Detecting Malicious Code by Binary File Checking

Marius POPA
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies, Romania
marius.popa@ie.ase.ro

The object, library and executable code is stored in binary files. Functionality of a binary file is altered when its content or program source code is changed, causing undesired effects. A direct content change is possible when the intruder knows the structural information of the binary file. The paper describes the structural properties of the binary object files, how the content can be controlled by a possible intruder and what the ways to identify malicious code in such kind of files. Because the object files are inputs in linking processes, early detection of the malicious content is crucial to avoid infection of the binary executable files.
Keywords: *Malicious Code, Binary File, Malware Detection*

1 Introduction

The term of *malicious code* is assigned to any code or script in any part of a software system, having the intent to cause undesired effects, security breaches and system damages. The malicious code gives the feature of malware to the software system which resides in. The most known forms of the malwares are viruses, worms, Trojans horses, spyware, trapdoors, adware, rootkits, malicious active content and so forth.

The *binary files* contains non-text data encoded in binary form as computer files that are stored and may be processed by a software system that knows how to deploy, manage and use a such file in the computer system or over a computer network. Usually, in software development process, the term of binary file is assigned to hard-disk recipient that stores instructions in binary form which can be executed by the central processing unit of the computer directly. Currently, the binary files have evolved as structure, content and their management as processes at runtime as the hardware, software development tools and challenges of Information and Communications Technologies (ICT) have advanced.

In [1], [3], [4], [5], [6], the following issues are addresses:

- Requirements of the secure software development process;
- Compiling and interpreting processes;
- Binary code and file formats;
- Binary and bytecode file structures;
- Disassembly process;
- Virtual machine architectures;
- Processes of secure code review;
- Techniques and tools used in reverse engineering;
- Methods and techniques for a secure program coding;
- Methods and techniques of code obfuscation;

The Windows executable file in the Portable Executable (PE) format is detailed in [4]. In [7], the specifications regarding the PE files and object files used by Microsoft product are presented. The object file is referred as Common Object File Format (COFF).

Object file is produces by a compiler, assembler or translator and represents the input file of the linker. After linking, an executable or library is generated and contain combined parts of the object file. The content of the object file is not directly executable, but it is a re-locatable code. The linking process is illustrated in Figure 1.

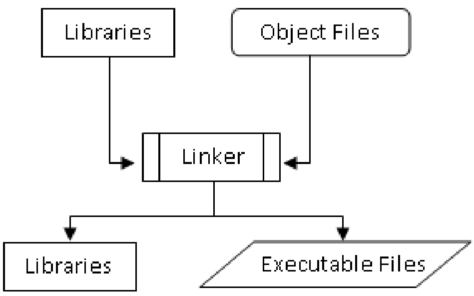

Fig. 1. The linking process

DOI: 10.12948/issn14531305/18.1.2014.10

A comprehensive image of the PE file layout is given in [7], Figure 2.

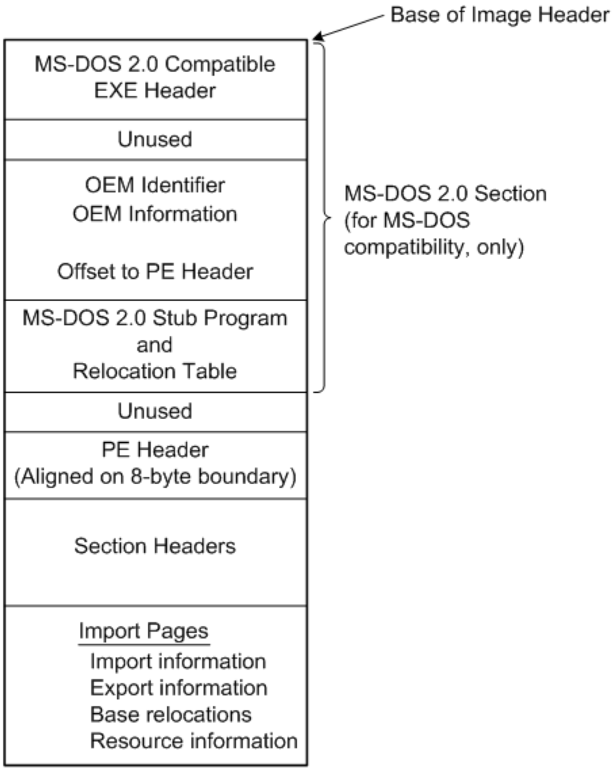

Fig. 2. Comprehensive PE structure as [7] states

Also, [7] illustrates the COFF file layout, Figure 3.

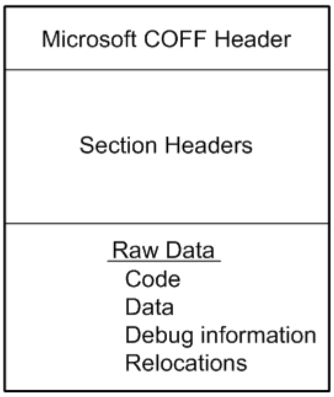

Fig. 3. Comprehensive COFF structure as [7] states

The COFF file header has a length of 20 bytes and is structured in several fields as [7] states and Table 1 highlights.

Table 1. The COFF file header structure [7]

Offset	Size	Field
0	2	Machine
2	2	NumberOfSections
4	4	TimeDateStamp
8	4	PointerToSymbolTable
12	4	NumberOfSymbols
16	2	SizeOfOptionalHeader
18	2	Characteristics

The COFF file header describes the environment which the object file can be used in and the file structure at highest level.
Each COFF section header has a length of 40 bytes and is structured in several fields as [7] states and the Table 2 depicts.

Table 2. The COFF section header structure [7]

Offset	Size	Field
0	8	Name
8	4	VirtualSize
12	4	VirtualAddress
16	4	SizeOfRawData
20	4	PointerToRawData
24	4	PointerToRelocations
28	4	PointerToLinenumbers
32	2	NumberOfRelocations
34	2	NumberOfLinenumbers
36	4	Characteristics

The object file is the output of the compiling process. Different source code programs lead to different contents of the object files compliant with the layout requirements and constraints. The COFF file is the foundation of the library and executable files.

2 The Object File Content

Let consider the following source code written in C++ programming language.

```
class Employee{
        public:
                char* Name;
          int id;

          Employee(char* aName, int nr){
```

DOI: 10.12948/issn14531305/18.1.2014.10

```
            this->Name = aName;
            this->id = nr;
            procData(aName, nr);
        }

        char* empName(){ return this->Name; }

        int empID() { return this->id; }

        void procData(char* sName, int snr) { }
};

void main() {
        Employee e ("Smith", 113);
        e.empID();
        e.empName();
}
```

The first 20 bytes represents the COFF file header generated by Visual Studio 2010 C++ compiler in **`Employee.obj`**. The COFF file header content is:

```
00000000  4C 01 10 00 A5 AB 29 53  E5 0F 00 00 41 00 00 00  L.....)S....A...
00000010  00 00 00 00 2E 64 72 65  63 74 76 65 00 00 00 00  .....drectve....
```

Fig. 4. The COFF file header content for the above example

The values of COFF file header fields are explained in Table 3.

Table 3. The COFF file header fields explained

Field	Value	Description
Machine	0x014C	Intel 386 or later processors and compatible processors.
NumberOfSections	0x0001	The size of section table (one section for above example).
TimeDateStamp	0x5329ABA5	Number of seconds since January 1, 1970, 00:00 when the file was created (1395239845 seconds/16148 days/about 44 years).
PointerToSymbolTable	0x00000FE5	The offset of COFF symbol table (4069 bytes).
NumberOfSymbols	0x00000041	The number of entries in the symbol table (65 entries). Also, the string table is located by this value.
SizeOfOptionalHeader	0x0000	It is not required for object files. Null value means an object file.
Characteristics	0x0000	Flags to indicate the attributes of the file. No flag for current object file.

Next structure item of the COFF file layout is *Section Headers*. The number of the section headers is given by *NumberOfSections* field from COFF file header that is 1 section. Each section header covers 40 bytes. The section header content of the COFF considered above is presented in below figure.

DOI: 10.12948/issn14531305/18.1.2014.10

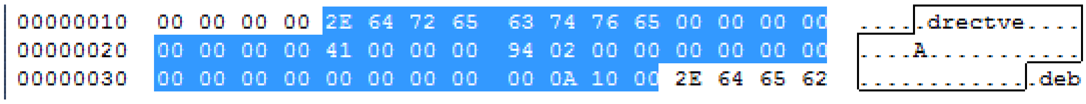

Fig. 5. The COFF section header content for the COFF file

The values of COFF section header fields are explained in Table 4.

Table 4. The COFF section header fields explained

Field	Value	Description
Name	0x2E64726563747665	An 8-byte size string: "**`.drectve`**". The string has no null terminator because the length is 8 bytes.
VirtualSize	0x00000000	Set to null value because the file is an object file.
VirtualAddress	0x00000000	The address of the first byte before applying the relocation. Set to zero for the considered object file.
SizeOfRawData	0x00000041	For object files, the field represents the size of the section that is 65 bytes.
PointerToRawData	0x00000294	It is a file pointer to the first page of the section. The value has to be aligned to 4-byte boundary for best performance: 660 bytes / 4 bytes = 165.
PointerToRelocations	0x00000000	The null value means no relocation.
PointerToLinenumbers	0x00000000	The null value means there are no object line numbers.
NumberOfRelocations	0x0000	The null value means no relocation entry for the section.
NumberOfLinenumbers	0x0000	The null value means no line-number entry for the section
Characteristics	0x00100A00	The following flags are set for the object file: IMAGE_SCN_LNK_INFO, IMAGE_SCN_LNK_REMOVE and IMAGE_SCN_ALIGN_1BYTES. That means the object file contains comments or other information (**`.drectve`** type), the section will not become part of the executable file, and data are align to 1-byte boundary.

When a section has set the flag IMAGE_SCN_LNK_INFO and the name of **`.drectve`**, then the section is a directive one. The section does not appear in the executable file because the linker removes it after information processing. It has not relocations and line number, and it is used to provide linking options to linker.

The data for the section is located at the file offset specified in *PointerToRawData* field of the section header. The size of the data is indicated by the *SizeOfRawData* field from the section header. For the object file **`Employees.obj`**, the offset is 0x00000294, and the size is 0x00000041. The content of the section is depicted in Figure 6.

DOI: 10.12948/issn14531305/18.1.2014.10

```
00000290  40 00 10 42 20 20 20 2F  44 45 46 41 55 4C 54 4C  @..B   /DEFAULTL
000002a0  49 42 3A 22 4D 53 56 43  52 54 44 22 20 2F 44 45  IB:"MSVCRTD" /DE
000002b0  46 41 55 4C 54 4C 49 42  3A 22 4F 4C 44 4E 41 4D  FAULTLIB:"OLDNAM
000002c0  45 53 22 20 2F 45 44 49  54 41 4E 44 43 4F 4E 54  ES" /EDITANDCONT
000002d0  49 4E 55 45 20 04 00 00  00 F1 00 00 00 83 03 00  INUE ...........
```

Fig. 6. The COFF file **`.drectve`** section content

The object file **`Employees.obj`** has no relocation and line numbers. The relocations specify how the section data is modified when is placed the executable file. The line numbers indicates the relationship with the code.

The COFF file symbol table is places to the offset 0x00000FE5 specified by the field *PointerToSymbolTable* within the COFF file header. The symbol table for the file **`Employees.obj`** has 65 symbols. Each symbol table entry is an 18-byte long array of records. The format of a record within the symbol table is presented in table as [7] states.

Table 5. The COFF symbol table record structure [7]

Offset	Size	Field
0	8	Name
8	4	Value
12	2	SectionNumber
14	2	Type
16	1	StorageClass
17	1	NumberOfAuxSymbols

For instance, the symbol **`.drectve`** is defined in symbol table as:

```
00001000  00 00 00 FF FF 00 00 03  00 2E 64 72 65 63 74 76  ..........drectv
00001010  65 00 00 00 00 01 00 00  00 03 02 41 00 00 00 00  e..........A....
```

Fig. 7. The symbol **`.drectve`** definition in symbol table of the COFF file

The symbol table record of **`.drectve`** is detailed in Table 6.

Table 6. The COFF symbol table entry explained

Field	Value	Description
Name	0x2E64726563747665	An 8-byte size string: "**`.drectve`**". The string has no null terminator because the length is 8 bytes.
Value	0x00000000	The null value means that the symbol is not assigned to section.
SectionNumber	0x0001	Identifier of the section (first section of the object file).
Type	0x0000	The null value means that the symbol is not a function. There is no type information.
StorageClass	0x03	The offset of the symbol table entry within the section. The entry represents the section name when the field *Value* is zero.
NumberOfAuxSymbols	0x02	2 symbol table entries follow the current symbol.

DOI: 10.12948/issn14531305/18.1.2014.10

The auxiliary symbol records keep the 18-byte size of the symbol table entries. The auxiliary symbols may have different formats than the symbol table entry format.

After the COFF symbol table, the COFF string table is stored. Based on the fields *PointerToSymbolTable* and *NumberOfSymbols*, the computations give the file offset 0x00001477 where the COFF string table for the file **`Employees.obj`** is stored. The content of the string table for the file **`Employees.obj`** is depicted in Figure 8.

```
00001470  00 00 00 00 00 00 00 FD  00 00 00 3F 3F 5F 43 40  ........ ...??_C@
00001480  5F 30 35 42 4E 48 48 41  4F 46 44 40 53 6D 69 74  _05BNHHAOFD@Smit
00001490  68 3F 24 41 41 40 00 5F  5F 52 54 43 5F 43 68 65  h?$AA@.__RTC_Che
000014a0  63 6B 45 73 70 00 40 5F  52 54 43 5F 43 68 65 63  ckEsp.@_RTC_Chec
000014b0  6B 53 74 61 63 6B 56 61  72 73 40 38 00 5F 5F 52  kStackVars@8.__R
000014c0  54 43 5F 53 68 75 74 64  6F 77 6E 2E 72 74 63 24  TC_Shutdown.rtc$
000014d0  54 4D 5A 00 5F 5F 52 54  43 5F 53 68 75 74 64 6F  TMZ.__RTC_Shutdo
000014e0  77 6E 00 5F 5F 52 54 43  5F 49 6E 69 74 42 61 73  wn.__RTC_InitBas
000014f0  65 2E 72 74 63 24 49 4D  5A 00 5F 5F 52 54 43 5F  e.rtc$IMZ.__RTC_
00001500  49 6E 69 74 42 61 73 65  00 3F 3F 30 45 6D 70 6C  InitBase.??0Empl
00001510  6F 79 65 65 40 40 51 41  45 40 50 41 44 48 40 5A  oyee@@QAE@PADH@Z
00001520  00 3F 65 6D 70 4E 61 6D  65 40 45 6D 70 6C 6F 79  .?empName@Employ
00001530  65 65 40 40 51 41 45 50  41 44 58 5A 00 3F 65 6D  ee@@QAEPADXZ.?em
00001540  70 49 44 40 45 6D 70 6C  6F 79 65 65 40 40 51 41  pID@Employee@@QA
00001550  45 48 58 5A 00 3F 70 72  6F 63 44 61 74 61 40 45  EHXZ.?procData@E
00001560  6D 70 6C 6F 79 65 65 40  40 51 41 45 58 50 41 44  mployee@@QAEXPAD
00001570  48 40 5A 00                                       H@Z.
```

Fig. 8. The COFF string table for the file **`Employees.obj`**

The size of the string table is indicated by the first 4 bytes that is 0x000000FD bytes (253 bytes), including the size field itself. The string table contains null-terminated strings that are pointed to by symbol table entries.

3 Detection Techniques of the Maliciousness and Security Management

Object files are binary files stored on computer systems in order to get other binary files that can be executed. Infection of the object files can lead to getting malicious applications executed by host computers or transmitted over the computer network.

Binary file checking inside its structure content is another level of detection of the potentially malicious files.

A detection method uses safe files and malicious files to identify the differences between two groups of files. It is the foundation to implement effective techniques in detection of the malwares by specialized software. The detection techniques consider the binary file layout in order to investigate the presence of the malicious content.

In [8], statistical analyses are performed to get valuable information regarding how the malicious binary files can be detected. Some fields in object file layout can be changed to hide the malicious behavior of the developed application.

In COFF file header **`Employees.obj`**, detection techniques aim the fields *TimeDateStamp*, *NumberOfSections*, *PointerToSymbolTable*, *NumberOfSymbols* and *Characteristics*. In [8], statistical analysis are performed to create indicators used to classify a binary file on fields of the file layout.

The value of the field *TimeDateStamp* is automatically created by compiler or linker. Its value can be changed when its place is exactly known. A range of dates and hours is established to identify the binary file within the normal distribution [8]. This detection technique can be used to establish the malicious file sources like time zone, country of origin and activity time slots.

In binary file layout, sections bound the file content in different areas whose content has a particular role in the executable application. Even though there are a large number of section types and names, the number of sections in a non-malware file is small. The statistical

DOI: 10.12948/issn14531305/18.1.2014.10

data achieved in [8] for the field *NumberOfSections* can be used as a detection technique. For the object file **`Employees.obj`**, the field value is 1, so there is a small possibility that the object file to be a malware, according to attribute *NumberOfSections* analysis.

The fields *PointerToSymbolTable* and *NumberOfSymbols* are used to bound the file area where debugging information is stored. Because the COFF debugging information is deprecated in favour of Program Debug Database (PDB) file, the value of field *PointerToSymbolTable* should be zero [8]. The values of two fields can be correlated to other fields in order to detect the file maliciousness.

The field *Characteristics* has a flag role to specify a combination of attributes of the binary file. There are some attribute combinations indicating a possible infection of the binary file [8].

The object file **`Employees.obj`** has not optional header.

Regarding the object section header of the file **`Employees.obj`**, the detection techniques aim the fields *VirtualSize*, *SizeOfRawData*, *NumberOfRelocations*, *NumberOfLinenumbers*, *PointerToRawData*, *PointerToRelocations*, *PointerToLinenumbers* and *Characteristics*.

For the file **`Employees.obj`**, the *VirtualSize* is zero and *SizeOfRawData* is 65. As detection technique, relation *VirtualSize* < *SizeOfRawData* highlights a possible issue, but the binary file is an object file, so it is normal that *VirtualSize* to be zero because the object code is not stored in memory [8].

As detection technique, the values of *NumberOfRelocations* and *NumberOfLinenumbers* are not malicious issues for the binary file **`Employees.obj`**.

According to [8], the detection rule *PointerToRawData = 0* has a high detection rate, but also a high false positive rate. For the file **`Employees.obj`**, *PointerToRawData* is a non-null value, so it cannot be used as detection technique. The null values of *PointerToRelocations and PointerToLinenumbers* are not used in maliciousness detection.

The three flags stored by the field *Characteristics* in the section header of the file **`Employees.obj`**, has not a big impact in malicious detection as states [8].

It is necessary to avoid that own computer application to be classify as malware because one or more binary files (object, libraries or executable) contains malicious code inserted accidentally or deliberately. Computer software producers hold the program source code of the software and they can perform code review processes for a robust and viable final software product.

BSIMM-V Project [9] proposes three levels of code review for a better quality control of the software product. The aim is detection and correction of the security bugs both the software quality and other software which reuses parts from another. The three levels of code review are [9]:

1. Code review is manual or automated and the reporting is centralized – all software projects have to be examined in terms of code review; the code review has to be imposed by management and the intelligence extracted from review processes is stored in a centralized repository; the level includes the following activities [9]:
   - Create a list of the most important security bugs – the reviewer attention is driven by the most common security bugs; the security bugs are extracted from public sources and the reviewer's experience gathered from code review, testing and actual incidents; the list has to be tailored to organization's bug priorities depending on the features of the software products developed by the organization;
   - Perform ad hoc review – the code review is made during the software development life cycle before reaching its maturity level;
   - Perform manual and automated review – increasing the efficiency and consistence of the code review process by including the static analysis in the process; also, automation brings additional information to the review-

DOI: 10.12948/issn14531305/18.1.2014.10

ers in a shorter time;
- Enforce the code review for all software projects – software release is possible when the code review process has been done and accomplishes a minimum acceptable standard before its shipping; software products addresses different security issues having low-risk or high-risk features;
- Implement a centralized reporting – a bug list repository is created to store the details of all identified security bugs during the code review and their tracking; the bug repository can be used to make summary and progress reports, and it is an excellent source of training curriculum;

2. Code review is made by standards enforcement – the rules and best practices stated in standards must be follow during a code review process; the standard content is the result of the best specialists experience and interdisciplinary points of view; the second level includes the activities [9]:
   - Enforce coding standards – coding standard that are not followed by developers are a sufficient reason to reject a software product of parts of it; coding standards can be published as developer guidelines or within the Integrated Development Environment (IDE);
   - Assign tool mentors – one or more experts in code review are assigned to a development team increasing the efficiency and effectiveness of the review process; tool expert provide information regarding configuration of the review tools and how the results have to be interpreted;
   - Custom rules for automated tools – static analysis is tailored to improve efficiency and reduce the false positives; it is made by the assigned tool mentors to find coding errors;
3. Automated code review with tailored rules – reviewers have to build a tool that find and remove the security bugs from the entire codebase; the level aims the following activities [9]:
   - Create the tool – the results of multiple analysis techniques are combined in a single information review flow and reporting solution; analysis techniques can combine static and dynamic analysis; it leads to better informed risk mitigation decisions;
   - Remove new bugs from the codebase – when a new security bug is found, the rule that has been used in bug seeking is used to find the all occurrences in the entire codebase;
   - Automate detection of the malicious code – malicious code is found by automate code review, out-of-the-box automation and custom rules for static analysis;

Malware detection techniques have the following approaches [2]:
- Static detection – uses the syntax or structural information to establish whether a program or process is malicious;
- Dynamic detection – uses the runtime information to determine whether a program or process is malicious; the runtime information aims the resources used by and how they are used by the process;
- Hybrid detection – combines static and dynamic detections

Detecting malicious code stored by binary files is a feature that must be implemented in detection software named malware detector.

4 Conclusions

The paper describes what structural properties of the binary object files must be considered by a malware detector during the static detection process. The most important structural parts of an object file are presented and described together with applying of statistical analyses the object file content.

The object file is the result of compile process. Lack of malicious code is also assured when the program source code is reviewed and accomplishes the minimum coding standards. If the program source code lacks, then a malware detector can be used to iden-

DOI: 10.12948/issn14531305/18.1.2014.10

tify the maliciousness of a process or program stored in a binary file.

References

[1] A. Danehkar, *Inject your code to a Portable Executable File*, December 27, 2005, http://www.codeproject.com

[2] N. Idika and A. P. Mathur, *A Survey of Malware Detection Techniques*, Purdue University, February 2, 2007

[3] M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format", *MSDN magazine*, http://msdn.microsoft.com /en-us/magazine/cc301805.aspx

[4] M. Popa, "Binary Code Disassembly for Reverse Engineering", Journal of Mobile, Embedded and Distributed Systems, vol. 4, nr. 4, 2012, pp. 233 – 248

[5] M. Popa, "Requirements of a Better Secure Program Coding", *Informatica Economică*, vol. 16, nr. 4(64), 2012, pp. 93 – 104

[6] M. Popa, "Techniques of Program Code Obfuscation for Secure Software", *Journal of Mobile, Embedded and Distributed Systems*, vol. 3, nr. 4, 2011, pp. 205 – 219

[7] Microsoft Portable Executable and Common Object File Format Specification, Revision 8.3, February 6, 2013

[8] J. Yonts, *Attributes of Malicious Files*, SANS Institute InfoSec Reading Room, June 30, 2012

[9] BSIMM-V Project, http://bsimm.com, http://bsimm.com/online/ssdl/cr/

Marius POPA has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2002. He holds a PhD diploma in Economic Cybernetics and Statistics. He joined the staff of Academy of Economic Studies, teaching assistant in 2002. Currently, he is Associate Professor in Economic Informatics field and branches within Department of Economic Informatics and Cybernetics at Faculty of Cybernetics, Statistics and Economic Informatics from Bucharest University of Economic Studies. He is the author and co-author of 9 books and over 140 articles in journals and proceedings of national and international conferences, symposiums, workshops in the fields of data quality, software quality, informatics security, collaborative information systems, IT project management, software engineering.

DOI: 10.12948/issn14531305/18.1.2014.10