Generic Unpacking Method Based on Detecting Original Entry Point

Ryoichi Isawa[1], Masaki Kamizono[1,2], and Daisuke Inoue[1]

[1] National Institute of Information and Communications Technology, Tokyo, Japan
[2] SecureBrain Corporation, Tokyo, Japan
{isawa,masaki_kamizono,dai}@nict.go.jp

Abstract. In this paper, we focus on the problem of the unpacking of packed executables in a generic way. That is, we do not assume specific knowledge about the algorithms used to produce the packed executable to do the unpacking (i.e. we do not extract/create a reverse algorithm). In general, when launched, a packed executable will first reconstruct the code of the original program, write it down someplace in memory and then transfer the execution to that original code by assigning the Extended Instruction Pointer (EIP) to the so-called Original Entry Point (OEP) of the program. Accordingly, if we had a way to accurately identify that transfer event in the execution flow and thus the OEP, we could more easily extract the original code for analysis (cf. by inspecting the remaining code after the OEP was reached). We then propose an effective generic unpacking method based on the combination of two novel OEP detection techniques, one relying on the incremental measurement of the entropy of the information stored in the memory space assigned to the unpacking process, and the other on the incremental searching and counting of potential Windows API calls in that same memory space.

Keywords: Packer, Kernel mode, NX bit, Malware analysis.

1 Introduction

Malware authors often make use of packers to protect their malware programs from code analysis. They can easily *pack* (compress and/or encrypt) the original code of a malware program with the packers (e.g., UPX, ASPack, and PECompact), therefore we have to extract the hidden original code from the packed malware before code analysis. Because many types of packer exist and their packing algorithms vary widely, manual unpacking operations which require us to detect the packer type used and to infer the packing algorithm can induce huge additional analyzing costs.

Many anti-malware analysts rely on automated generic unpacking techniques to skip the manual unpacking operations. The existing methods [1–6] can be classified into three groups according to their purposes: to obtain the original code of a packed malware program [1], to detect the OEP in addition to obtaining the original code [2–4], and to detect malware with an anti-virus software program even though the malware program is packed [5, 6], where OEP means the address which indicates the beginning point on the original code loaded into the memory. In this paper, our purpose is the second one, which is to detect the OEP in addition to obtaining the original code.

Finding the OEP of a packed program should provide us with several benefits in terms of code analysis. If we disassemble the original (binary) code starting from a wrong address, we are to get a wrong assembly code because some architectures like x86 architecture apply an instruction set with a variable bit length for the representation of instructions. In contrast, starting from the OEP, we can transfer the original code correctly and obtain useful information from the program, (e.g., the names of stolen files and servers used by attackers). For instance, such information can be used in computer forensics to track malware authors or attackers.

We focus on the elementary behavior of packed executables to perform generic unpacking. A typical packer such as UPX and ASProtect compresses or encrypts a program code and adds its unpacker code to the packed one. When the packed program is run, it executes the unpacker code first, then the unpacker code decrypts and writes the original code into the memory. After the unpacker code completes the decryption, the extended instruction pointer (EIP) moves from the unpacker code to an address in the original code. Note that OEP is the address to which the EIP moves from the unpacker code. To summarize, the elementary behavior is that the original code will be decrypted and written first and then executed, whichever packer is used.

In this paper, we propose a novel generic unpacking method featuring two OEP detection approaches: one is based on an entropy analysis and the other is based on the number of API-call instructions present in the memory. The first approach focuses on the entropy score of the decrypted original code. Generally, the entropy score of non-random data is low, and the entropy score of a set of instructions will be also low because it consists of often-used instructions (e.g., `mov`, `push`, `call`, `cmp`, and `add`). When caching a page fault, if the entropy score is lower than a given threshold value, that approach decides that the address to which the EIP points is the OEP. At the end of the decryption, the number of instructions should become the highest in the decrypting process. The second approach focuses on API-call instructions such as '`call APIaddress`'. The intuition is that we cannot find API-call instructions contained in the packed original code, but we can find them after the decryption is completed. If the number of API-call instructions is higher than another given threshold value, that second approach decides that the address of the EIP is the OEP. Our method outputs an address as the OEP when both the first approach and the second one reach the same conclusion at which point it outputs a memory dump of all memory areas that the packed program loaded into the memory.

Our main contribution is to propose two OEP detection approaches which are greatly different from existing methods. In particular, our method focuses on the changing information of the original code, whereas almost all existing methods focuses on the unpacker code. Distinct OEP detection techniques are strongly needed because we can combine unpacking methods to implement a simple but practical solution. That is, each method independently detects the OEP from OEP candidates and we finally determine the best candidate as the OEP which wins most votes. The experiment shows that our method can unpack 14 of 20 files packed with distinct packers.

The rest of the paper is organized as follows: Section 2 introduces related works. Section 3 presents our method. Section 4 shows an experimental evaluation of our method. Finally, Section 5 concludes the paper.

2 Related Works

2.1 Commonly Used Techniques

The original code of a packed file, independently of the packer that was used, will be decrypted and written, then the original code will be executed. Therefore the original code will be contained inside the instructions and data that are dynamically generated.

There exist two techniques that can detect dynamically generated instructions and data: one relies on disassembly and the other relies on page protection settings named W $\oplus$ X page protection [7], where the 'W' and the 'X' stand for write and execute, respectively. The former technique disassembles each instruction and catches write operations (e.g., '`mov %eax, [%edi]`' and '`push %eax`' [2]). The technique can learn the address of the current execution from the EIP. If a dynamically generated code such as the code written by '`mov`' or '`push`' are executed, the technique memorizes the address of the written code and the code itself as an OEP candidate and part of the original code, respectively. The latter technique modifies the page protection settings to be executable/read-only and read/write-only in sequence. The technique gives the settings of executable/read-only to the whole memory or a certain memory area just after loading a packed file on the memory. When the unpacker code writes instructions to one of the unwritable areas, the technique can catch a page fault and modifies the area to be read/write-only. Similarly, when the unpacker code executes instructions of the non-executable area, the technique can catch a page fault and recognize dynamically generated instructions.

As we mentioned above, we can recognize dynamically generated instructions, but many page faults occur in practice. For example, Guo et al. show that 11 exceptions occur when they execute a file packed with UPX under only W $\oplus$ X page protection [6].

2.2 Existing Methods

Renovo [2] monitors jump instructions such as '`jmp`' to detect the OEP of a packed file. If the EIP moves to a dynamically generated instruction through a jump instruction, Renovo decides that the address of the generated instruction is the OEP. Because Renovo monitors and disassembles each instruction, it incurs a significant overhead. Note that we must apply single stepping and disassembly if we have to recognize instructions such as jump instructions.

Kawakoya et al. [4] focus on a packed file's memory access behavior to detect the OEP, where memory access means operations of read, write, and execute. They define an equation that represents the memory access behavior and they input a type of memory access, which is read, write, or execute, to the equation in order to obtain a representing value of memory access. If values that the equation outputs are changing rapidly, Kawakoya et al.'s method decides that the changing point is the OEP.

OmniUnpack [5] applies W $\oplus$ X page protection. Every time a dynamically generated instruction is executed and the instruction is for executing dangerous system calls (e.g., registry/network/file-write operations and process creation), OmniUnpack invokes anti-virus software to scan newly generated code. Indeed, if OmniUnpack was to invoke an anti-virus software program for all exceptions, it would incur a significant overhead. OmniUnpack aims to detect a malware program with anti-virus software even if the malware program is packed, and it does not aim to detect the OEP. OllyBonE [8] is a tool that is used for implementing OmniUnpack. OllyBonE supplies W $\oplus$ X page protection.

Justin [6] applies W $\oplus$ X page protection and aims to detect packed malware programs with anti-virus software as well as OmniUnpack. One of the differences between Justin and OmniUnpack is that Justin finds out OEP candidates and an anti-virus software program scans the whole memory starting from each OEP candidate. Guo et al. propose three approaches to guess if the address on which a page fault occurs is the OEP. The first is to check if dynamically generated code area contains unpacker code when a page fault occurs. The second is that several stack pointers are the same as their initial state. The third is to check if a packed file accesses the command-line argument. Justin independently combines each approach with W $\oplus$ X page protection. If the address appears to be the OEP, an anti-virus software program scans the whole memory starting from the address. Justin can supply OEP candidates but does not specify the OEP.

3 Proposed Method

3.1 Workflow

Our system consists of a manager program and an unpacking driver. For generic unpacking, just after the manager runs a packed PE file, it stops the process. At which time, a signal is sent to the driver saying that the process just ran. Having received it, the unpacking driver memorizes the initial state of the memory that the process uses. After the driver turns on NX bit flags on the memory, the manager restarts the process. When the process executes an instruction whose NX bit flag is on, a page fault occurs. After catching it, the driver stops the process. If a memory page that process is accessing at the time is different from the initial state, the driver registers/saves the address on which the page fault occurs as an OEP candidate. The driver then checks if the candidate is correct with the entropy-based approach and the approach focusing the number of API-call instructions. If both approaches decide that the candidate is correct, the driver outputs the OEP and obtains a memory dump; otherwise, the manager restarts the process. Our system repeats the unpacking work until the OEP is determined or at most 50 times.

3.2 OEP Detection Technique Based on Entropy Analysis

We apply Lyda et al.'s file type identification method [9] as an entropy-based OEP detection technique. Their method can recognize which file type, text, executable, encrypted, or packed file as follows. It divides a whole file into successive blocks of size q bytes and calculates an entropy score for each block using the following equation:

$$H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i) \tag{1}$$

where q is a given value and $q \in \{1, 2, \cdots\}$, x means one of the blocks, n denotes the number of unique one-byte values in x, and i and $p(i)$ denote the i-th smallest one-byte value in x and a frequency of the i-th smallest one-byte value in x, respectively. We express, for example, x as hexadecimal numbers "12 34 AB 34 56". In this case, $q = 5$, $n = 4$, the first smallest one-byte value – the fourth one are 12, 34, 56, and AB, and pairs of $(i, p(i))$ are $(1, 0.2)$, $(2, 0.4)$, $(3, 0.2)$, and $(4, 0.2)$, respectively. Lyda's method then calculates the average and the maximum of the entropy scores of all blocks. If both the average and the maximum are smaller than given threshold values, Lyda's method infers that the file is neither encrypted nor packed.

As Lyda's method is not suitable for generic unpacking, we examine entropy of packed programs and customize the method. That is, we consider encrypted data as packed data and do not use the maximum value of entropy. The reason why we do not use the maximum is that a packed file definitely contains packed data, and the maximum is always high due to the packed data, no matter how much progress the unpacker code makes. If the average entropy score of the file is smaller than given threshold m, our entropy-based approach decides that the OEP candidate is correct.

3.3 OEP Detection Technique Focusing on API-call Instructions

This approach searches for API-call instructions as follows. It takes as a list API addresses of DLLs exported in the memory and searches for the API addresses based on two ways. The first one simply searches for each API address, which, for example, corresponds to the right of '`call APIaddress`', on the memory areas whose NX bit flags are on. The other searches for each API address added by given values to counter stolen bytes techniques. Figure 1 shows an example of an anti-debugging technique called the stolen bytes technique. When we call ShellExecuteW API, we just directly jump to address '`0x73813C59`' like the left disassembly code. If one intends to thwart code analysis with a stolen bytes technique, it copies several instructions in an API to the main module and write a jump instruction that points to the address just below the last copied instruction like the right code. When a code is modified with the stolen bytes technique,

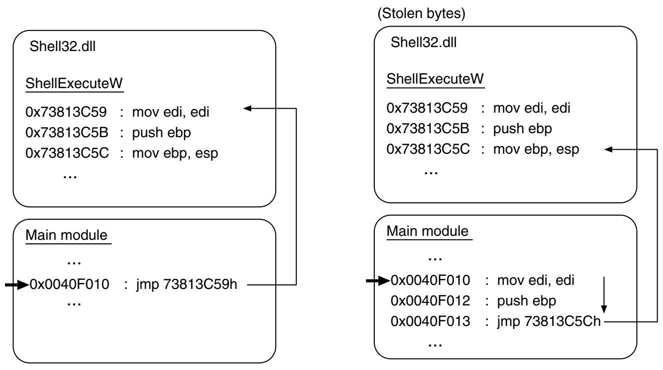
Shell32.dll
ShellExecuteW
0x73813C59 : mov edi, edi
0x73813C5B : push ebp
0x73813C5C : mov ebp, esp
...
Main module
...
0x0040F010 : jmp 73813C59h
...
(Stolen bytes)
Shell32.dll
ShellExecuteW
0x73813C59 : mov edi, edi
0x73813C5B : push ebp
0x73813C5C : mov ebp, esp
...
Main module
...
0x0040F010 : mov edi, edi
0x0040F012 : push ebp
0x0040F013 : jmp 73813C5Ch
...

Fig. 1. An example: the left is usual and the right is used with a stolen byte technique

API addresses of the main module do not appear. For example, '`jmp 73813C59`' in the left, which is ShellExecuteW address, is modified as '`jmp 73813C5C`' in the right. To fill a gap between an original address and a modified address for search, our method searches for from an API address added by 1 through the API address added by y for each API on the memory areas, where y is a given value and $y \in \{1, 2, \cdots\}$.

When a page fault occurs, if our API-call-instruction based approach can find more than z API addresses using both the first way and the second one, the approach infers that an OEP candidate is correct, where z is a given threshold value and $z \in \{1, 2, \cdots\}$.

4 Experiments

4.1 Setup

We use 20 packers described in Table 1. As sometimes a packer cannot pack an executable file correctly, we pack three executables, calc.exe, comp.exe, and vim.exe, in turn with each packer. The calc.exe and comp.exe are obtained from the system folder of Windows XP and vim.exe is a variant of an editor tool vi for Windows XP. We take the first packed file that can run for each packer. We unpack the 20 packed files described in Table 1 with our method.

The environment for unpacking is as follows. We used a PC whose CPU and host OS are Intel Xenon 3.10 GHz and Ubuntu 12, respectively. We installed KVM to the host OS, and installed Windows XP Professional SP3 to KVM as a guest OS. We implemented our method on the guest OS.

We set block size q and threshold value m of average entropy, which are described in Section 3.2, to 256 and 5.5, respectively. We picked these values from our experiments. We set y for API address search and threshold value z of the number of API addresses, which are described in Section 3.3, to 15 and 5. The reason why we set z to 5, which is small, is that there exist executables which call just a few APIs.

Table 1. The unpacking results

No.	Packer name	exe	(1)	(2)	No.	Packer name	exe	(1)	(2)
1	ACProtect 1.32	calc	√	√	11	PE-Pack 1.0	vim	√	√
2	ASPack 2.12	calc	√	√	12	Upack 0.39	calc	√	√
3	Exe32Pack 1.4.2	comp	√	√	13	UPX 3.08	calc	√	√
4	ExePack 1.4	vim	√	√	14	WWPack32 1.20	vim	√	√
5	eXPressor 1.5.0.1	calc	√	√	15	ASProtect 2.1	calc	√	×
6	FSG 2.0	calc	√	√	16	Mew11 1.2	calc	√	×
7	Molebox pro 2.6.4	calc	√	√	17	Armadillo 4.20	calc	×	×
8	Npack 1.1.300	calc	√	√	18	Obsidium 1.4.5	calc	×	×
9	Nspack 3.7	calc	√	√	19	Morphine 1.7	vim	×	×
10	PECompact 2.79	calc	√	√	20	Themida 1.8.5.5	calc	×	×

(1) : To obtain all the original code
(2) : To detect the OEP

4.2 Unpacking Results

Table 1 shows the unpacking results for our method. '(1)' in the table shows whether or not our method is able to obtain a memory dump that contains all the original code, where '√' and '×' denote success and failure, respectively. If a result do not satisfy '(1)', our method failed to obtain any of the original code. '(2)' shows whether or not our method is able to detect the OEP. We consider results that satisfy both '(1)' and '(2)' as unpacking success.

Our method is able to unpack the packed files of cases no. 1 – 14 successfully. In cases no. 15 (ASProtect 2.1) and 16 (Mew11 1.2), our method cannot detect the OEPs of the two packed files. To find the reasons, we manually monitored each instruction of the packed file of case no. 15. When our method decides an OEP candidate is correct, the EIP has not indicated the OEP. After we skip several page faults, the EIP indicates the OEP. Threshold value m of average entropy and z of the number of API addresses are not suitable for case no. 15. The reason for case no. 16 is the same as that of case no. 15. The results of cases no. 17 – 20 do not satisfy both '(1)' and '(2)'. The failure reason for case no. 19 (Morphine 1.7) is a consequence of the implementation of our method. The packed file of case no. 19 dynamically allocates memory for writing its original code. Our implementation does not turn on NX bit flags of such dynamically allocated memory areas. When our method tries to unpack the packed files of cases no. 17 (Armadillo 4.20), no. 18 (Obsidium 1.4.5), and no. 20 (Themida 1.8.5.5), no page faults occur. We tried to uncover the reason for that, but could not find an acceptable explanation. Armadillo, Obsidium, and Themida apply strong anti-debugging techniques. We guess that they detect KVM and stop their process in the experiment to thwart analysis.

We can tell from 14 success results that our method is efficient for generic unpacking. The results of cases no. 15 and 16 are failures, but our method should be able to unpack the two if we tuned the parameter values of our method. To unpack files packed with Morphine, we will implement our method such that it can turn on NX bit flags of dynamically allocated areas. For cases no. 17, 18, and 20, we will consider how to bypass anti-analysis techniques.

5 Conclusion

In this paper, we propose a generic unpacking method featuring two OEP detection approaches: one is an entropy-based approach and another focuses on the number of API-call instructions. Our key ideas are greatly different from those of the existing methods. For implementing a practical unpacking solution, we can independently use several methods to detect the OEP and we can decide the OEP from candidates as an election. Thus several types of OEP detection approaches are required. The experiment shows that our method can defeat 14 of 20 packers. In our future work, we plan to apply machine learning approaches instead of just using threshold values in order to overcome the issues we had with trials of cases no. 15 and 16 in the experimental phase.

References

1. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: 22nd Annual Computer Security Applications Conference, ACSAC 2006, pp. 289–300 (2006)
2. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode, WORM 2007, pp. 46–53. ACM, New York (2007)
3. Kim, H.C., Orii, T., Yoshioka, K., Inoue, D., Song, J., Eto, M., Shikata, J., Matsumoto, T., Nakao, K.: An empirical evaluation of an unpacking method implemented with dynamic binary instrumentation. IEICE Transactions 94-D(9), 1778–1791 (2011)
4. Kawakoya, Y., Iwamura, M., Itoh, M.: Memory behavior-based automatic malware unpacking in stealth debugging environment. In: 2010 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 39–46 (2010)
5. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 431–441 (2007)
6. Guo, F., Ferrie, P., Chiueh, T.-C.: A study of the packer problem and its solutions. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 98–115. Springer, Heidelberg (2008)
7. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44(2), 6:1–6:42 (2008)
8. Stewart, J.: Ollybone v0.1, break-on-execute for ollydbg, html document (2006), `http://www.joestewart.org/ollybone/tutorial.html`
9. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Security and Privacy 5(2), 40–45 (2007)