

COS20030

**Malware Analysis**

*Lab 5*

**Unpacking**

# Purpose

In this lab, we are going to unpack a packed file. The use of packers is a very common technique for obfuscating malicious code.

Although there are various off-the-shelf packers that offer both packing and unpacking capabilities to eliminate the need for unpacking a file manually, when it comes to malicious files, this is not always the case. When dealing with packed malware samples, more often than not, a custom packer is used and we need to unpack the file manually. Therefore, recognising whether a file is packed and finding a way to unpack it is an important skill every malware analyst should have.

# Outcome

* Learn how to recognise a packed file
* Learn what is the purpose of stub and how it can be used in progressing in the unpacking task
* Use a disassembler to try and find the jump to the Original Entry Point
* Use a debugger to unpack a packed file
* Use the Scylla plugin of X64dbg to dump the unpacked file from memory and reconstruct its Import Address Table and other headers

# 

# Steps in manual unpacking

Manually unpacking a file is done in a number of steps:

1. Looking for distinctive signs of packed file and potentially recognizing the packer
2. Inspecting the disassembled code of the unpacker stub and finding the jump to the Original EntryPoint (OEP)
3. Executing the program in a debugger until it reaches the OEP
4. Dumping the unpacked code from memory to a file
5. Reconstructing the Import Address Table (IAT) and other headers of the unpacked file

Now let’s do these steps one by one.

# Step 1: Look for signs of a packed file

File Overview and Basic Static Analysis to gather initial information about the file without executing it, potentially revealing signs of packing.

1. Extract the lab5\_unpacking.exe\_ from the zip file using the password “infected”
2. Open the file in DiE to inspect the headers and see the characteristics of the file.

You can see immediately that the tool recognises the packer used for packing this file as “Packer: UPX”.

A screenshot of a computer

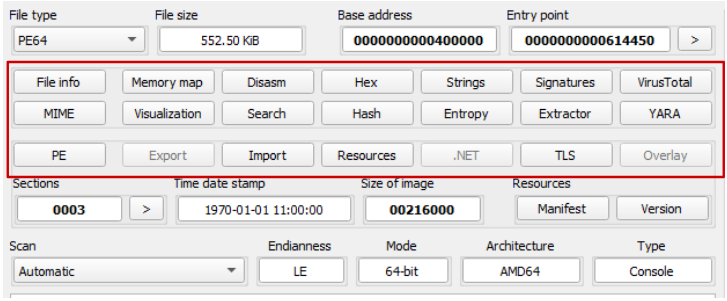
Description automatically generated

UPX is one of the most popular Packers. It’s used on various legitimate software but at the same time, it’s also very popular among malware authors.

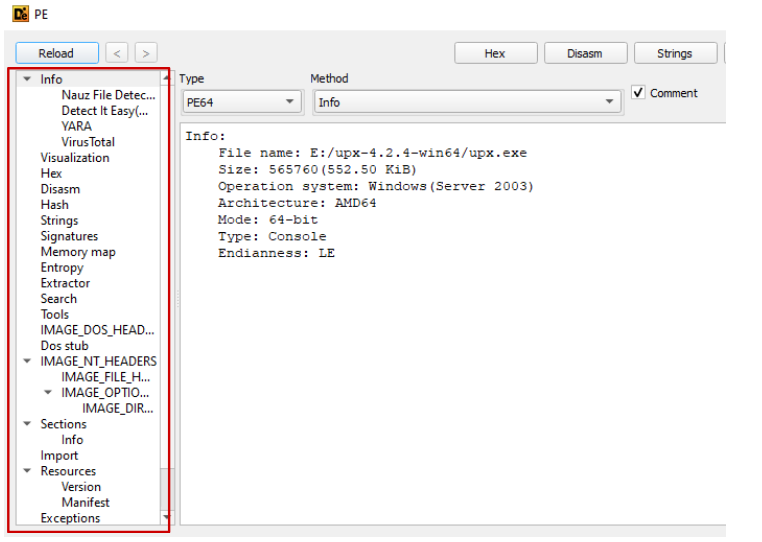
Tools like DiE have signatures for a variety of known packers and can even in some cases tell which version of the packer was used.

However, we need to confirm that the file is actually packed by looking for the signs in the structure of the file.

1. Browse the information extracted from the file by clicking on any of the sections in the middle part of the UI.



1. Navigate through various parts from the left tab to learn about this file.



1. Each question from the table below describes one sign of a packed file. Explore different sections of DiE and answer the following questions. (hint: look at the “Import” tab)

|  |  |
| --- | --- |
| Question | Answer |
| Does this file have less than 3 imported DLLs? | yes |
| What are the two APIs used for dynamically resolving an API? Do you see them in the Import Table of this file? |  |
| Does this file have less than 4 sections? | yes |
| What are the section names? | UPX0, UPX1, and .rsrc |
| Is there a section with SizeOfRawData equal to 0 while having a large VirtualSize? | Yes, UPX0 |
| Are there very few meaningful strings in the file? | A few (header, and informations) |
| Is the Total Entropy of the file over 7? What about any sections? | Yes, its 7.931 |

We have confirmed that the file is packed. Now, Step 1 in unpacking is completed. Let’s go to step 2.

# Step 2: Find the jump to the OEP using GHIDRA

In this task, we locate the beginning of the program's execution, which is likely the start of the unpacking stub. We also identify code patterns or functions related to unpacking and memory manipulation.

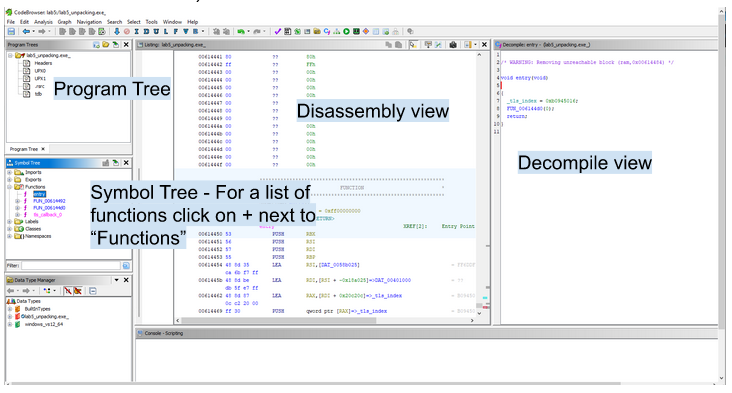
A packed file has a relatively small but a little complex code which is called a stub. This code is responsible for mainly three tasks:

1. Uncompressing the original content of the program from a section (section with a high entropy) and storing the generated values in another section (section with 0 size on disk, but large size in memory)
2. Dynamically resolving API addresses required by the original code
3. Jumping to the OEP (original entry point).

Restoring the OEP could be done by a JUMP instruction, a CALL instruction or some other creative way, even a RET instruction.

Our goal is to find where this happens in the code without actually reading and understanding the instructions.

1. Execute Ghidra via its desktop shortcut.
2. Create a new project and name it Lab5.
3. Import lab5\_unpacking.exe\_ into the project and open it by double-clicking.
4. Allow the file to be analysed by Ghidra using default analysis settings.
5. In Ghidra you can look at the assembly and decompiled instructions side by side. Navigating between functions or just scrolling up and down in each window, results in the other window moving to the corresponding part of the code as well. (See the screenshot below)



The number of functions recognised by Ghidra is very limited. So, let’s quickly navigate through them. (Except for the tls\_callback\_o which is an irrelevant function)

1. From the “Functions” list, choose the functions one by one and skim through their code to find the unpacker stub function.
2. “entry” and “FUN\_664492” have very small routines, therefore don’t seem to contain the functionality to unpack the compressed data and load it in memory.
3. However, “FUN\_006144D0” has a relatively large code compared to the other functions. So this function seems to contain the unpacker routine. Let’s try to find the JUMP to the Original Entry Point.
4. The first thing to notice when looking at the FUN\_006144D0 function’s code in the decompilation window is a warning shown at the top of the decompiled code. (Note: don’t mind the second warning. It has no significance for us)

|  |
| --- |
| /\* WARNING: Control flow encountered bad instruction data \*/ |

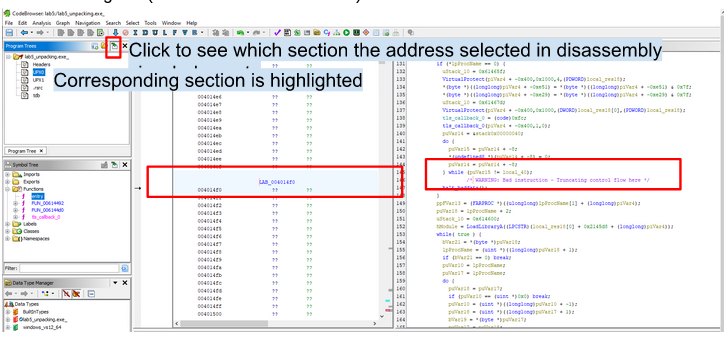
This warning tells us that the control flow of the function encountered some unusual code.

Keep in mind that the OEP address should contain no valid instruction at this point because it’s only going to be populated at runtime when the code is unpacked. A jump instruction to an address without any valid instruction could be a cause of such a warning.

1. Take note of this and scroll down in the decompiled code to find where the control flow encountered such a situation.

Somewhere in the middle of the function, you will see another warning about truncating control flow.

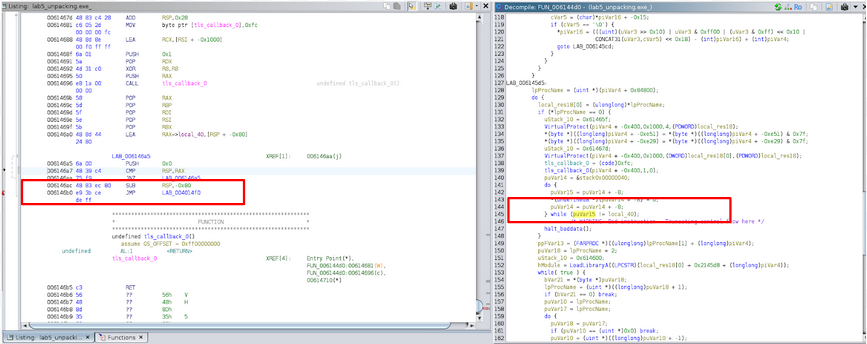
1. Click on the warning line. You can see that in the disassembly window, you are taken to an address with no valid code. Take note of this address, it might be what we are looking for. (See the screenshot below)



Now that we found a jump to an address that has no valid code in the file, let’s also see which section of the PE file this address belongs to.

1. Click on this address. Then click on the button with a green arrow in the Program Trees window to see the section this address belongs to becomes highlighted. (See the screenshot above)
2. Now, click on the line of code above the warning in the decompiled code to see the corresponding disassembly code.

You will see a JMP instruction which is in fact the last instruction of the current function in the disassembly view.



1. Take note of the address where the JMP instruction jumps to. This address should be the same as what we saw a little bit earlier corresponding to the warning line.
2. Check which section the current code belongs to (Instruction similar to step 14).
3. It seems that we found the address of the OEP. Answer the following questions to help confirm that this is the address of the OEP. (hint: locating and analyzing the JUMP, and Understanding the XREF Notation are very important to find the correct answers)

|  |  |
| --- | --- |
| Question | Answer |
| Write down the address of the JUMP instruction | 006146b0 |
| Write down the address where it jumps to(address of OEP). | LAB-004014f0 |
| Is this address in a different section than the unpacker stub code? | yes |
| Which section this address belongs to? Is this the section with a large size in memory and zero size on disk? | UPX0 , yup |
| Is there any function or instruction there? Or does it seem to be populated dynamically? | no |

Voila! We found the jump to OEP! Now, keep Ghidra open and proceed to step 3 of unpacking.

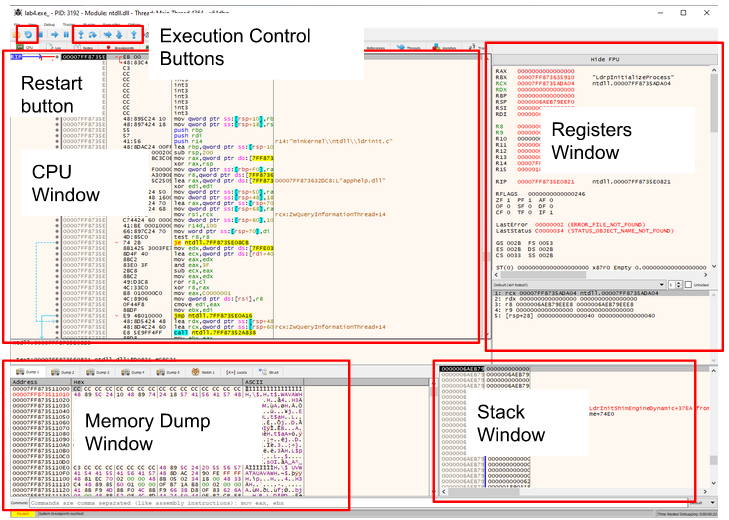
# Step 3: Unpack the file with a debugger

The goal of this section is to execute the packed program in a debugger and create a breakpoint on the JUMP instruction. Once the code is executed until the breakpoint, the original code is already unpacked in memory.

1. Execute the X96dbg.exe debugger via its desktop shortcut and choose “x64dbg”.
2. From File -> Open, load the lab5\_unpacking.exe\_ file.

This program is a debugger that can be used to debug an executable program and inspect its behaviour.

At any point during the execution of a program in a debugger, you can see the code instructions in the CPU window, the memory content, the values of the registers and also the stack values. (See the screenshot below)



Once the program is loaded, you can debug the code by executing the instructions one after another. You can manage the execution of the program using the following keys or the corresponding buttons. (See the screenshot above)

F7 - Step Into

F8 - Step Over

F9 - Execute until a breakpoint is hit

F2 - Toggle/untoggle breakpoint

F4 - Execute until the position of the cursor

F5 - Run

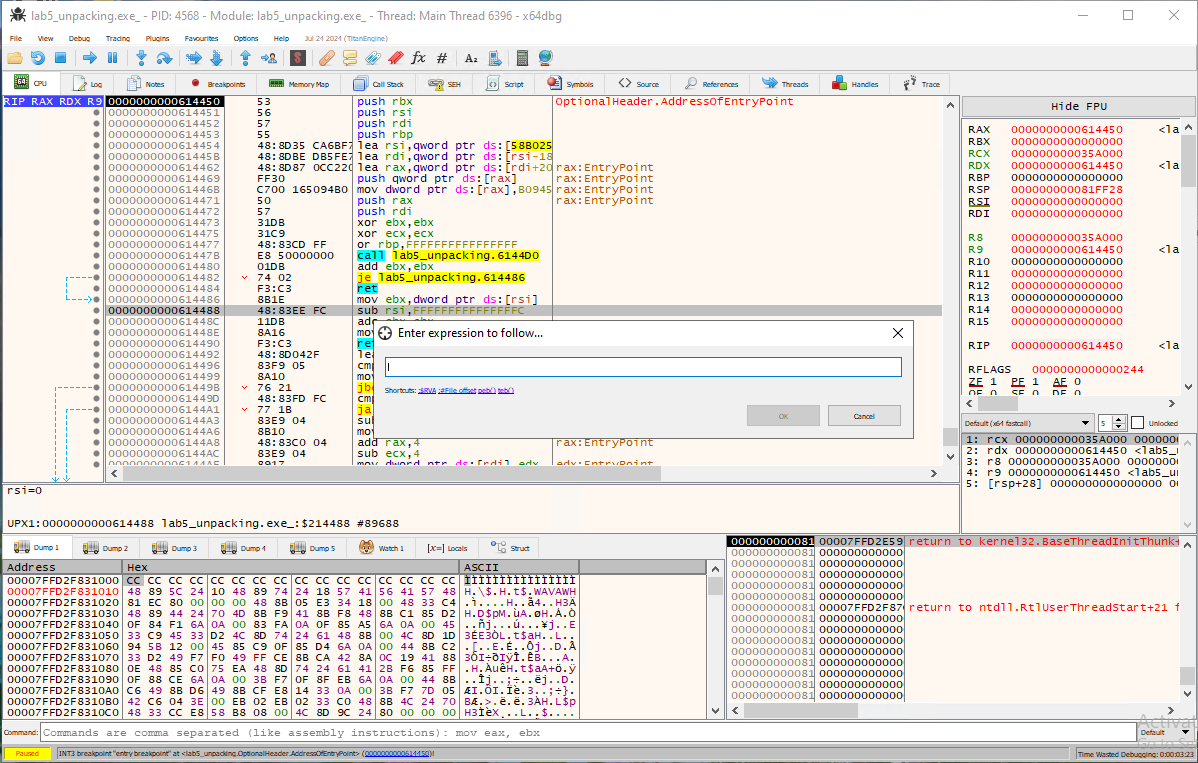
Ctrl+G - Enter an address to jump to

1. The X64dbg debugger has a number of default breakpoints that we don’t need. To remove them, go to Options -> Preferences and in the Events tab, remove all the check marks except for Entry Breakpoint. (See the screenshot below)

A screenshot of a computer program

Description automatically generated

1. Click on Save and restart the program using the restart button or Ctrl+F2.
2. Click inside the CPU window and press Ctrl+G.
3. Enter the address of the JUMP instruction we found in the previous section and click on “OK”. (See the screenshot below)



1. Click on the little circle next to the address of the JUMP instruction from the left side of the CPU window to create a breakpoint. (See the example screenshot below)



If a breakpoint is set, you should see the address highlighted in red. (See the example screenshot below)



1. Run the program by pressing F9 and allow it to stop at the breakpoint.
2. From the Registers window, check the value of the RIP register to confirm that the program execution is stopped at the correct instruction. At any point in execution, the RIP register has the address of the instruction to be executed.
3. Perform one “Step into” to jump at the OEP. Use F7 or the “Step into” button for that.

What you see at this address should be very different from what we found in this address in Ghidra, because the original code is decompressed and copied here at this point.

Now the file is unpacked and step 3 in the unpacking steps is completed.

# Step 4-5: Dump and reconstruct the original file’s headers

Now we need to dump the memory of the program which now contains the unpacked code and then using a plugin of X64dbg called Scylla, reconstruct the headers of the unpacked file. The reconstructed headers will also contain the correct IAT(Import Address Table) of the file.

1. While the execution is stopped at the OEP of the program, open the Scylla plugin by clicking on its icon in the bar above the main tabs. (See the screenshot below)



1. Confirm that the OEP recognised by Scylla is correct. If it is not, you probably have missed a step. Restart the debugger by clicking on the Restart button then go back to the beginning of Step 3 (Step 3: Unpack the file with a debugger) and try again. (Note: Don’t worry about executing the lab sample multiple times, it’s not malicious)



1. Click on “IAT Autosearch” to let Scylla try and find the dynamically resolved API addresses in memory. You will be shown an alert saying IAT was found. This value is needed to reconstruct the original IAT. Click on “OK”
2. Click on “Get Imports” to retrieve all the imported APIs present in the IAT found in memory. If any invalid part is found (with a red X next to the line), right-click on it and choose “Delete tree node”. (See the example screenshot below)



Answer the following questions to confirm that everything has been done correctly so far.

|  |  |
| --- | --- |
| Question | Answer |
| List the DLLs found in memory and their addresses. | Kernel32.dll (0020E55C), msvcrt.dll (0020E784), kernel32.dll (00215614) |
| What is the size of the IAT found by Scylla? | 0x70D0 |
| What is the VA(Virtual Address) of the IAT? | 060E55C |
| Why do you think the same DLL is found in two different addresses? |  |

1. Click on the “Dump” button in the Scylla window and make a memory dump of the program.
2. Click on the “Fix Dump” button in the Scylla window and choose the dump file, Then click on Open.

When you see the “Import Rebuild success” note in the log window of Scylla, our job is finished! (See the example screenshot below)



The unpacked file is stored next to the dump file you created with the \_SCY suffix added to the dump file’s name.

1. Open the unpacked file in DiE and answer the following questions to make sure the whole unpacking is done correctly.

|  |  |
| --- | --- |
| Question | Answer |
| What is the md5 hash of the unpacked file? | 128c91d07d268702b7da164d05345ced |
| How many sections does this file have? | 4 |
| How many DLLs are in the import table? How many APIs are imported from each of them? | 3 |
| What is the Total Entropy of the file? | 6.32094 |
| Are there more than 40 readable strings found in the file? | yes |

*End of Lab*