Lab 4: Reverse Engineering Lab

Signature-based detection is a process where a unique identifier (i.e., a signature) is established about a known threat so that the threat can be identified in the future. In the case of a virus scanner, this is typically a unique pattern of code that attaches to a file, or it may be a hash of a known bad file. Attackers may use a runtime packer, or self-extracting archive, to obfuscate malware in an effort to hide its existence from signature-based anti-virus software. However, we will see in this lab that even after applying obfuscation, software code can still be cracked and viewed.

Today, in this lab we are going to use a tool-based technique to unpack a UPX-packed executable to reveal the original program structure.

First, let’s see how using a runtime packer affects the following "Hello World" program in C++:

Graphical user interface, text, application

Description automatically generated with medium confidence

Type up this small program as HelloWorld.cpp on your Windows machine and compile, calling the resulting executable HelloWorld.exe.

**IDA Freeware**

If *IDA Freeware* is not already downloaded on your Windows machine:

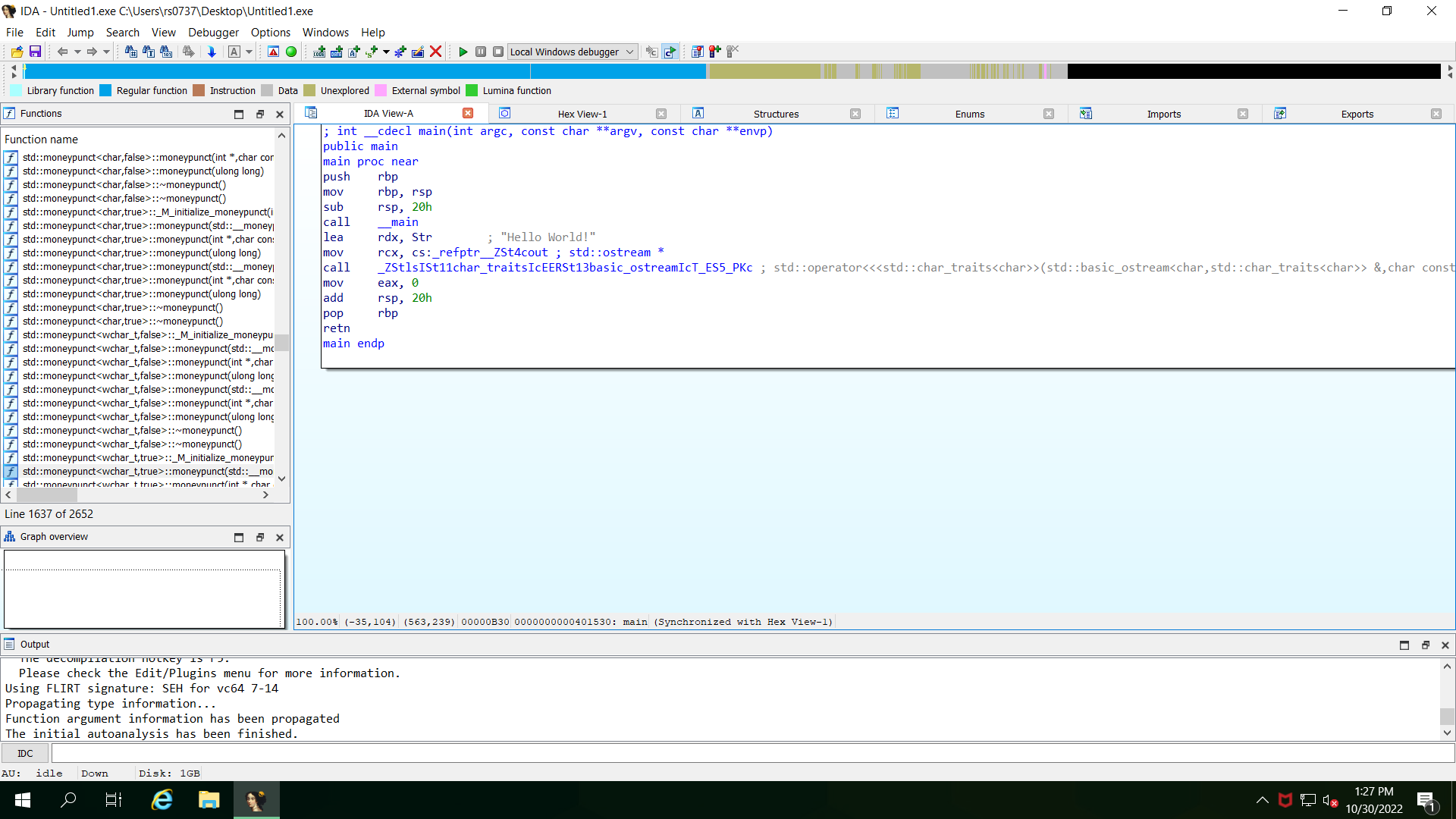
* + Go to <https://hex-rays.com/ida-free/#download>
  + Select *IDA Freeware for Windows*, which should start the downloading process.
  + Install on your machine (following the instructions).

We will use *IDA Freeware,* an interactive disassembler often used in reverse engineering, to see how packing software affects the executable.

Let’s disassemble the HelloWorld.exe file to see what it looks like:

* + Start *IDA Freeware*.
  + Click *New* to "Disassemble a new file" and select the HelloWorld.exe file.
  + In the *Load a new file* dialog box, select *Portable executable for…* and leave all other defaults as is. If you get a prompt to load DWARF debug information, leave the defaults as is and select *Yes*.
  + Under the IDA View-A tab on the right, you should see some assembly instructions with the "Hello World!" text clearly visible.
* Note that if you do not see the assembly code with the "Hello World!" text in the IDA View-A tab on the right, you can select the \_main function name in the left *Functions* box, which should make it visible in the IDA View-A tab.

**Q1: Attach a screenshot of the entire *IDA Freeware* application, including the title bar (with your EUID and the name of the file), *Function* box, and assembly code in IDA View-A tab.**



* In *IDA Freeware*, select *File* 🡪 *Close* to close the executable file in IDA Freeware, and when prompted to save the database, check the *DON'T SAVE the database* box and click *OK*.

You can leave the *IDA Freeware* application open as we will be back to using it in just a little bit.

**Ultimate Packer for eXecutables (UPX)**

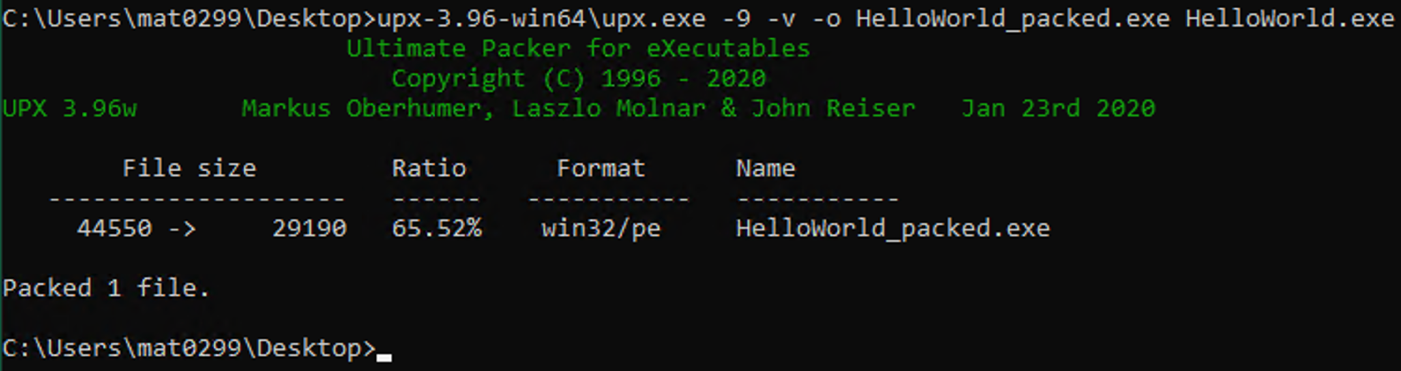
If *UPX* is not already downloaded on your Windows machine:

* + Go to <https://github.com/upx/upx/releases/tag/v3.96>
  + Select the appropriate version for your machine (e.g., upx-3.96-win64.zip), which should start the downloading process.
  + Extract the ZIP file to a selected location on your Windows machine.

We will useUltimate Packer for eXecutables (*UPX*), a free, opensource executable packer used to reduce the size of programs and DLLs, to "pack" our executable in an effort to obfuscate the code’s footprint (i.e., signature). Programs and libraries compressed by UPX are completely self-contained and run exactly as before, with no runtime or memory penalty for most of the supported formats.

* Pack the HelloWorld.exe executable using *UPX* using the –9 compression level option as shown in the example, saving the resulting packed executable as HelloWorld\_packed.exe.

If successful, the results of the *UPX* command should look something like this, where the file size is visibly reduced:



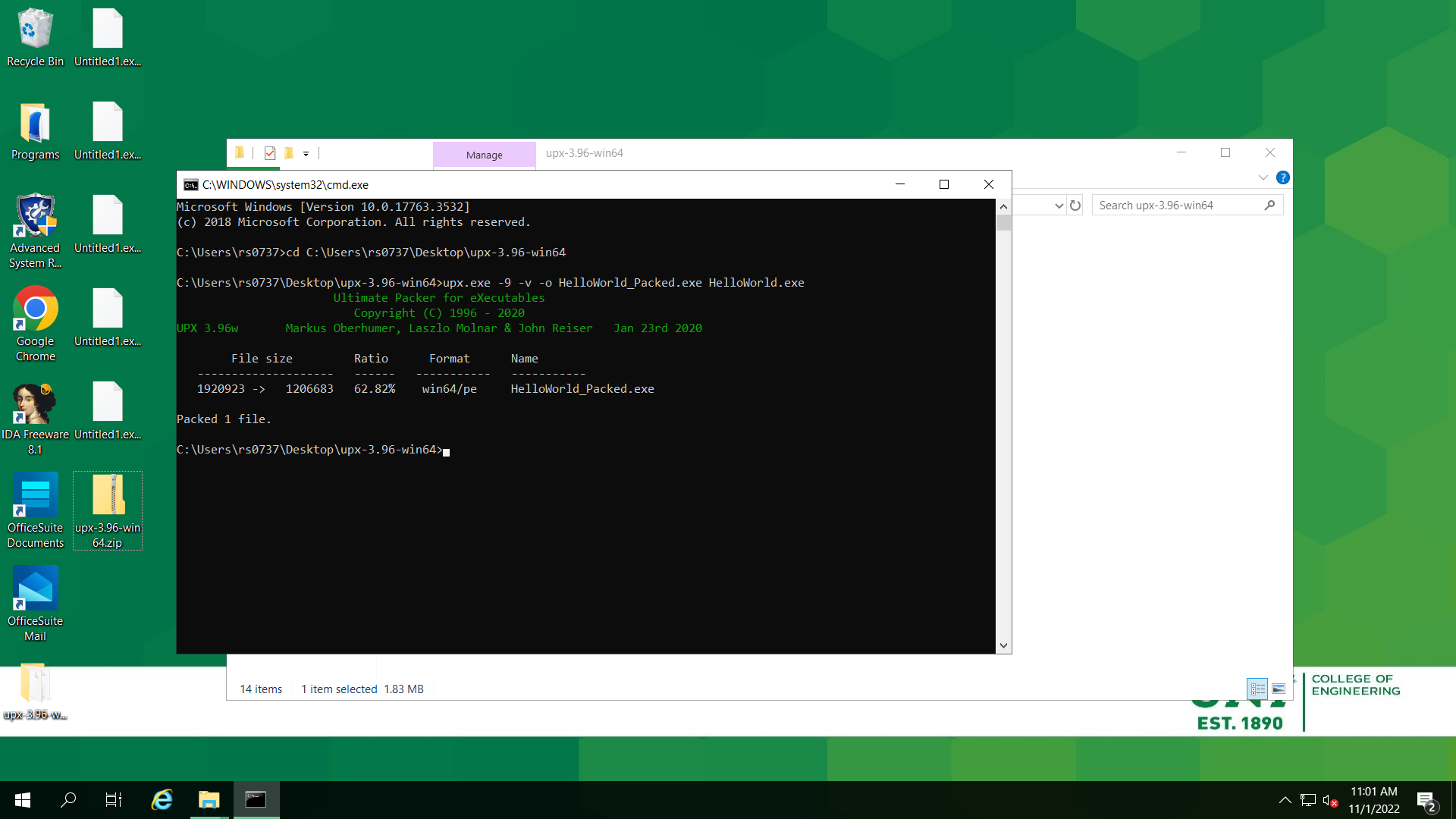
* Now, open up this new packed executable HelloWorld\_packed.exe in *IDA Freeware* using the same options as before. You should notice a warning similar to the following, which indicates that the executable is most likely "packed". Click OK.

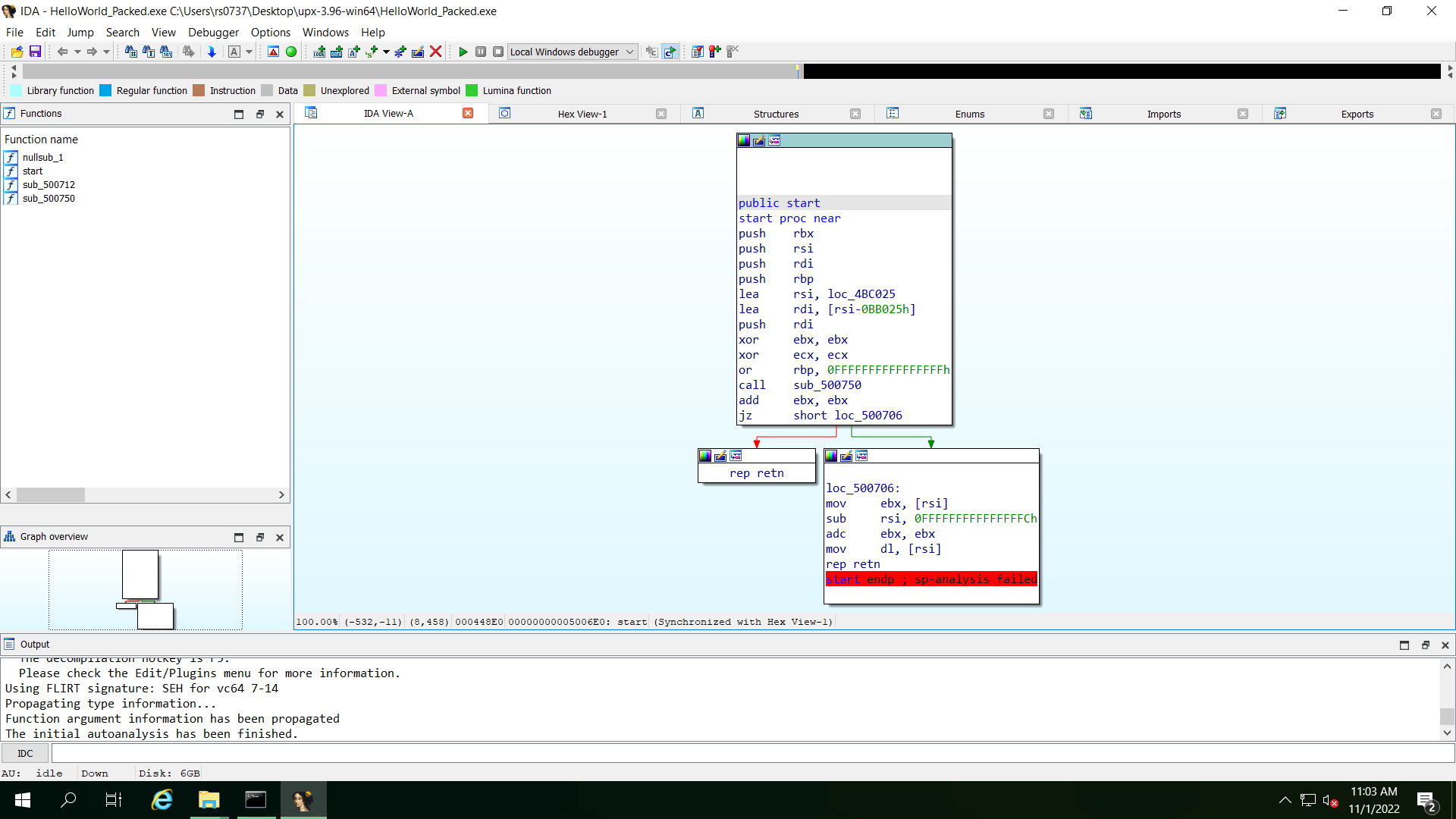
Graphical user interface, text, application

Description automatically generated

You should notice that the sections are reduced to maybe a couple UPX segments, which means that it may be more difficult (but not impossible) to determine what this executable does!

**Q2: Attach a screenshot of the *Functions* box showing the UPX segments in the *IDA Freeware* application.**





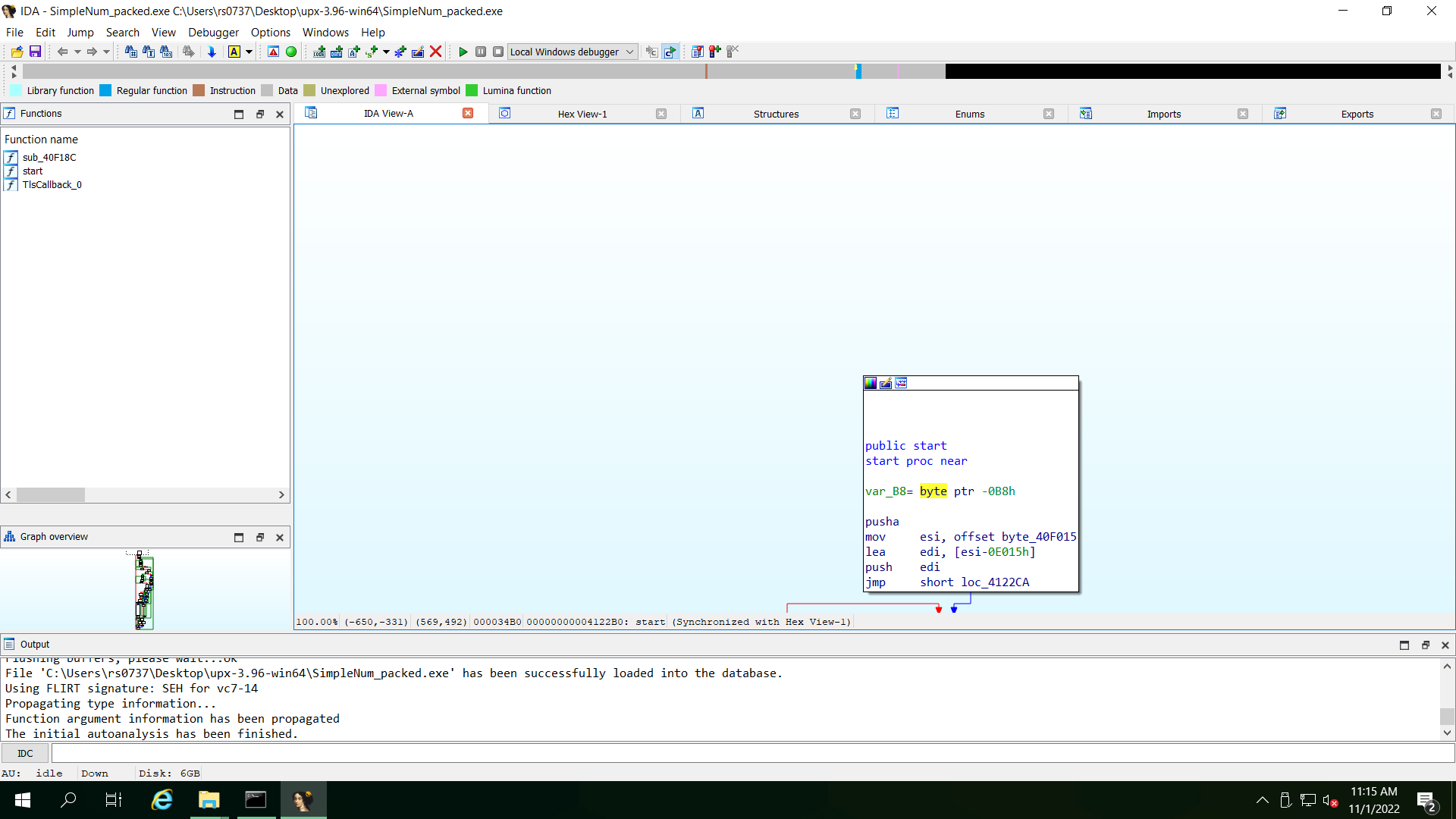
* In *IDA Freeware*, select *File* 🡪 *Close* to close the executable file in IDA Freeware, and when prompted to save the database, check the *DON'T SAVE the database* box and click *OK*.

Download the SimpleNum\_packed.exe file to your Windows VM and run it to see what it does, paying close attention to the strings that are displayed to the terminal.

* With the *IDA Freeware* application still open, select *File* 🡪 *Open* to disassemble the given file SimpleNum\_packed.exe, using the same options as before. Again, you should notice a warning that indicates that the executable is most likely "packed" as before.

Verify that the sections are reduced to maybe a couple UPX segments similar to what you observed in your HelloWorld\_packed.exe file.

**Q3: Attach a screenshot of the one assembly code box in the IDA View-A tab with the text sp-analysis failed, meaning that the stack analysis failed since the stack pointer does not point to the address expected by the function type (since the executable was packed).**



* In *IDA Freeware*, select *File* 🡪 *Close* to close the executable file in IDA Freeware, and when prompted to save the database, check the *DON'T SAVE the database* box and click *OK*.
* Now, exit the *IDA Freeware* application.

**x32dbg**

If *x32dbg* is not already downloaded on your Windows machine:

* + Go to <https://x64dbg.com/>
  + Click the Download button, and then select the appropriate version for your machine (e.g., snapshot\_2022-10-18\_22-09.zip), which should start the downloading process.
  + Extract the ZIP file to a selected location on your Windows machine.

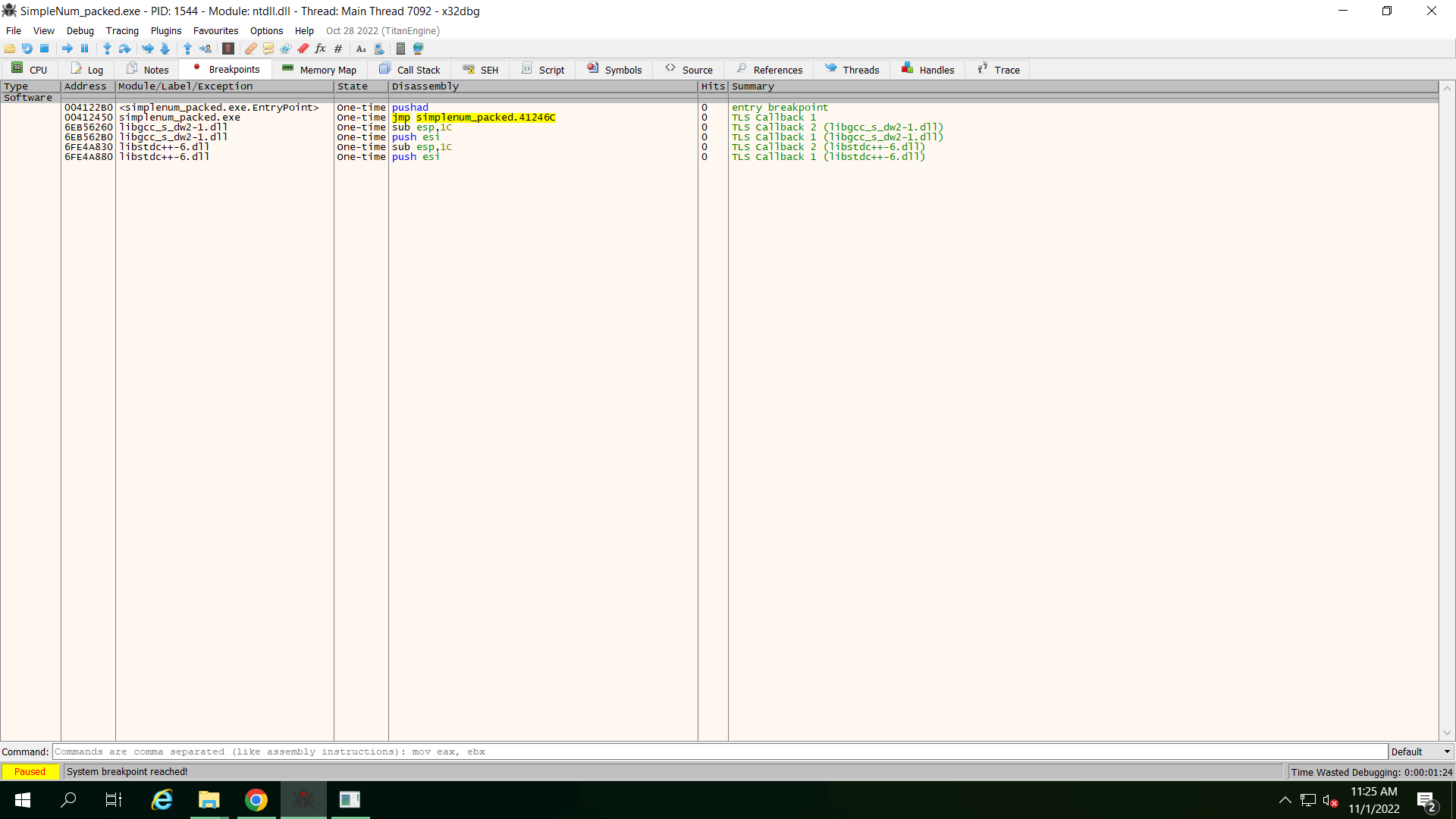
We will use *x32dbg*, which is an open-source binary debugger for Windows that is aimed at malware analysis and reverse engineering of executables that you do not have the source code for. In particular, we want to use a "breakpoint trick" to see if we can restore the original binary. And while this should work with almost any debugger, we will use *x32dbg* for this one.

Let’s run this debugger on the SimpleNum\_packed.exe file to see what it looks like:

* + Start *x32dbg*.
  + In *x32dbg*, select *File* 🡪 *Open* to open the SimpleNum\_packed.exe executable file.

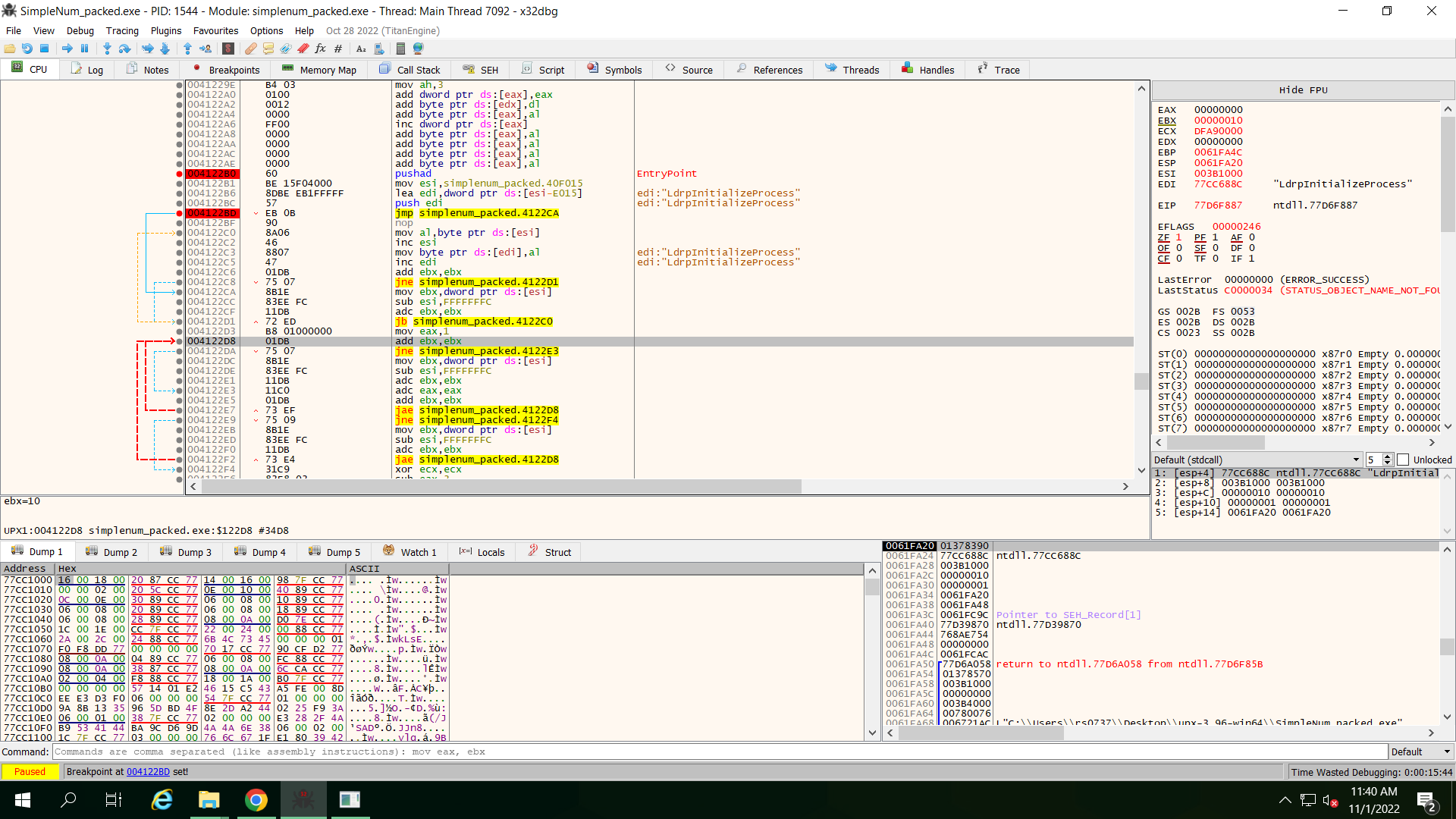
When loading is complete, you should initially be positioned at the Extended Instruction Pointer, EIP, under the CPU tab. However, x32dbg will put a breakpoint at pushad or *EntryPoint* of your binary by default that pushes the contents of the general-purpose registers onto the stack. To see this entry breakpoint, simply select the Breakpoints tab.

**Q4: Attach a screenshot of all of the default breakpoints in the debugger for this executable. Be sure to include the x32dbg window header information with the name of the executable and process ID (i.e., PID).**



* Now, in your list of breakpoints, double-click on the pushad operation to take you to this location.
* Then, scrolling down the binary a little bit (about 178 lines or so), search for the first jmp perhaps only a few lines down from the popad and set your breakpoint there.
* The popad operation pops all general-purpose registers. You can set the breakpoint by placing your cursor on the currently gray dot to the left of the instruction address for the jmp instruction (hovering over the gray dot should reveal a pop-up window that indicates "Breakpoint Not Set"). Click the gray dot until it turns red, indicating "Breakpoint Enabled". This action should also turn the instruction address red.

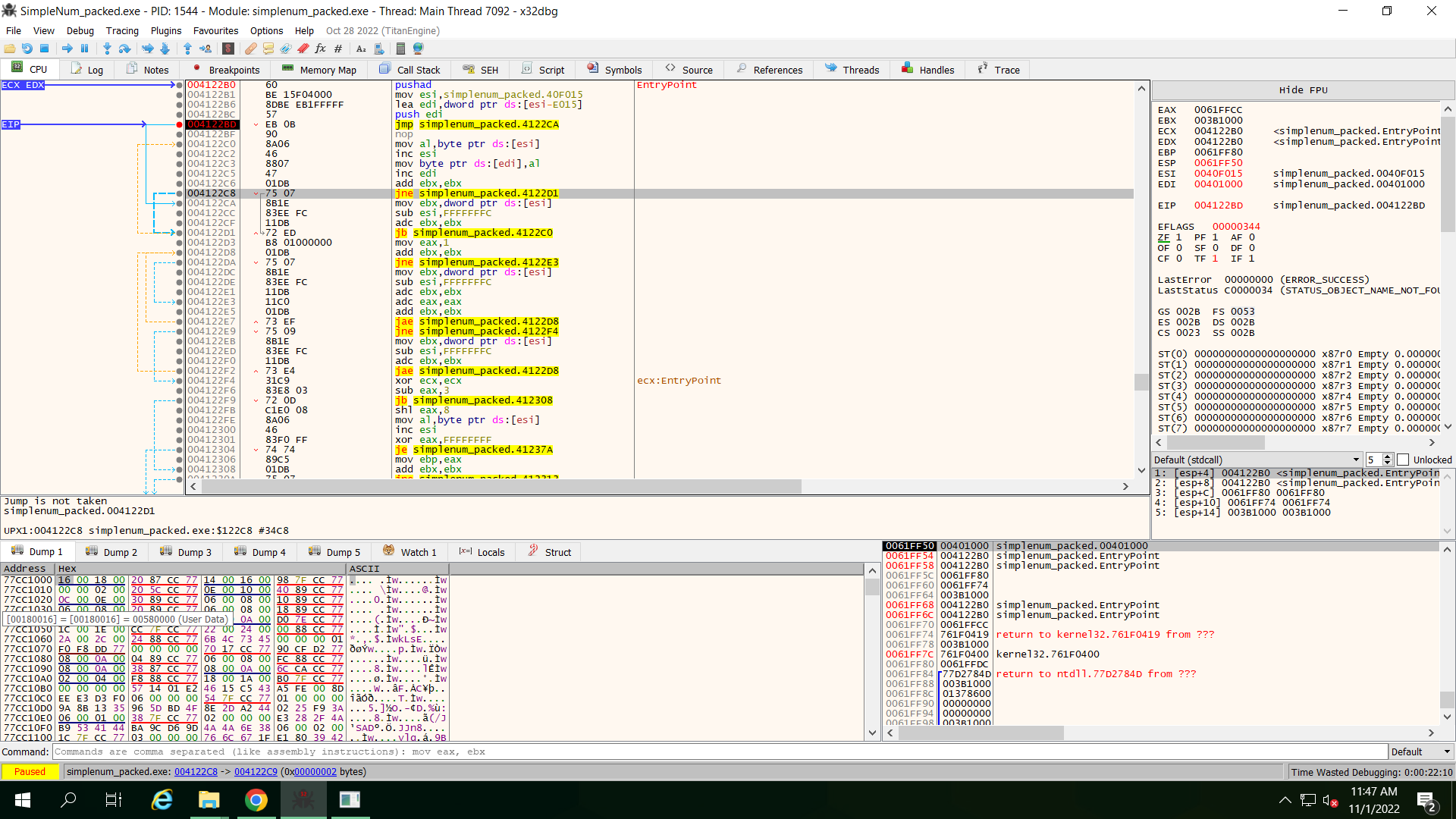
**Q5: Attach a screenshot of the binary instructions from the popad instruction to the jmp instruction where the breakpoint was set. Again, be sure to include the x32dbg window header information with the name of the executable and process ID (i.e., PID).**

**c**

We are going to search for the OEP (Original Entry Point) of the packed binary.

* Run the program until it hits the jmp breakpoint. You can run the program by selecting Debug 🡪 Run, or clicking on the blue arrow facing the right in the *x32dbg* toolbar. Note that you may have to perform Debug 🡪 Run several times to get to your jmp breakpoint, if there are more breakpoints already set.
* Now, single step into the jmp one time, which will take you to the OEP of the packed binary and take note of the instruction address, which should be a sub instruction. This can be done by selecting Debug 🡪 Step into, or clicking on the blue arrow facing down to a blue circle.

**Q6: Attach a screenshot of the binary instructions from the sub instruction to include six lines or so, making sure to include a couple of the comments that indicate EntryPoint. Again, be sure to include the x32dbg window header information with the name of the executable and process ID (i.e., PID).**



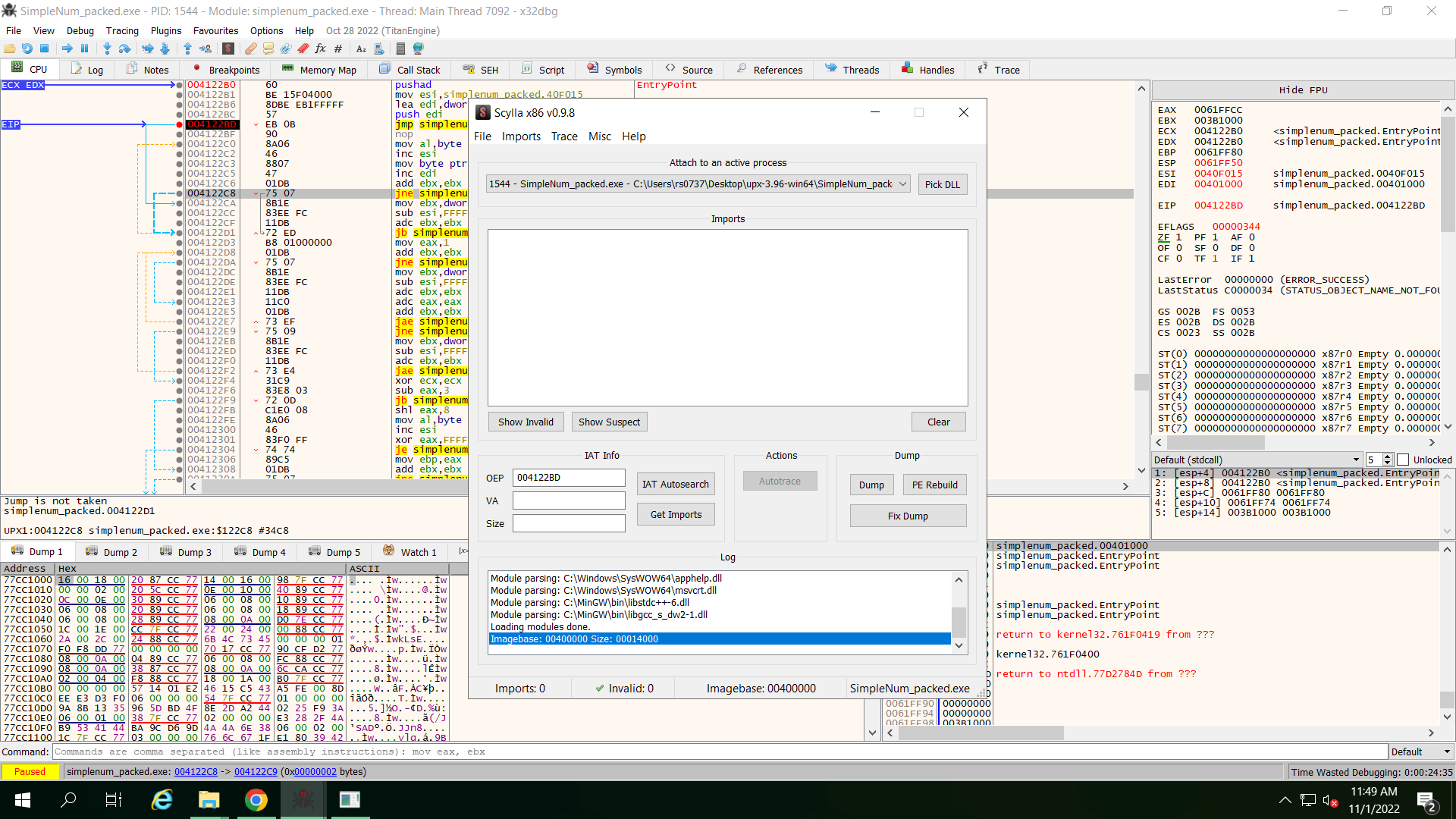
We are going to dump the PE (Portable Executable) starting from its OEP using the *Scylla* plugin for *x32dbg*. This should already be included in your x32dbg and is shown here circled in red:

Graphical user interface, application

Description automatically generated

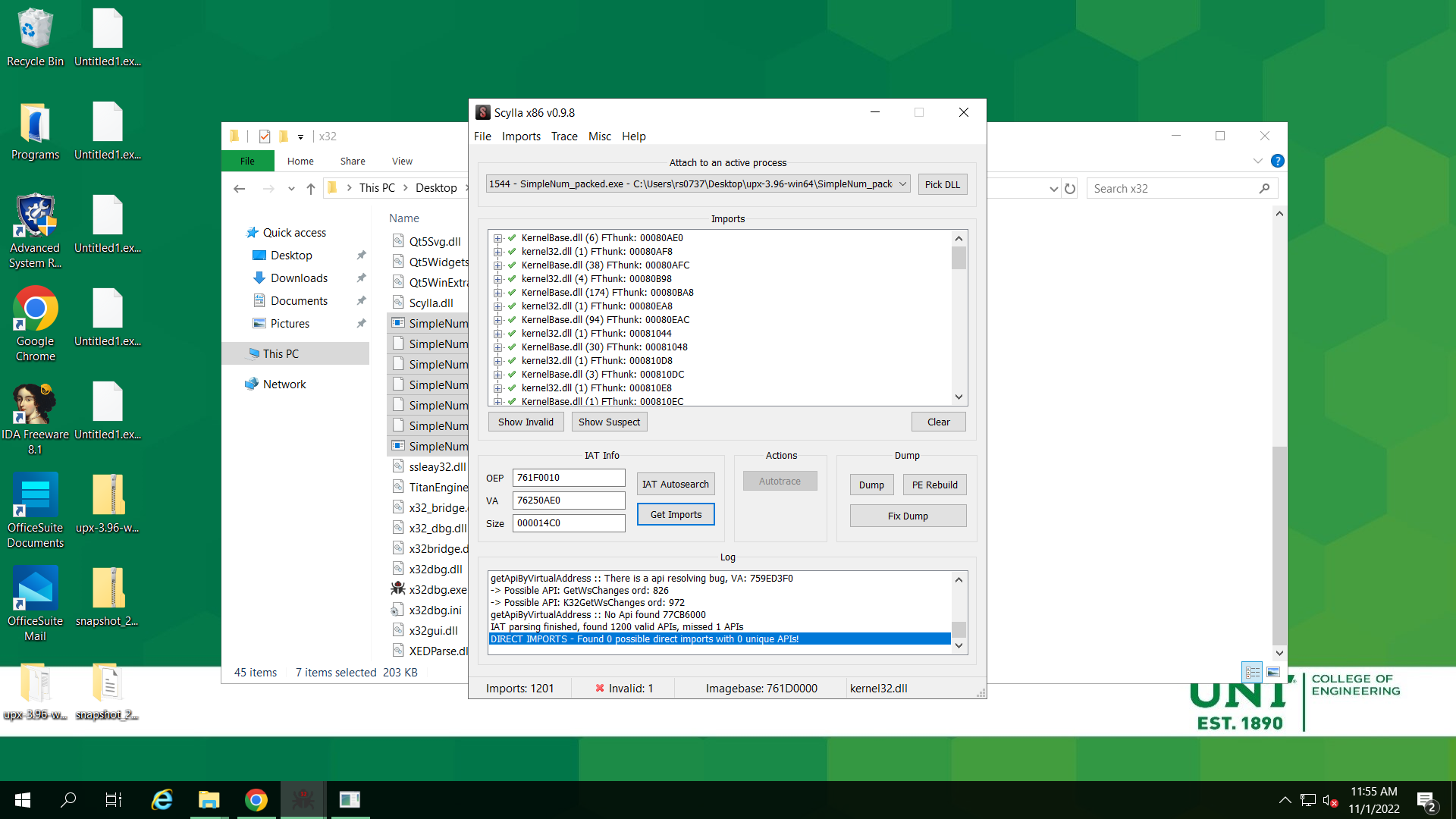
* With the EIP pointing to this address, click on this 'S' button in your toolbar that should bring up a new window for *Scylla*.
* In the Attach to an active process box, select your current process, which should be for SimpleNum\_packed.exe and make sure your OEP is correct (i.e., the instruction address you found when stepping into the jmp instruction).

**Q7: Attach a screenshot of the Scylla dialog box, showing the current process selected as well as the correct OEP address.**



* Then, click the IAT Autosearch button to scan for imports. Click Yes when prompted for advanced results. Click OK in the IAT found dialog box.
* When completed, click the Get Imports button to list all the imports found.

**Q8: Attach a screenshot of the Scylla dialog box, showing the results of the Get Imports listing.**



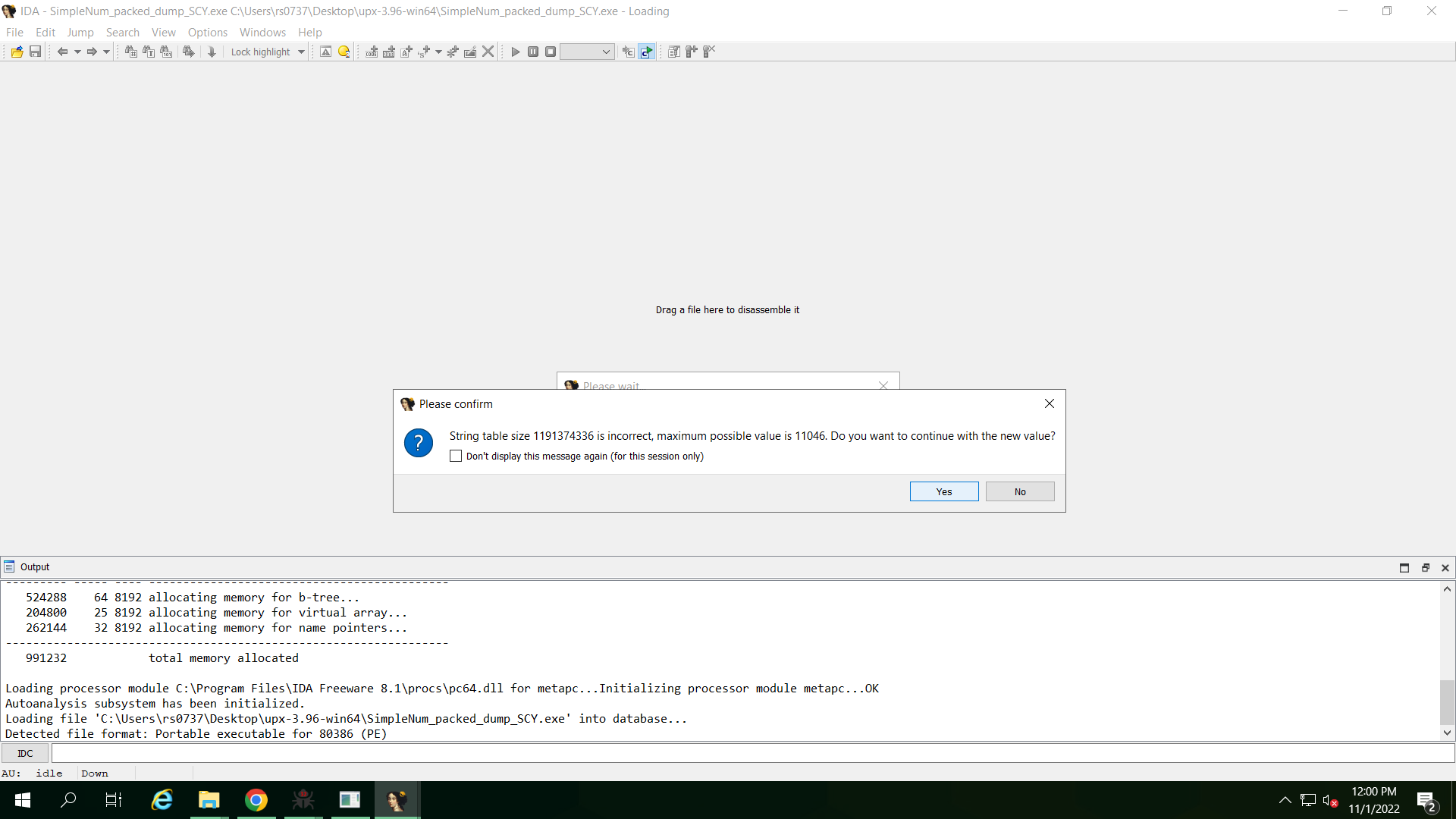
* Click the Dump button to dump the extracted binary. This should bring up a Save As dialog box. Save the binary file as SimpleNum\_packed\_dump.exe, which should already be filled in.
* Finally, click the Fix Dump button and select the recently dumped SimpleNum\_packed\_dump.exe binary to fix its IAT.

You should now have the SimpleNum\_packed\_dump\_SCY.exe file in your directory, which is the final unpacked binary that we can open in *IDA Freeware* and compare the call graphs.

* Start *IDA Freeware* and then click *New* to "Disassemble a new file" and select the SimpleNum\_packed\_dump\_SCY.exe file.
* In the *Load a new file* dialog box, select *Portable executable for…* and leave all other defaults as is. Although IDA Freeware still indicates that the file was most likely packed, click OK as we shall see otherwise!

Unlike before (when we opened the original SimpleNum\_packed.exe file in *IDA Freeware*), we should see a number of functions in the left Function name box. Remembering the strings you noted from running the packed version of this executable, double-click on the functions in the list until you find the call graph that includes all of the original strings found in the executable, which shows our original program structure!

**Q9: Attach a final screenshot of the Functions box and call graph, with the "function" selected that shows the call graph of the function with the original strings from the program. Again, be sure to include the x32dbg window header information with the name of the executable and process ID (i.e., PID).**



Graphical user interface, application

Description automatically generated

* In *IDA Freeware*, select *File* 🡪 *Close* to close the executable file in *IDA Freeware*, and when prompted to save the database, check the *DON'T SAVE the database* box and click *OK*.
* Now, exit the *IDA Freeware* application.