

.NET/MSIL Malicious Code and AV/Heuristic Engines

Markus Schmall 2002-11-18

The .NET strategy/technology from Microsoft has caused quite a stir amongst the security community. While the Windows .NET strategy incorporates numerous aspects, this article will focus on what aspects to cover in order to develop an AV/heuristic engine for this new platform. Specifically it will address the additions introduced by .NET technologies to standard Windows PE (portable executable) file format and how that will affect the development of an effective heuristic engine. It will also briefly discuss the existing malicious codes for the .NET environment.

To better understand the PE file format and the .NET extensions, this article will use "HelloWorld" as an example. It can be downloaded [here](#). At this location, readers will also find basic sources for parsing the Microsoft Intermediate Language (in the following referred to as MSIL) and a complete AV module for scanning .NET/MSIL malicious codes.

A full implementation of a .NET Common Language Runtime (CLR) engine can be found at [Rhys Weatherley's home page](#).

Basics of the .NET MSIL Tokens

The program functionality (i.e., the program code) is represented by MSIL tokens, which will be interpreted by the runtime engine. As various required data -- such as strings etc. -- is placed in different parts of the file and not directly combined with the code, the parsing is not trivial. (That said, it is not nearly as complicated as parsing Visual Basic for Applications pcode structures.) The MSIL tokens themselves can be only found within a file if a so-called meta data header (later also referred to as "#~" section) is completely analyzed. Furthermore there is certain additional information within the .NET related parts of the file, such as security related permission tables, which are not needed for the actual code analysis.

HLLP/Sharpej.A

Since the .NET Framework SDK was released to the public, we have seen very few examples of malicious code (actually less than 10) targeting this platform. An early "attempt", known as W32/Donut (released in the beginning of 2002), was a simple Win32 virus with a simple .NET payload, which simply displayed a message alert. Therefore, it is clearly not correct to classify

this malicious code as a .NET malicious code.

The first "real" virus, which appeared for this platform, is called HLLP/Sharpej.A, which was released in the middle of 2002. Accordingly, this discussion will concentrate on the .NET component of the HLLP\Sharpej.A worm. It should be noted that the other main functionality of this worm is located within "plain" x86 assembly code and a Visual Basic Script (VBS) file, which contains a mass mailing routine.

Besides the core replicating routine in the .NET part of the malicious code, the virus drops an additional VBS file called "sharp.vbs", which contains a simple MsgBox() payload. The replication itself is fairly straightforward. The HLLP/Sharpej.A virus searches various directories (Windows directory plus three other directories) for possible targets (these must have the file ending "*.exe"). An infected file contains the viral part, which is placed in front, followed by the original file behind. In order for an infected file to be started, the original file will be written to a temporary file and a new process will be created based on this file. Similar technologies are known from various other malicious codes for the Windows platform.

Overall, it can be said that HLLP\Sharpej.A is a relatively straightforward malicious code without any technical specialties.

HLLP/Flatei.A-E

Besides HLLP/Sharpej.A and Win32/Donut, HLLP/Flatei.A-E is one of the few examples of .NET malicious codes known to the AV community. Unlike HLLP\Sharpej, the HLLP\Flatei family is a set of pure .NET malicious codes. This malicious code is a rather simple virus, which searches for "*.exe" files in the current directory. If a file matching these standards is found, the code of the virus (actually 0x1404 bytes) is copied to the beginning of the original file. The body of the original file will be not changed, only a block is written in front of it. If the infected file is started, then the virus receives initially the control and (besides the replication operation) writes the original file to the static filename "hostbytes.exe". This file will be then executed using the Win32 API call "ShellExecute()". After this API call returns, the suspicious file will be deleted.

Having taken a cursory look at the currently known .NET malicious codes, it can be said that existing .NET malicious codes are rather simple and are comparable to early malicious codes found on the Windows platform in general. The heuristic detection of malicious functionality for both examples of malicious code can be easily performed due to the lack of anti-heuristic

encryption and other heuristic-avoiding routines. Looking at the MSIL tokens, it is rather easy to detect the file search routines and the infection routines. Furthermore the creation of a new file, the creation of a new process (i.e., the execution of the new file) and the removal of suspicious traces such as the temporary file, are implemented in a fairly straightforward manner.

None of the current .NET malicious codes contain polymorphic routines or self-encryption approaches, which are generally rather complicated to realize using MSIL. Instead, we are more likely to see "external" routines, which parse the entire file and make all changes.

Coming Back to the Technical Stuff?

Assuming that we can expect to see more malicious code dedicated to .NET/MSIL, it is reasonable to think that the idea of the development of a dedicated MSIL heuristic engine is one obvious next step. The following section will do just that. It is expected that the reader has at least basic knowledge about Windows file formats. The file format described below is based on the final version of the .NET framework, certain changes to the .NET Beta 1 format have been introduced.

As stated earlier, the example used for this discussion is a simple "Hello World" application, written in managed C++. The source code looks as follows:

```
#using
#include

using namespace System;

// This is the entry point for this application
int _tmain(void)
{
    // TODO: Please replace the sample code below with your own.
    Console::WriteLine(S"Hello World");
    return 0;
}
```

Built using the Win32 debug configuration, the file is exactly 114688 bytes long. This example applications contains three PE file format sections:

| Name | Virtual Size | Virtual Offset | Raw Size | Raw Offset |
|--------|--------------|----------------|----------|------------|
| .text | 0x15924 | 0x1000 | 0x16000 | 0x1000 |
| .rdata | 0x3e88 | 0x17000 | 0x4000 | 0x17000 |
| .data | 0x2498 | 0x1b000 | 0x1000 | 0x1b000 |

Figure 1: Structure of the example file

In the PE header, there exists a new field at offset 0xe8, which describes the RVA of the MSIL runtime header and the size of the runtime header. Within MSDN documentation, the runtime header is referred to as IMAGE_COR20_HEADER .

The runtime header contains several interesting and relevant values, wherein the long word (32 bit value) at offset 0x8 describes the RVA (short form of relative virtual address) of the metadata header and the long word at offset 0xc describes the size of the metadata.

In our example file, the metadata is located at offset 0x19dca as shown below.

```

00019DC4      0000 0000 3164 4100 5F66 7265 6562      ....1dA._freeb
              7566 2E63 0000 4253 4A42 0100 0100      uf.c..BSJB....
00019DE0      0000 0000 0C00 0000 7631 2E30 2E33      .....v1.0.3
              3730 3500 0000 0000 0500 6C00 0000      705.....1...
00019DFC      2C02 0000 237E 0000 9802 0000 D802      ,...#~.....
              0000 2353 7472 696E 6773 0000 0000      ..#Strings....
00019E18      7005 0000 1C00 0000 2355 5300 8C05      p.....#US...
              0000 1000 0000 2347 5549 4400 0000      .....#GUID...
00019E34      9C05 0000 3003 0000 2342 6C6F 62      ....0...
#Blob
    
```

Figure 2: Hex dump of the metadata tables

The beginning of the metadata header is marked with ?BSJB?. At offset 0x1e is found the number of sections within the metadata stored (please note, this may vary, if the version identifier, V1.0.3.705 in this case, changes). In the example above, we have five sections, "#US", "#GUID", "#Blob", "#Strings" and "#~", that are all necessary and useful for scanning

MSIL code. The latter section contains the metadata tokens, which have to be analyzed by the heuristic engine to be able to detect the MSIL opcodes.

The sections mentioned above do not match the PE file format sections. Unlike the Beta 1 file format, not all .NET-related data is placed within the same PE file section. In our example file, the metadata-related code (such as runtime header etc.) is placed within the PE ".rdata" section, and the MSIL opcodes itself are placed within the PE ".text" section.

Scanning the File, Stage One: File Collection Information

To scan the MSIL parts of a file, it is clearly necessary to parse the "#Strings", the "#Blob", "#us" and the "#~" section. Within the "#~" section we find references to all functions within the code. Obviously, in our example file, we only have one relevant function, named "main()".

NB: For complete disassembles/dumpers like "ildasm" or the fabulous "akrino" it is necessary to parse even more information in the file format but a heuristic engine does not necessarily need to parse security permission-related information.

As has already been mentioned, there are certain tokens placed within the "#~" section that describe the module name and other relevant information. The tokens within this section are grouped together in incremental order (meaning that the opcode with type 0 is placed before the opcode with type 1 etc.). One of the most interesting tokens (0x6) is the token called "MethodDef". This token defines such critical values like the RVA to the existing methods. However, the length of the designated method is not stored within this function definition but can be found in the header of the method itself. After this method header, referenced by the RVA value, the MSIL opcodes for the function itself can be found. Having reached this point, the file can be analyzed by the main components engine.

Scanning the File, Part Two : Collection and Analysis of the opcodes

As previously discussed, in our example the "main()" function is the most interesting function. A dump generated by the "ildasm" tool from the .NET Framework SDK looks as follows:

```
//Global methods
//~~~~~
.method /*06000001*/ public static int32 modopt([mscorlib/* 23000001
*/]System.Runtime.CompilerServices.CallConvCdecl/* 01000012 */)

```

```

        main() cil managed
// SIG: 00 00 20 49 08
{
    .vtentry 1 : 1
    // Method begins at RVA 0x1000
    // Code size      14 (0xe)
    .maxstack 1
IL_0000:  /* 72  | (70)000001      */ ldstr      "Hello World" /* 70000001 */
IL_0005:  /* 28  | (0A)00000F      */ call      void [mscorlib/* 23000001
*/]System.Console/* 01000013 */::WriteLine(string) /* 0A00000F */
IL_000a:  /* 16  |                */ ldc.i4.0
IL_000b:  /* 2B  | 00              */ br.s      IL_000d

    IL_000d:  /* 2A  |                */ ret
} // end of global method main

```

The scan engine needs to read token by token, as the tokens are not always only one byte long. Therefore, the engine must know about each token and the required parameters. For example, the first line of code can be analyzed as follows:

1. A token 0x72 is found (= ldstr)
2. at offset 0x70 in the relevant section the string "Hello World" can be found

As a result, this code can be decompiled to "ldstr "Hello World"".

The scan engine should always start looking at code referenced by the MSIL/PE entry point (for ordinary, unchanged normal executable files, this should be always the function "main").

The scanning process should then follow the program flow and analyze all possibilities/permutations of possible program flows. As this could become a very time-consuming task, a straightforward analysis of all functions (methods) on its own can also be sufficient. When performing this simplified approach, certain false detections/results may be possible.

The engine should then remember what kind of functionality exists within the functions and whether or not this functionality is dependant of function parameters. This feature will be required to enable a more exact form of understanding/rating the overall functionality. It should

be noted that in the early stage of .NET based malicious codes, simplified emulation should be fully sufficient to detect malicious operations.

Scanning the File, Part Two and a Half: the Need for General Emulation

As has already been discussed in other papers on heuristic technologies, it can be very helpful to implement a processor emulation to detect certain cleverly programmed malicious codes. Looking extant .NET malicious code, the need for such a complex routine obviously does not exist right now. However, as certain runtime implementations have already shown, this task can be done without dependency on the Microsoft Windows platform.

What would be needed to implement an emulator for .NET?

- an emulator for the stack itself;
- emulation/interpretation of the instructions at MSIL level (at least an interpretation of MSIL opcodes must exist for a standard .NET/MSIL heuristic engine);
- emulation of a minimal file system (for example, to be able to fake a Windows (system) directory, a "c:" drive and objects, which could be relevant for malicious codes like certain initialization files such as "win.ini");
- emulation of certain functionality, such as file system related or registry related; and,
- emulation of the Windows registry.

As the complete environment is based on a stack, it should be ensured that even if a certain operating system function is not emulated, faked return values have to be placed on the stack to keep the program from stopping based on a simple stack "underrun" failure (i.e., trying to remove an element from the stack, even if the stack is already empty).

Furthermore, it should be taken into account that malicious code could use rarely used MSIL functions, which are most likely not emulated by AV solutions. The result could be used to detect the presence of an emulator. This kind of "AV check" technology has already been seen a couple of times in Windows-based malicious codes.

Scanning the File, Part Three: Rating

Initially, we can identify two separate basic rating methods:

- Weight/threshold based systems; and,

- Rule based systems.

Basically, both approaches have been discussed in great detail in a couple of papers. However, rule-based systems often offer better granularity and more possibilities for special detection rules, which can be used to detect complete families of malicious codes.

Within the scanning process, it appears to be useful to rate all functions on their own. Functions depending on parsed arguments should be rated additionally in a second process, where the possible program flow is also taken into account.

Based on the generic available information, two types of heuristic "rules" are thinkable:

1. A rule created out of an array of flags like {flag1, flag2}; and,
2. A rule using more advanced definition features, which can define flags per function, such as {subroutine_start, flag1, flag2_subroutine_end) and {subroutine_start, flag4, flag5_subroutine_end) .

The latter type particularly provides enough information to be able to do "family" detections. A "family" detection is able to detect using one rule a major percentage of malicious codes belonging to a family. Furthermore, the second approach seems to fit much better with the .NET/MSIL concept of storing the MSIL functions one by one and not directly in a big module, as is often seen in Visual Basic for Applications.

Moving Back One Step...Checksums...?

Checksums are still a very popular way of detecting malicious code. The fact that polymorphic/metamorphic approaches will probably not often be seen within .NET/MSIL malicious codes makes the development of checksum-driven scan engines for .NET/MSIL a simple project, as no complex exclusion or mathematical-linking operations need to be performed. The easiest method is to implement a checksum algorithm for MSIL/.NET malicious code based on a per-function basis and perform a "smart" checksum approach, meaning that all operands will be ignored. This approach is also reasonably exact.

Our demo project (more precisely, the main() function) would then be seen as the following block (bold characters are the opcodes itself):

7200000028000000162B002A

It could be easily checksummed, for instance using a standard CRC32/ZLIB algorithm. Therefore, there would not be a single checksum representing a malicious code but instead the combination of several checksums, one checksum for every function. As a next step it one could give every checksum an "importance" rating, meaning that malicious codes can be compared based on the existence of certain functions.

Conclusion

The development of an AV/heuristic engine for .NET/MSIL within a reasonable short time frame is possible. Based on the excellent development of the existing .NET runtime systems like PNET and the [ECMA documents](#), the file format can be investigated relatively quickly and parsing systems can be written. No other components of AV scan engines represent new technology inventions.

Without question, more.NET-based malicious codes (also thinking on the .NET Compact Framework) will appear and cause problems to both end users and corporate users. It is good to see that standard AV scanning systems already have the ability to detect the existing .NET based malicious codes.

References

[1] Philipp Hannay & Richard Wang, "MSIL for the .NET framework: The next battleground?", Proceedings of the 11th Virus Bulletin Conference, September 2991, Prague

[2] Markus Schmall, "Heuristic Techniques in AV Solutions: An Overview", <http://online.securityfocus.com/infocus/1542>

[3] Markus Schmall, "Building an Anti-Virus Engine", <http://online.securityfocus.com/infocus/1552>

[4]MSILTestHelloWorld.exe <http://www.mschmall.de/msil/msiltesthelloworld.exe>

[5] Rhys Weatherley, "Unofficial guide to CLR Metadata", Copyright 2001 Southern Storm Software, Pty Ltd.

[6] DotGNU Portable.NET, Southern Storm Software, http://www.southern-storm.com.au/portable_net.html

[7] Common Language Infrastructure, ECMA-335 <http://www.ecma.ch/ecma1/STAND/ECMA-335.htm>

Markus Schmall works in the IT Product Security Department of T-Mobile Germany and can be reached at markus@mschmall.de.

Relevant Links

[Heuristic Techniques in AV Solutions: An Overview](#)

Markus Schmall, SecurityFocus

[Building an Anti-Virus Engine](#)

Markus Schmall, SecurityFocus

[Privacy Statement](#)

Copyright 2006, SecurityFocus