# Module 6.3

# Anti-emulation techniques

## Emulation

We've covered the two main detection techniques antiviruses rely on. Now it's time for the third approach that underpins the previous two. `Emulation` is the way in which an `AV` simulates the operation of an analyzed program. An antivirus is able to read our encrypted code by emulating it in its memory. One way to beat emulation is to use strong cryptographic algorithms that make emulation complex and time-consuming. Some antiviruses 'give up' when faced with this challenge. Algorithms that do just that are for example `AES`, `RC4` and `Twofish`. Implementing `RC4` is relatively simple and within the grasp of even inexperienced developers. On the other hand, implementing `AES` or `Twofish` without a background in cryptography is strongly discouraged; attacks on implementations of algorithms have been recently occurring more often than actual algorithm attacks.

## RC4 overview

`RC4` is a stream cipher. First, it generates a 256-element identity permutation. Every item in these arrays has a value identical to the element number (the array is marked with `S`).

Or, `S[0]=0; S[1]=1;` etc.

In pseudocode:

```
for i from 0 to 255
S[i] := i
```

The next step is permutating the array, or 'swapping' its elements. The permutation consists of 256 swaps of `S[i]` and `S[j]`. For each iteration, the variable `i` is incremented by `1`, and `j` can be computed from the following equation:

```
j=j+S[i]+key[i%sizeof(key)]%256
```

In pseudocode:

```
j := 0
for i from 0 to 255
   j := (j + S[i] + key [i mod key_length]) mod 256
   swap(S[i],S[j])
```

As you can see, at this point the initial array is combined with an encryption key.

Once the `S`-array, or the `IV` (Initial Vector) is ready, the encryption can begin. Encryption is a process of generating values from 0 to 255 (1 byte) and `xor`-ing the current state with the next values of data to be encrypted.

The algorithm generating the values:

```
i := 0
j := 0
while CreatingCipherStream:
   i := (i + 1) mod 256
   j := (j + S[i]) mod 256
   swap(S[i],S[j])
   result S[(S[i] + S[j]) mod 256]
```

In all iterations the output is `xor`-ed with the next data byte.

As all algorithms, `RC4` has both advantages and problems. On the plus side, it's incredibly simple to implement while providing relatively high security (although decidedly not on the level of the more sophisticated algorithms like `AES` and `Twofish`). For our needs here, using a stream cipher is a plus. We don't have to worry about the last block, which doesn't have the required size

for a full data block (needed in block ciphers). Likewise, agonizing over the encryption mode to use (block ciphers have more than 10 of them) is no longer a trouble.

Emulation and code execution are not the same. We can build code that will run and produce an output we anticipate and know in advance. Meanwhile, an antivirus can emulate the program and produce a radically different output. The code below is an example:

```
FARPROC ZwR=
GetProcAddress(GetModuleHandle("ntdll.dll"),"ZwResumeThread");
FARPROC ZwS=
GetProcAddress(GetModuleHandle("ntdll.dll"),"ZwSuspendThread");
DWORD sys_res,sys_sus;
memcpy(&sys_res,(char*)((char*)ZwR+1),4);
//Get the syscall number of ZwResumeThread
memcpy(&sys_sus,(char*)((char*)ZwS+1),4);
//Get the syscall number of ZwSuspendThread
DWORD old;
VirtualProtect(ZwS,5,PAGE_EXECUTE_READWRITE,&old);
//Set the writing permission for ZwSuspendThread
memcpy(((char*)ZwS+1),&sys_res,4);
//change the syscall number of ZwSuspendThread to ZwResumeThread
SuspendThread(GetCurrentThread());
memcpy(((char*)ZwS+1),&sys_sus,4);                    //Restore the syscall number
```

First, let's learn more about `syscalls` and their ordinals. A `syscall` is how a program calls a function from the kernel. The system prohibits programs from accessing the kernel and its internals directly, and due to this, an alternative communication pattern needs to exist to tell the kernel for example to create a process or manage threads. This role is filled by `syscalls`. Most `ntdll.dll` functions have the following layout:

```
76F564A8 >  B8 30010000          MOV EAX, 130
76F564AD    BA 0003FE7F          MOV EDX, 7FFE0300
76F564B2    FF12                 CALL DWORD PTR DS:[EDX]
76F564B4    C2 0800              RET 8
```

`0x130` is the `syscall` ordinal for `ZwResumeThread` in `Windows 7`. All functions have an assigned number used to call them. `7FFE0300` is the address of the `KiFastSystemCall` function. Here's the code of this function:

```
76F57090 > 8BD4          MOV EDX, ESP
76F57092   0F34          SYSENTER
76F57094 > C3            RET
```

## System calls and 64-bit architecture

A `syscall` calling convention in 64-bit systems is different than in 32-bit architectures. While the general conception is identical, the code that executes a `syscall` has a different layout on top of using 64-bit registers. Below is the general structure of a `syscall` in 64-bit systems:

```
mov r10,rcx
mov eax,[syscall_number]
syscall
ret
```

The convention is still different in the case of running 32-bit applications in a 64-bit system. The `WOW64` subsystem introduced in 64-bit systems to allow them to run 32-bit applications on 64-bit processors is then initialized. Rather than calling `KiFastSystemCall`, the invoked function is `X86SwitchTo64BitMode`, with a call address of `0xC0` in the `TEB` structure. To access it, execute the following:

```
MOV EAX,DWORD PTR FS:[0xC0]
```

`EAX` should now contain the address of the needed function. Its code is very simple: it consists of a jump instruction (`JMP`) to an address in segment `0x33`. At this point, the operation mode is changed, and the system call is executed just like for a 64-bit application run on a 64-bit operating system. After the function terminates, the execution moves back to the 32-bit mode.

Back to our code: the SYSENTER instruction jumps to the kernel and executes the remainder of the code of a given function there. Again, back to the code: why is the function number retrieved in this manner?

```
memcpy(&sys_res,(char*)((char*)ZwR+1),4);
```

It's because the first instruction in ZwR is the following:

```
76F564A8 >  B8 30010000     MOV EAX, 130
```

If you look at the code bytes, the first byte 0xB8 is responsible for MOV EAS,digit. The remaining 4 bytes specify the number (here, 0x130).

And why do we tweak the ZwSuspendThread function but call SuspendThread? This is possible as SuspendThread calls ZwSuspendThread itself. SuspendThread stops a thread from executing code, while ResumeThread wakes it up. If a thread is not suspended, the function doesn't change it.

With function numbers in sys_sus and sys_res established, we need to find a way to swap these numbers inside the ZwSuspendThread function's code with their corresponding ZwResumeThread values. VirtualProtect can be used for this task. Next, we switch the numbers using memcpy and make the following call next:

```
SuspendThread(GetCurrentThread());
```

We'll try to suspend the current thread. The antivirus should interpret this as program termination and won't associate the earlier change of syscall ordinals with the current function call – and by failing to do so, it misreads the events.

By the end of the code, we need to restore everything to the initial state. We'll restore the original number by again calling memcpy.

**What happened:** `ZwResumeThread` is called through `SuspendThread`. In reality, it did not change the thread. The program still executes the function.

**What happened according to the AV:** `SuspendThread` suspended the current thread of the program and the rest of the code was not executed.

The next step could involve for instance decoding the program. If we allowed the antivirus to do it, this would reveal the code's true function and make it easy to detect.

Apparently, there's no silver bullet to prevent discovery. A better precaution is using all these solutions in conjunction, just like `AVs` combine their three basic virus detection methods.

## Practice: video module transcript

Welcome to the final part of the sixth module of our training. In this part we'll learn how to counter emulation, which is the most advanced technique used by anti-virus programs.

The principle behind emulation is that it simulates the operation of the program without actually executing it. We'll base our anti-emulation code on the assumption that it's not a real program execution, but just a simulation. In order to counter emulation, we'll write a code with the purpose of fooling the anti-virus software so that it assumes that the program closes, enters an endless loop or has an error that forces it to close.

Obviously, we can't really close the program - it just has to seem as if we did so. Mind you, we mustn't use the if-else construction in our code because during the emulation the antivirus will check both conditions anyway. We have to use an operation the result of which is known to us, but which during the emulation will return a different value than the one we assumed. It will result, for instance, in a jump to a wrong address or the termination of the program.

The result of the GetLastError function, may serve as a relevant example, if we call the htons function without the previous initialization of the winsock

library using the WSAStartup function. As we know, the htons function will return an error, and GetLastError - the error code we know from the documentation. As it seems, some anti-virus programs during the emulation claim that the GetLastError function returns 0 in such cases, so there is no error. Based on the error code, we can calculate the address for the jump. If GetLastError returns a different value than the one we know, the jump address will be wrong and the emulation will obviously fail. Let's try to use such an approach in practice.

Here, we have the keylogger code we wrote earlier to counter heuristics. As we remember, it was detected by two anti-virus programs. Let's send our file to the VirusTotal to check the current result of the scanning. During the analysis we can see the code of our program.

```
FARPROC a1=GetProcAddress(GetModuleHandle("ntdll.dll"),"ZwResumeThread");
    FARPROC a2=GetProcAddress(GetModuleHandle("ntdll.dll"),"ZwSuspendThread");
    DWORD sys_res,sys_sus;
    memcpy(&sys_res,(char*)((char*)a1+1),4);
    memcpy(&sys_sus,(char*)((char*)a2+1),4);
    DWORD old;
    VirtualProtect(a2,5,PAGE_EXECUTE_READWRITE,&old);
    memcpy(((char*)a2+1),&sys_res,4);
    SuspendThread(GetCurrentThread());
    memcpy(((char*)a2+1),&sys_sus,4);
```

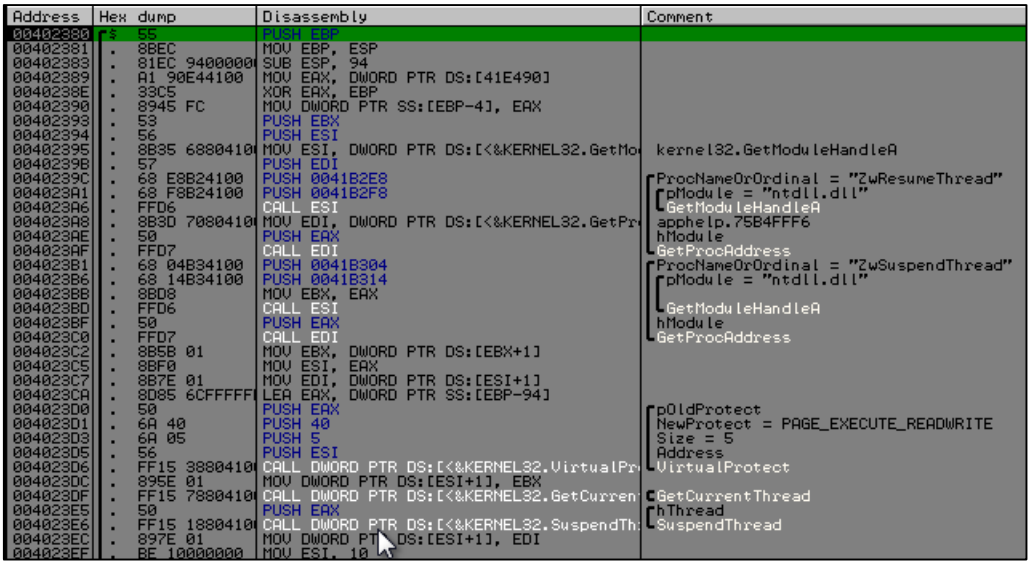
As we can see, we've added only this code fragment to the main function. First, we get the address of the ZwResumeThread, and then we get the address of the ZwSuspendThread function. These functions are only syscalls. Our code will change the places of system calls of these functions, namely the function ZwResumeThread will become ZwSuspendThread.

First of all, we get the call numbers the way we've done it before, that is we add 1 to the address and copy 4 bytes from this address. Next, we have to grant writing rights here. Then, we add 1 to the ZwSuspendThread address and copy there the value of the sys_res variable, that is the syscall number of the ZwResumeThread function.

Next, we call the SuspendThread function. We provide GetCurrentThread as a parameter, so we want to freeze the thread which is currently operating. The

anti-virus program will consider that the execution didn't complete, but the currently executing thread was frozen and is no longer operating. However, we know that in fact it is ResumeThread that executed, instead of SuspendThread. It means that if we resume the current thread, absolutely nothing will happen. After executing SuspendThread, the execution will move on to the next line where we restore the previous bytes in ZwSuspendThread, that is the previous syscall number we got, after which the execution moves on.

In other words, executing our code has no influence on the further operation of the program. Anyway, we can check on our own whether everything executes correctly. If the SuspendThread function actually executed there, the code execution should halt. Let's have a look at our code in the debugger. As usual, we use F8 and F7 to go to the main function.




Here we are, in the main function of the program. We can see our anti-emulation function. If the numbers hadn't been changed, this code line would interrupt the program execution. Let's check it step by step. Here we get the ntdll address. It's written in the EAX register. Next, we get the function address.

As we can now see, the EAX register includes the address of the ZwResumeThread function. Again, we get the ntdll address and the

ZwSuspendThread address. We have this address in the EAX register one more time.

Once they are obtained, the addresses are placed in the relevant registers. In the EDX register we have the ZwResumeThread address, while the EAX includes the address of the ZwSuspendThread function. We get the syscall number from ZwResumeThread for the EDX register. In the register we can see the number 130, which is the number of this call. To the EDI register, in turn, we get the ZwSuspendThread number. Next, we execute the VirtualProtect function in order to grant appropriate permissions. No error is returned, so we get the handle of the current thread. As we already know, it's -2. This function always returns such a value.

Now we execute the SuspendThread function. If we hadn't replaced the syscall numbers, the call would halt. We press F8 and, as we can see, the thread wasn't stopped. The execution carries on. In the next line, the syscall number is restored. Now let's see whether the scanning results are ready.

| | |
|---|---|
| SHA256: | 495e5339c87b0f132c2feb532dbd15676a7156098cc7ead7c6b4396631000352 |
| File name: | keylogger.exe |
| Detection ratio: | 1 / 41 |
| Analysis date: | 2012-04-18 13:44:19 UTC ( 16 minut ago ) |

The previous version of our application is now considered dangerous by only one anti-virus – Norman. It should also be detected by Kaspersky but it's currently inactive in the VirusTotal service. This happens sometimes when the page is under load.


Now let's scan our modified program. Again, we have to wait for the results. We can see that even though Kaspersy is already active, it doesn't detect any threats in our program. We can also see the Norman application, which just a while ago detected our program. Now, it considers it entirely safe. As a result, the detectability of our code falls to zero.

In less than twenty minutes we managed to decrease the detectability of our application from the initial 7 anti-viruses in the second part to zero in this part of the module. Let's summarise what we've learnt so far.

In this part of the module we've dealt with emulation, the most advanced technique used by anti-virus software. We've seen with our own eyes how much can be changed by adding just a few code lines. It can effectively fool an anti-virus program. Similar techniques are commonly used by malware creators. We have to keep that in mind.

Encoding the code, for instance using the AES algorithm, would be equally effective. In such a case, however, as we've already mentioned, there would be an issue with the increase of entropy. The anti-virus software wouldn't be able to emulate the code encoded this way, not because the emulation as such would fail, but because such an operation would be too complex to calculate. However, it doesn't necessarily mean that the encoded code would be considered safe. The conclusion is as follows - don't encode the code if you don't have to. We can possibly use encoding together with anti-emulation methods, which is used in various programs which encode or obfuscate the

code. However, we need to find some kind of an equilibrium, the so-called golden mean.

I strongly encourage you to explore and experiment with your own methods based on threads which influence each other. Also, seek functions which, under specific conditions, have well known results but can be skipped by anti-virus software during the process of emulation. Thank you for your attention and I hope to see you in the next module, where we'll discuss the issue of bypassing the firewall.