TECHNICAL FEATURE 2

MALWARE DESIGN STRATEGIES FOR CIRCUMVENTING DETECTION AND PREVENTION CONTROLS – PART 2

Aditya K. Sood and Richard J. Enbody
Michigan State University, USA

Anti-virus scanners have shown tremendous advances with the passage of time. There have been significant improvements in the tactics used by anti-virus developers to detect malicious code. However, malware writers are still running upfront. Let's look at some of the methods used by anti-virus engines to detect malicious code:

- The most common technique used by anti-virus engines is pattern matching and string detection. When a set of malware samples are analysed, a signature is generated using information extracted from the samples. The signature is usually built using byte code (memory data) that maps to a string that is unique to the malware. Wildcards are also deployed for detecting the different variants of malicious code. The signatures are stored in databases that are updated regularly. Additionally, cryptographic checksums are used for large signatures to produce a one-way unique hash for detecting complex malicious code. Implementation of cryptographic checksums makes the process faster and more accurate than using generic signatures.
- Code emulation is a technique in which malicious code is run in a virtual environment in order to detect its behaviour and infection tactics. Primarily, code emulation is used to detect encrypted and polymorphic malware by tracing different patterns in the memory. It is also easy to implement code optimization in emulators, making the process faster by removing junk code (code that has no relevance during analysis) from the malicious program.
- Heuristic analysis is used by anti-virus engines to make an educated guess about unknown malware based on a set of rules which determine whether a file meets any suspicious criteria. Heuristic analysis can harness the power of deployed signatures and code emulation strategies to detect unknown families of malware. It can be static in nature, utilizing file formats and dependencies to characterize a program's functionalities. Dynamic heuristics requires code emulation with self-learning capabilities. While heuristics-based detection is prone to false positives, it is valuable for enhancing the anti-virus engine's capabilities.

Generally, the majority of scanners use these techniques. More details about the working of scanning engines can be found at [1]. Many researchers have done a good amount of work in analysing and discussing obfuscation, anti-debugging and anti-emulation techniques. However, to continue our discussion we will dissect these principles again.

OBFUSCATION TACTICS

Malware writers employ a range of obfuscation tactics to make their code harder to analyse.

- *Embedding garbage code*: To make code more complex, malware writers add garbage code to their programs. Garbage code is also known as 'dead code'. Generally, such code is a set of instructions that do not modify the state or execution of the program, but which make the code complex when used with other obfuscation techniques. For example, NOP instructions are used heavily for this purpose. The garbage code can also be a set of subroutines. However, most present-day anti-virus engines have the capability to remove ineffective instructions from the code during analysis.
- *Subroutine randomization*: Malware writers also use a technique of randomizing subroutines to reorder the flow of instructions in the code. Typically, instead of placing subroutines in a hierarchical manner, they are placed randomly in different locations within the program. It is possible to have a number of variants of the same code based on subroutine randomization. This is possible because subroutines are individual pieces of program code that are independent in nature and can be imported during execution.
- *Code substitution*: Code substitution is an obfuscation technique in which certain instructions are substituted with equivalent ones. This can change the structure of code substantially and make it more complex to understand. Typically, this technique is used to subvert the pattern-based scanning tactics of anti-virus engines.
- *Obscuring entry points*: Generally, malware writers manipulate the entry point of the infected program (which is present in the code section) and relocate it to the malicious code. However, this can easily be detected as the entry point is present outside of the code section. To avoid detection, malware writers use Entry Point Obfuscation (EPO) techniques in which malware does not gain control directly from the system program but injects a JMP/CALL routine to subvert the execution. There are many variants of EPO.
- *Register shuffling*: Register shuffling is another technique used by malware writers to transform the layout of instructions in the code. Registers are shuffled

so that the code pattern is changed, but the code is executed in the correct manner. This is primarily used to mask real code, making its interpretation complex.

- *Code encryption*: Polymorphic viruses encrypt their code differently with each infection (or each generation of infections) in order to make it difficult for anti-virus engines to detect them. Emulation-based malware detection came about as a result of the fact that polymorphic malware decrypts itself during run time to trigger infection. This means that, at some point, polymorphic encryption/decryption has to produce the real code in memory, and that's where emulation succeeds. To thwart this, malware writers developed metamorphic malware in which the code itself mutates with every infection. Figure 1 shows the execution pattern of a metamorphic virus:

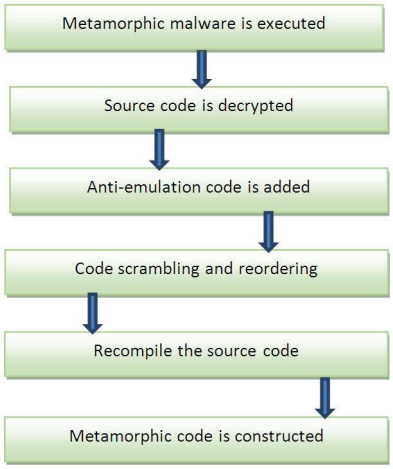

Figure 1: Metamorphic virus execution flow.

Both the decryption routine and the decrypted code are different in every generation of metamorphic malware, whereas in polymorphic malware the original source code does not change. Different types of encryption have been discussed in [2].

ANTI-TRAFFIC ANALYSIS

Since the advent of Zeus, several classes of malware have been using anti-traffic-analysis code. This allows the malware to detect the presence of traffic analysis systems and kill them before it starts communicating with the Command & Control (C&C) server. Typically, on *Windows Wireshark* and *Microsoft*'s *Network Monitor* are used to monitor the traffic going in and out of the system. There are many ways to trigger anti-traffic code in the *Windows* operating system. Malware writers generally prefer to implement anti-traffic

```
DWORD main_pid =  // Get the Process ID of the Traffic
Monitoring Program [Wireshark | ]
PROCESSENTRY32 proc_en;

memset(&proc_pe, 0, sizeof(PROCESSENTRY32));
proc_en.dwSize = sizeof(PROCESSENTRY32);

HANDLE handle_snap = CreateToolhelp32Snapshot(TH32CS_
SNAPPROCESS, 0);

if (Process32First(handle_snap, &proc_en))
{
    BOOL continue = TRUE;
    while (continue)  {
      if (proc_en.th32ParentProcessID == main_pid)
      {
        HANDLE h_child_proc = OpenProcess(PROCESS_
ALL_ACCESS, FALSE, proc_en.th32ProcessID);
          if (h_child_proc)
          {
            TerminateProcess(h_child_proc, 1);
            CloseHandle(h_child_proc);
            }
          }
            continue = ::Process32Next(handle_snap,
&proc_en);  }

   HANDLE h_proc = ::OpenProcess(PROCESS_ALL_ACCESS,
FALSE, main_pid);
   if (h_proc) {
          TerminateProcess(h_proc, 1);
          CloseHandle(h_proc); }   }
```

Listing 1: Killing a process in the system.

```
char* window_handle[] = { "The Wireshark Network
Analyzer", "Microsoft Network Monitor" };
void anti-traffic_routine( )
{
  for( int temp = 0; temp < ( sizeof( sText ) /
sizeof( char* ) ); temp++ )
    {
      HWND handle_find = FindWindow( 0, sText[ temp ]
);
        if( handle_window != NULL )
          {
            SendMessage(handle_find, WM_CLOSE, 0, 0 );
          }
    }
}
```

Listing 2: Anti-traffic routine using FindWindows().

routines in the form of assembly code which is embedded as inline code. However, this is not the only way. Listing 1 shows the code that is used to kill a process in the system. All the running processes are enumerated first, then once the active processes have been found, the malicious code looks for the target process and kills it.

Other methods involve retrieving a handle to the window of the running program using FindWindow() and FindWindowEx(). This method simply requires finding a handle to the window of the running traffic-monitoring program. A WM_CLOSE message is then sent to kill it. Listing 2 shows how this is achieved. When this kind of code is triggered in the system, the traffic analysis program is killed and an exception is raised, as shown in Figure 2.

Malware writers also try to open the file handle to \\.\NPF_NdisWanIp to query information about the interface and verify the state of the adapter. Registry-based detection is another viable method for detecting the state of programs in the system – for example, some of the *Wireshark* registry entries present in *Windows XP* are presented in Listing 3. If the *Wireshark* program is installed properly then these registry entries must exist. The registry path might vary with different operating systems. Listing 4 shows the registry entries for the presence of *Microsoft Network Monitor* on *Windows 7*.

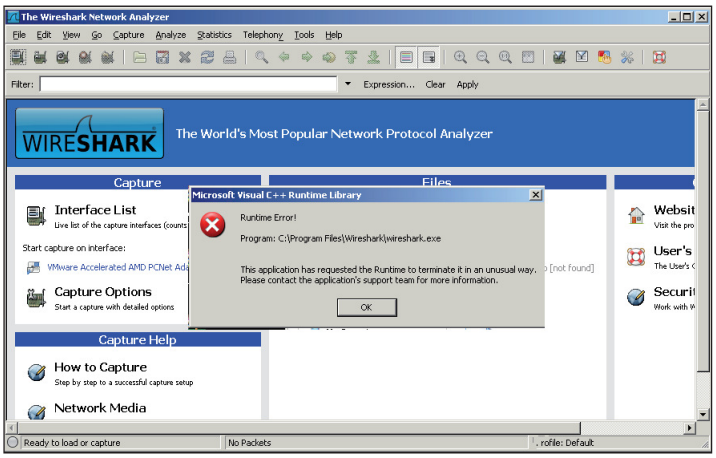

Figure 2: Running anti-traffic analysis code results in an exception.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Applications\
wireshark.exe

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\wireshark-
capture-file

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\App Management\ARPCache\Wireshark

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\App Paths\wireshark.exe

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\Uninstall\Wireshark
```

Listing 3: Registry entries for Wireshark in Windows XP.

```
HKEY_CURRENT_USER\Software\Microsoft\Netmon3

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Netmon3
```

Listing 4: Registry entries for Network Monitor in Windows 7.

ANTI-PROTECTION ANALYSIS

All of the above techniques can also be applied to detect the presence of *SysAnalyzer*, *Windows Defender* and *Microsoft*'s *Security Essentials* as well as other anti-virus engines running in the system. There are also other methods, such as querying Windows Management Instrumentation (WMI), that can be used to gain information about the state of the *Windows* operating system. Listing 5 shows how WMI is used for querying installed programs in *Windows*. Similarly, firewalls that are installed on a system can also be enumerated.

ANTI-DEBUGGING TRICKS

Anti-debugging is a method that malware writers use to prevent active debugging of the executable/binary when a relevant process is triggered in the system. This method plays a significant role in disrupting the analysis process. Malware writers can implement several methods to trigger anti-debugging routines in the malicious code.

Generally, Application Programming Interface (API) calls are used for this purpose. *Windows* provides inbuilt API calls such as IsDebuggerPresent() and CheckRemoteDebuggerPresent(), which are used to detect the presence of user-mode debuggers. IsDebuggerPresent() is used to determine whether the running process has a user-mode debugger attached to it. CheckRemoteDebuggerPresent() is used to determine whether a given process is being debugged. Generally, IsDebuggerPresent() is used extensively in malicious codes that are forced to execute in user mode.

- *Bypassing IsDebuggerPresent()*: There are several techniques that can be used to bypass this debugger detection API. The first involves tampering with the code flow. In this, the primary task of the reverse engineer is to remove (overwriting with NOP) the instructions that perform comparisons in the code and then manipulate the flow of code by tampering with JMP (JNZ → JZ, and so on). A second technique

```
[Listing Spyware Programs]

wmic:root\cli>/Node:localhost /Namespace:\\root\
SecurityCenter2 Path AntiSpywareProduct Get displa
yName,productState,instanceGuid,pathToSignedProduc
tExe,PathToSignedReportingExe /format:list

displayName=Windows Defender
instanceGuid={D68DDC3A-831F-4fae-9E44-
DA132C1ACF46}
pathToSignedProductExe=%ProgramFiles%\Windows
Defender\MSASCui.exe
pathToSignedReportingExe=%SystemRoot%\System32\
svchost.exe
productState=393472

displayName=Microsoft Security Essentials
instanceGuid={2C040BB5-2B06-7275-5A21-
2B969A740B4B}
pathToSignedProductExe=C:\Program Files\Microsoft
Security Client\msseces.exe
pathToSignedReportingExe=C:\Program Files\
Microsoft Security Client\MsMpEng.exe
productState=397312

[Listing AntiVirus Programs]
wmic:root\cli>/Node:localhost /Namespace:\\root\
SecurityCenter2 Path AntiVirusProduct Get displayN
ame,productState,instanceGuid,pathToSignedProductE
xe,PathToSignedReportingExe /format:list

displayName=Microsoft Security Essentials
instanceGuid={9765EA51-0D3C-7DFB-6091-
10E4E1F341F6}
pathToSignedProductExe=C:\Program Files\Microsoft
Security Client\msseces.exe
pathToSignedReportingExe=C:\Program Files\
Microsoft Security Client\MsMpEng.exe
productState=397312
```

Listing 5: Enumerating anti-virus and anti-spyware programs in Windows.

involves overwriting the flags. Here, the reverse engineer overwrites the _PEB.BeingDebugged flag in the Process Execution Block (PEB), which is pointed to by the Thread Execution Block (TEB).

- *Bypassing CreateRemoteDebuggerIfPresent()*: To successfully bypass this API, it is necessary to hook the process. This function is different from IsDebuggerPresent() because it takes two different arguments as handles to the process and pointer referencing a variable (true, false). The IsDebuggerPresent() function does not call any specific arguments, rather it relies on the internal NT calls to determine the presence of a user-mode debugger. To bypass CreateRemoteDebuggerIfPresent(), it is necessary to perform hooking in the running code.

Practical examples of these techniques have been discussed in [5, 6]. Another method for detecting the presence of a debugger in the system is based on a simple registry key check to verify if the specific key related to the debugger (such as *OllyDbg*) is present in the registry. This is an easy tactic but it is not a robust way to detect debuggers in the system because registry entries can easily be tampered with. The FindWindow() trick is an old one, but works well to check the presence of any active window pointing to the running state of a debugger in the system. Considering program code, the presence of code-based debugging APIs such as OutputDebugString() with error handing APIs such as GetLastError()/ SetLastError() is another easy way to detect the presence of a debugger. Several interesting anti-debugging techniques have been discussed in [7].

CONCLUSION

We have discussed several techniques and tactics used by malware writers to analyse the operating system environment before the malware is executed. This enables them to bypass several host-based protection solutions and detection tools. The methods discussed here are not the only ones used, but these concepts should provide a good basic understanding. Some of these techniques have been widely researched, demonstrating the importance of these issues. We believe that there are still more robust techniques available that might not yet have been seen in the wild.

REFERENCES

[1] Hunting Metamorphic Engines. http://www.truststc.org/pubs/237/hunting.pdf.

[2] Advanced Polymorphic Techniques. http://www.waset.org/journals/waset/v34/v34-45.pdf.

[3] Advanced self-modifying code. http://migeel.sk/blog/2007/08/02/advanced-self-modifying-code/.

[4] Fighting EPO Viruses. http://www.megapanzer.com/wp-content/uploads/fighting_epo_viruses.pdf.

[5] CheckRemoteDebuggerPresent Bypass. http://gunboundinfo.blogspot.com/2008/10/checkremotedebuggerpresent-bypass-by.html.

[6] IsDebuggerPresent Bypass. http://gunboundinfo.blogspot.com/2008/10/isdebuggerpresent-bypassing-by-wiccaan.html.

[7] Anti-Debugging – A Developers View. http://www.shell-storm.org/papers/files/764.pdf.