Excel@FIT 2015
http://excel.fit.vutbr.cz

Generic Unpacker of Executable Files

Marek Milkovič*

```
.MPRESS2:0042D339 loc_42D339:                             ; CODE XREF: sub_42D175+267↓j
.MPRESS2:0042D339                 shl     [esp+8Ch+var_4C], 1
.MPRESS2:0042D33D                 mov     ecx, [esp+8Ch+var_4C]
.MPRESS2:0042D341                 lea     edx, [esi+esi]
.MPRESS2:0042D344                 mov     ebp, [esp+8Ch+var_78]
.MPRESS2:0042D348                 and     ecx, 100h
.MPRESS2:0042D34E                 cmp     [esp+8Ch+var_44], 0FFFFFFh
.MPRESS2:0042D356                 lea     eax, [ebp+ecx*2+0]
.MPRESS2:0042D35A                 mov     [esp+8Ch+var_50], ecx
.MPRESS2:0042D35E                 lea     ebp, [edx+eax]
```

Abstract

Executable file packing is a process used for compression or protection of these files. The behavior and intent of such packed executable files is difficult or even impossible to analyze. If we want to analyze the original code, we need to detect the used packer and unpack the executable file with a tool called unpacker. This paper describes the methods used for packing and unpacking of the executable files as well as the implementation of an easily and quickly extensible unpacker, which is going to be used in a decompiler developed by AVG Technologies. This unpacker provides the interface for plugins, which extend the set of supported packers. Unpacking plugins aim at the methods for unpacking without actually running the packed program; thereby providing security measures and targeting the architecture and platform independent unpacking. A newly proposed generic unpacker achieves comparable results with unpackers used in practice and even outpace them in a few aspects. It shows that even static unpacking methods can produce accurate results.

Keywords: Unpacking — Decompilation — Retargetable Decompiler — Reverse Engineering — Executable File — Packing — Malware — Compression

Supplementary Material: *N/A*

*xmilko01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Reverse engineering is a process of gaining knowledge about an engineered object while it is deconstructed; so much that its inner structure and architecture are revealed [1]. Software reverse engineering is a reverse engineering in which the object of research is software. This paper focuses on this kind of reverse engineering.

One of the main pillars of software reverse engineering is decompilation—reverse compilation [2]. It is a process of translating machine code to high-level language (HLL). The author of the program might have taken measures that protect his or her program against reverse engineering. This procedure is often seen in malicious software—malware. One kind of protection is packing. It is stated that 80% to 90% of malware is packed [3]. Packed software is compressed into a self-decompressing executable file, which may also implement anti-debugging methods. Unfortunately, the content is decompressed just into memory in most cases. Decompilation of such files may end up with false results or fail completely. Therefore, the file needs to be unpacked by unpacker at first.

The goal of this paper is to describe the design and implementation of a generic unpacker. It is part of the decompilation chain in a retargetable decompiler developed by AVG Technologies[1]. The generic unpacker aims to fill the gap between decompilers and unpackers by providing the unpacked file in decompilable format. It also focuses on the unpacking without running the

[1] https://retdec.com/

packed program, thus achieving more security, and architecture/platform independence. Extensibility is provided by the plugin interface.

AVG Technologies already has an internal generic unpacker. However, it is dedicated just for the malware detection. It does not solve any problems that are noticable only when the file is being decompiled. The newly proposed generic unpacker deploys several techniques that are absolutely redundant in the antivirus unpackers. This makes it unique.

Section 2 describes the retargetable decompiler and its structure. Section 3 gives an overview about the executable file and Section 4 about the packing and unpacking of such files. Section 5 is dedicated to the generic unpacker itself, while Section 6 describes the generic unpacker plugins. In the end, the tests are described in Section 7.

2. Retargetable Decompiler

The retargetable decompiler is a decompiler developed by AVG Technologies. The targeted architecture can be configured via the input configuration file, which describes the architecture. The supported architectures are x86, ARM, MIPS, PowerPC, and PIC32.

The executable file needs to go through a preprocessing phase before entering the decompiler. Since this is the phase we need to modify to integrate the generic unpacker, we will describe it in more detail than the others.

Preprocessing analyzes the input executable file and tries to gather as much information as possible, e.g. class of executable file (32 or 64 bit), target architecture, executable file format, used compiler and packer. The compiler and the packer detection is based on the signature database or heuristics. It is comparing the bytes at the entry point of the program and calculates the best matches. The executable file needs to be unpacked in the event of positive packer detection. Currently, it uses the single purpose third-party unpackers . The result of preprocessing is a configuration file in XML format and an unpacked executable file. The scheme of preprocessing is displayed in Figure 1.

The three main modules of the retargetable decompiler are front-end, middle-end, and back-end. They perform various transformations to translate the machine code of the program to the HLL. First, machine code is transformed into LLVM IR (Low Level Virtual Machine Intermediate Representation)[2] code in the front-end. It is being optimized in the middle-end and finally transformed into the HLL in the back-end. Currently supported HLLs are C and Python.

[2] http://llvm.org/docs/LangRef.html

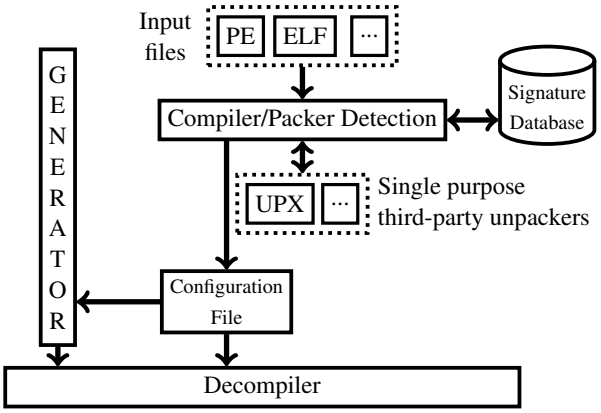

Figure 1. Preprocessing scheme. [3]

3. Executable Files

An executable file is a file that contains the transcript of a program using executable instructions. Executable files belong to the family of object files together with linkable files and libraries.

The content of an executable file has to be in a format the targeted system can understand. It contains the following information in general [4]:

Header — Basic information about the file, identification of the format, and many others.
Object code — Binary executable instructions.
Relocation entries — The addresses that need to be modified while linking or loading the program into memory.
Symbols — Symbols (functions and variables) exported by the file and the locally defined symbols.
Debugging information — Variable names, line number relations with the instructions, etc. This information is optional, but useful for debugging.

The loader performs the start of a program in the system. It takes care of importing the required symbols, creating runtime structures, etc.

The executable files are divided into parts called **sections**. Sections separate the content of the file logically by its purpose. There are often sections separating the code from the data. The code should be executable and readable, while the data should be readable and writable.

Programs do not always use just their own code. They also use the system or third-party functions from libraries. The program can be linked against a static or dynamic library. The code from the library is put directly into the program itself when linked against the static library. On the other hand, the linking against the dynamic library causes just the import symbol table to be filled with the information that helps to find these symbols during the start-up of the program.

The general structure of the executable file can be seen in Figure 2. Information such as relocation entries, symbols, etc. are placed in their own sections or they are included in the headers. This is specific for the different executable file formats used among various platforms.

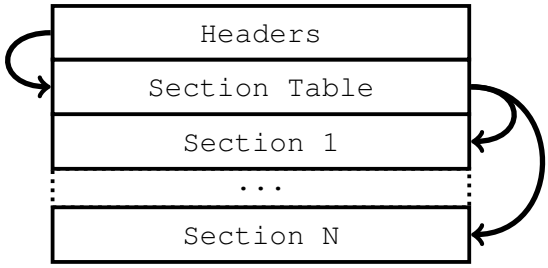

Figure 2. The general structure of an executable file.

For instance, the format used on the current Windows systems is called Portable Executable (PE)[3]. It stores the import data structures in tables called ILT (Import Lookup Table) and IAT (Import Address Table). The loader reads the entries in ILT and based on their values, it imports the symbols while filling the IAT with actual addresses of the symbols on the program start or load. The code that uses the imported symbol accesses the IAT to get the required symbol address. As we will mostly focus on the PE format further in the paper, we are not going to describe the other formats.

4. Packing & Unpacking

Packing is a process of compression of the code or even data of an executable file. More importantly, the new packed file still remains executable and performs the same action as the original file. Tools that perform packing are called packers.

There are multiple reasons for packing. One reason can be a reduction of the executable file size. However, it is mostly an effort to hide the real code of the program because of its malicious intent, or to protect the proprietary solutions. Malware actually uses packing mostly as it complicates the analysis of such file. Packed malware may not be identified by an antivirus software as malicious and remain in the system.

Even though many packers exist and new ones are still being created, all of them work on a similar basis. They create the new file and insert the compressed content of the original file as its data. The code that is able to decompress the data, is then inserted too. This code is often referred to as an **unpacking stub** [5]. The import symbol table is modified or completely erased to hide the behavior even more by the vast majority of packers. The structure of the original and packed file can be seen in Figures 3 and 4.

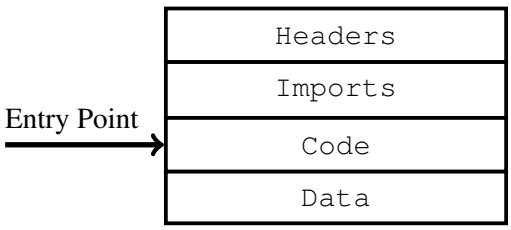

Figure 3. Structure of original file

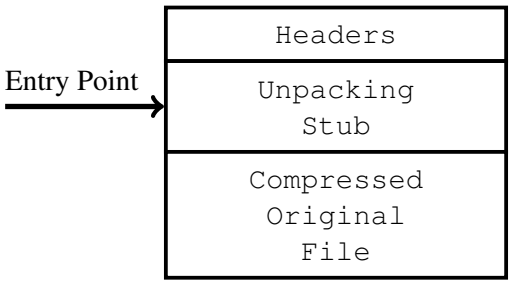

Figure 4. Structure of packed file

The purpose of an unpacking stub is to decompress the original file content to the memory or disc. However, the most packers just decompress to the memory nowadays. The unpacking stub fixes the imports if they are damaged. In the end, the control of the program is redirected to the original entry point (OEP).

There are more methods of unpacking that can be divided into two groups by different criteria. First, we can divide it into **manual** or **automatic** unpacking [5] by what performs the unpacking. **Manual** unpacking is being done by using debugging, disassembling, and other tools by the analyst himself. **Automatic** unpacking is performed by unpackers. The analyst does not even need to have underlying knowledge of how the packer works. The second division is based on the approach that is taken to unpack the program. Options are **static** and **dynamic** unpacking [5]. While **static** unpacking does not run the packed program and tries to simulate the unpacking stub, the **dynamic** approach runs the packed program and lets the unpacking stub do all the work. Memory is just dumped to the file as soon as it contains the decompressed content.

The majority of the available unpackers are using the **automatic dynamic** method as it is easy to implement, but at the cost of security. The origin of a packed program may not be known, putting the user at risk. Some advanced packers are nearly impossible to unpack automatically [5]. Manual methods then need to be used together with automatic.

Generic unpackers are group of special unpackers. They can use various techniques mentioned above. They are intended for unpacking the set of specific

[3] http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

packers, not just one single packer. Very few generic unpackers exist. Here is a short list of them:

FUU[4] — The generic unpacker written in assembly. It uses the dynamic approach. The development is stalled for years now.
PackerBreaker[5] — This one uses emulation technique. It is a dynamic approach in a virtual environment. It provides only a graphical interface, making it hard to integrate with other tools. This unpacker even protects itself from being reverse engineered. It cannot run alongside a debugger or any process monitor. This raises a lot of controversy in the community. The last PackerBreaker release was in 2012.
Internal antivirus unpackers — Nearly all antivirus softwares use their own unpackers for malware detection. However, they are not publicly available as standalone tools. AVG antivirus uses one too. It was provided for the purpose of this work.

5. Generic Unpacker

The inner structure of the generic unpacker consists of three main parts—modules. These are **core**, **unpacking library**, and **plugins**. Each of these modules has its own purpose.

Core is responsible for plugin management. It loads the plugins at the start-up and it registers them. It initiates the unpacking process by finding the matching plugins for currently unpacked file and redirect control to the plugin through the API.

Unpacking library is a library used by plugins. It contains a set of common utilities that makes the unpacking easier such as decompression algorithms and functions to extract bytes from file. It focuses on the static unpacking methods.

Plugins represent the standalone unpackers themselves. Every plugin should take care of one packer but multiple versions. The packer and version it supports is passed to the core during the plugin registration. They are just regular shared/dynamic libraries, which must follow certain API rules to be treated as a valid generic unpacker plugin.

5.1 Integration to Retargetable Decompiler

The structure of the retargetable decompiler was described earlier in Section 2. It was using single purpose third-party unpackers, which bring the disadvantages mentioned in Section 4. This part of preprocessing needs to be replaced completely with the new generic unpacker and its plugin system. The updated schema is depicted in Figure 5. The new parts are highlighted with a red color.

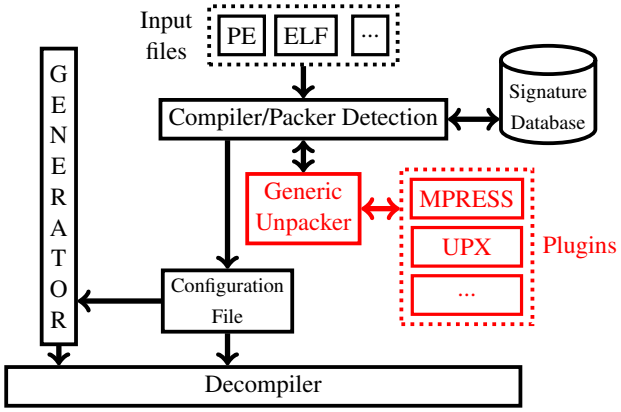

Figure 5. Updated preprocessing scheme. [3]

The generic unpacker uses an XML configuration file to get a list of possible packers and percentage probabilities for every packer and its version. The matched plugins are then run in order from the highest to the lowest probability.

6. Generic Unpacker Plugins

This section is dedicated to existing plugins in the generic unpacker and those that are still in development. We will take a look at two real packers; how they work, and how they are unpacked.

6.1 MPRESS

MPRESS[6] is a packer from Matcode Software. It is capable of packing PE and .NET executable files, but we will focus on PE. The current version is 2.19, but there are still files packed with 1.xx versions on the Internet. It uses an LZMAT[7] compression algorithm for small files and LZMA[8] for larger files. Both algorithms belong to the LZ compression algoirthms family. LZMA is well known and partially documented. However, LZMAT is custom made and not documented at all.

The MPRESS packed file contains just two sections, `.MPRESS1` and `.MPRESS2`. Exceptions are only resources, which are left untouched if present in the original file. The imports are totally destroyed. The section `.MPRESS1` contains the compressed content and `.MPRESS2` represents the unpacking stub. After the unpacking stub decompresses the content of section `.MPRESS1` into the same section, the code jumps to the routine that fixes imports and relocation entries. The decompressed section contains import

[4] https://code.google.com/p/fuu/
[5] http://www.sysreveal.com/tag/packerbreaker/
[6] http://www.matcode.com/mpress.htm
[7] http://www.matcode.com/lzmat.htm
[8] http://www.7-zip.org/sdk.html

hints, which are just ill-formed ILT. The fixing routine uses them to fill the real IAT, taking the role of a loader. The control is then transferred to the OEP.

To unpack MPRESS, it is required to recognize whether LZMA or LZMAT was used. This can be detected from the bytes at the entry point, as each have its own specific signature. This can also recognize the different versions of MPRESS. After the content of the section `.MPRESS1` is decompressed, the positions of the real IAT and OEP can be read from it. The last step is to fix the imports based on the import hints.

A problem arises when the unpacked file is decompiled. Almost the whole original file is present in just one huge section in this new unpacked file. This section contains the OEP so it is marked as the code. The retargetable decompiler tries to decompile everything in this section, even though there are data. This leads to false results of decompilation.

This is the reason why we have deployed heuristics that analyze the unpacked section based on known facts and observations. These heuristics detect where are borders of the original sections. The big unpacked section is split at specific point into two sections. This continues until it is not able to split the section anymore. The sections of the packed file can be seen in Figure 6. The unpacked file without any heuristics applied is displayed in Figure 7. The impact of the heuristics can be seen in Figure 8.

6.2 UPX

UPX[9] is one of the most popular packers since it is open-source. It is often modified, thus cannot be unpacked by a default UPX unpacker. It uses NRV[10] compression algorithms, but it also has an option for LZMA compression. It has a wide scale of supported executable file formats like PE, ELF, Mach-O, and also supports multiple architectures like x86, ARM, MIPS, and PowerPC.

It works very similar to MPRESS. It creates two sections `.UPX0` and `.UPX1` in the packed file. The section `.UPX0` does not contain anything in the beginning. The unpacking stub with the compressed content is placed in the section `.UPX1`. The content is unpacked into the section `.UPX0`. Routines for fixing imports, relocations, and others are part of the unpacking stub.

The imports are not destroyed completely. There is one remaining import from each library. In this way, the unpacking stub just needs to add the other imported symbols to the import symbol table. The loader takes care of the library load.

This plugin is still in development. The UPX samples are being reverse engineered and analyzed.

7. Experiments & Testing

The tests aim at the MPRESS plugin of the generic unpacker. We have downloaded 99 in-the-wild samples from VirusTotal[11] that were tagged as an MPRESS samples. These were analyzed prior to the testing. Only 91 of them were truly executable. These 91 samples are used as the test suite for MPRESS unpacking.

The two important criteria are observed—successful unpacking ratio and executability ratio. The successful unpacking ratio compares how many executable files were successfully unpacked to the number of files in the whole test suite. The executability ratio then compares how many of the unpacked files remained executable to the number of successfully unpacked files. The reason why we observe the executability ratio is that we may still be able to statically analyze the file and get the right results.

The results of the MPRESS plugin are compared to the results of the internal AVG unpacker and PackerBreaker as can be seen in the Table 1.

Table 1. The results of unpacking

Unpacker	Successful unpacking ratio	Executability ratio
Generic unpacker	**93.41%**	**100%**
Internal AVG unpacker	93.41%	100%
PackerBreaker	93.41%	95.29%

The generic unpacker and internal AVG unpacker ended up with the same results. They unpacked 85 of 91 samples, while every single unpacked file remained executable. The PackerBreaker also unpacked 85 of 91 files, but 4 of them were not executable. The 6 samples that neither of the unpackers was able to unpack require manual unpacking. They are all packed with MPRESS, but they have obscured important information for unpacking. The most of them are packed twice using a very simple packer on the top of MPRESS.

The decompilation testing is a hard task on these downloaded samples, since we do not have the original source code to compare the results. Even if we make our own samples, the output of the other unpackers is not decompilable. The other unpackers focus mainly on the executability, they do not care about the getting

[9] http://upx.sourceforge.net/
[10] http://www.oberhumer.com/products/nrv/
[11] https://www.virustotal.com/en/

Name	Virtual Size	Virtual Address	Size of Raw Data	Pointer to Raw Data	Characteristics	Pointing Directories
.MPRESS1	0001E000h	00401000h	00005000h	00000200h	E00000E0h	
.MPRESS2	00000D87h	0041F000h	00000E00h	00005200h	E00000E0h	Import Table; Import Address Table
.rsrc	0001095Ch	00420000h	00010A00h	00006000h	C0000040h	Resource Table

Figure 6. The sections in the packed file.

Name	Virtual Size	Virtual Address	Size of Raw Data	Pointer to Raw Data	Characteristics	Pointing Directories
.MPRESS1	0001E000h	00401000h	00005000h	00000200h	E00000E0h	Import Address Table
.MPRESS2	00000D87h	0041F000h	00000E00h	00005200h	E00000E0h	
.rsrc	0001095Ch	00420000h	00010A00h	00006000h	C0000040h	Resource Table
.imports	00001000h	00431000h	00000E00h	00016A00h	40000040h	Import Table

Figure 7. The sections in the unpacked file (no heuristics used). `.MPRESS1` contains the whole unpacked content of the original file. The unpacking stub in `.MPRESS2` is zeroed, but the section is left in the file as it may still be referenced from the header. The new section containing ILT was added.

Name	Virtual Size	Virtual Address	Size of Raw Data	Pointer to Raw Data	Characteristics	Pointing Directories
.text	00002000h	00401000h	00002000h	00000400h	E0000060h	
.data3	00007000h	00403000h	00007000h	00002400h	C0000040h	
.data0	00015000h	0040A000h	00015000h	00009400h	C0000040h	Import Address Table
.MPRESS2	00000D87h	0041F000h	00000E00h	0001E400h	C0000040h	
.rsrc	0001095Ch	00420000h	00010A00h	0001F200h	C0000040h	Resource Table
.imports	00001000h	00431000h	00000E00h	0002FC00h	40000040h	Import Table

Figure 8. The sections in the unpacked file (heuristics used). The section `.MPRESS1` was divided into smaller sections `.text`, `.data3` and `.data0`. The section `.text` contains the entry point so it is the only one marked as code. Even though the others are data, it does not matter during the execution.

the file into its original state as much as possible. The only decompilable unpacked files are those from the new generic unpacker.

8. Conclusions

This paper described the generic unpacker of executable files. First, it provided information about the retargetable decompiler it is going to be integrated in. Then, brief insight of executable files, packing, and unpacking was given. In the end, the inner structure of the new generic unpacker was described together with its plugins. The existing unpacking plugins were then tested on the real malware samples.

The described design, implementation, and packer analysis are all author's contribution. The decompression algorithms are inspired by an existing solutions.

The generic unpacker was shown to be reliable and produced results comparable to the unpackers already used in practice for many years. Even the static unpacking resulted in accurate results. It unpacked 93.41% of all tested packed executable files, while all of them remained executable. It even defeated the PackerBreaker unpacker. The static unpacking allowed us to test the unpacker on the real malware samples without any damage. Last, but not least, it filled the gap between unpackers and decompilers by providing the required output using hueristics for the section detection.

Its future depends on how fast can plugins be developed and our ability to keep up with the packer creators. These two points are the biggest issues in this field as we can see in cases of PackerBreaker and FUU.

A possible improvement for the future is to extend the unpacking library, which is very thin at the moment. This requires the implementation of more plugins to combine all the common techniques together.

References

[1] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Indianapolis, IN, 2005.

[2] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[3] Jakub Křoustek and Dušan Kolář. Preprocessing of binary executable files towards retargetable decompilation. In *ICCGI'13*, pages 259–264. IARIA, 2013.

[4] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, 2000.

[5] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.