btrisk

CYBER SECURITY AND IT GOVERNANCE SERVICES

Capturing MSSQL Credentials from an Executable

"With Dynamic Analysis"

ISMAIL ONDER KAYA [OSCP]

10 / 20 / 2020

Contents

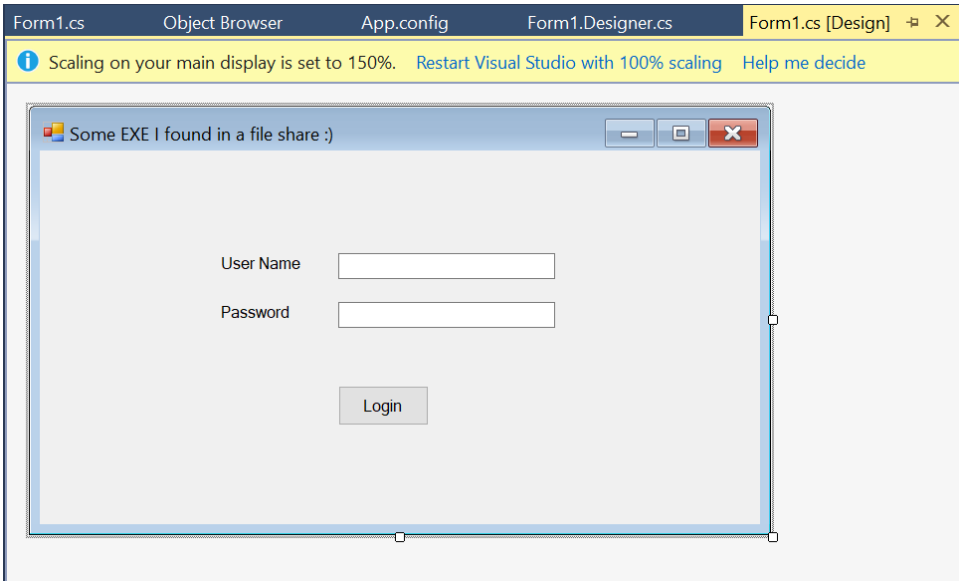
How to capture MSSQL credentials dynamically?

Suppose during a penetration testing we found a file share that we could access. In that share we discovered an enterprise database application executable (a .Net forms application to be precise) and its corresponding libraries and config files. Is this a possibility in a real penetration testing engagement? Yes, in 1 out of 15 – 20 penetration tests you may encounter that situation, desktop apps are not dead yet. What we would probably be doing would be in the following order:

- Looking through the config files if we could find any database connection string in the open.
- Retrieving the strings within the executable files to find a connection string or any other interesting information.
- Decompiling / disassembling the executables and using our reversing skills to recover any information hidden from plain sight (perhaps by encoding / encrypting the connection strings and other interesting information)

However, there is another way to reap the benefits of our discovery: Making the executable send us the MSSQL username and password to connect to the database server (which would be us in this case). It is much easier than you might thing with today's tools and resources.

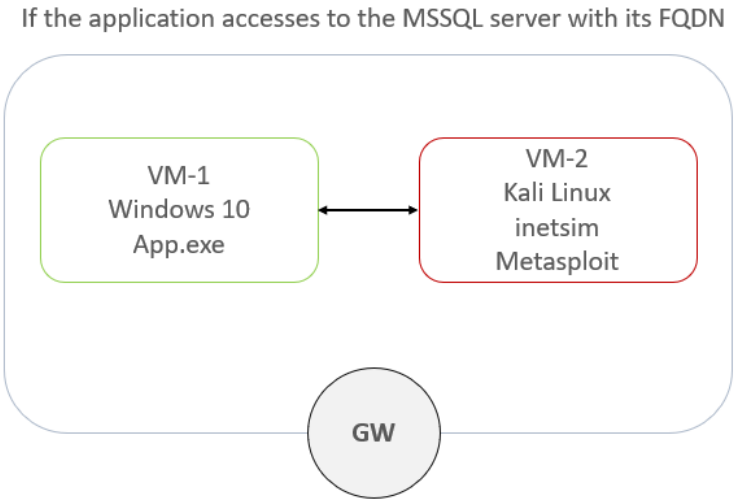
Below is a very tiny forms application which connects to a database server once the user hits the "Login" button.



```
14    public partial class loginForm : Form
15    {
16        public loginForm()
17        {
18            InitializeComponent();
19
20        }
21        private void loginForm_Load(object sender, EventArgs e)
22        {
23        }
24
25        private void BTN_login_Click(object sender, EventArgs e)
26        {
27            try
28            {
29                SqlConnection conn =
30                    new SqlConnection("server=sqlserver.btrisk.com;user=sa;pwd=123456;database=MyDB;");
31                conn.Open();
32            } catch(Exception ex) {}
33        }
34    }
35 }
```

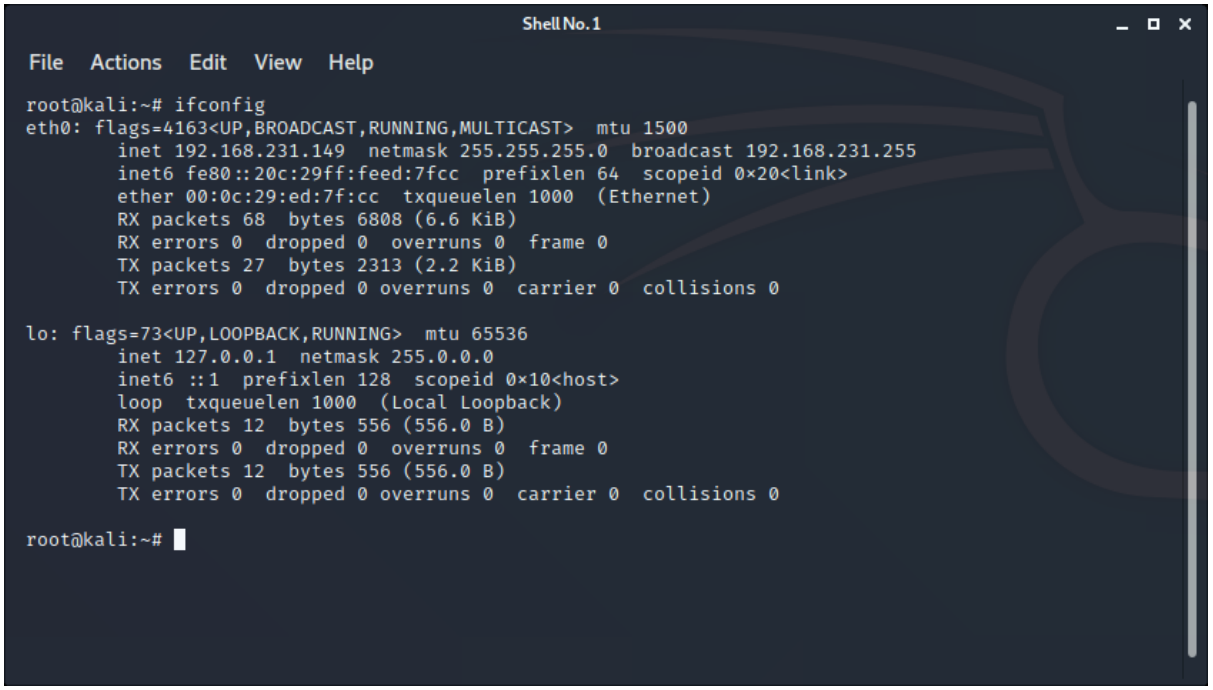
Decompiling a .Net executable and pointing the connection string or finding the decoding / decryption routine for the encoded / encrypted connection string in it would be no problem for a penetration tester with minimum development skills. However, in a rare occasion, the developer could have been used a commercial obfuscator to make reversers' job complicated. So, you may still need to resort to the dynamic analysis method which I will describe below.

Our setup for the dynamic analysis of the database connection is not complicated. One machine (VM-1 in our case) is to run the executable (with its accompanying libraries and data files of course). Another machine (VM-2 in our case) to act as a honeypot to attract all the traffic from the other machine to itself, and to provide an MSSQL server simulator waiting to respond to a connection request. That's all (well, if the executable is trying to connect to the MSSQL server with its FQDN anyways).



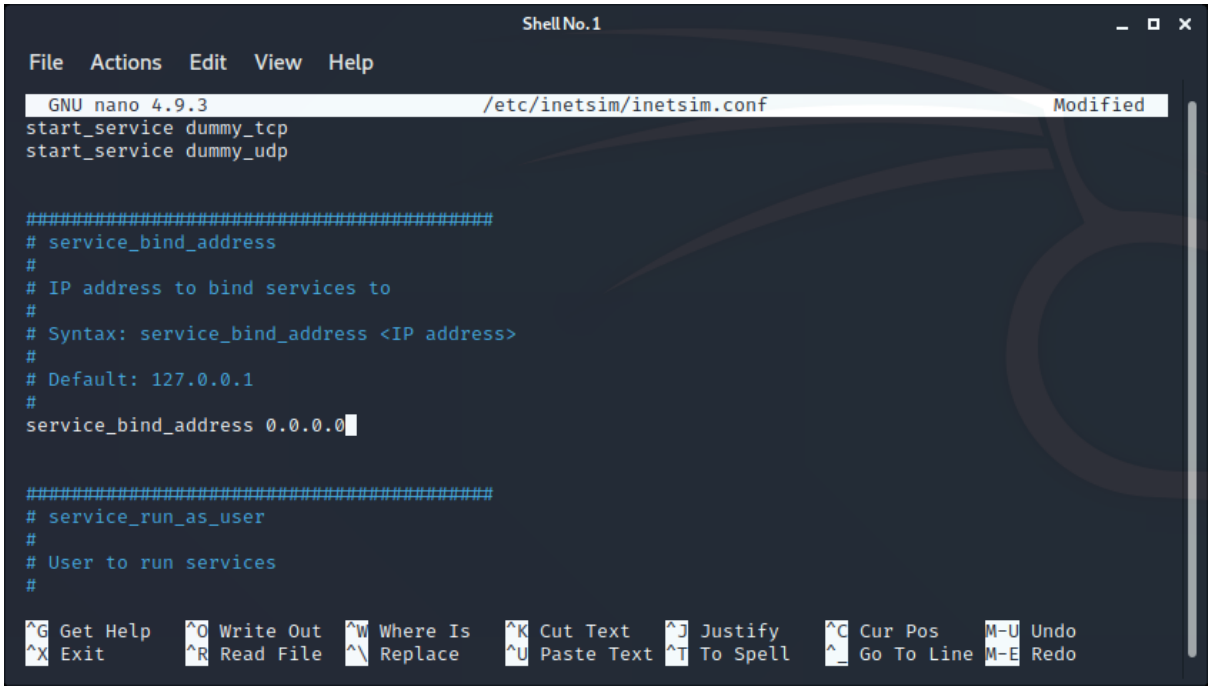
We used a Windows 10 (it could be any Windows version as long as the executable runs on it with no problem) and a Kali Linux (since we don't want to deal with installation problems for the tools that we need).

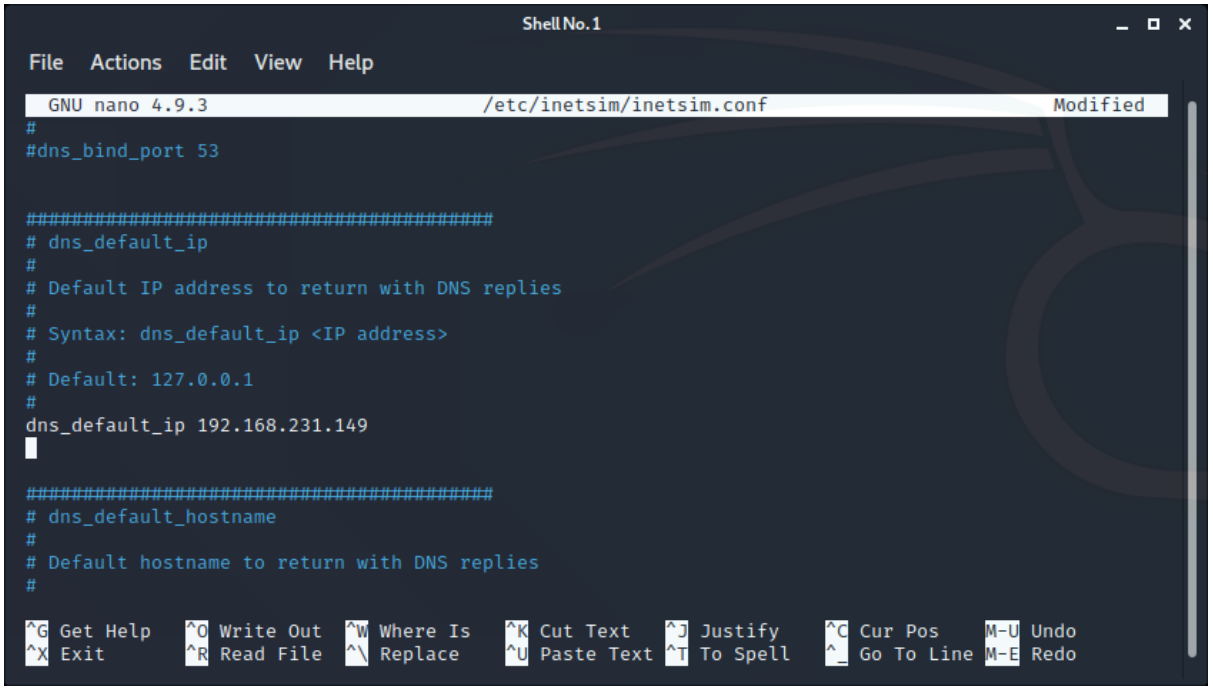
The IP address of the Kali machine is as below in my case (I will need this to use as a DNS server address on the Windows machine):



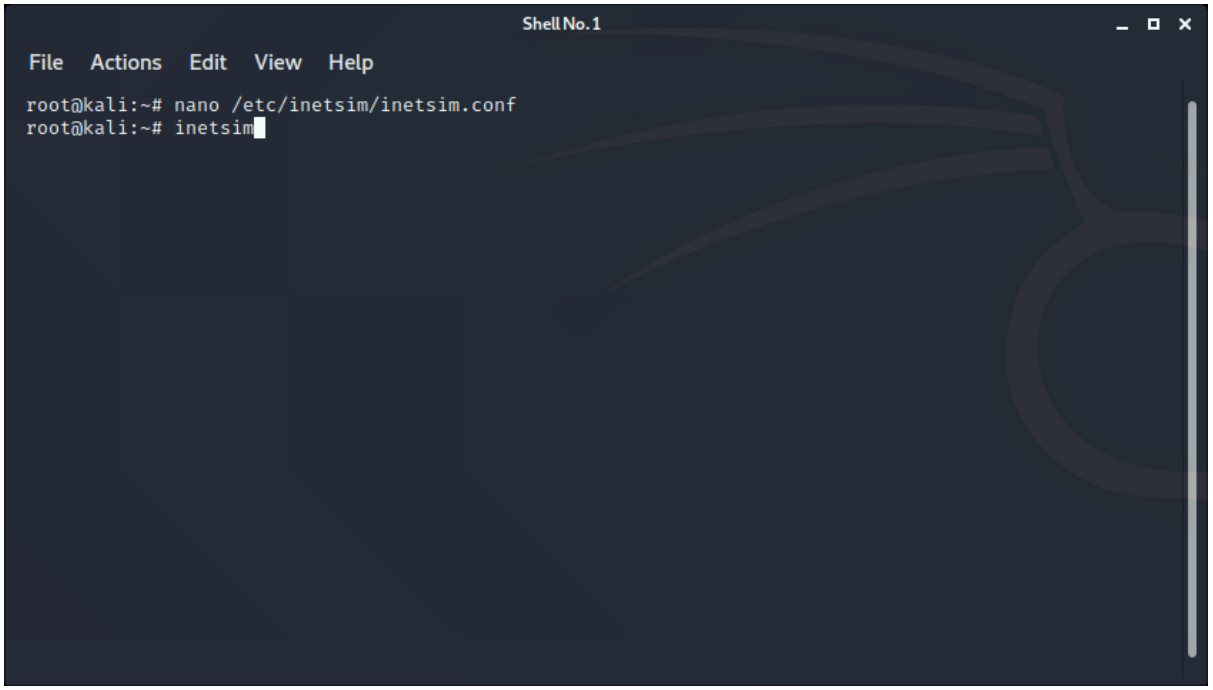
While I'm on the Kali machine I can configure my honeypot in accordance with my needs. What I need are:

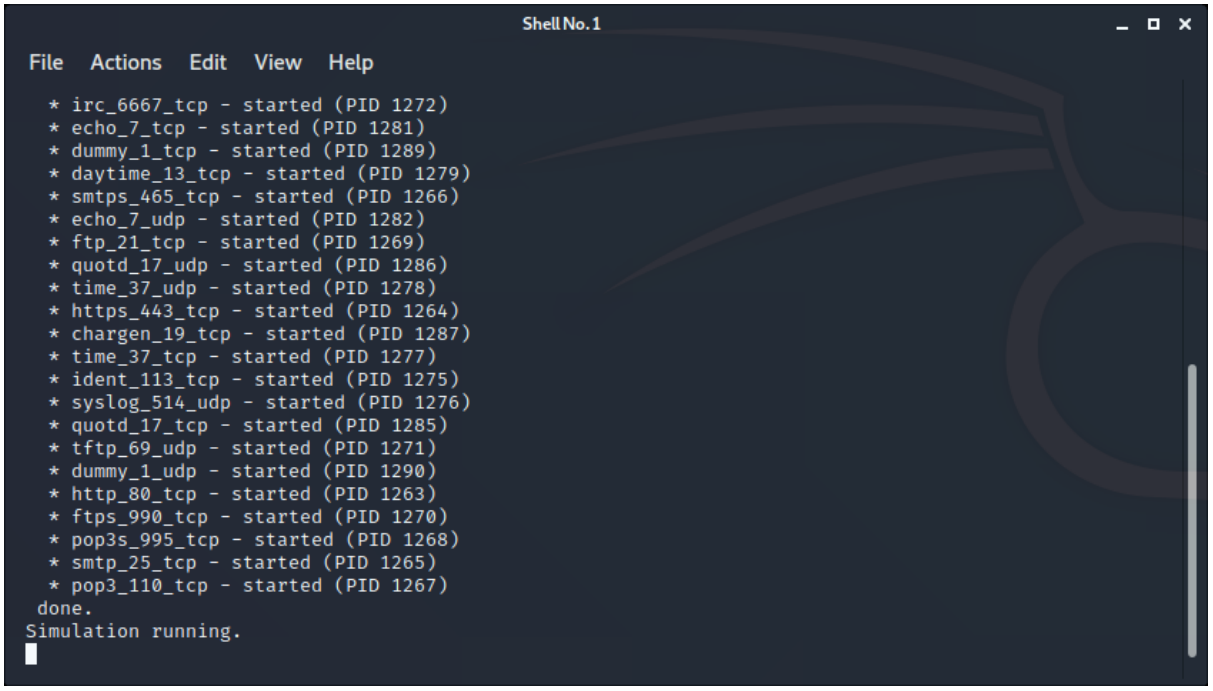
- To bind the DNS service to the machine's external network interface (0.0.0.0, meaning all interfaces actually)
- To respond to all name queries with the IP address of the Kali machine (you'll understand why in a few moments)





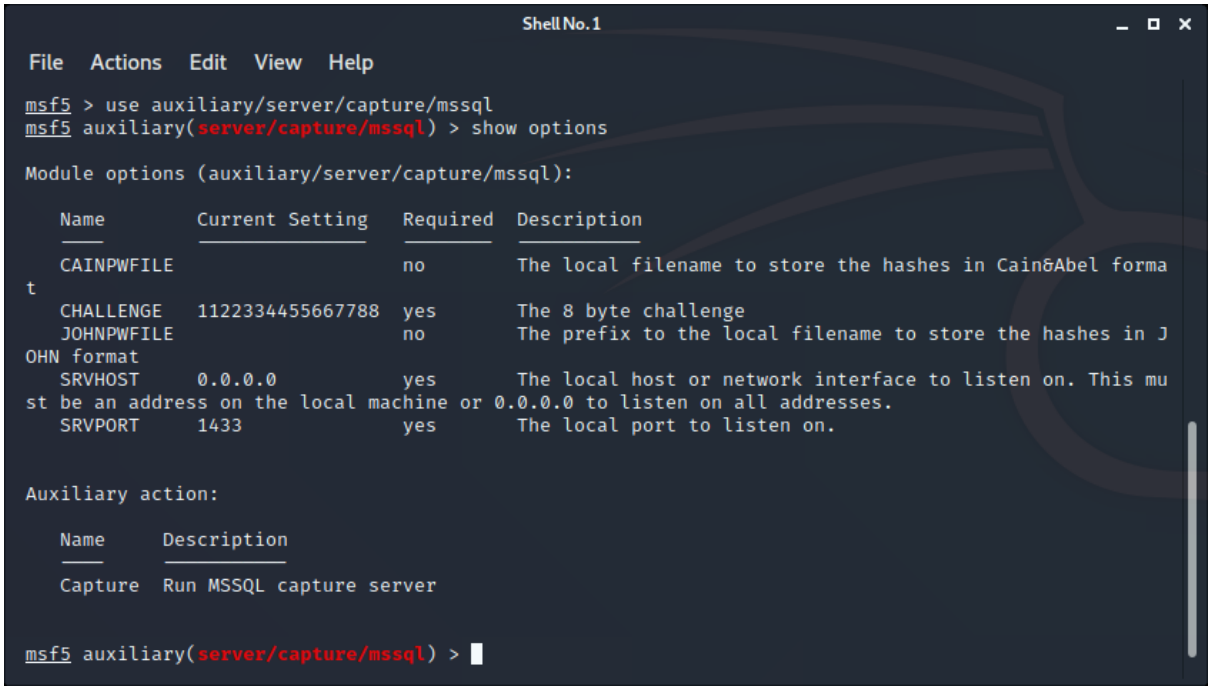
Let's start the "inetsim" to start the DNS service. It will start a myriad of other services, but they will not hurt us so I wouldn't bother disabling them.



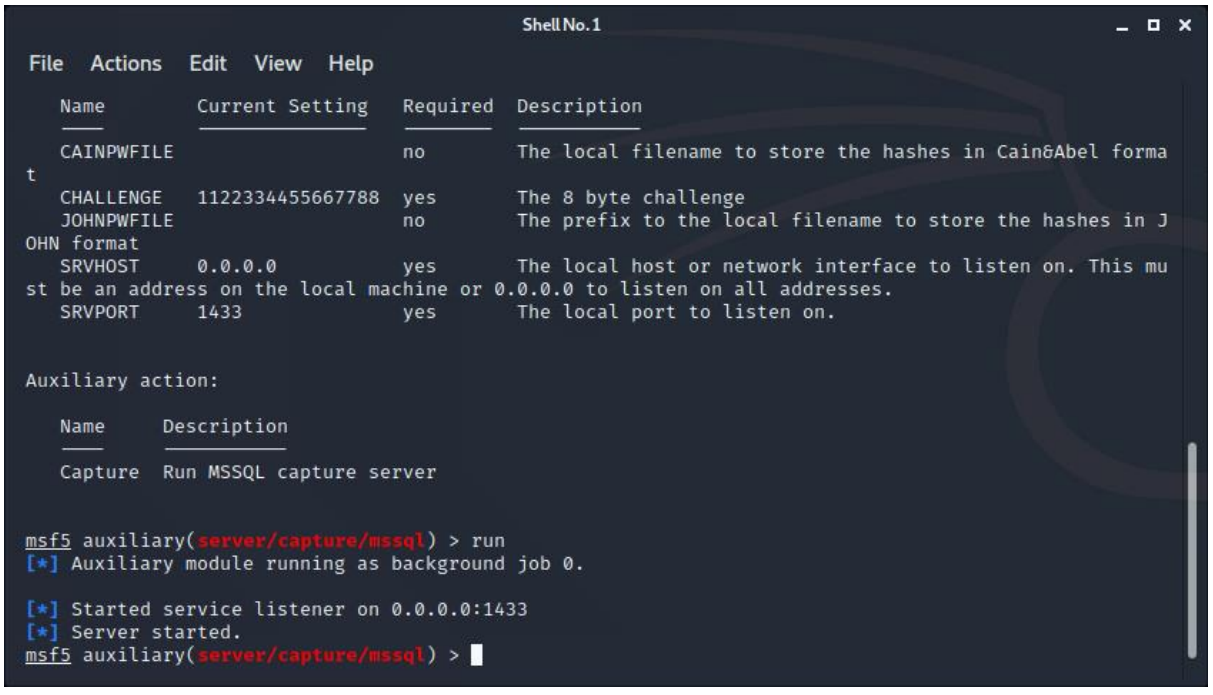


The real hero here is Metasploit as it is in many cases. Let’s start the Metasploit console first.

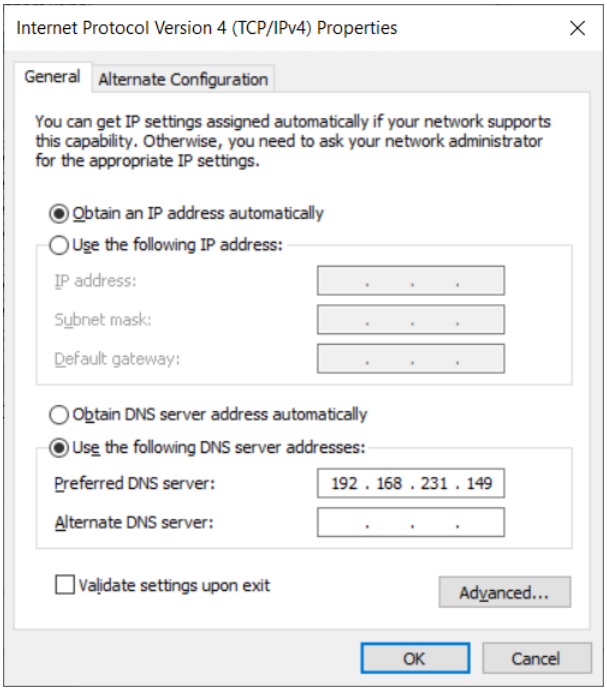
Use the following module to act as an MSSQL simulator. This module's sole purpose is to get the MSSQL server logon request and nothing else (we'll get into the internals later).



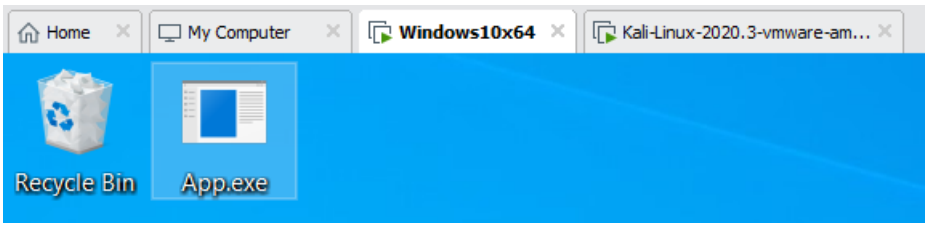
We don't mess with the config parameters and run with the defaults. At this point our Kali machine will be accepting the MSSQL connection requests. So, let's make the client executable try to communicate this service.



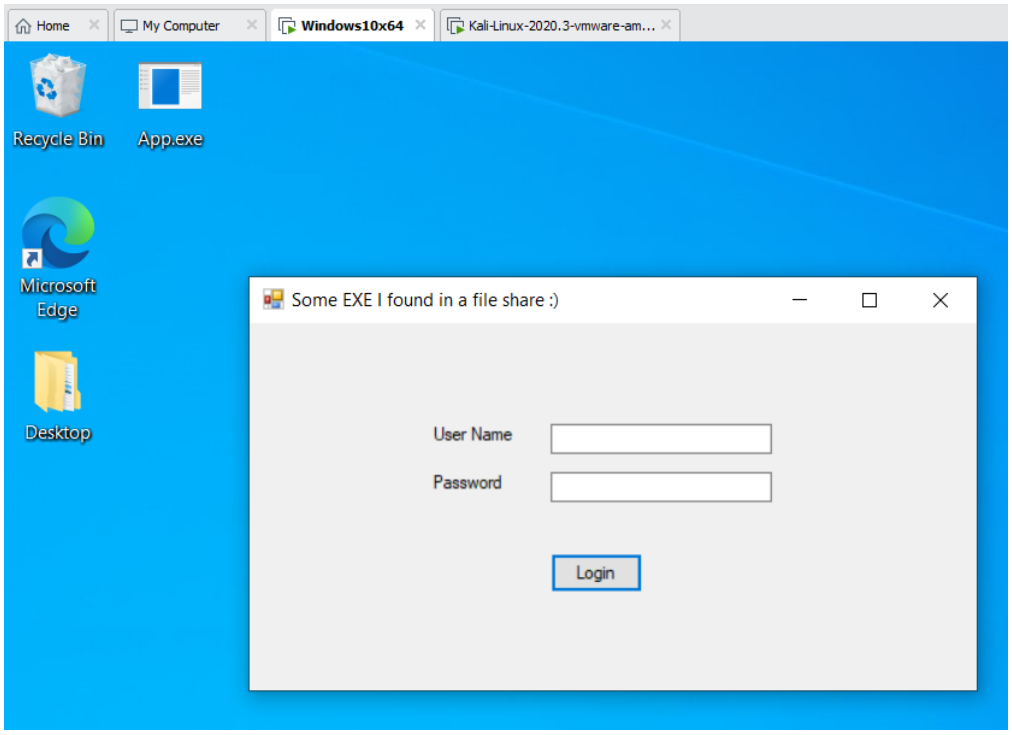
On the VM-1 (the Windows machine) we simply define the VM-2's (the Kali machine) IP address as the DNS server address. Remember this service will respond to any address resolution requests with its IP address.



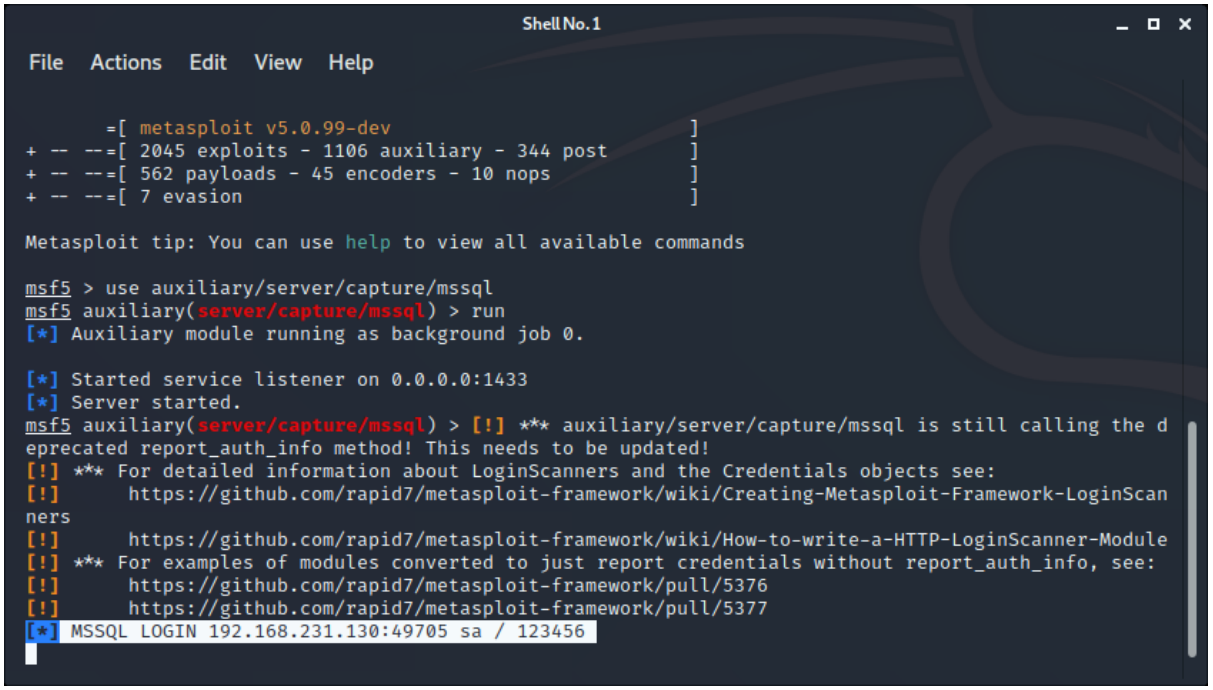
Let's run the application and get done with our job.



I push the Login button to initiate the connection process (in many cases you wouldn't need to do this since the application would try to connect to the database server on application start).



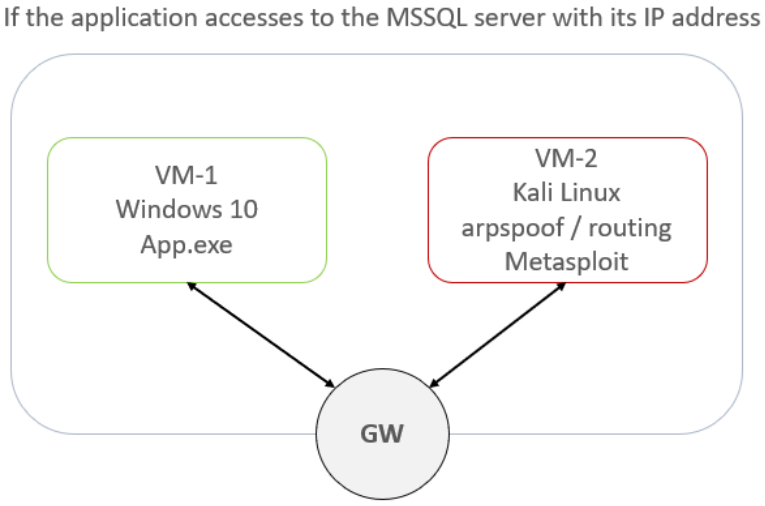
We magically see the database user name and password which the executable used to connect to the database (you may be asking what is the database server's FQDN, well you can use Wireshark to track the DNS query and find the server address easily).



What if the executable connects the Database Server with its IP address?

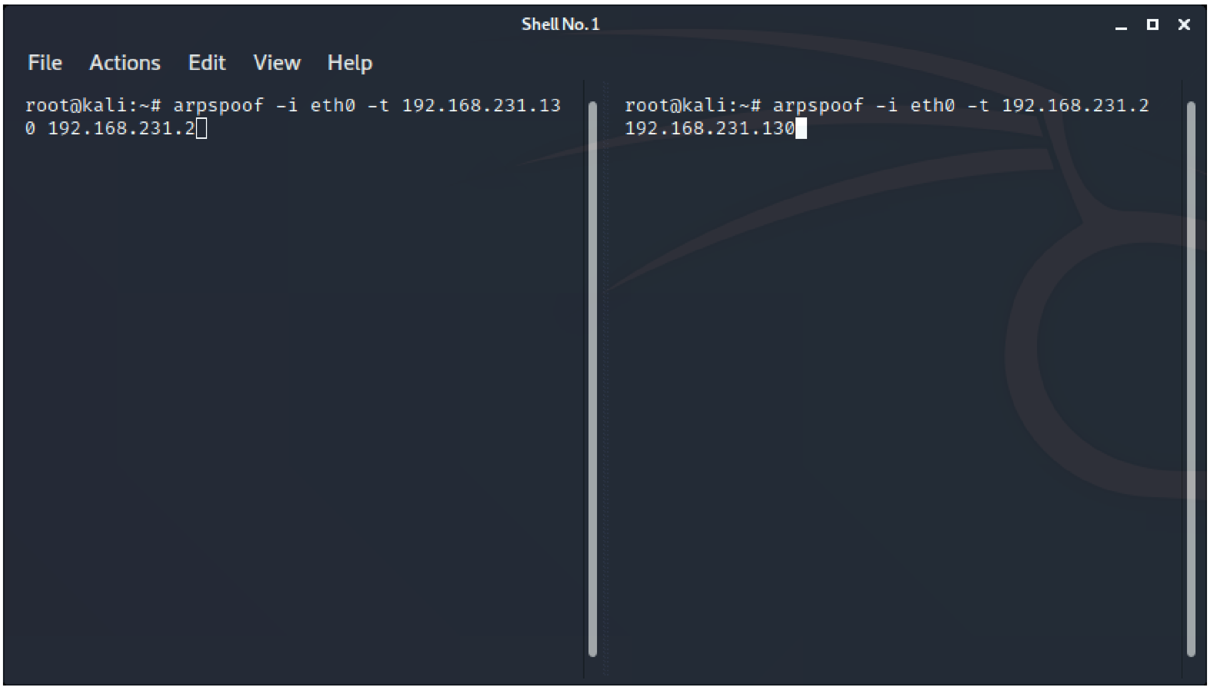
One other obstacle with analyzing your prey application could be that the developer could be using a hardcoded IP address as the MSSQL database server address. In that case, the DNS trap we used would be meaningless obviously, but we still have tricks to make the client come to us, instead of the real database server (thanks to the conveniences provided by Linux).

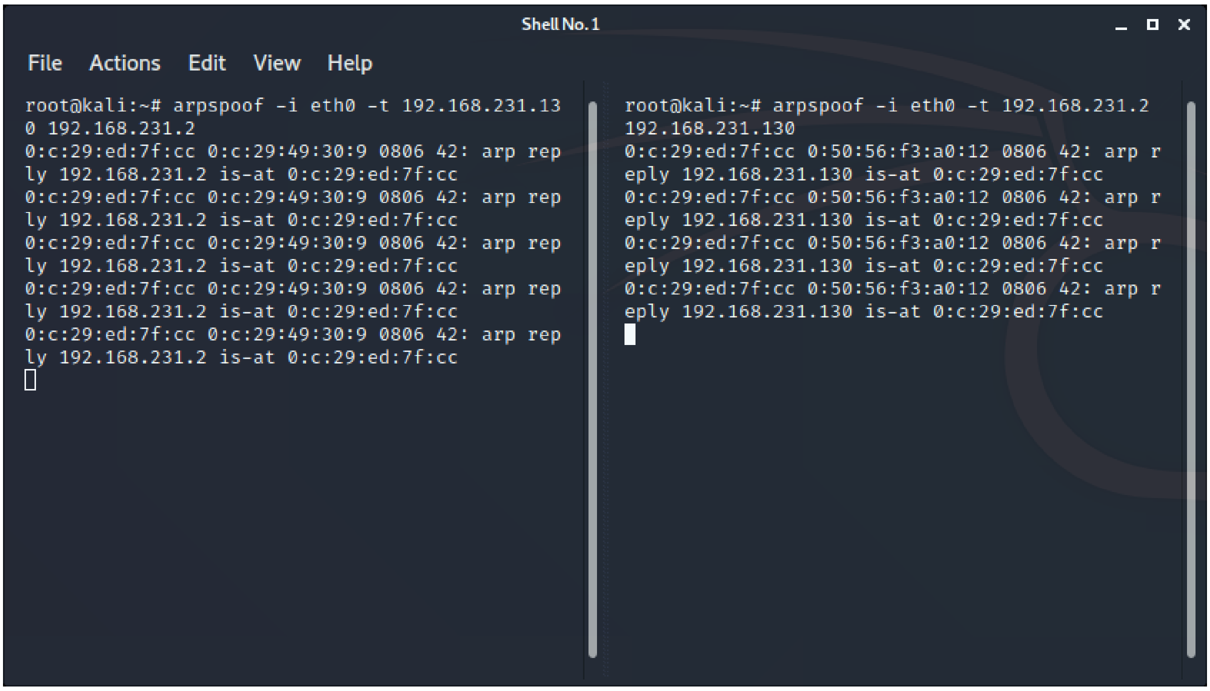
In this case we will use the same machines again but will trick the client to come to us with our ancient Man In The Middle technique: Arp Poisoning. Well, not only that, we will also use the routing and port forwarding utilities on the Kali Linux machine.



There are other tools to start our Arp Poisoning campaign but most of them make MITM and other staff. We don't need all that fuss, so we will stick to the simple arpspoof tool. First, we need to install it with the "dsniff" package. Then run the following two arpspoof commands to poison the ARP caches of the gateway and the Windows machine (you need to change the IP addresses in accordance with your environment of course):

```
# apt install dsniff
# arpspoof -i eth0 -t 192.168.231.130 192.168.231.2
# arpspoof -i eth0 -t 192.168.231.2 192.168.231.130
```





The next thing you should do to make the Windows application to talk to the MSSQL simulator on the Kali Linux machine to enable routing and port forwarding with the following two commands:

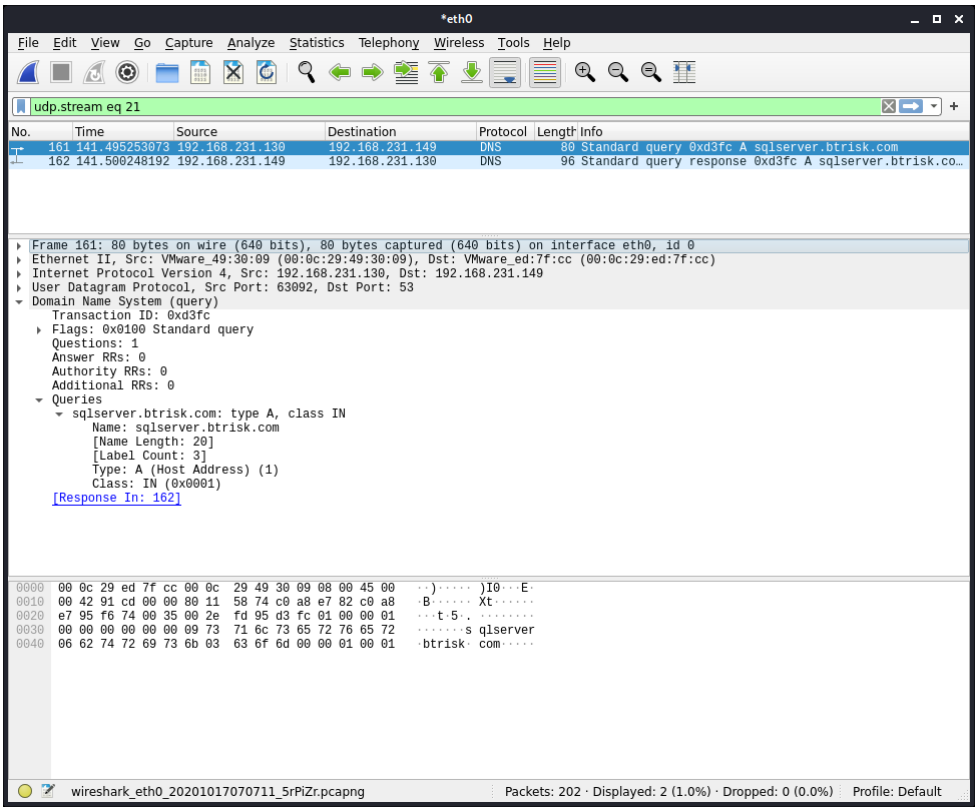
```
# echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables -t nat -I PREROUTING --dst 172.16.4.30 -p tcp --dport 1433 -j
REDIRECT --to-ports 1433
```

From this point on you will have the same results as before.

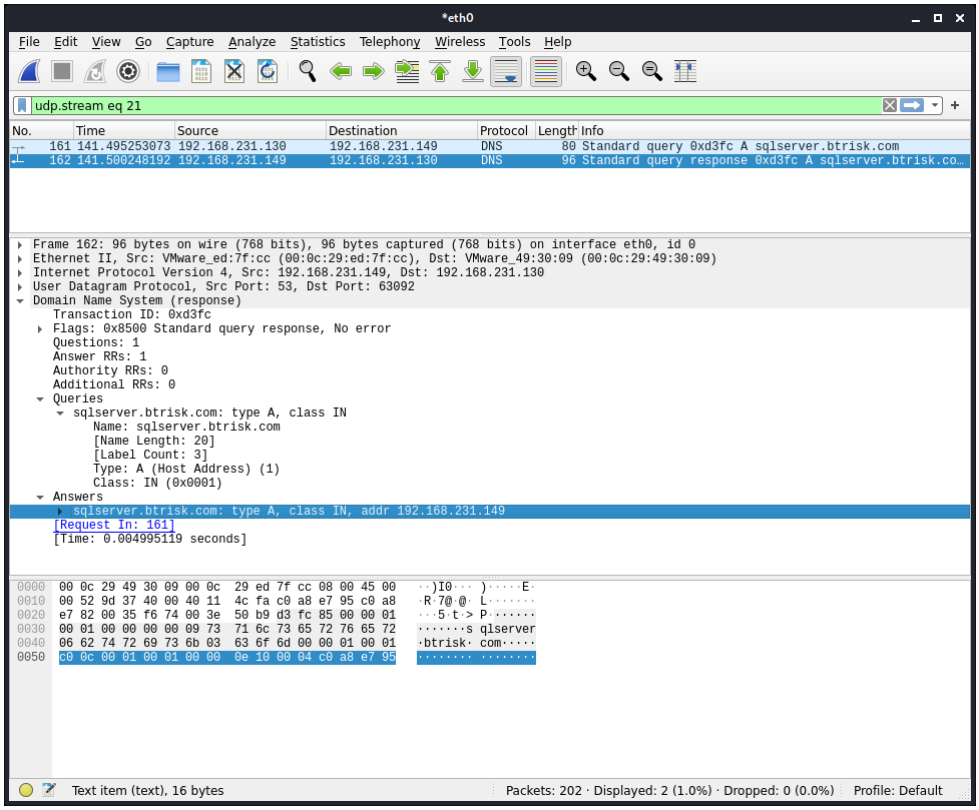
How did it happen?

In order to understand how did the Metasploit module achieved that we can first look at the network traffic. We can read the module code to get better insight about the process later.

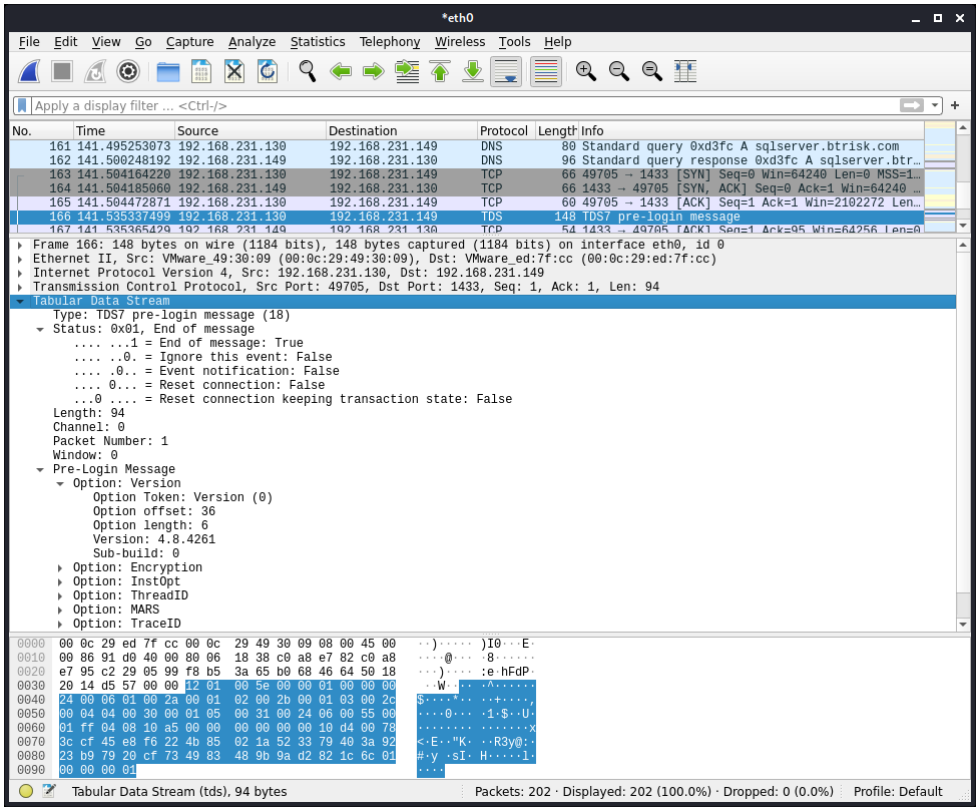
First, we see the DNS query for the MSSQL server address (Remember the question I mentioned before, here you can see the address of the MSSQL server. Obviously, we need it to proceed our reconnaissance further during our penetration test.)



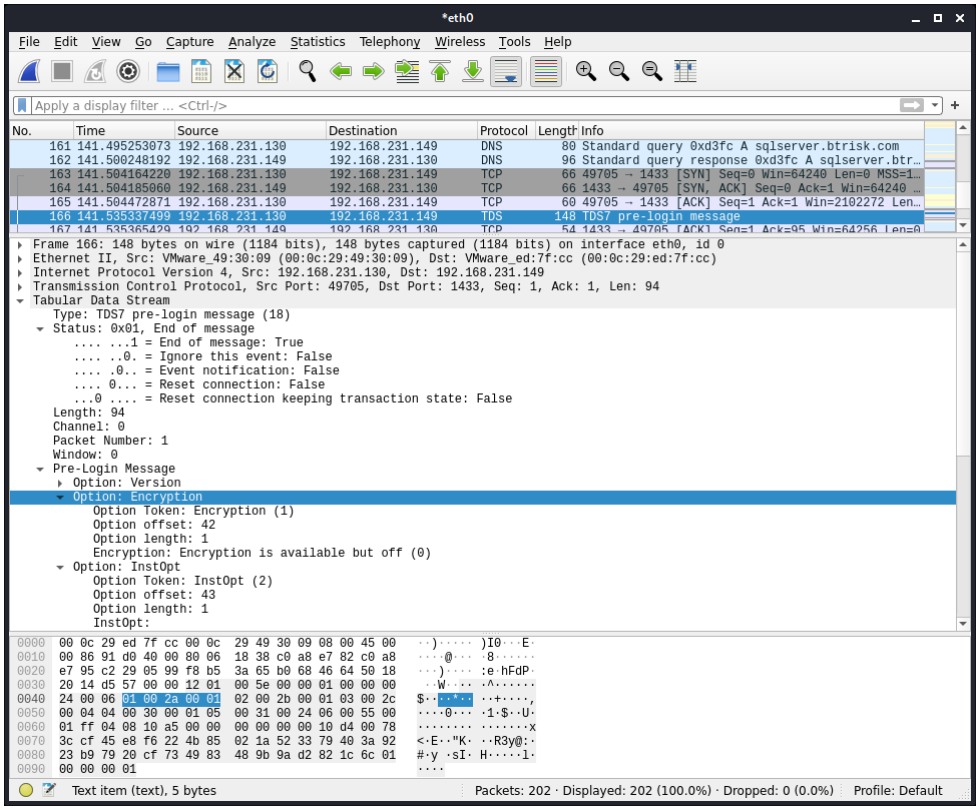
The response suggests the IP address to be the Kali Linux's IP address (since we configured inetsim to do this).



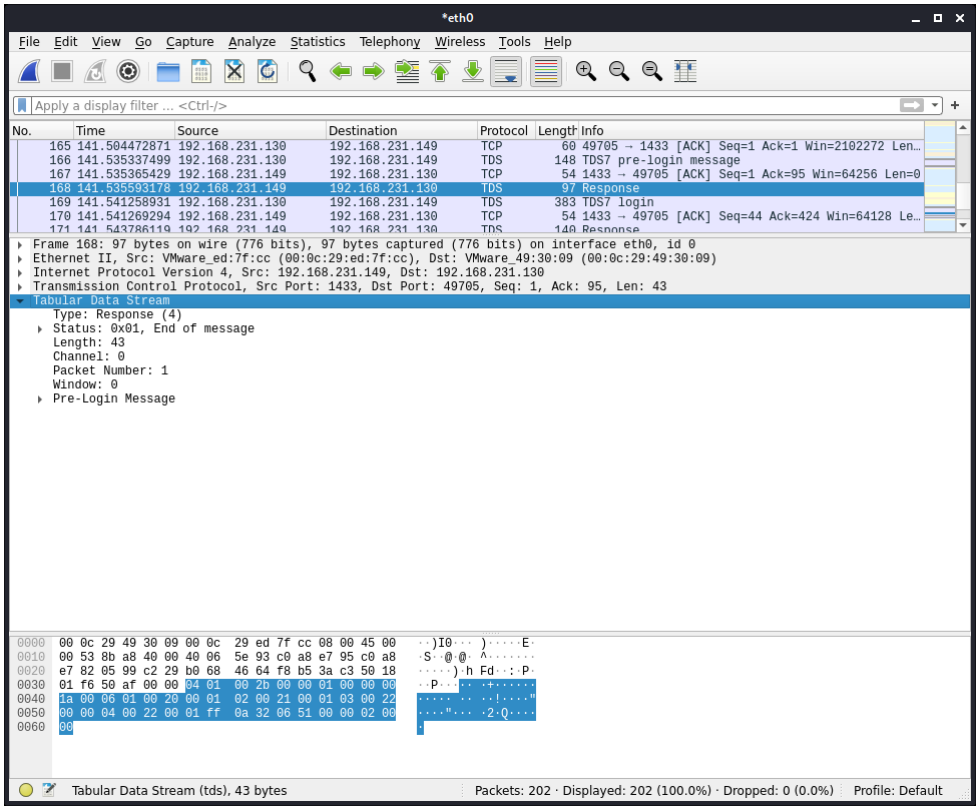
Below is the pre login request to the MSSQL server (which would be our malicious simulator service started from Metasploit).



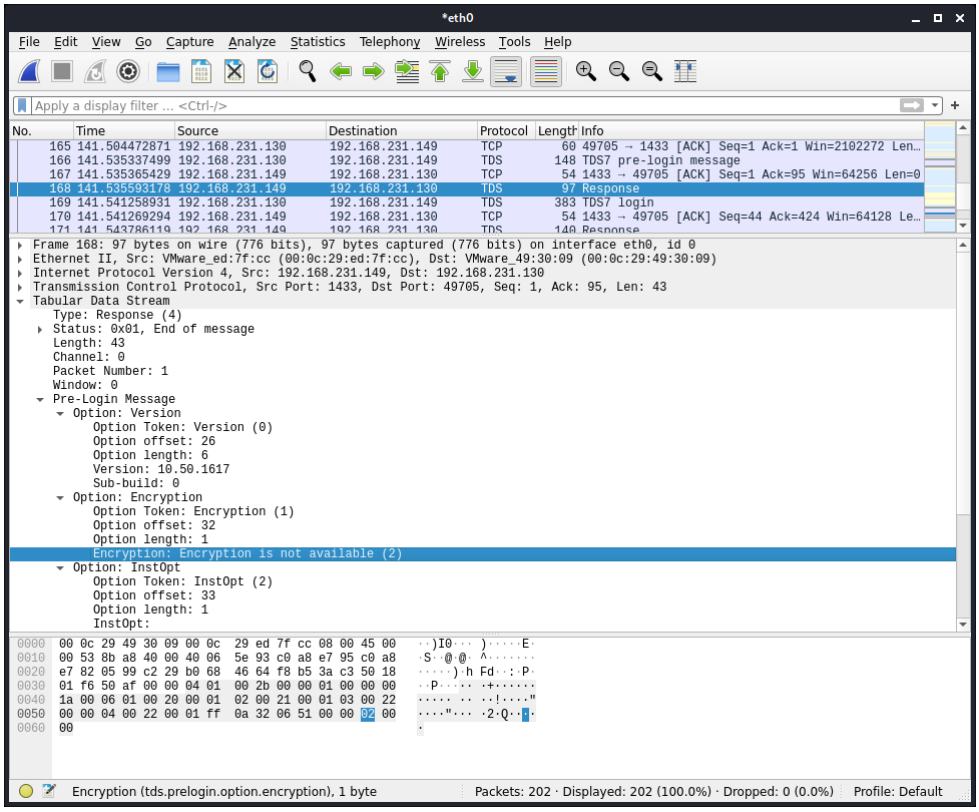
Below I highlighted that the client did not insist on an encrypted logon session. That knowledge will give us a hint to prevent the problem of stealing MSSQL login credentials (from the developer's perspective).



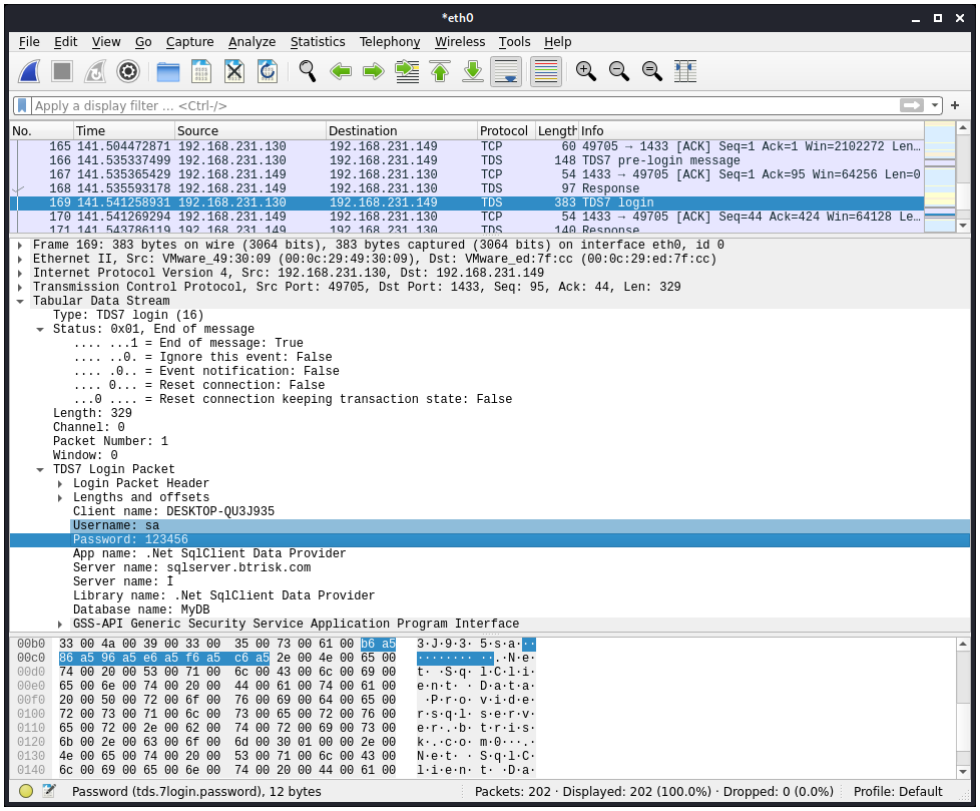
Below is the response of the malicious MSSQL simulator. We can compare this response data later to our static analysis of the MSSQL capture module.



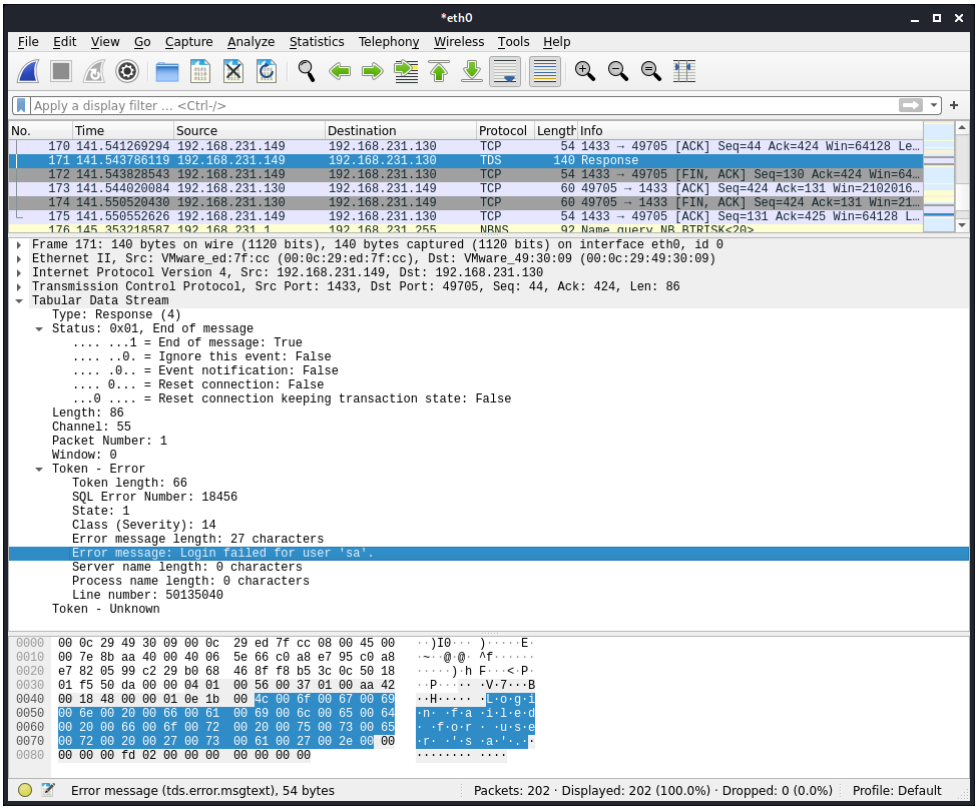
The essence of the credential capture tactic used is here, the malicious MSSQL simulator informs the client that it DOES NOT have support for the ENCRYPTION.



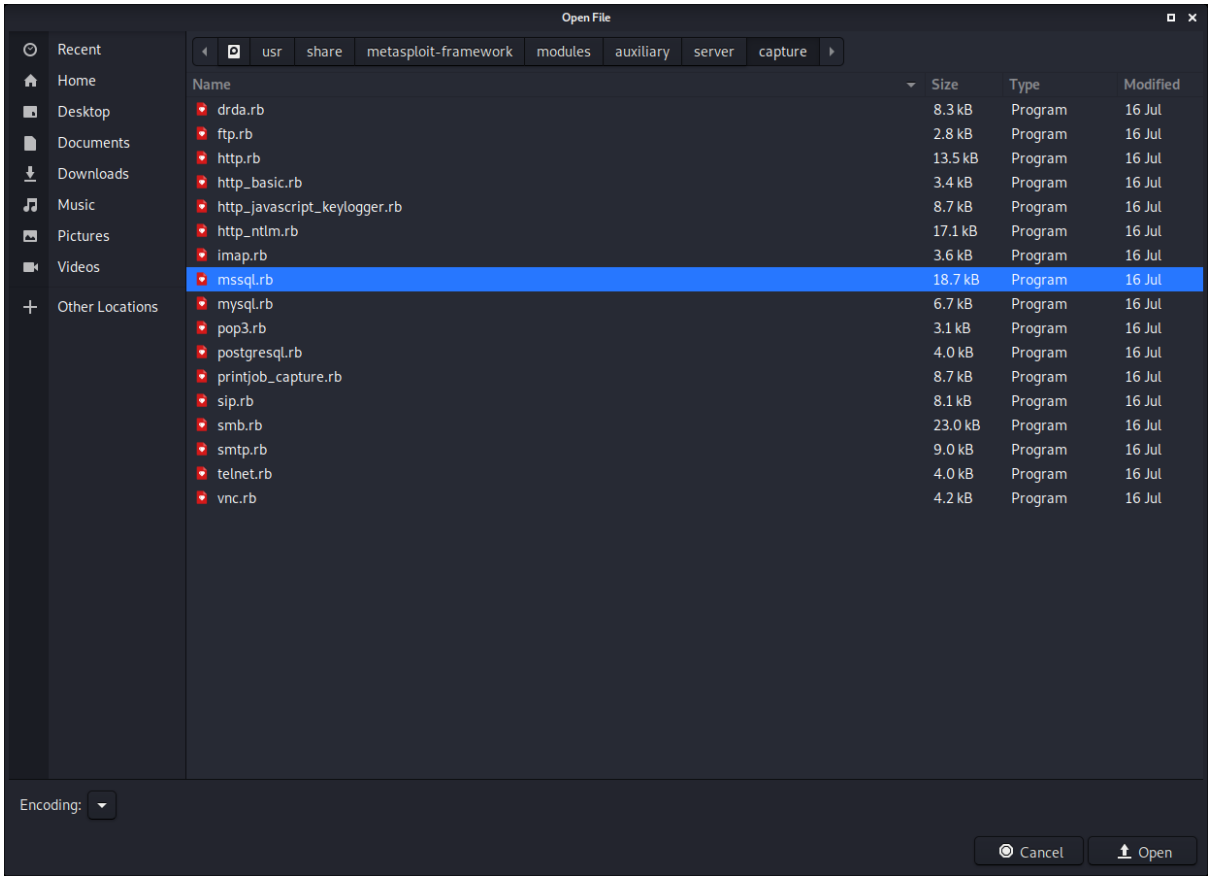
That results in the client not to use encryption. An astute reader might notice that the raw data is not clear text really. However, as we will see from the module code later, the password is encoded (not encrypted) somehow and obviously Wireshark is intelligent enough to decode it and show it to us.



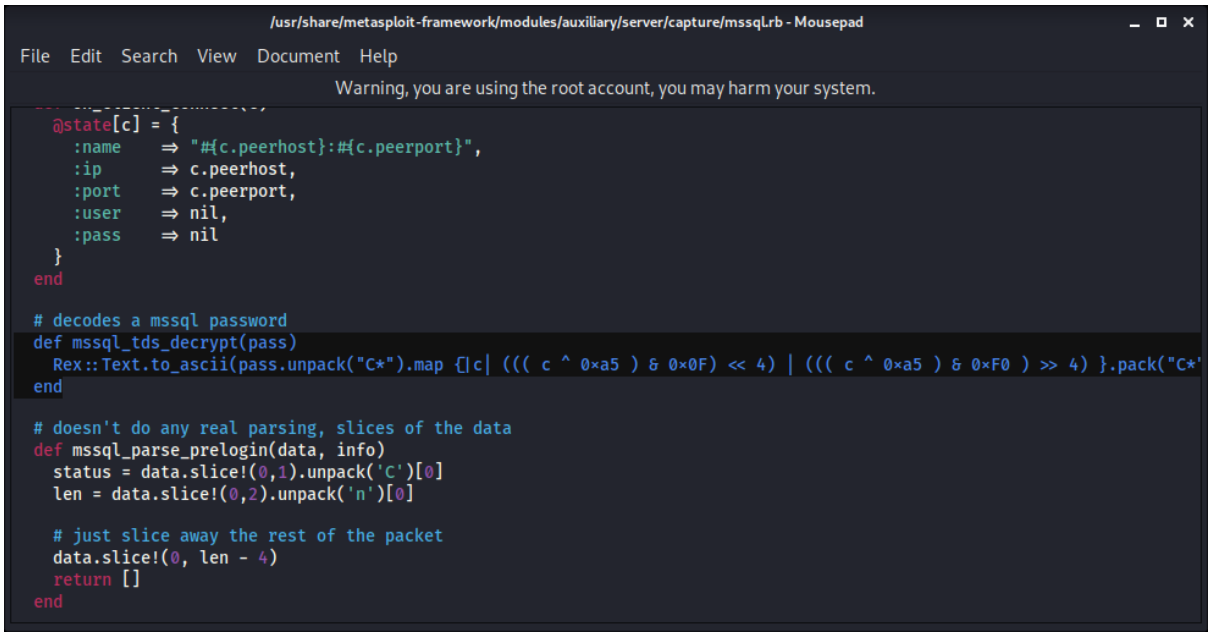
Our malicious MSSQL simulator service just says that the credentials are not correct. This suggests that the service did not intend to relay the traffic to the real server at all, just steal the credentials.



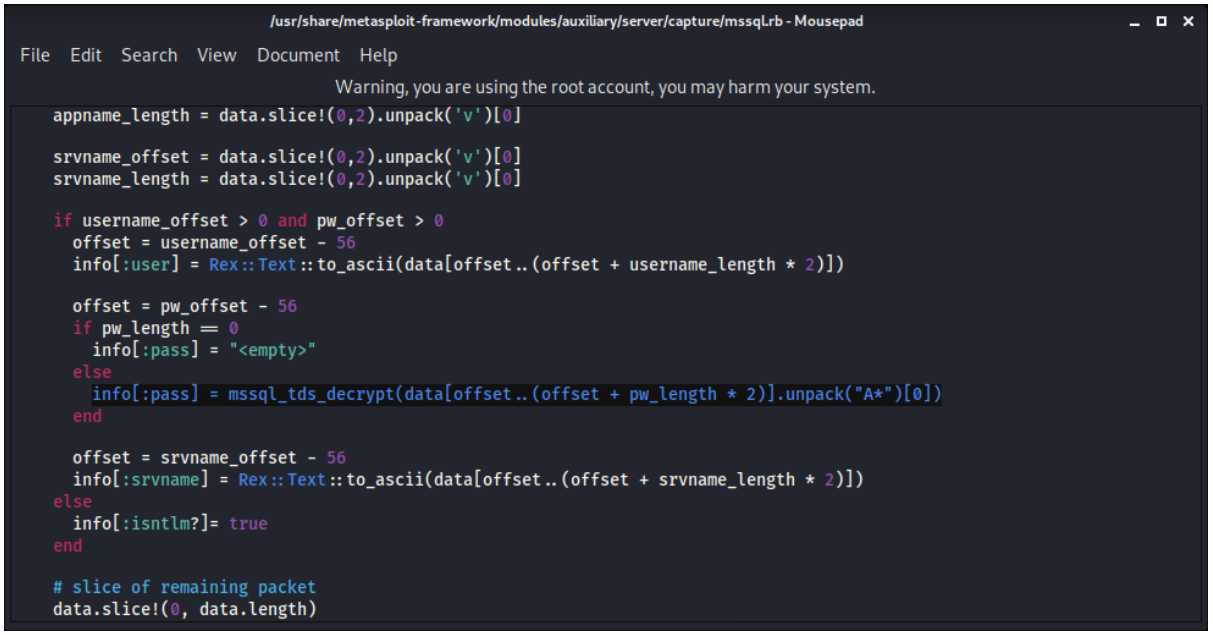
To understand the internals of our MSSQL service simulator we can look for the files named “mssql.rb” in our Kali Linux machine. There it is.



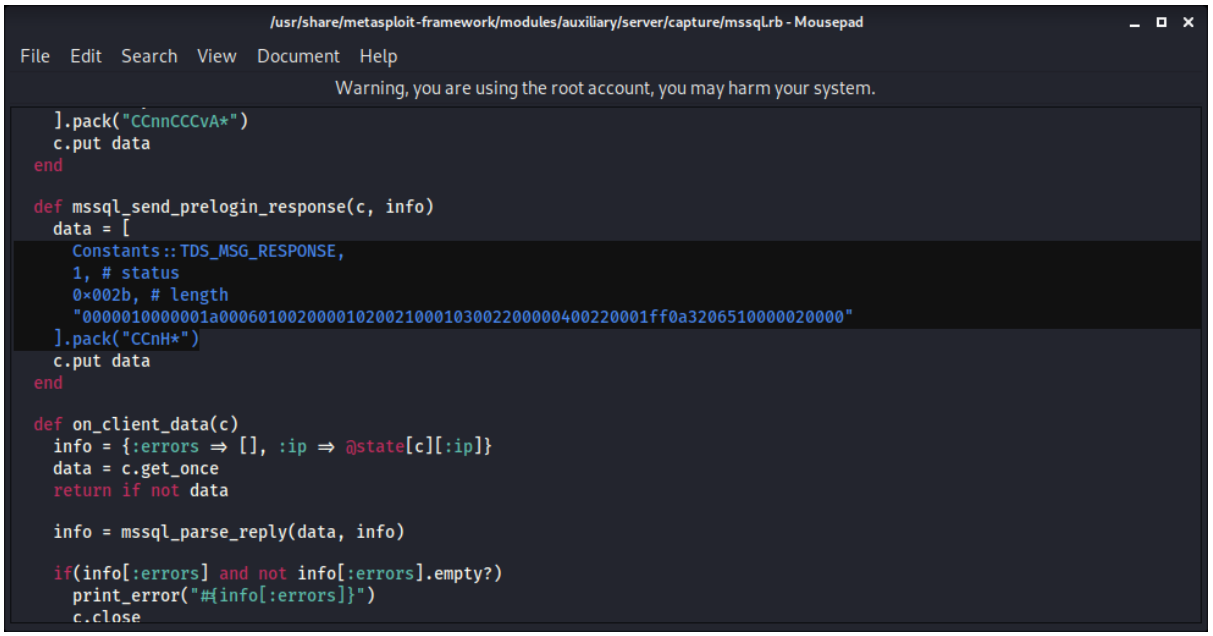
One of the things we can see in the code is the password decoding algorithm (although the function name is mssql_tds_decrypt).

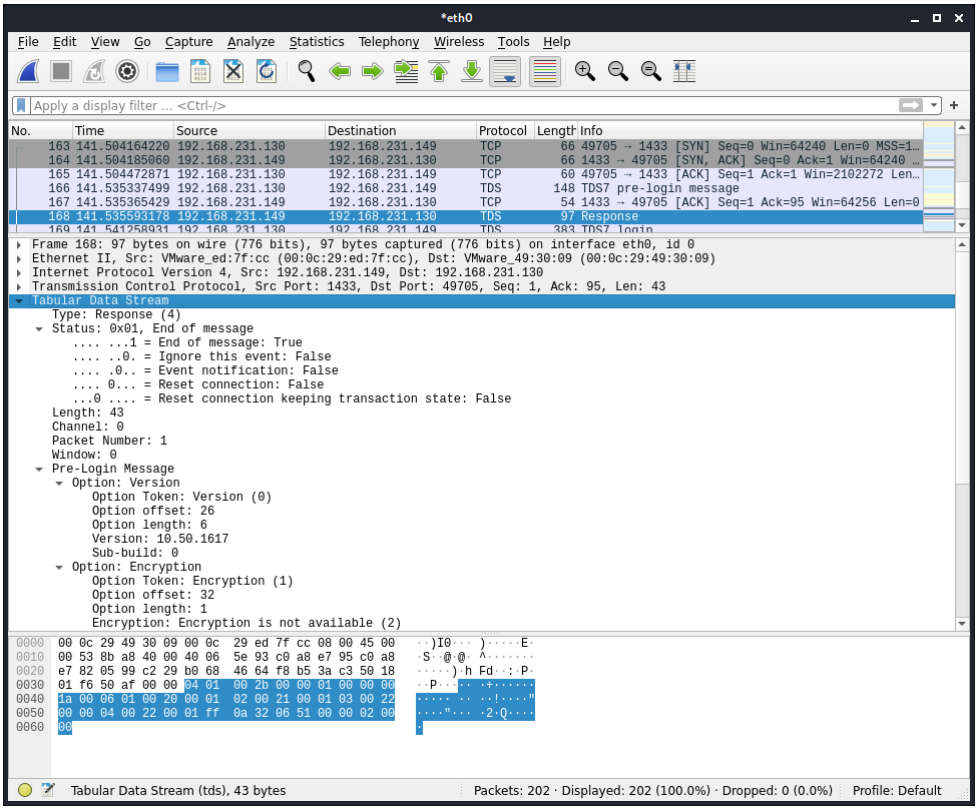


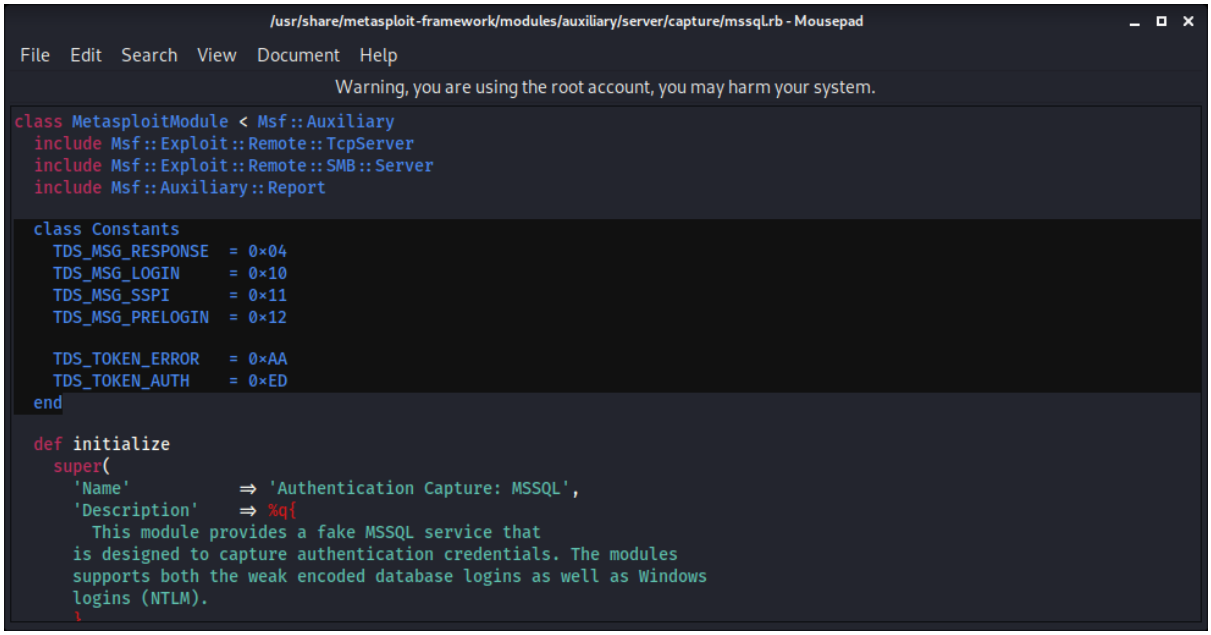
This is where the decoder is called.



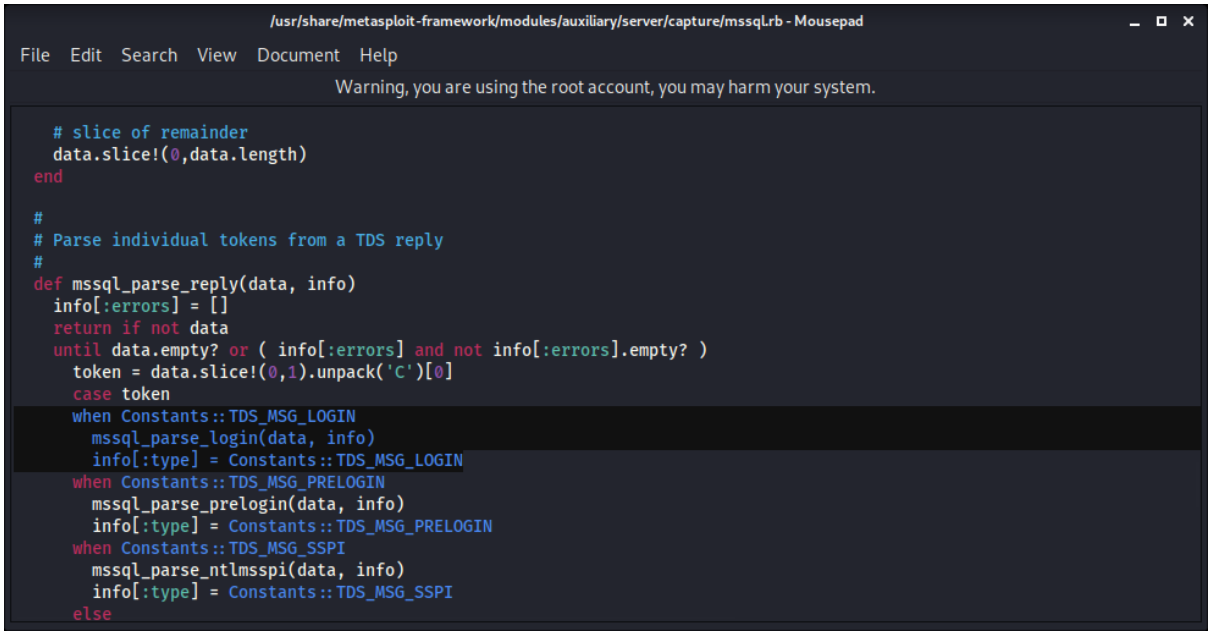
The pre-login response function includes the response data that we had seen earlier while we were analyzing the network packets in Wireshark (we'll see the constant's below).



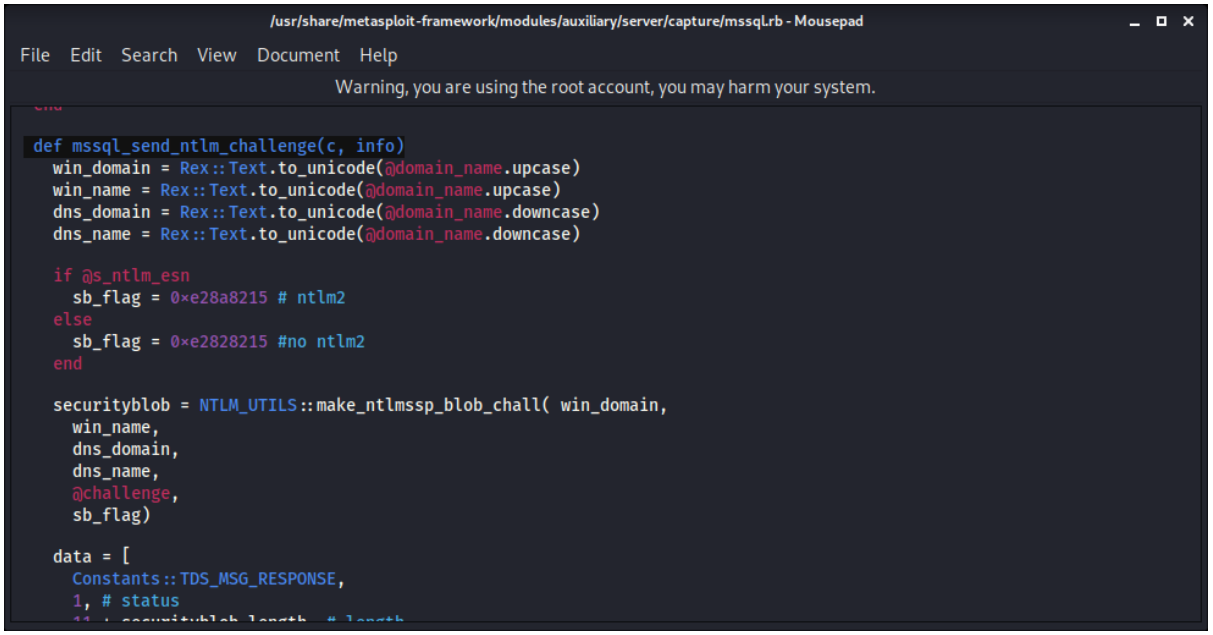




This is where different phases and kinds of client requests are handled.



The benefit of searching the internals of a tool is to discover its other capabilities. Here we can see that our simulator module is also capable of stealing LM and NTLM authentication credentials (hashes to be precise).



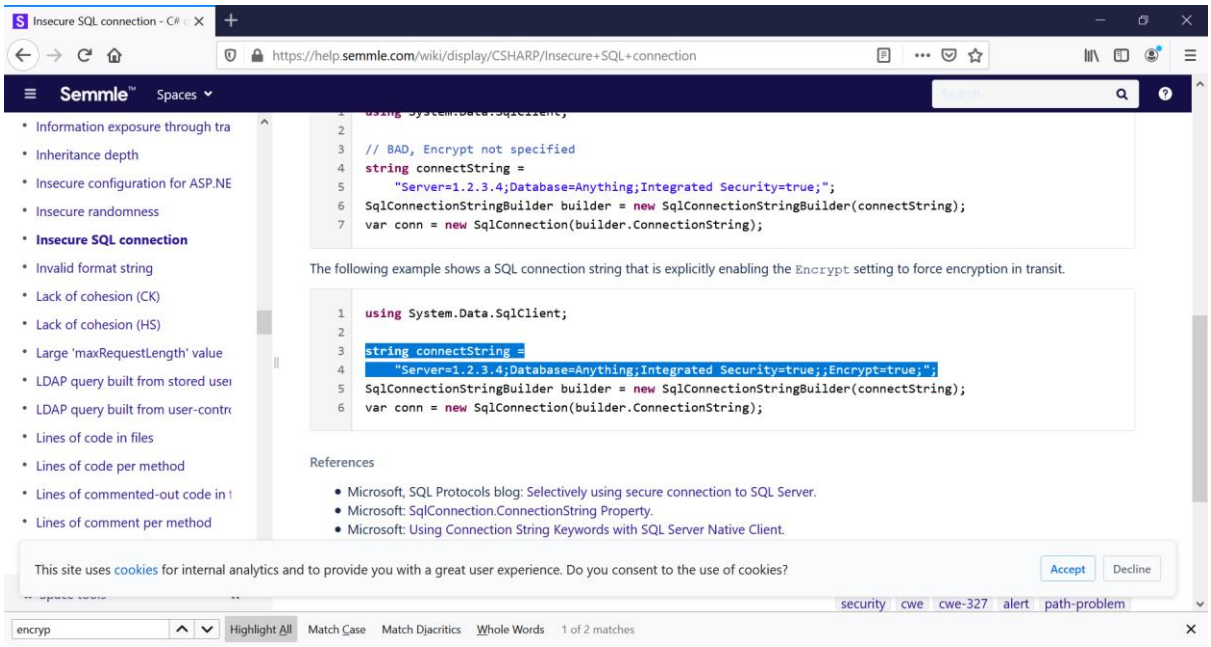
We can also see that the collected credentials can be recorded into the files with right format for two password cracker tools (i.e. Cain and JTR).


/usr/share/metasploit-framework/modules/auxiliary/server/capture/mssql.rb - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.
fd = File.open(datastore['CAINPWFILE'], "ab")
fd.puts(
[
user,
domain ? domain : "NULL",
@challenge.unpack("H*")[0],
lm_hash ? lm_hash : "0" * 48,
nt_hash ? nt_hash : "0" * 48
].join(":").gsub(/\n/, "\\n")
)
fd.close
end
end
if(datastore['JOHNPWFILE'] and user)
case ntlm_ver
when NTLM_CONST::NTLM_V1_RESPONSE, NTLM_CONST::NTLM_2_SESSION_RESPONSE
fd = File.open(datastore['JOHNPWFILE'] + '_netntlm', "ab")
fd.puts(
[
user,"",
domain ? domain : "NULL",
lm_hash ? lm_hash : "0" * 48,

How to defend against this attack (from a developer's perspective)?

The root cause of this problem is that the application is developed with a 2-tier architecture. Thus, the client must have MSSQL server credentials on the client side (whether in a config file, in its code or use user credentials to access the MSSQL server). However, if we were stuck with this architecture (i.e. we cannot switch to a 3 tier application because of the operational dependencies to the application, financial constraints or for another reason) we can do the following to protect ourselves from the kind of dynamic test attack described in this article.

We can use the "Encrypt" attribute to refuse the client to connect to the MSSQL server without encryption.



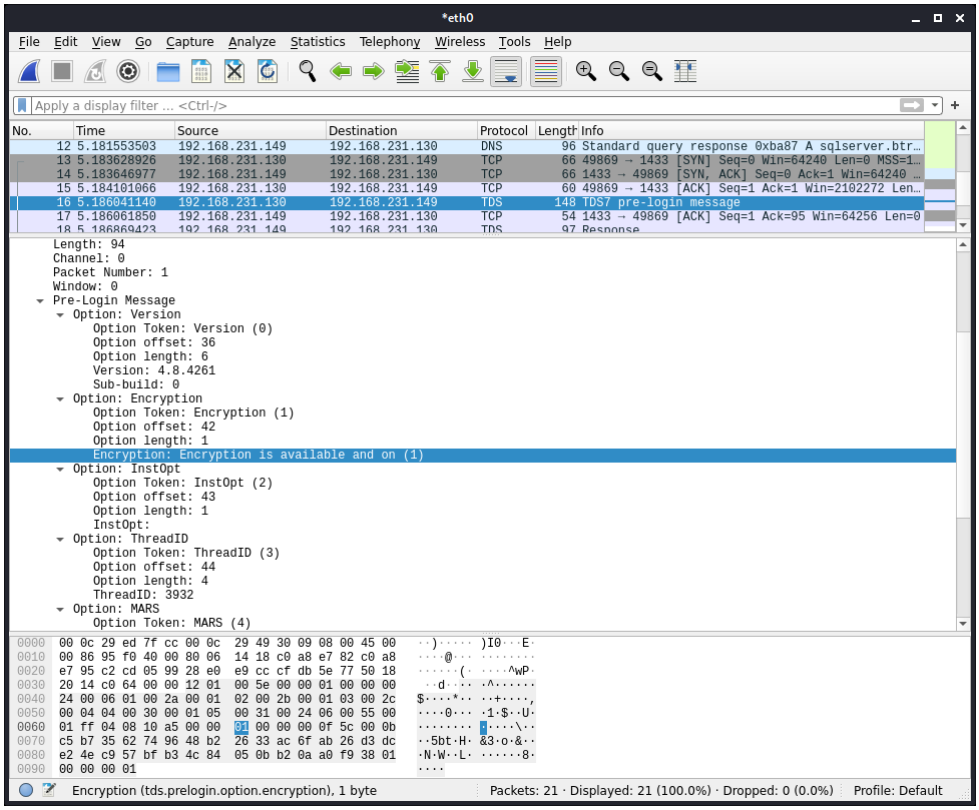
Below is the change we implemented in our existing connection string. We added "Encrypted=true" parameter and value to the connection string.

```
14      public partial class loginForm : Form
15      {
16          public loginForm()
17          {
18              InitializeComponent();
19
20          }
21          private void loginForm_Load(object sender, EventArgs e)
22          {
23          }
24
25          private void BTN_login_Click(object sender, EventArgs e)
26          {
27              try
28              {
29                  SqlConnection conn =
30                      new SqlConnection("server=sqlserver.btrisk.com;user=sa;pwd=123456;database=MyDB;;Encrypt=true;");
31                  conn.Open();
32              } catch(Exception ex) {}
33          }
34      }
35  }
```

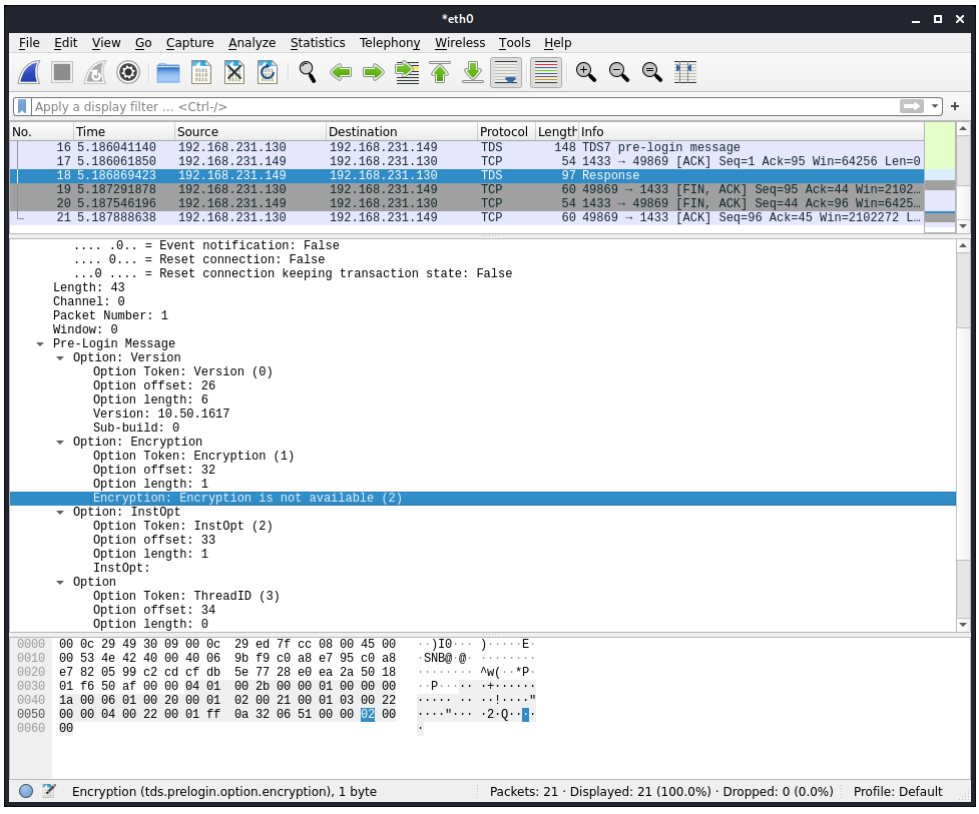
Let's try the new code to see if it will fall victim to our attack.

```
                                    Shell No.1
File   Actions   Edit   View   Help
   Name         Current Setting   Required  Description
   ----         ---------------   --------  -----------
   CAINPWFILE                     no        The local filename to store the hashes in Cain&Abel forma
t
   CHALLENGE    1122334455667788  yes       The 8 byte challenge
   JOHNPWFILE                     no        The prefix to the local filename to store the hashes in J
OHN format
   SRVHOST      0.0.0.0           yes       The local host or network interface to listen on. This mu
st be an address on the local machine or 0.0.0.0 to listen on all addresses.
   SRVPORT      1433              yes       The local port to listen on.


Auxiliary action:

   Name     Description
   ----     -----------
   Capture  Run MSSQL capture server


msf5 auxiliary(server/capture/mssql) > run
[*] Auxiliary module running as background job 0.

[*] Started service listener on 0.0.0.0:1433
msf5 auxiliary(server/capture/mssql) > [*] Server started.
```

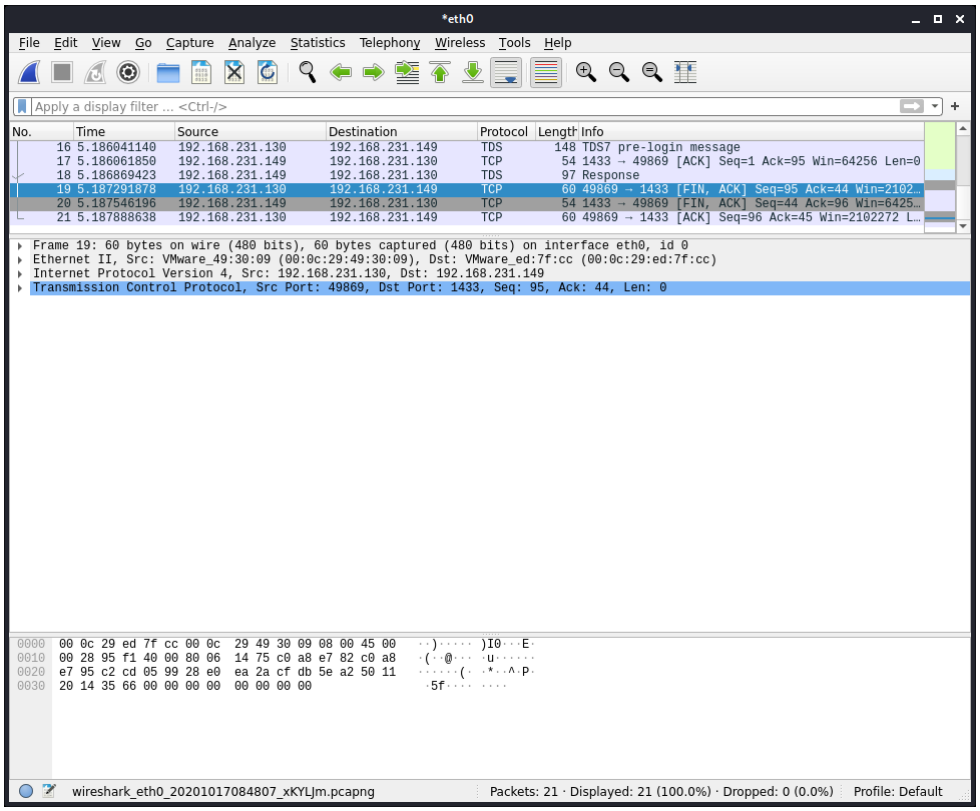
This time our executable specifically requires encryption, it is not optional anymore.



The MSSQL simulator service responds as usual indicating that it does not want encryption.



However, this time the client does not care and tears down the TCP connection immediately. Hence, we (the attacker) do not have the chance to discover the password.

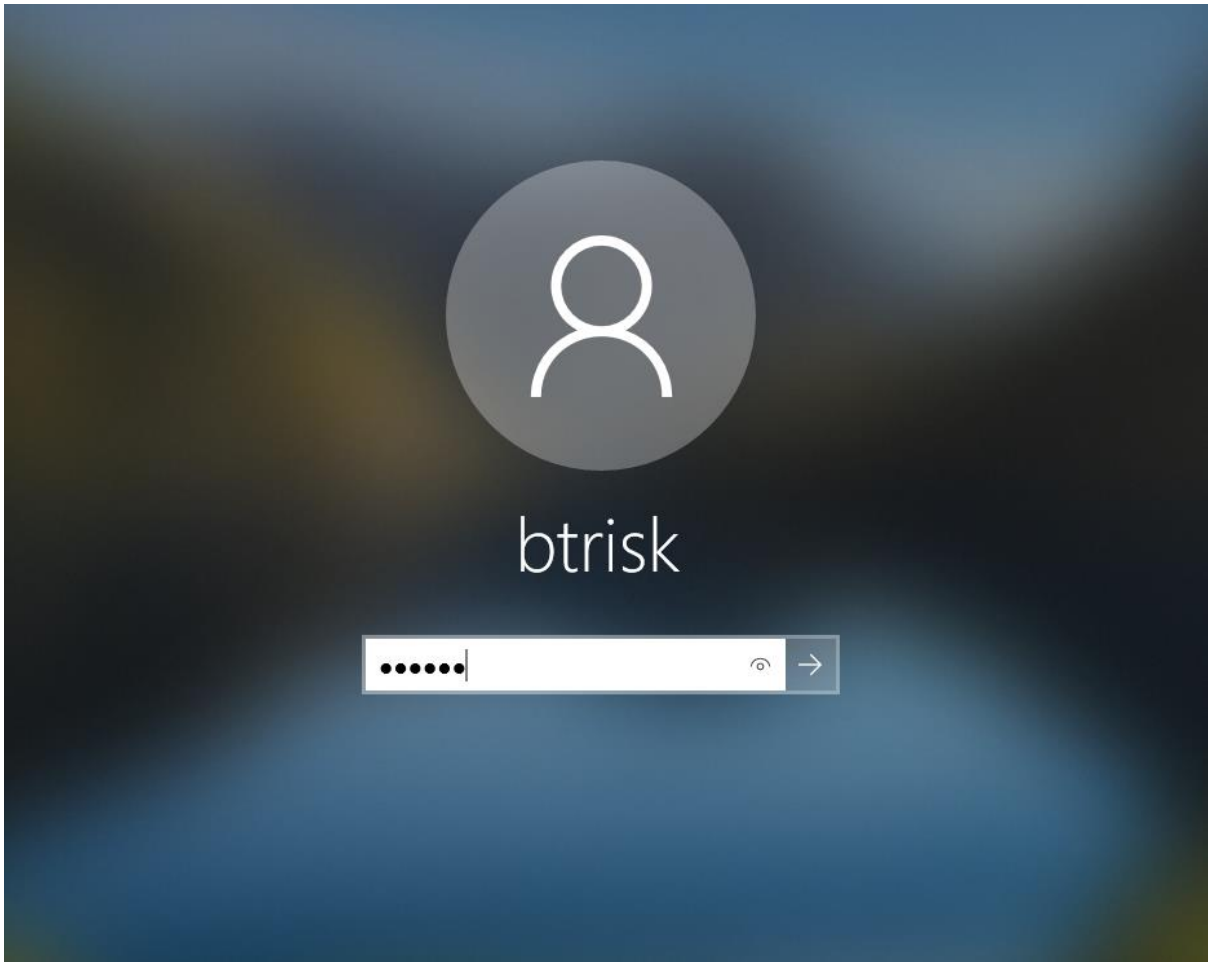


What happens if the client executable uses Windows Authentication?

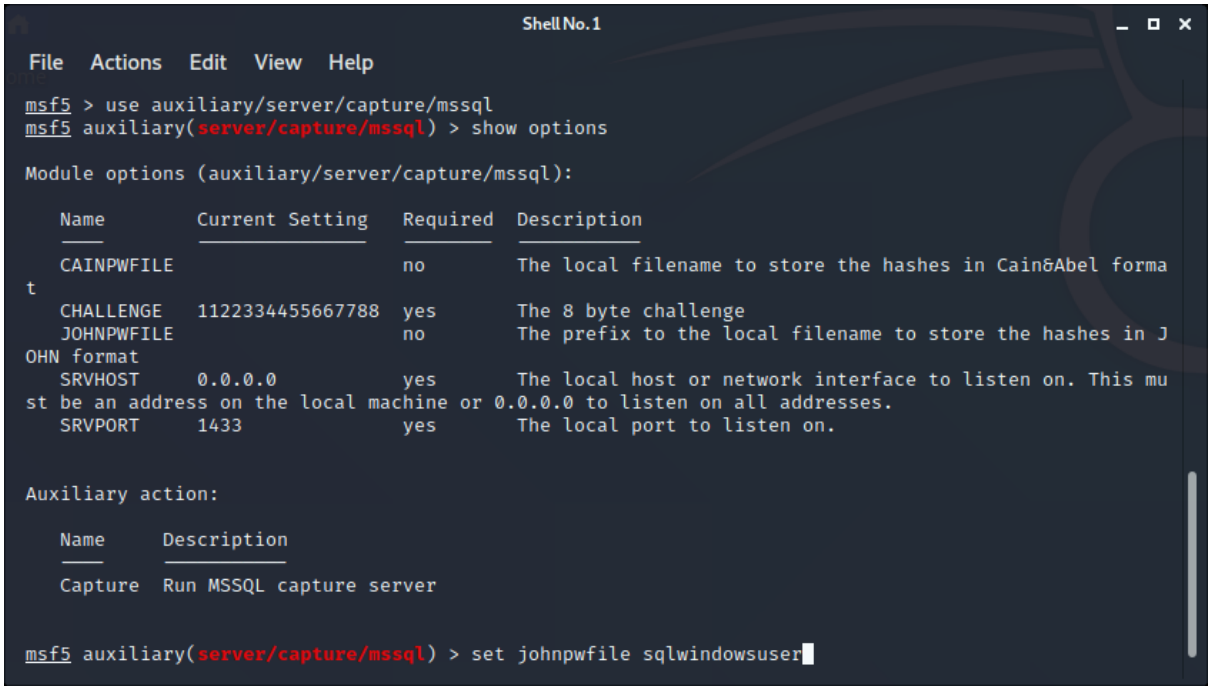
To impersonate the connecting user, we need to start the process as a different user. In order to keep things simple and just discover our tools extra capabilities we will try to connect to the database server as the PC user. To do this we change the connection string as below:

```
23          private void loginForm_Load(object sender, EventArgs e)
24          {
25          }
26
27          private void BTN_login_Click(object sender, EventArgs e)
28          {
29              try
30              {
31                  SqlConnection conn =
32                      // new SqlConnection("server=sqlserver.btrisk.com;user=sa;pwd=123456;database=MyDB;;Encrypt=true;");
33                      new SqlConnection("Data Source=sqlserver.btrisk.com;;Integrated Security=true;");
34
35                  conn.Open();
36              } catch(Exception ex) {}
37          }
38      }
39  }
40
```

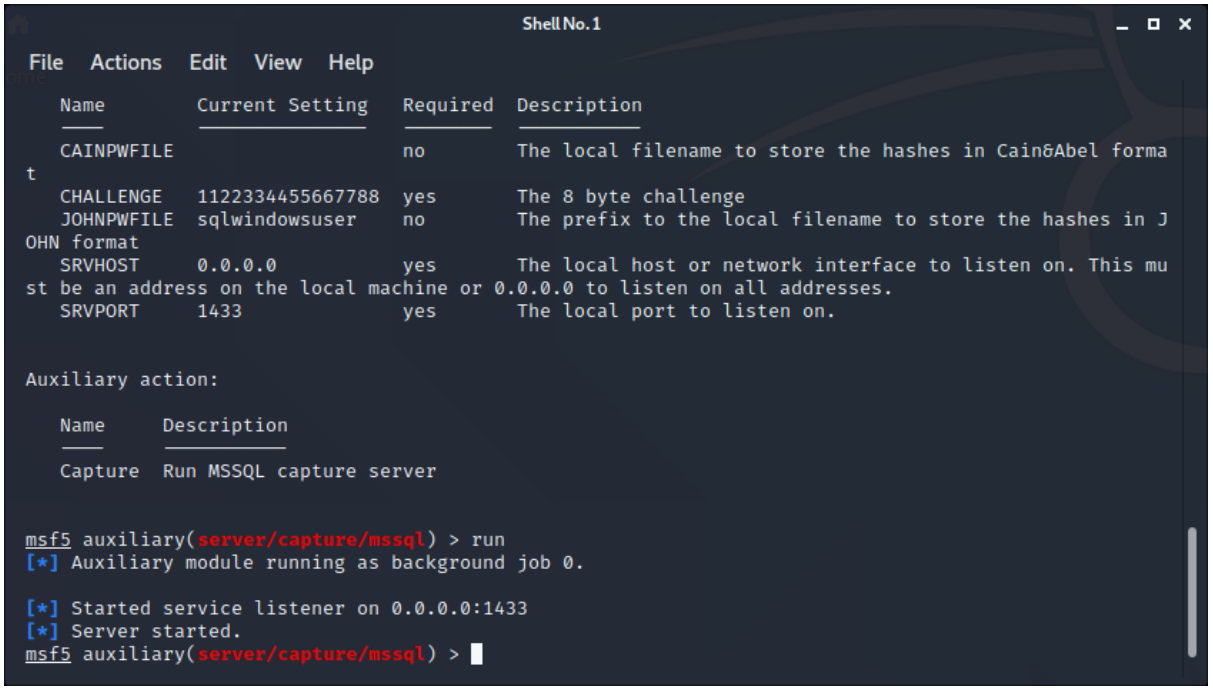
The PC username we use (i.e. the user we used to login to the Windows 10 machine) is “btrisk”. That user’s password is “123456”.



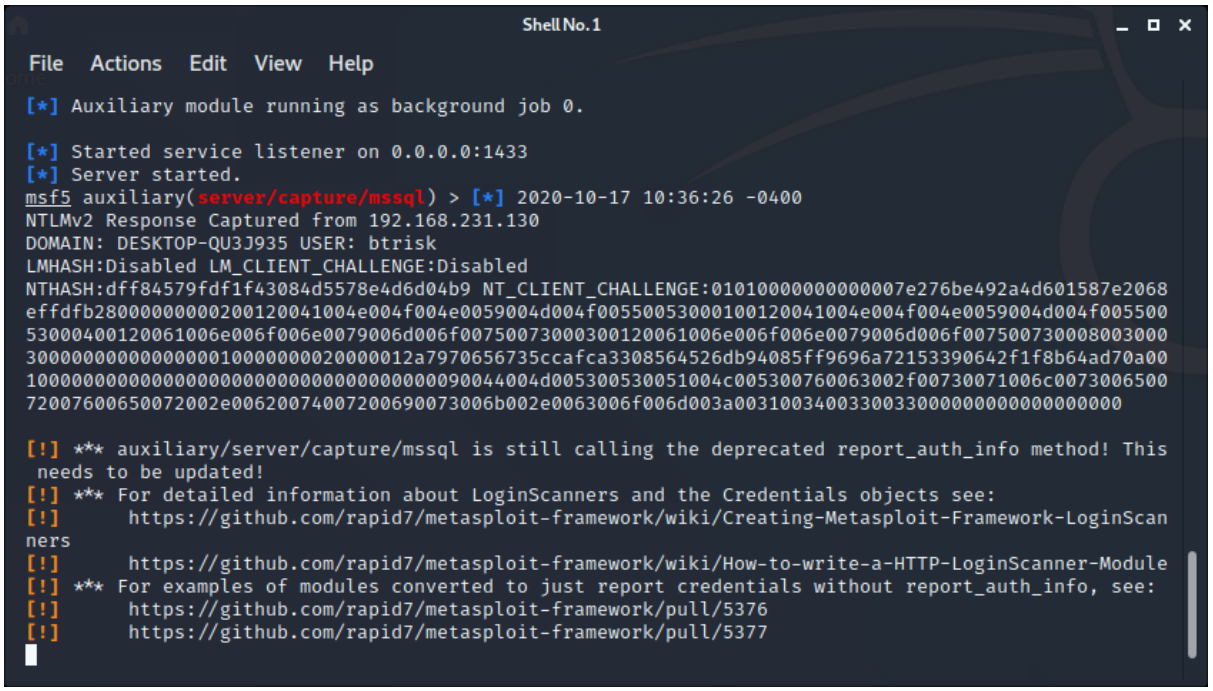
This time we will try to crack the NTLM password hashes. The capture mssql module prints out the captured password hash and other information, but we can use the recording feature of the tool for practicality. We set the prefix for the password hash file “sqlwindowsuser”.



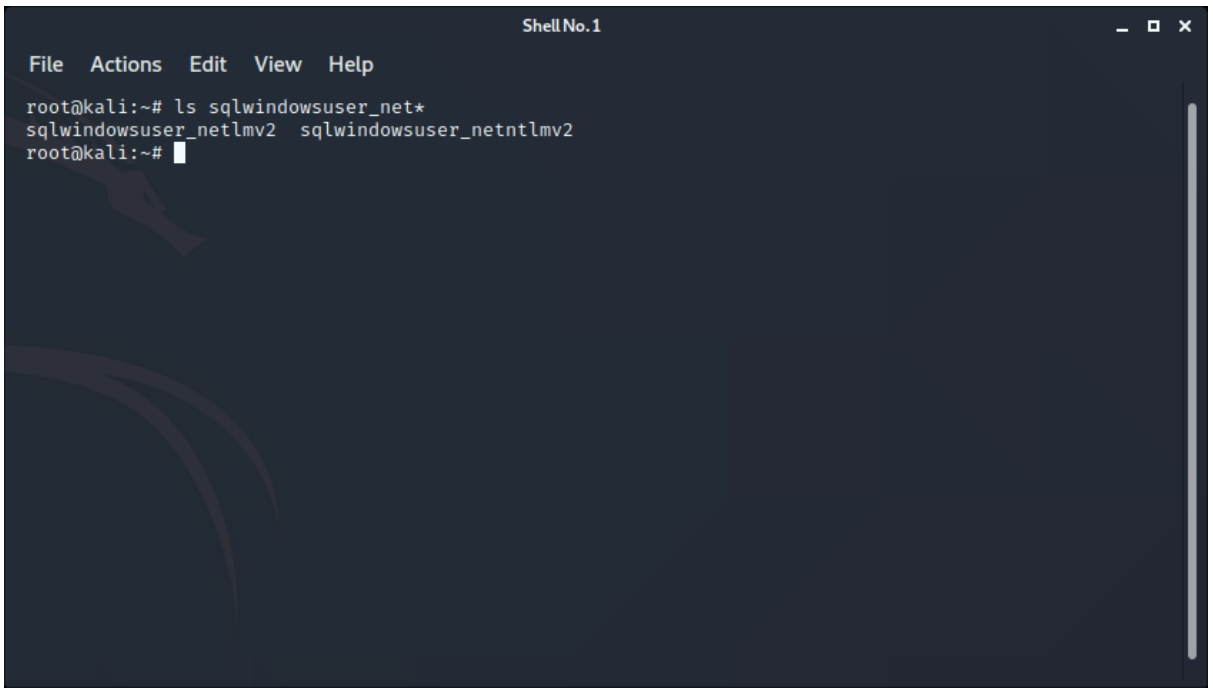
We start our module to imitate the MSSQL service and wait for the victim.



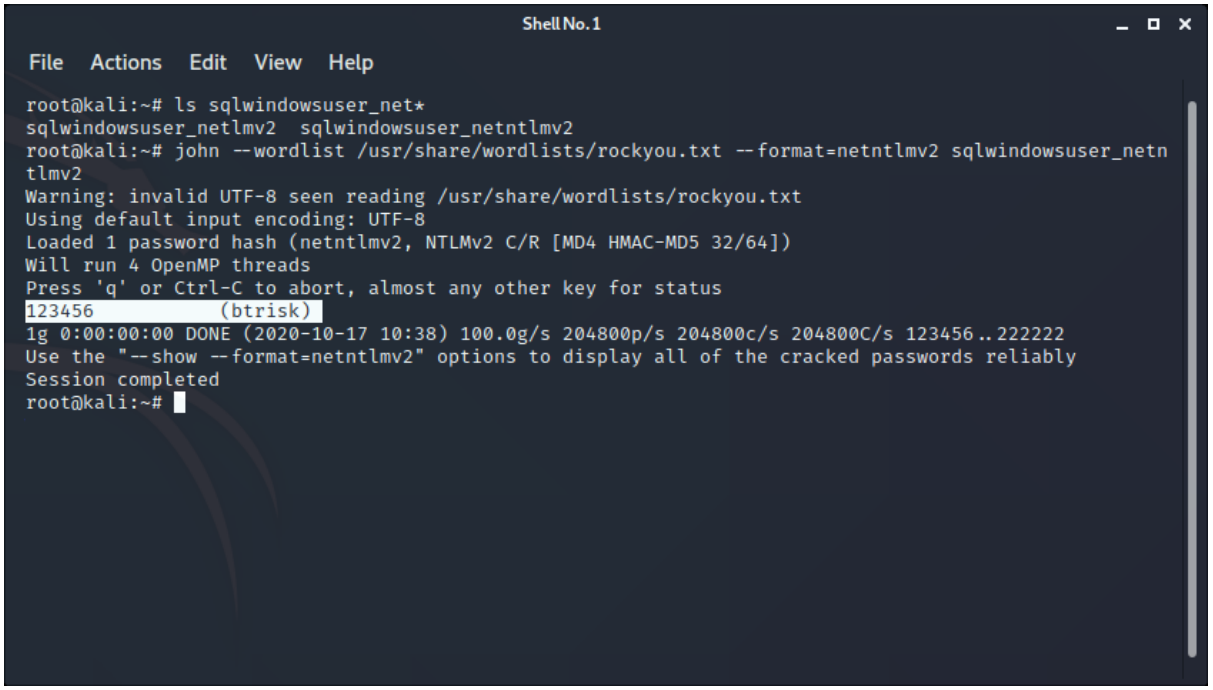
After we run the new client executable, we find the NTLM hash and challenge information both in the console output and the file for which we provided the prefix information before.



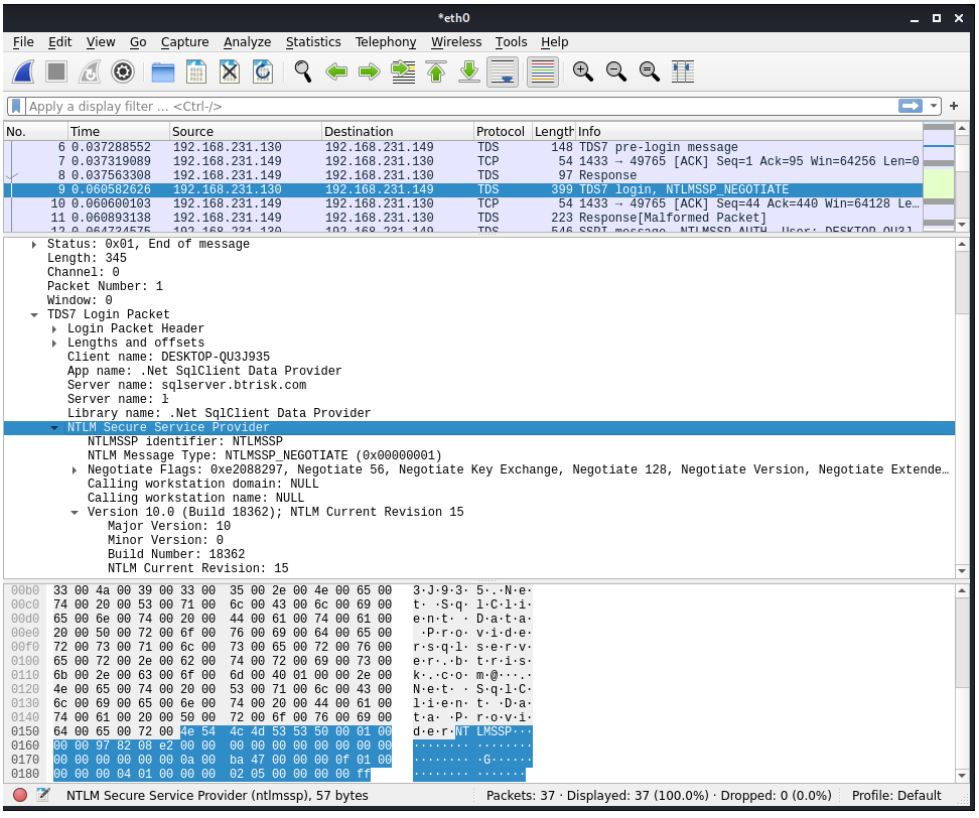
We can see that two files were created for LM and NTLM hashes (although the LM hash file has no information in it).

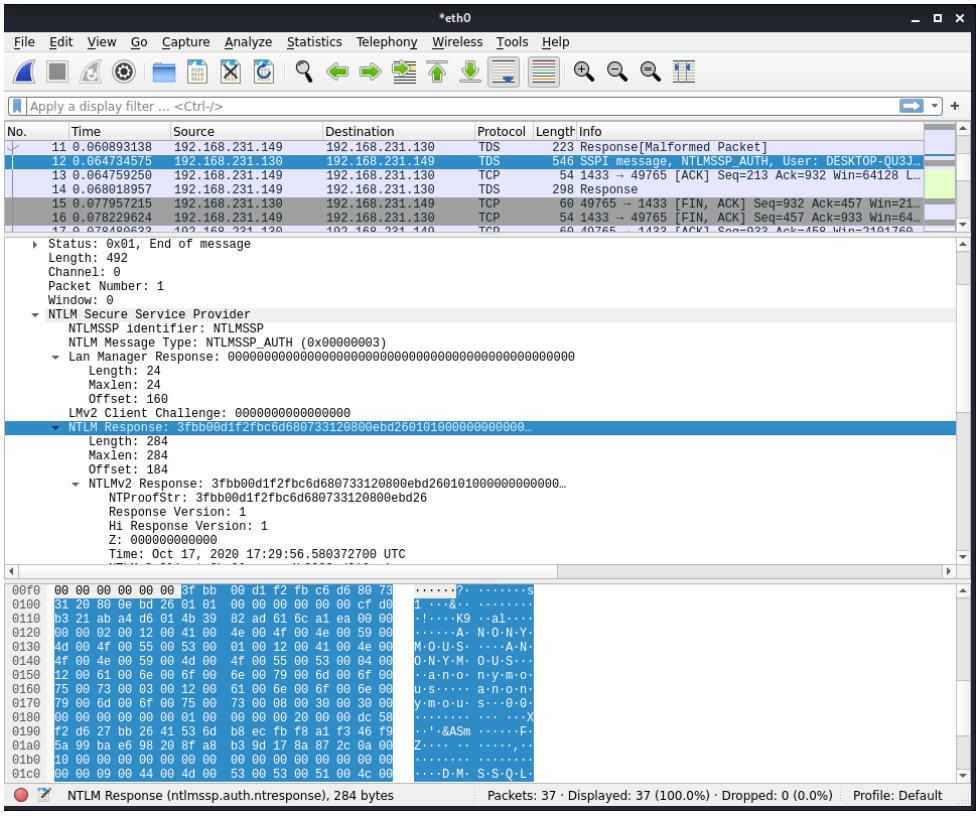


We can give the rockyou wordlist a go with the JTR tool and see that the “123456” password is cracked.



If we analyze the network packets for the Windows authentication trial, we see that the NTLM challenge response packets took place.




*eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
11 0.060893138 192.168.231.149 192.168.231.130 TDS 223 Response[Malformed Packet]
12 0.064734575 192.168.231.130 192.168.231.149 TDS 546 SSPI message, NTLMSSP_AUTH, User: DESKTOP-QU3J…
13 0.064759250 192.168.231.149 192.168.231.130 TCP 54 1433 → 49765 [ACK] Seq=213 Ack=932 Win=64128 L…
14 0.068018957 192.168.231.149 192.168.231.130 TDS 298 Response
15 0.077957215 192.168.231.130 192.168.231.149 TCP 60 49765 → 1433 [FIN, ACK] Seq=932 Ack=457 Win=21…
16 0.078229624 192.168.231.149 192.168.231.130 TCP 54 1433 → 49765 [FIN, ACK] Seq=457 Ack=933 Win=64…
Status: 0x01, End of message
Length: 492
Channel: 0
Packet Number: 1
Window: 0
NTLM Secure Service Provider
NTLMSSP identifier: NTLMSSP
NTLM Message Type: NTLMSSP_AUTH (0x00000003)
Lan Manager Response: 000000000000000000000000000000000000000000000000
Length: 24
Maxlen: 24
Offset: 160
LMv2 Client Challenge: 0000000000000000
NTLM Response: 3fbb00d1f2fbc6d680733120800ebd260101000000000000…
Length: 284
Maxlen: 284
Offset: 184
NTLMv2 Response: 3fbb00d1f2fbc6d680733120800ebd260101000000000000…
NTProofStr: 3fbb00d1f2fbc6d680733120800ebd26
Response Version: 1
Hi Response Version: 1
Z: 000000000000
Time: Oct 17, 2020 17:29:56.580372700 UTC
00f0 00 00 00 00 00 00 3f bb 00 d1 f2 fb c6 d6 80 73
0100 31 20 80 0e bd 26 01 01 00 00 00 00 00 cf d0
0110 b3 21 ab a4 d6 01 4b 39 82 ad 61 6c a1 ea 00 00
0120 00 00 02 00 12 00 41 00 4e 00 4f 00 4e 00 59 00
0130 4d 00 4f 00 55 00 53 00 01 00 12 00 41 00 4e 00
0140 4f 00 4e 00 59 00 4d 00 4f 00 55 00 53 00 04 00
0150 12 00 61 00 6e 00 6f 00 6e 00 79 00 6d 00 6f 00
0160 75 00 73 00 03 00 12 00 61 00 6e 00 6f 00 6e 00
0170 79 00 6d 00 6f 00 75 00 73 00 08 00 30 00 30 00
0180 00 00 00 00 00 00 01 00 00 00 00 20 00 00 dc 58
0190 f2 d6 27 bb 26 41 53 6d b8 ec fb f8 a1 f3 46 f9
01a0 5a 99 ba e6 98 20 8f a8 b3 9d 17 8a 87 2c 0a 00
01b0 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01c0 00 00 09 00 44 00 4d 00 53 00 53 00 51 00 4c 00
NTLM Response (ntlmssp.auth.ntresponse), 284 bytes
Packets: 37 · Displayed: 37 (100.0%) · Dropped: 0 (0.0%)
Profile: Default

Result

During a penetration test if we capture a database application, we have a myriad ways to retrieve database credentials from it. The method explained here with the Metasploit module is another effective method in our tool belt.

Obfuscation and encrypted connection options might mitigate the risk here, but we should not forget that no controls on the client side can be hundred percent effective. We should always suggest a 3-tier architecture as a long-term solution for these kinds of problems.