Applied Mechanics and Materials Vols 198-199 (2012) pp 343-350 *Online: 2012-09-26*

doi:10.4028/www.scientific.net/AMM.198-199.343

Unpacking Techniques and Tools in Malware Analysis

Peidai Xie[1, a], Meijian Li[1, b], Yongjun Wang[1, c], Jinshu Su[1], Xicheng Lu[1]

[1]School of Computer, National University of Defense Technology, Changsha Hunan, China

[a]peidaixie@gmail.com; [b]limeijian961@sohu.com; [c]wwyyjj1971@126.com

Keywords: unpacking; unpacker; runtime packer; spotting OEP; malware analysis; packed binaries.

Abstract. Nowadays most of malware samples are packed with runtime packers to complicate the task of reverse engineering and security analysis in order to evade detection of signature-based anti-virus engines. In the overall process of malware analysis, unpacking a packed malicious binary effectively is a necessary preliminary to extract the structure features from the binary for generation of its signature, and therefore several unpacking techniques have been proposed so far that attempt to deal with the packer problem. This brief survey article provides an overview of the currently published prevalent unpacking techniques and tools. It covers the operation process of packing and unpacking, packer detection methods, heuristic policies for spotting original entry point (OEP), environments for runtime unpacking, anti-unpacking techniques, and introduces several typical tools for unpacking.

Introduction

Malicious software is the main threat to internet today. According to Symantec's latest Internet Threat Report [1], more than 286 million unique by hash samples of malware are detected in 2010, which have caused huge economic damages estimated at more than ten billion dollars. In order to combat malware effectively, security researchers have made substantial effort[2] to study malicious code so that they can learn the logical structure and data content stored within a malware binary and extract signature used for malware detection.

Today malware authors heavily relay on code obfuscation techniques[3-6], such as runtime packing, encryption and polymorphism, to evade detection of signature-based anti-virus scanners and prevent their productions from being reverse-analyzed without striking a blowing by security researchers. The most prevalent of evasion detection techniques is runtime packing of compressed and encrypted code[7] According to[8, 9], over 80% of malware samples are packed using different packing techniques, and at the same time more than 50% new samples is simply re-packed version of existing one.

Runtime packing is a code transformation method[10]. A runtime packer is a program which uses packing techniques to transform an executable binary into another one with a different appearance so that the packed binary can evade detection of signature-based anti-virus engines. The most prevalent one used by malware authors is UPX[11], which is a free and open source, generic purpose runtime executable packer.

The prevalence of runtime packing techniques used by malware authors has driven the creation of various unpacking methods[10, 12-16]. The operation process of unpacking a packed binary is the reverse of packing it. Unpacking of a malicious packed binary correctly and automatically is the first step to dissect it to extract its structure features and uncover its behavior so as to improve detection accuracy of anti-virus engines. Now there are lots of efficient unpacking techniques are proposed in academic literature. This paper briefly summarizes unpacking techniques that are used in malware analysis and detection. It covers the operation process of runtime packing and unpacking, packer detection methods, heuristic policies for spotting original entry point (OEP), environments for unpacking, anti-unpacking techniques, and some excellent unpackers.

The main contribution of this paper is that it provides an overview of unpacking techniques for malware analysts to comprehend the problem of runtime packing and unpacking a binary in order to lift the burden of manual post processing.

The remainder of this paper is organized as follows. Section II gives an overview of operation process of packing and unpacking. Section III describes unpacking techniques. Several typical unpacking tools are introduced in section IV. We conclude this paper in section V.

Runtime Packing and Unpacking

The executable file in Microsoft Windows operating system is portable executable file format[4] which consists of PE header and Section Table. PE header contains information that is required by the loader in Windows, and Section Table includes sections, such as standard code section .text and standard data section .data, which constitute the main portion of an executable file.

Typically, a runtime packer takes as input an executable P, merges all sections of P into one, compresses and/or encrypts the section as data, and constructs a new executable P' together with a piece of code named stub attachment which is responsible for uncompressing and/or decrypting the data. Correspondingly, after the packed executable P' is loaded into memory, the stub code is running to uncompress/decrypt packed data into reserved memory space. When the original code in P is recovered, the program control is transferred to original entry point by a jump instruction and then the executable P' is running as its unpacked version P is running.

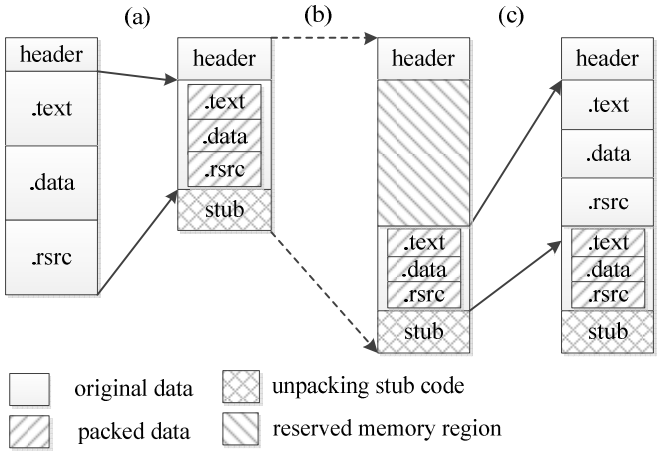

Figure 1. Typical operation process of packing and unpacking.

The operation process of runtime packing and unpacking[10] is shown in figure 1. Fig. 1(a) is a typical operation process of packing a packer performs on files; Fig. 1(b) is the process of a executable file being loaded into memory; Fig. 1(c) is a typical operation process of unpacking an unpacker performs in memory.

The packing process shown in Fig. 1(a) can be carried out on an executable binary iteratively and result in a multiple layer packed version. In [16] we can see that some malware samples have been packed hundreds of times.

It is must be noted that the process shown in fig.1 is the most typical and conceptual. Lots of packers use advanced runtime packing techniques[17, 18] which are sophisticated and heavily time consuming to perform unpacking correctly.

Unpacking Techniques

Unpacking techniques fall into two major categories: *static unpacking* and *dynamic unpacking*, according to the packed binary is executed or not, the same as categories of program analysis techniques[2]. A static unpacking method employs a specific unpacker which is developed by reversing the corresponding packing algorithm or stub code to unpack packed binaries directly, so it is

not necessary to run packed binaries. After unpacking, the unpacker is responsible for building the binary of original version. The main feature of static unpacking is fast, but a prior knowledge of the packing algorithm is required.

While a dynamic unpacking method runs a packed binary actually in a restricted analysis environment, and its execution is monitored to determine when unpacking routine is completed and dump a memory image of the monitored process to build the binary of original version with other information gathered during monitoring. The main feature of dynamic unpack is slow and detectable, but it is generic for packer types.

A static unpacking method is type-aware of a packer, in another word, given a packer, a corresponding unpacker should be developed, while this development process is typically complex and time consuming. Therefore, static unpacking methods are usually used for packer detection. On the other hand, a dynamic unpacking method is type-unaware of packers because the executable to be unpacked will run actually, so it is universal and can be used to develop a generic unpacker.

Packer Detection Methods. As universal unpacking is computationally expensive, it is necessary to differentiate a packed binary from unpacked one for a signature-based anti-virus scanner in order to improve efficiency. Packer detection methods[14, 19, 20] are devised to distinguish between packed and non-packed executables and find out the type of packers used in it. Some unpackers have ability of packer detection. It is demonstrated that the problem of distinguishing between packed and non-packed executables is undecidable[21], so some heuristic policies are proposed to do packer detection.

There are several packer detection methods, including *signature-based packer detection*[22], *entropy threshold* [8] and *pattern recognition* [20], all of which are static detection methods.

Signature-based Packer Detection. The characteristic of byte sequences (signature) of unpacking stub can be used to detect packers, the same as malware can be detected using a signature-based anti-virus engine. Before performing the potential packed executable detection, the signature of unpacking stub code must be extracted by reversing it manually to form a database. If a packed executable is detected, the corresponding unpacking routine is invoked to unpack it if the detector has implemented one. Signature-based packer detection method is simple and efficient, but it cannot detect a packer without its signature. PEiD is the most popular signature-based packer detector with plugin support.

Entropy Thresholds. Entropy, or information density, is a term from information theory describing the amount of information that each symbol within a series of symbols carries. A runtime packing method transforms one byte sequence into another, where the new byte sequence typically has higher entropy than the original one. This property can be leveraged to differentiate the packed executables from non-packed ones. In [8], the authors use an entropy-based metric to detect packed executables. This approach represents PE file as byte strings and use individual byte as a symbol. The entropy of a symbol X is defined as $\log(1/p_x)$ where p_x is the frequency of X. The higher the entropy of a PE file holds, the more possible it is a packed one. Some thresholds are inferred statically from a given database of packed binaries. If the entropy of a PE file under detection is found to be above the respective thresholds, it will be classified as packed. Entropy threshold method is used in most of unpackers commonly and it can be attacked by manipulating data within a PE file explicitly to decrease its entropy.

Pattern Recognition. This is a typical statistic and supervised learning method[19]. Firstly some structure features of the executable binary, such as the number of writable-executable sections, the code and data entropy etc., should be extracted using binary static analysis techniques and translated into a pattern vector. And then pattern recognition techniques are applied to distinguish between packed and non-packed executables. Pattern recognition method uses a classifier trained on a database of packed executables to distinguish between packed and non-packed executables, so it is prone to inaccurate.

Heuristic Policies for Spotting OEP. Spotting OEP[10, 12] in a static unpacking method is fairly straightforward[15], because a static unpacker can change value of OEP in PE header directly after unpacking as it has gained the value of OEP after reversing corresponding unpacking stub code or packers. While in a dynamic unpacking method there is no prior knowledge of detected executables that the whole process is to execute an executable in a restricted environment. So it is a crucial target to find the moment that a packed binary finishes running of its stub code and transfers control to OEP, because at that moment layout of memory is the same as that a non-packed version of the binary is just loaded, and a whole memory image of the binary can be dumped to construct an original one. Spotting OEP correctly is key point for an unpacker to unpack a packed binary successfully using dynamic unpacking methods[13].

While in a dynamic unpacking method, spotting OEP is not straightforward, as the OEP of a packed executable is usually obscured by malware authors to prevent from unpacking[23]. There are some heuristic policies implemented in unpackers, including *JMP-based spotting OEP*[11], *static code model*[21], *W⊕X* [2]and some *assistive methods*[7].

JMP-based Spotting OEP. After a packed binary is loaded into memory, it is common that unpacking stub code of the binary is in one section and the memory region for unpacked data is in others. An inter-section JMP instruction which is rarely used is responsible for transferring program control from stub code to original code, namely OEP. An executable is in all probability packed if an inter-section JMP instruction is detected as it is running.

Static Code Model. The instruction sequence of a packed executable being running is checked against its static code model before each instruction executes. The OEP is spotted if a divergence appears.

Write xor Execute (W⊕X). This method bases on page protection mechanism, the key idea of which is to monitor behaviors of write memory and execution of an instruction sequence. Some heuristic policies feasibly in a dynamic analysis environment in which address space of memory is monitored should be leveraged to spot OEP, including *dirty flag*[16], *page-fault handler debugging mechanism*[24] and *instruction statistic*[12].

- *Dirty Flag*. A shadow memory is used to mark a memory region as dirty whenever the region is written by a memory write instruction of the monitored program, which means it is newly generated. When the program control is transferred to one of these newly generated regions, it is determined that OEP is spotted. It is classified into dirty page execution and dirty byte execution by dirty granularity.
- *Page-fault handler Debugging Mechanism*. Using page property mechanism, memory pages of a binary under analysis is set to be read/write only (i.e., NX) except unpacking stub code resides, and page-fault handler is instrumented. When unpacking stub code is completed, it transfers control to the newly generated code. This will lead to a page fault due to the page protection settings. The instrumented page-fault handler is invoked to deal with this situation and OEP is found.
- *Quantification of Code Revelation*. A shadow state of memory is used to maintain 1) a written byte is executed [W->X] and 2) an executed byte is written [X->W] which respectively reflect the newly revealed code by unpacking and disappearing code by possible repacking. If the count of [W->X] is suddenly increasing, OEP is found.

Assistive Methods for Spotting OEP. A part of packers leverage sophistical code obfuscation techniques to hide OEP in order to make OEP spotting very difficult. According to some behaviors that unpacked binaries should perform after transferring to OEP, several assistive methods for spotting OEP are proposed, including:

- *The first API Invoked*. The first API exported from dynamic link library is invoked at the moment that stub code within a packed binary has finished its run.
- *Stack Pointer Check*. The current stack pointer at the first time of a control transfer to a dirty memory region is the same as that it is at the very start run.

- *Command-line Arguments Access*. The command-line arguments supplied with a packed binary's run will be moved from heap to stack at the first control transfer to the dirty memory region.

Most of unpackers perform unpacking directly and then to construct original binaries from memory image dumped when OEP is spotted by leveraging heuristic policies. If OEP is not spotted in a given time interval, it will report that unpacking fails and the binary under analysis is unpacked.

Environments for Unpacking. Some unpackers are implemented as a kernel driver for Microsoft Windows operating system as reason of lower overhead. But mostly the implementation of a dynamic unpacking method requires a restricted environment in which the execution of binaries under analysis can be monitored.

Dynamic Binary Instrumentation Framework. Dynamic binary instrumentation (DBI)[25, 26] is a technique for inserting extra code into address space of an application being running to observe its behavior. Pin[27] is a typical software system that performs runtime binary instrumentation. DBI is widely used in malware analysis[28] and implementations of unpacking techniques.

Sandbox. A sandbox is a software system that emulates CPU and memory of a physical machine[29]. If a potential malicious binary runs in a sandbox, there is no side effect to the host operation system at the same time behaviors of the binary are observed. So a dynamic unpacking method can be implemented in a sandbox.

Virtual Machine. According to [30], a virtual machine (VM) is "an efficient, isolated duplicate of the real machine" which is presented by a virtual machine monitor (VMM). A VMM can be used as an analysis environment to perform unpacking of packed binaries which run in a VM.

Full System Emulation. A full system emulator[31, 32] emulates all components of a real machine, including CPU, memory and devices such as network interface card. It is transparent to the programs running in its guest operating system so that it has the advantage of performing malware analysis. In such an emulator it is allowed to perform instruction-level analysis[33]. Some universal unpackers are implemented in such an emulator.

Anti-unpacking Techniques and Executable Protectors. Anti-unpacking techniques[7] are designed to make it difficult to comprehend an unpacker and dump address space of a running process, and fall into two major categories: passive and active. *Passive anti-unpacking techniques* focus on static unpacking methods with the purpose of making it difficult to identify and reverse the packing algorithm. *Active anti-unpacking techniques* are intended to protect the fully unpacked image of a running process being dumped, and classified into three subcategories: *anti-dumping* including virtualization obfuscators[17], *anti-debugging* and *anti-emulating* including detection of analysis environment in which an unpacker runs[34-36].

Some runtime packers can also perform protecting the unpacking stub code from reverse engineering attempts, or obfuscate parts of original code. Such packers are called executable protectors[10]. Some executable protectors design *virtualization* to transform from original code into byte code of a virtual machine, *stolen bytes* to hide OEP, *shifting decode frame* and *unpacking page-by-page* to prevent memory image of a running binary from being dumped successfully.

Typical Unpackers

PEiD. PEiD[22] is a most famous tool used for packer detection based on signatures of packed binaries. If a packer is identified, the corresponding unpacker can be invoked to unpack the binary if it is implemented as a plugin.

PolyUnpack. PolyUnpack[21] is a universal unpacker proposed for automatically extracting the hidden-code bodies of packed binaries. It makes use of the Microsoft Windows Debugging API for single-step executing and implemented as a plugin for OllyDBG debugger.

Renovo. Renovo[16] is a universal unpacker based on QEMU. It manages a shadow memory that keeps track of memory write accesses and uses W⊕X method to find OEP before dumping the memory. Renovo can repeat the unpacking process to deal with multi-layer packing.

OmniUnpack. OmniUnpack[24] is a universal unpacker implemented as a kernel driver of Windows XP. It monitors memory and detects executing attempts of written code. It is proposed to improve accuracy of signature-based anti-virus software.

Justin. Justin[7] is an automatic solution to reverse packed binaries for scanning by a signature-based scanner. It uses W⊕X to find OEP and can deal with multi-layer packing.

OllyBonE. OllyBonE[37] is a semi-automatic unpacker implemented as a plugin of OllyDBG which leverages page protection mechanism implemented to spot OEP.

Saffron. Saffron[38] is an automated unpacker that employs dynamic binary instrumentation based on Pin to monitor execution and memory reads and writes of a packed binary.

Bintropy. Bintropy[8] is a tool used for packed executables detection based on entropy thresholds method when performing binary static analysis.

Pandora's Bochs. Pandora's Bochs[39] is an generic unpacker based on Bochs[32] which leverages similar principles as Renovo.

Eureka. Eureka[40] is a static analysis framework which includes an automatic binary unpacking unit based on heuristic and statistical method with system call granularity.

Conclusion

Runtime packing of malware samples is prevalent in order to evade detection of signature-based anti-virus engines. This briefly survey paper provides an overview of unpacking techniques in malware analysis, covers operation process of packing and unpacking, unpacking techniques and typical unpackers, which is important for research of malware analysis. In the future, malware authors will exploit more complex and sophistical methods to consume time of reverse engineering performed by security researchers. So advanced unpacking techniques should be proposed to deal with corresponding packing methods.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61003303 and No. 60873215, the Program for Changjiang Scholars and Innovative Research Team in University (NO.IRT1012), Aid Program for Science and Technology Innovative Research Team in Higher Educational Institutions of Hunan Province "network technology", and Hunan Provincial Natural Science Foundation of China (11jj7003).

References

[1] Internet Security Threat Report, Vol. 16. Symantec Corporation, Jan. 2012. Available: http://www.symantec.com/business/threatreport/

[2] T. S. Manuel Egele, Engin Kirda, and Chrstopher Kruegel, "A Survey on Automated Dynamic Malware Analysis Techniques and Tools," ACM Computing Surveys, pp. 1-49, 2010.

[3] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Tech. Report, No.48, Department of Computer Science, the University of Auckland, New Zealand, July 1997.

[4] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware Normalization," Tech. Report, No.1539, University of Wisconsin, Madison, Wisconsin, USA, Nov. 2005.

[5] T. Brosch and M. Morgenstern, "Runtime Packers: The Hidden Problem," in Black Hat briefings USA, 2006, p. 3.

[6] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," in Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), 2008.

[7] F. Guo, P. Ferrie, and T. Chiueh, "A Study of the Packer Problem and Its Solutions," in In proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID' 08), 2008.

[8] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," in Proceedings of the IEEE Symposium on Security and Privacy (SSP'07), March 2007, pp. 40-45.

[9] A. Stepan. Improving Proactive Detection of Packed Malware. March 2006. Available: http://www.virusbtn.com/virusbulletin/archive/2006/ 03/vb200603-packed.dkb

[10] L. Bohne, "Pandora's Bochs: Automatic Unpacking of Malware," Diploma Thesis, 28th January 2008.

[11] UPX. Available: http://upx.sourceforge.net/

[12] H. C. Kim, D. Inoue, M. Eto, Y. Takagi, and K. Nakao, "Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation," in Joint Workshop on Information Security, August. 2009.

[13] K. Babar and F. Khalid, "Generic Unpacking Techniques," in IEEE Proceedings of the 2nd International Conference on Computer, Control and Communication (IC4'09), 2009, pp. 1-6.

[14] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, "Structural Feature based Anomaly Detection for Packed Executable Identication," in Proceedings of the 4th International Conference on Computational Intelligence in Security for Information Systems (CISIS'11), 2011, pp. 50-57.

[15] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic Static Unpacking of Malware Binaries," in Working Conference on Reverse Engineering, October 2009.

[16] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A Hidden Code Extractor for Packed Executables," in Proceedings of the ACM Workshop on Recurring Malcode, New York, NY, USA, 2007, pp. 46-53.

[17] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," in Proceedings of the IEEE Symposium of Security and Privacy (SSP'09), 2009.

[18] R. Rolles, "Unpacking Virtualization Obfuscators," in Proceedings of 3rd USENIX Workshop on OensiveTechnologies (WOOT'09), 2009.

[19] I. Santos, X. Ugarte-Pedrero, and B. Sanz, "Collective Classification for Packed Executable Identification," in Proceedings of the 8th Annual Collaboration, Electronic messaging, AntiAbuse and Spam Conference (CEAS'11), 2011, pp. 231-238.

[20] R. Perdisci, A. Lanzi, and W. Lee, "Classification of Packed Executables for Accurate Computer Virus Detection," in Pattern Recognition Letters, 2008, pp. 1941-1946.

[21] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06), Washington, DC, USA, 2006, pp. 289-300.

[22] PEiD. 2007. Available: http://www.peid.info/

[23] P. Bania. Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs. March 2009. Available: http://piotrbania.com/all/ articles/pbania-dbi-unpacking2009.pdf

[24] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07), Miami Beach FL, USA, 2007, pp. 1-4.

[25] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," in Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003.

[26] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'07), 2007.

[27] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05), 2005, pp. 190-200.

[28] M. Li, Y. Wang, P. Xie, Z. Huang, S. Jin, and S. Liu, "Reverse Engineering of Security Protocol Format Based on Dynamic Binary Analysis," in International Conference on Computer Convergence Technology (ICCCT'11), October 2011.

[29] T. H. Carsten Willems, and Felix Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," in Proceedings of the IEEE Symposium on Security and Privacy (SSP'07), 2007.

[30] R. P. Goldberg, "Survey of Virtual Machine Research," vol. IEEE Computer Magazine, pp. 34-45, June 1974.

[31] F. Bellard, "Qemu: A Fast and Portable Dynamic Translator," presented at the Usenix Annual Technical Conference, 2005.

[32] Bochs. Bochs: The open source IA-32 emulation project.

[33] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in Proceedings of the 4th International Conference on Information Systems Security (ICISS'08, keynote invited paper), Hyderabad, India, December 2008.

[34] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to Automatically Generate Procedures to Detect CPU Emulators," in Proceedings of the USENIX Workshop on Offensive Technologies (WOOT'09), 2009.

[35] P. Ferrie, "Attacks on Virtual Machine Emulators," ed: Symantec Advanced Threat Research, 2006.

[36] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10), San Diego, CA, USA, 2010.

[37] J. Stewart, "OllyBonE: Semi-Automatic Unpacking on IA-32," in Defcon 14, Las Vegas, NV, 2006.

[38] D. Quist and Valsmith, "Covert Debugging: Circumventing Software Armoring Techniques," in Black Hat Briengs, USA, August 2007.

[39] L. Boehne, " Pandora's Bochs: Automated Unpacking of Malware," Diploma thesis, January, 2008.

[40] V. Yegneswaran, H. Saidi, P. Porras, and M. Sharif, "Eureka: A Framework for Enabling Static Analysis on Malware," Tech. Report, No.SRI-CSL-08-01, SRI Project 17382, Computer Science Laboratory SRI International and the College of Computing Georgia Instititue of Technology, 12 April 2008.

Applied Mechanics, Mechatronics Automation & System Simulation
10.4028/www.scientific.net/AMM.198-199

Unpacking Techniques and Tools in Malware Analysis
10.4028/www.scientific.net/AMM.198-199.343