' | grep -v grep | We can see in this case that vncserver chose the next available X11 port on the server. |
| # Client1  # Open vnc session  echo $DISPLAY | We can also see that the DISPLAY variable is set differently in this desktop environment. |
| # Client1  export DISPLAY=:1.0 | We can just as easily start applications on this desktop as we did in the Gnome desktop. First, we need to change the DISPLAY variable because “xhost” executes commands against the server identified by this variable. |
| # Client1  xhost +client2.example.com | Now we can add access to this desktop with “xhost” just like we did before. |
| # Client2  export DISPLAY=client1.example.com:1.0  xeyes | We need to change the DISPLAY variable on the remote server to match the new desktop and we have success. |
| # Client1  ssh -X client2 | A more common and secure method is to create a tunnel and forward X11 traffic through an SSH connection. Fortunately, most SSH clients have this functionality built-in. In my terminal I can forward X11 traffic by specifying dash capital x. |
| # Client2  echo $DISPLAY | Echo the DISPLAY variable to see the traffic is being directed to localhost colon ten or eleven. |
| # Client2  lsof -nP -i 4tcp@localhost:x11-ssh-offset | Looking at the output of the “lsof” command, we can see that location of the DISPLAY variable is being forwarded through an SSH tunnel, specifically a remote tunnel. |
| # Client2  xauth list | The magic cookies are stored in the X authority file in the user’s home directory and can be listed with the “xauth” command. These cookies belong to the root user only. |
| # Client2  su - oracle  xeyes | The other users are not aware of these cookies. Without the cookie associated with the connection, the X11 tunnel cannot be used. |
| # Client2  xauth list | If we list “oracle’s” cookies, we won’t see any of “root’s” cookies. |
| # Client2  xauth list  su - oracle  xauth add ...  xauth list  xeyes | A simple workaround for this so we don’t have to reconnect directly to “oracle” is to add the cookie generated for the SSH connection to “oracle’s” authority list. Now we can see this cookie in “oracle’s” authority list and successfully launch “xeyes”. |
| # Windows  # show putty X11 connection | This stuff is pretty cool but many of us run a Windows OS on our PC’s. Fortunately, most Windows SSH clients also have this functionality built-in. In Putty, it is a simple check box. |
| # Windows  # show Cygwin X Server | In order to forward X applications to our local desktop, we need to be running some form of X Windows desktop manager. My PC runs Cygwin’s X Server and this particular server that is running is listening on zero colon zero. |
| # Windows  # start new putty session to client1  echo $DISPLAY | Let’s start the session and look at the DISPLAY variable. It looks like we are forwarding the X applications through localhost display 10. |
| # Client1  lsof -nP -i 4tcp@localhost:x11-ssh-offset | The “lsof” command output tells us that “sshd” is actually using port 6010 to forward the X11 traffic. |
| # Client1  xauth list | The “xauth” command tells us that a new magic cookie was generated for authorization. |
| # Client1  xeyes | Now, we can try to start an X application on the remote server that will be forwarded through the SSH connection and run on my local desktop. |