

ON THE FEASIBILITY OF MALWARE UNPACKING
WITH HARDWARE PERFORMANCE COUNTERS

by

JAY MAYANK PATEL

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2019

Copyright © by Jay Mayank Patel 2019

All Rights Reserved



ACKNOWLEDGMENT

I would like to thank my supervisor, Dr. Jiang Ming, for his kind co-operation, inspiring discussion and realistic suggestion at all stages of my work and giving me such opportunities to work on this topic. Without his guidance and efforts, I could not be able to finish my work smoothly.

I would also like to thank Dr. Jeff Lei and Dr. Jia Rao for taking time out from their schedule and attending my dissertation. Thank you, Erika and Haotian for helping me with my research directly or indirectly.

Last but not the least, I would like to thank my wife, parents, my family, and friends for encouraging me and supporting me throughout my research.

April 18, 2019.

ABSTRACT

ON THE FEASIBILITY OF MALWARE UNPACKING WITH HARDWARE PERFORMANCE COUNTERS

Jay Mayank Patel, MS

The University of Texas at Arlington, 2019

Supervising Professor: Jiang Ming

Most of the malware authors use Packers, to compress an executable file and attach a stub, to the file containing the code, to decompress it at runtime, which will turn a known piece of malware into something new, that known-malware scanners can't detect. The researchers are finding ways to unpack and find the original program from such packed binaries. However, the previous study of detection for unpacking in the packed malware using different approach won't provide many promising results.

This research explores a novel approach for the detection of the unpacking process using hardware performance counters. In this approach, the unpacking process is closely monitored with Hardware Performance Counters. The HPCs shows hot spot during the unpacking process. By performing the per-process filtration, HPCs show a close relation with the decompression algorithm. For this research, the analysis is performed on a bare-

metal machine. The packed executable is profiled for hardware calls using Intel® VTune™ Amplifier.

TABLE OF CONTENTS

Acknowledgment.....	iii
Abstract.....	iv
List of Figures.....	viii
List of Tables.....	ix
Chapter 1 Introduction.....	1
About Malware Analysis	2
Art of Packing	3
Packing with UPX.....	4
Hardware Performance Counters	4
Chapter 2 Background and Related Work.....	6
Unpacking.....	6
Unpacking with UPX	6
Manual unpacking with OllyDbg	7
Issues in previous work	10
Chapter 3 Problem Identification and Proposed Solution.....	12
Defining Problem	12
Potential Solution.....	14
Chapter 4 Evaluation.....	15

Experimental Setup	15
HPC Collection with Intel® VTune™ Amplifier	15
Result Summary and Performance Measurement.....	15
Data Modeling with Eureqa.....	18
Chapter 5 Conclusion and Future Work	21
Appendix A Configure and Build UPX in Linux (Ubuntu 18.04 LTS)	22
References.....	24
Biographical Information.....	26

LIST OF FIGURES

Figure 1-1 File System View for Packing of Windows PE	3
Figure 1-2 Screenshot of Packing a File with UPX.....	4
Figure 2-1 Memory View during Execution of Packed File	6
Figure 2-2 Representation of pf.exe in OllyDbg	7
Figure 2-3 Locating PUSHAD Instruction in the Packed Binary	8
Figure 2-4 Breakpoint Setup at POPAD Instruction	8
Figure 2-5 Location of OEP at the End of the Unpacking.....	9
Figure 2-6 Use of OllyDump Plugin to extract the Original Program	10
Figure 3-1 Hotspot View Summary for Manual Unpacking of pf.exe with UPX	12
Figure 3-2 Bottom-Up View of Manual Unpacking of pf.exe with UPX	13
Figure 4-1 File pf_lzma_upx_int.exe with Interrupt Before OEP.....	17
Figure 4-2 File pf_ucl_upx_int.exe with Interrupt Before OEP.....	17
Figure 4-3 Data Model by Eureka for pf_ucl_upx_int.exe	19
Figure 4-4 Data Model by Eureka for pf_ucl.exe	20

LIST OF TABLES

Table 1-1 Types of Malware [2].....	1
Table 4-1 File Name Convention with Packer and Algorithm Used.....	18

CHAPTER 1
INTRODUCTION

Malware is known as malicious software or malicious code. Malware can be defined as a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim [1]. Table 1-1 represents different types of malware.

Table 1-1 Types of Malware [2]

Malware	Description
Virus	Mostly found in programs or executables. The processor executes such malware code with the program.
Worm	Like a virus in functional behavior except that they are stand-alone software which does not require any assistance from a host program or human aid for broadcasting.
Polymorphic Virus	A virus that can alter its payload to evade detection, while maintaining its functionality.
Metamorphic Virus	A virus that alters both the payload and functionality
Trojan	Malware that appears legitimate but acts maliciously once activated.
AdWare	Malware that floods a web-page with unwanted advertisements.
SpyWare	Malware that secretly gathers reports the user's personal information and grants access to such information to another entity without the user's consent.
Ransomware	Malware that blocks access to user data and threatens to delete or publish it unless the user makes a predetermined payment.
Botnet	Malware that employs an infected system as a node in a network controlled by a central malicious unit called the bot herder.
Rootkit	Malware that provides privileged access to a system while hiding its or any other malicious software's presence.

The malware is categorized based on their purpose of use like mass or targeted. Mass malware is designed to affect many machines at once, where targeted malware is created to infect specific organization and are a bigger threat to networks than mass malware [3].

About Malware Analysis

Malware analysis is the art of dismembering malware to identify, know about its working, and find a way to overcome or remove it. Malware can be identified by its host and network-based signatures. There are two main approaches to malware analysis: Static and Dynamic. Examine the malware without running it is Static approach, while Dynamic approach involves running the malware. The detailed categorization of these technique involves basic and advanced methods for both.

Malware authors use several tricks to avoid detection and analysis. The most popular way is use technique called packing. Software packing is a method of compressing or encrypting an executable or modifying a file's format. The payload, which is the actual malware is protected against reverse engineering and security software detection. This is done by adding code that is not strictly malicious but only intended to hide the malicious code. The goal is to hide the payload from the victim and from researchers that get their hands on the file. Packing an executable changes file signature to avoid signature-based detection. Most decompression techniques decompress the executable code in memory. Utilities used to perform software packing are called packers. Though there are many kinds of sophisticated packers available, malware writers also create custom packers to make it harder for an analyst to detect.

Art of Packing

Portable Executable (PE) file format is developed by Microsoft as a common file format to support all windows versions and on all supported systems. PE consists of PE header, section table, and other sections. The PE Header includes information about the machine, number of sections, time date stamp, pointer to the symbol table, number of symbols, size of optional header and characteristics. Section table provides the reference to various sections in the PE file and maintains section permissions for the file. Each of the sections is maintained in the section table contains information related to how the program runs. Section includes .text, .data, .rsrc and .reloc. Original file from Figure 1-1 describes PE file structure.

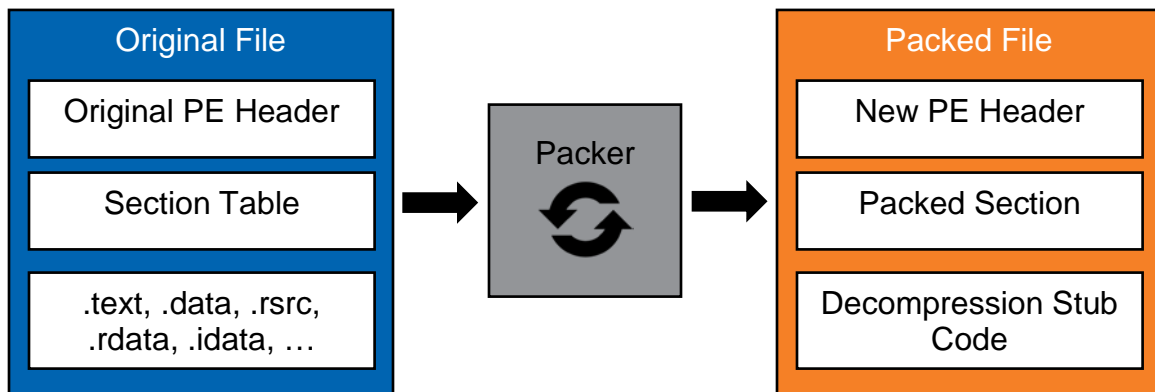
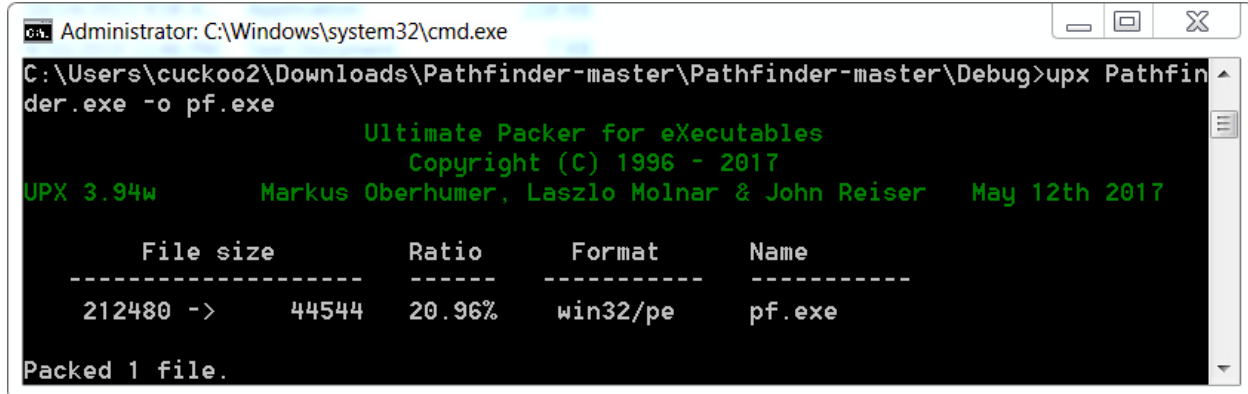


Figure 1-1 File System View for Packing of Windows PE

The binary packing is a benign process where the original file is being converted to a packed file using compression and/or encryption algorithm by the packer. The input to packer can be any executable file such as the Windows Portable Executable (PE) or the Linux Executable and Linkable Format (ELF) and the output will be packed/compressed executable as illustrated in Figure 1-1. In the packed file, the packed section contains the original file and the decompression stub code is used to automatically decompresses and runs the original file.

Packing with UPX



```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\cuckoo2\Downloads\Pathfinder-master\Pathfinder-master\Debug>upx Pathfinder.exe -o pf.exe

      Ultimate Packer for eXecutables
      Copyright (C) 1996 - 2017
UPX 3.94w      Markus Oberhumer, Laszlo Molnar & John Reiser      May 12th 2017

      File size      Ratio      Format      Name
-----
      212480 ->      44544      20.96%      win32/pe      pf.exe

Packed 1 file.
```

Figure 1-2 Screenshot of Packing a File with UPX

As shown in Figure 1-2, the file can be packed using UPX packer. When one pack any Executable with UPX, all existing sections (.text, .data, .rsrc, etc.) are compressed. Each of these sections is named as UPX0, UPX1, etc. Then it adds a new code section at the end of the file which will decompress all the packed sections at execution time. As shown, a file named as Pathfinder.exe (which is 32-bit Portable Executable) is packed into pf.exe. During this process, the original size is compressed 20.96% as from 208 KB to 44 KB.

Hardware Performance Counters

Hardware Performance Counters are sets of special-purpose registers built into modern microprocessors to record architectural and microarchitectural events precisely and accurately as they occur. These counters are part of a special, dedicated unit in the central processing unit (CPU) called the Performance Monitoring Unit (PMU). They have the ability to access detailed information regarding the processor's functional units and caches, as well as the memory, etc. The availability of HPCs depends on the CPU architecture and vendor. The HPCs are highly hardware dependent and that's why, even across the same vendor, each CPU generation has its own implementation. There are no additional overheads of using HPCs because they are built-in CPU. Each time the

programmed event occurs, the count register is incremented, that's why they provide accurate results [4].

The HPCs are used to conduct low-level performance analysis or tuning [5]. From performance analysis tools their usage has extended to detect firmware medication in embedded systems [6], estimating system power utilization [7], and even detection of malware [8]. Essentially, software engineers use HPCs for measuring the performance of their code and thus optimizing it.

Some of the commonly used software interfaces include PAPI [9] which provides standard APIs for accessing the HPCs. For Linux, based on perf event, perf [10] is a popular tool provides support for HPCs and for Windows operating systems, Intel® VTune™ Amplifier [11] for the Intel® processors and AMD's CodeAnalyst [12] for the AMD processors is used. In this project, HPCs are used to construe a time-series trace of N microarchitectural events by profiling packed benign application. Each packed binary executed on CPU may or may not generate a different performance counter trace.

CHAPTER 2

BACKGROUND AND RELATED WORK

Unpacking

During the execution of the packed file, decompression stub, stored in the packed file will decompress the packed section. The original file is then loaded into memory as per described in Figure 2-1.

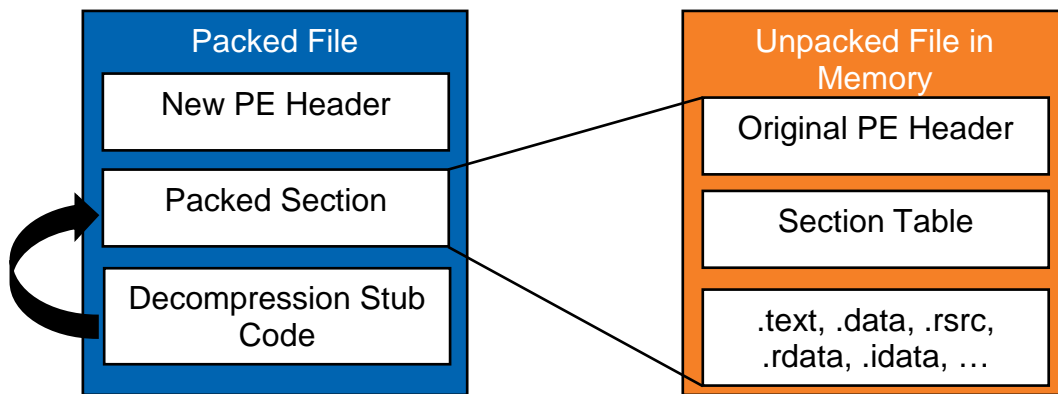


Figure 2-1 Memory View during Execution of Packed File

During this process, the original entry point(OEP), the memory address where the program starts, is relocated in the unpacked section.

Unpacking with UPX

To illustrate unpacking, consider the packed file pf.exe from Chapter-1. The following process will occur on the execution of a UPX packed pf.exe file.

- First, it saves the current Register Status using PUSHAD instruction.
- All the Packed Sections are Unpacked in main memory.
- Resolve the import table of the original executable file.
- Restore the original Register Status using POPAD instruction.
- Finally, Jumps to Original Entry Point(OEP) to begin the actual execution.

- Execution starts from new OEP (from the newly added code section at the end of file).

Manual unpacking with OllyDbg

To perform this process manually, we will debug pf.exe with OllyDbg [13]. OllyDbg is a 32-bit assembler level analyzing debugger for Microsoft® Windows®. Emphasis on binary code analysis makes it particularly useful in cases where the source is unavailable.

As the pf.exe is loaded in OllyDbg for analysis, it will be represented as shown in Figure 2-2 below.

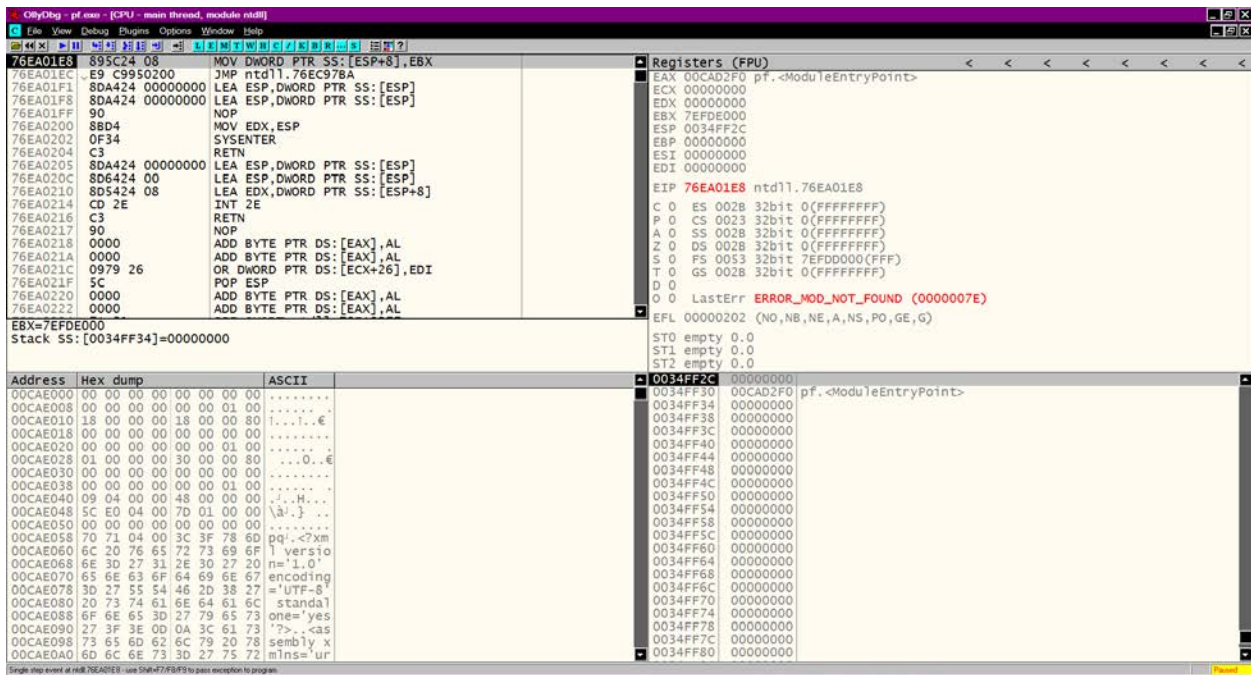


Figure 2-2 Representation of pf.exe in OllyDbg

Start tracing the EXE, until one encounter a PUSHAD instruction shown as Figure 2-3.

Usually, this is the first instruction, or it will be present after the first few instructions based on the UPX version.

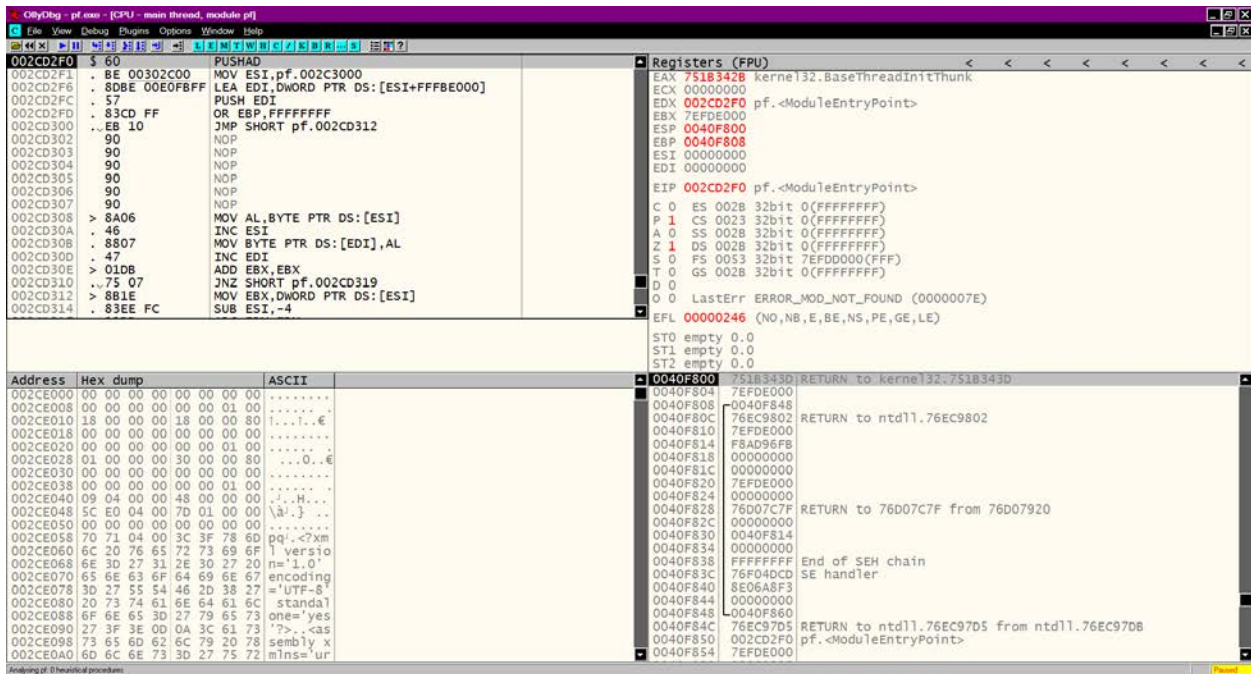


Figure 2-3 Locating PUSHAD Instruction in the Packed Binary

When one reaches PUSHAD instruction, put the Hardware Breakpoint to stop at POPAD instruction as described in Figure 2-4. Another way is to manually search for POPAD instruction and then set Breakpoint on it.

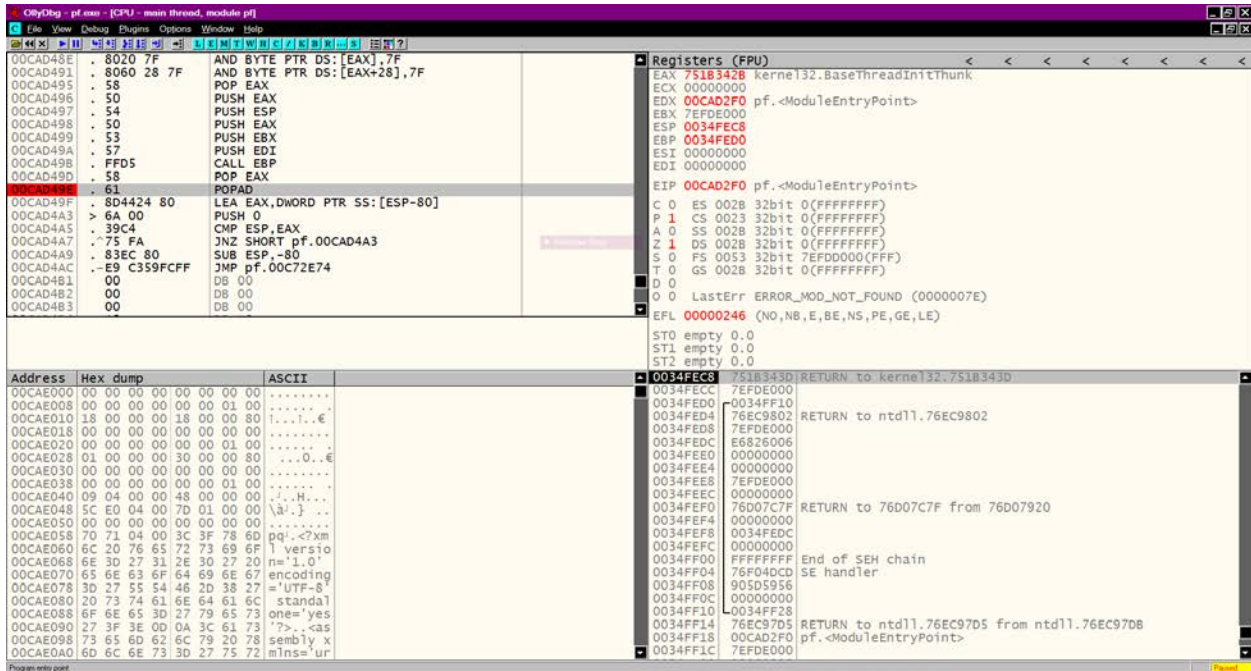


Figure 2-4 Breakpoint Setup at POPAD Instruction

Once set up the breakpoint, continue the execution. Shortly, it will break on the instruction which is immediately after POPAD or on POPAD instruction. Now start step by step tracing and soon one will encounter a JMP instruction which will take to actual OEP in the original program as shown in Figure 2-5.

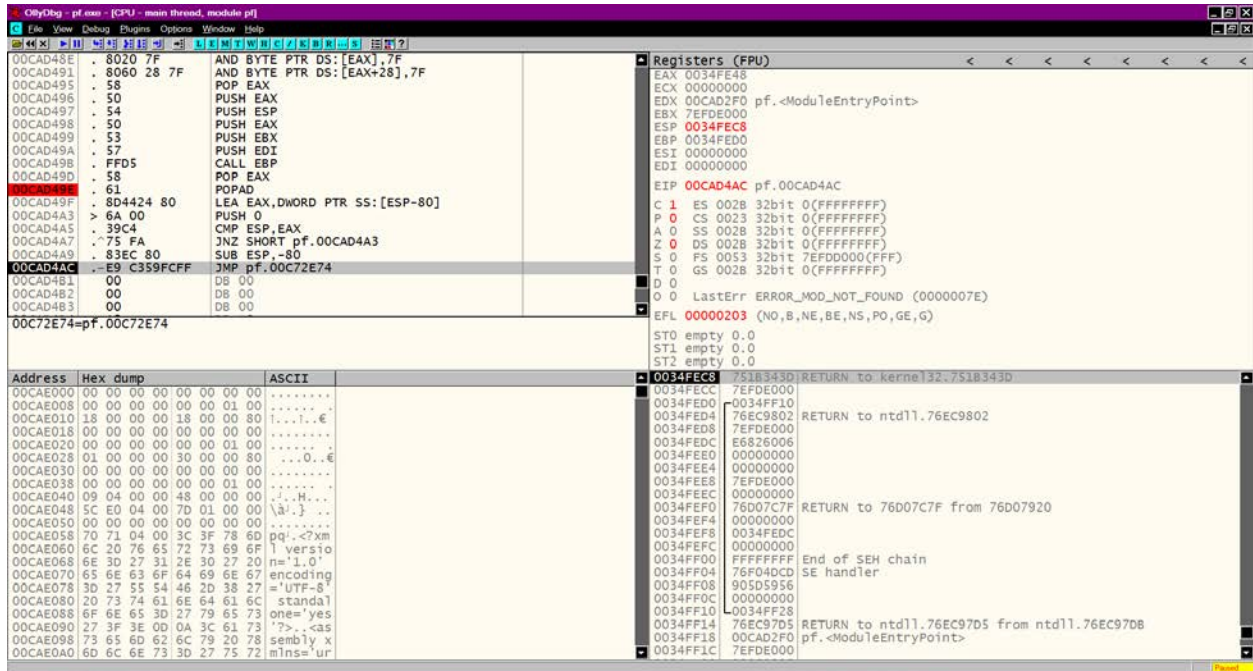


Figure 2-5 Location of OEP at the End of the Unpacking

When you reach OEP, dump the whole program using OllyDump plugin (use default settings) as mentioned in Figure 2-6. It will automatically fix all the Import table as well.

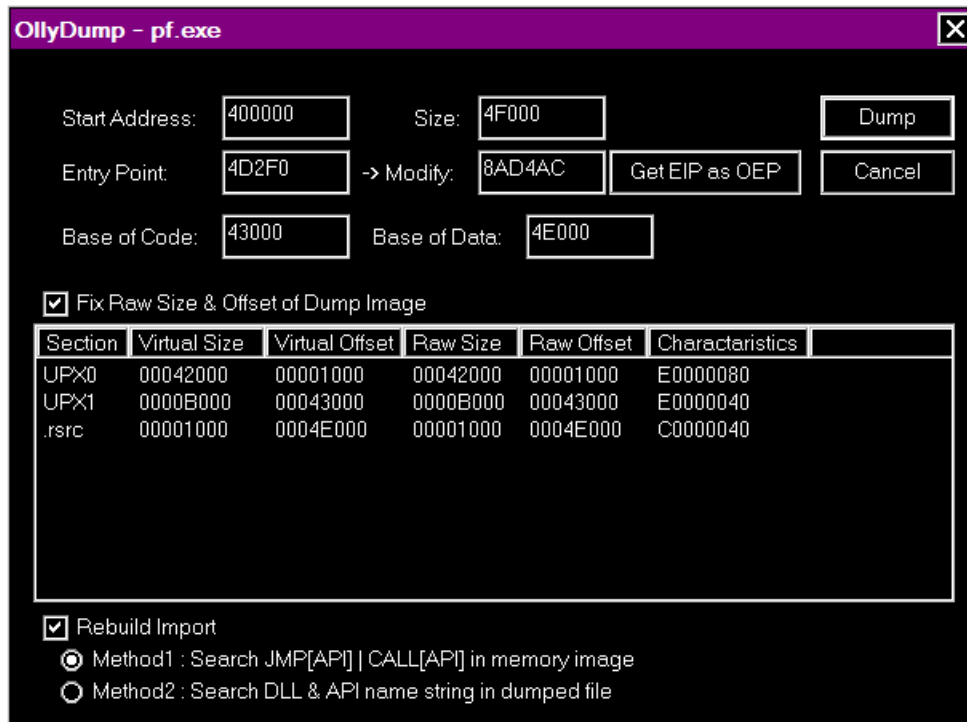


Figure 2-6 Use of OllyDump Plugin to extract the Original Program

However for most of the packers, one can use an advanced tool called ImpREC (Import Reconstructor) [14]. ImpREC is a highly advanced tool used for fixing the import table. It provides multiple methods to trace the API functions as well as allow writing custom plugins.

Issues in previous work

In [15], Das et al. have proposed some challenges, pitfalls, and risks of using HPCs for security. The authors have provided reasons lead to inaccurate measurements as the effect of external sources on the runtime environment, Non-Determinism, Overcounting, Variations in tool implementations. The authors have suggested proper instantiation and usage of various fundamentals like,

Context Switch Monitoring – In order to profile the runtime behavior of a process, performance counter values must be saved during context switches to avoid any contamination due to events from other processes.

Interrupt Handling – The performance counters are typically used in conjunction with performance monitoring interrupts (PMI). This feature is not essential when reading events in sampling mode; it can also profile events at a finer granularity.

Process Filtering Upon Process Monitoring Interrupt (PMI) – It is necessary to implement a technique for filtering performance counter data relevant solely to the process of interest. Otherwise, counter data will be contaminated by the events of other processes.

Minimizing the impact of non-deterministic events – It is important to consider only deterministic events. A deterministic event is defined as an event whose value does not vary between identical runs and matches the expected values that would be obtained through alternative means.

They have assessed 56 papers using HPCs in various field and pointed common mistakes. To overcome issues,

No per-process filtering – Any implementation that does not apply per-process filtering will capture events from other processes.

PMI-Based Filtering Only – Many papers did not save and restore the counter data during context switches. This made the data have contamination of counter data from other processes. To overcome this, obtain performance counter data by applying process filtering only at PMI.

Lack of compensation for non-determinism and over overcounting issues – The non-determinism and overcounting issues are a significant oversight.

CHAPTER 3

PROBLEM IDENTIFICATION AND PROPOSED SOLUTION

As described in Chapter 2, the process of unpacking is performed in a controlled environment, but in actual, this packed code will be unpacked and loaded into main memory and begun its execution, that's why it is not possible to know exactly when the original program has started. For the malware analyst, the most difficult task is to differentiate between benign unpacking process and malicious program execution.

Defining Problem

As the packers use decompression and decryption functions, they mostly utilize CPU and memory. To record this using HPCs, I have conducted profiling of such unpacking process with Intel® VTune™ Amplifier. As shown in Figure 3-1, it is obvious that the hotspot shows most of the HPCs related activities during unpacking.

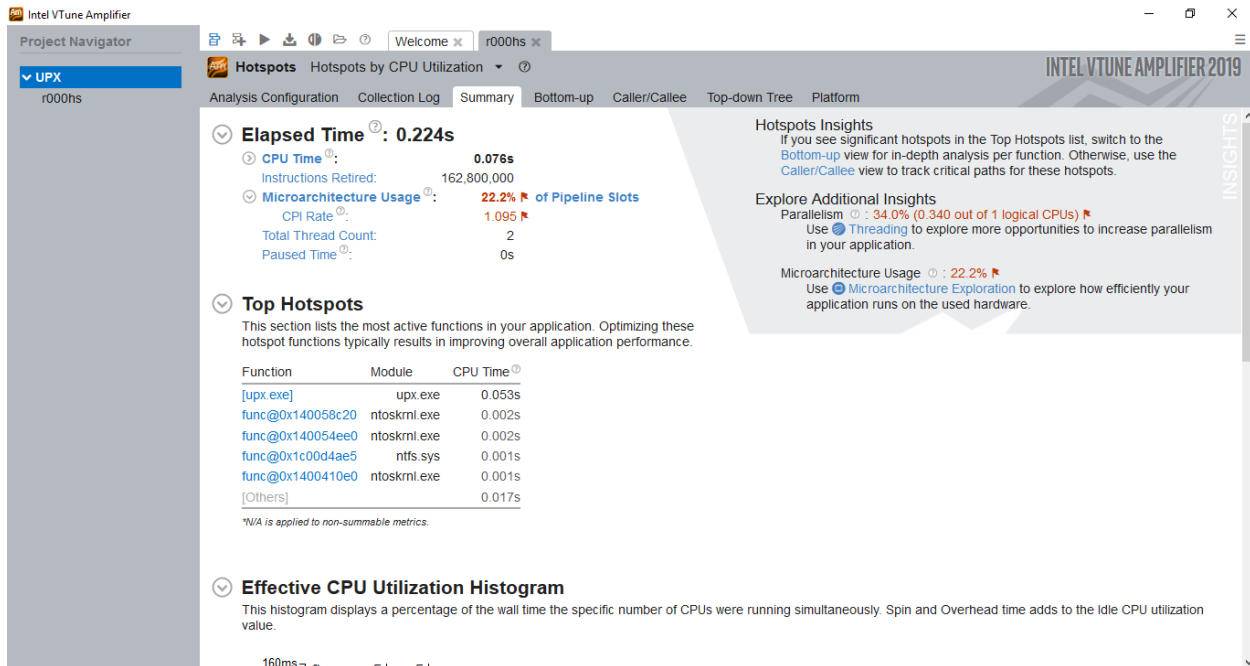


Figure 3-1 Hotspot View Summary for Manual Unpacking of pf.exe with UPX

From Figure 3-2, the microarchitecture usage for the unpacking is high, which is 26.8%. This gives the motivation to use HPC to explore the unpacking process.

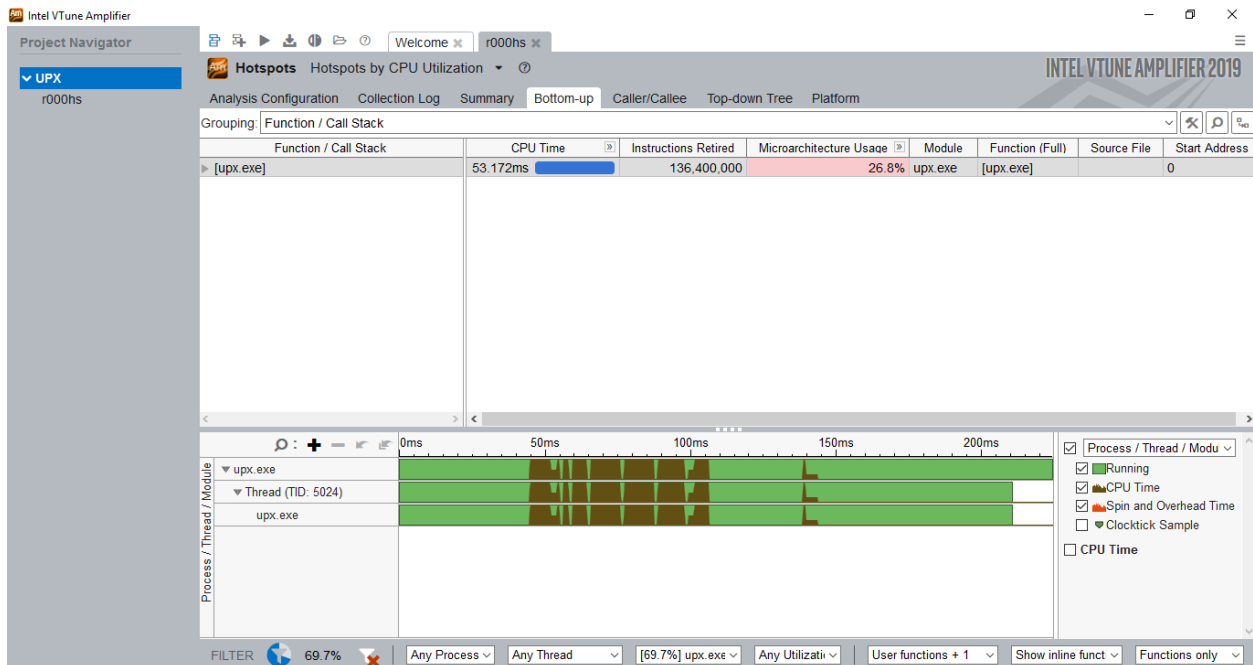


Figure 3-2 Bottom-Up View of Manual Unpacking of pf.exe with UPX

To analyze more about unpacking process in UPX, download the source from [16] and compile and build in Linux (Refer Appendix A). I have noticed that for the unpacking process in UPX uses inbuilt source file, written in assembly code. This file takes the original program as input and check for its packing method and based on that it will automatically select the unpacking method and unpacks the packed code to main memory. For 32-bit windows executable file, one has to look for i386-win32.pe.S file and for 64-bit windows executable file, amd64-win64.pep.S.

Once one look into the assembly code, one can find different unpacking algorithms as LZMA and different version of NRV. The NRV algorithms are part of UCL data compression library, which has 3 variants of as NRV2B, NRV2D, and NRV2E. These NRV algorithms are block compression algorithm, which takes 32-bit of data blocks to

perform packing or unpacking. To learn more about HPC activity during unpacking, I have checked the result of decompression with the standalone application of LZMA and UCL algorithms. By comparing them with the unpacking of UPX with their respective version, I find out that the unpacking is closely related to the algorithm used for packing rather than a different packer. The algorithms used to perform packing are less, different packers use these algorithms with other techniques to make packers more complex.

Potential Solution

To investigate in this direction, I have conducted different experiments with Pathfinder.exe as input to Lzma#.exe and uclpack.exe. I have generated packed file Pathfinder.exe for both the algorithms and recorded HPCs with Intel® VTune™ Amplifier and compared the HPCs with manual unpacking and interrupted unpacking just before OEP. I have collected the HPCs recorded, which are around 575 [17].

CHAPTER 4

EVALUATION

Experimental Setup

I have used the bare-metal computer to perform experiments. This environment can be helpful to work with the malicious packed file. As HPCs are CPU dependent, if one creates a virtual environment to conduct such experiments, one cannot record the correct values of HPCs. The HPCs can also capture the overhead of virtual machines, so one should avoid using virtual environment of any HPCs related experiments.

Again from the suggestions of Das et al. [15], I have conducted all the experiments using a single core of the processor. As the HPCs are related to CPU and using more core of processor shows different values of HPCs, it is highly recommended to use a single core. For this, I have to make a change in BIOS to use the processor as a single core.

HPC Collection with Intel® VTune™ Amplifier

During the collection of different HPCs, I have set CPU sampling interval as 1 ms and enabled per-process filtration from advance settings. The sampling starts as soon as the program starts its execution. During the sampling, HPCs generated by other processes running are also included with the result. So, it is necessary to perform per-process filtration of the result to computer correct HPCs for the unpacking process.

Result Summary and Performance Measurement

In this research, we have used a modified UPX source code to insert interrupt to stop the execution of packed executable just before OEP. The source code of UPX is compiled and built in Linux. To modify UPX source code and insert the interrupt refer to Appendix

A. The Pathfinder.exe should be packed in Linux then shifted to Windows for further analysis. During primary analysis, we found hotspot using Intel® VTune™ Amplifier in the unpacking process. By further investigating, I came to know that the algorithm behind the unpacking is the only reason for such HPC deviations.

By default, UPX uses UCL compression library to pack any file. So, to pack or unpack any file with LZMA in UPX is not straight forward, one must look into the source code of UPX. After looking into the conf.h and main.cpp file its command line argument --lzma for LZMA algorithm and --nrv2b, --nrv2d and --nrv2e for respective UCL algorithms NRV2B, NRV2D, and NRV2E. In this experiment, I have used the default UCL algorithm, which is NRV2B.

During the packing with UPX with LZMA and UCL algorithms, the interrupt was inserted to the end of unpacking or just before OEP. To verify this, refer the Figure 4-1, which shows the file pf_lzma_upx_int.exe in OllyDbg, where int 15 is the interrupt, inserted before the jump to OEP.

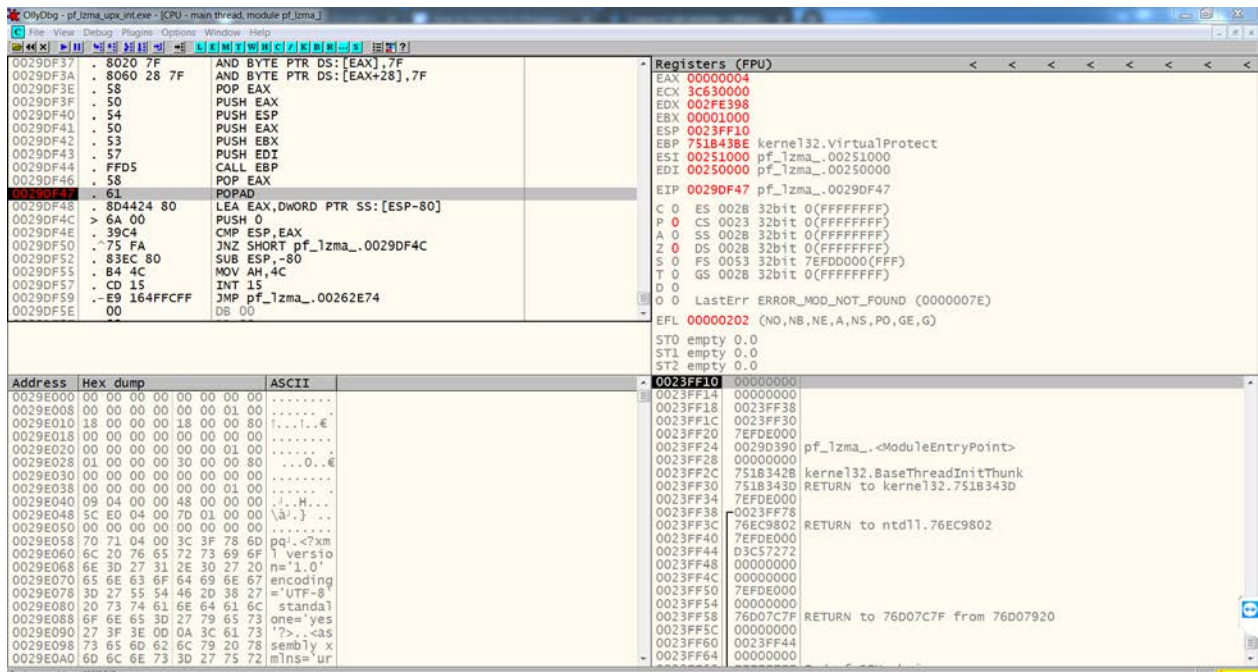


Figure 4-1 File pf_lzma_upx_int.exe with Interrupt Before OEP

In Figure 4-2, the interrupt, int 15, inserted before at the end of unpacking in file pf_ucl_upx_int.exe.

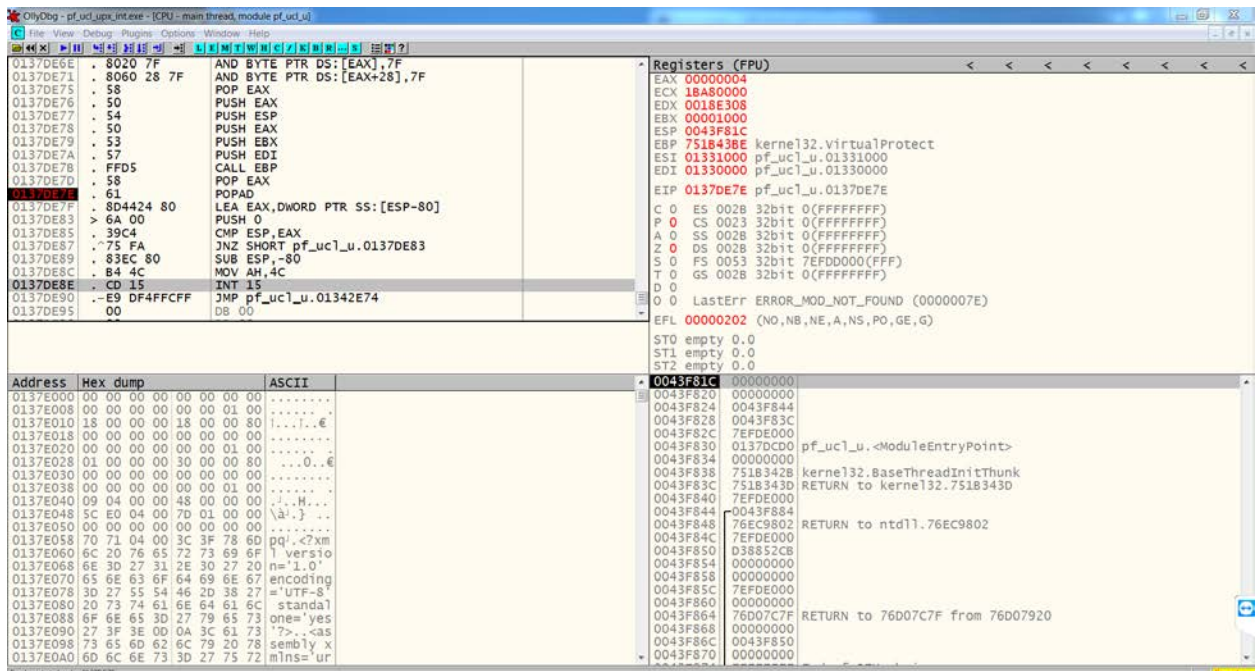


Figure 4-2 File pf_ucl_upx_int.exe with Interrupt Before OEP

These packed binaries while executed, they will stop automatically when the interrupt occurred. So, during their execution, they will just perform unpacking and decompress original code of the program into main memory and about to start its execution. The HPCs will be recorded for these binaries using Intel® VTune™ Amplifier.

Consider the Table 4-1 for the filename convention as the files are being used throughout the research experiments. The file used to pack, or compress is Pathfinder.exe.

Table 4-1 File Name Convention with Packer and Algorithm Used

No	Packer Program	Algorithm	Interrupt	File Name
1	lzma#.exe	LZMA	No	pf_lzma.exe
2	upx.out	LZMA	Yes	pf_lzma_upx_int.exe
3	uclpack.exe	NRV2B	No	pf_ucl.exe
4	upx.out	NRV2B	Yes	pf_ucl_upx_int.exe

To compare the recorded HPCs of UPX packed file, I have recorded the HPCs with the standalone application of compression and decompression LZMA and UCL algorithms.

Data Modeling with Eureqa

For simplicity, we have decided to use a software tool for detecting mathematical relationships in data. This tool is called Eureqa [18] and by providing the tool with HPCs values we found significant, talked about in earlier sections, we were able to create a simple linear relationship. The various selected HPC events during unpacking were inputted into the software especially focusing on the maximum values and minimum values. From there, the Eureqa provided a linear relationship algorithm that is based on a certain threshold for the minimum and maximum values.



Figure 4-3 Data Model by Eureqa for pf_ucl_upx_int.exe

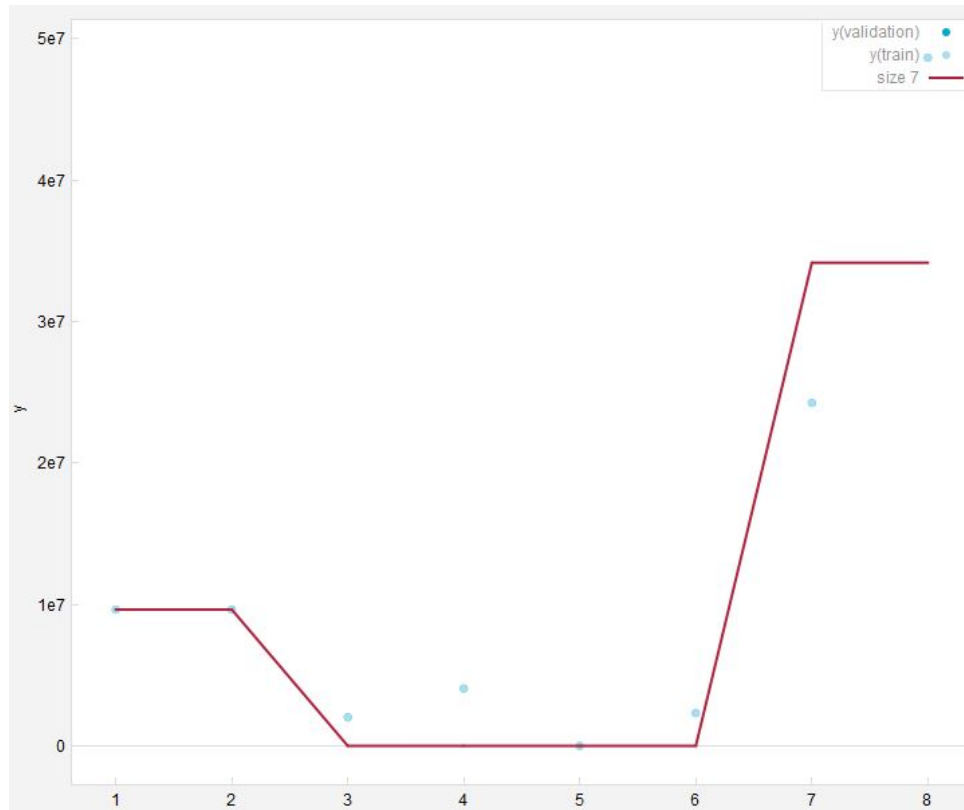


Figure 4-4 Data Model by Eureqa for pf_ucl.exe

Originally, we provided Eureqa with 10 HPCs that were found significant to unpacking between pf_ucl.exe and pf_ucl_upx_int.exe. However, Eureqa had 7 HPCs out of the 10 HPCs not used when creating the relationship. After taking these 7 HPCs and only running the Eureqa tool on the 3 remaining HPCs, it then created a relationship on 2 HPCs. We can see though, that when using these 2 HPCs, the model built using Eureqa look very similar when comparing Figure 4-3 to Figure 4-4.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Though this is a good place to start, it is to be said that if only using 2 HPCs out of the 10 HPCs we found significant, it might be easier for malware writers to evade presented model. In theory, what our plan is moving forward, is to be able to input an entire program's HPCs into this model and if we see a similar generated model out of the entire model for the program, we can infer that unpacking is happening there.

APPENDIX A

CONFIGURE AND BUILD UPX IN LINUX (UBUNTU 18.04 LTS)

To install dependencies, Open terminal and perform following commands for building UPX.

```
sudo apt-get install gcc
sudo apt-get install make
sudo apt-get install zlib1g
sudo apt-get install zlib1g-dev
sudo apt-get install zlib1g:i386
sudo apt-get install python
```

To provide link between libmpfr.so.4 and libmpfr.so.6 (As in new version libmpfr.so.6 is available)

```
sudo ln -s /usr/lib/x86_64-linux-gnu/libmpfr.so.6
/usr/lib/x86_64-linux-gnu/libmpfr.so.4
```

Download UPX [16] (For more information about how to build and configure refer README.SRC). Extract it in the home directory.

```
cd ~
```

To download UCL data compression library [19] and configure. Create folder in home directory (`$(HOME)/local/src/`).

```
cd ~
mkdir local
cd local
mkdir src
cd src
```

Decompress UCL folder as ucl-1.03 and build

```
cd ucl-1.03
./configure "CC=gcc -std=gnu89"
make all
```

To compile the UPX packer sources, Set the environment variable UPX_UCLDIR to point to your UCL build directory

```
export UPX_UCLDIR=$HOME/local/src/ucl-1.03
```

Download LZMA SDK [20] and copy content of LZMA SDK to UPX. Go to UPX's folder in home directory (`$(HOME)/upx.../`).

```
cd src/lzma-sdk
```

Extract downloaded LZMA SDK content here.

To modify the stub sources, Download upx-stubtools [21] (a number of cross-assemblers and cross-compilers) and go to the local folder in the home directory. (`$(HOME)/local/`).

```
cd ~/local
mkdir bin
cd bin
```

Decompress upx-build-20160918 folder as bin-upx

To make changes in .S file, go to upx's folder in home directory (`$(HOME)/upx.../`).

```
cd src/stub/src
gedit i386-win32.pe.S
```

Find section PEDOJUMP and modify (Also read `upx/doc/loader.txt`). Add instructions below after PEDOJUMP for setting exit interrupt just before Original Entry Point encountered.

```
mov ah,0x4c
int 21
```

Save this file and exit.

Go to src/stub to build(`$(HOME)/upx.../src/stub/`).

```
cd ..
make all
```

To build and get UPX's executable, go to UPX's folder in home directory (`$(HOME)/upx.../`).

```
make -B all
cd src
```

You can find `upx.out` here,

To execute UPX , for packing or compression

```
./upx.out <input_filename>.exe -o <output_filename>.exe
```

And for unpacking or decompression

```
./upx.out <input_filename>.exe -d <output_filename>.exe
```


REFERENCES

- [1] P. Mell, K. Kent, and J. Nusbaum, "Guide to malware incident prevention and handling," *Comput. Secur. Div. Inf. Technol. Lab. Natl. Inst. Stand. Technol.*, vol. 800–83, p. 101, 2005.
- [2] A. Brijesh, "Assessing malware detection using hardware performance counters," Boston University, 2017.
- [3] A. Mylonas and D. Gritzalis, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.
- [4] E. W. L. Leng, M. Zwolinski, and B. Halak, "Hardware performance counters for system reliability monitoring," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017, pp. 76–81.
- [5] J. Bulpin, "Hyper-threading aware process scheduling heuristics," *Proc. Annu. Conf. USENIX*, pp. 103–106, 2005.
- [6] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "ConFirm: Detecting firmware modifications in embedded systems using Hardware Performance Counters," *2015 IEEE/ACM Int. Conf. Comput. Des. ICCAD 2015*, pp. 544–551, 2016.
- [7] G. Contreras and M. Martonosi, "Power prediction for Intel XScale/spl reg/processors using performance monitoring unit events," *ISLPED '05. Proc. 2005 Int. Symp. Low Power Electron. Des. 2005.*, pp. 221–226, 2005.
- [8] J. Demme, M. Maycock, J. Schmitz, and A. Tang, "On the Feasibility of Online Malware Detection with Performance Counters Categories and Subject Descriptors," *Proc. 40th Annu. Int. Symp. Comput. Archit.*, vol. 41, no. 3, pp. 559–570, 2013.
- [9] P. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," *Proc. Dep. Def. HPCMP users Gr. Conf.*, vol. 32, pp. 7–10, 1999.
- [10] L. Kongress and A. Carvalho De Melo, "The New Linux 'perf' tools," 2010.
- [11] J. Reinders, "VTune Performance Analyzer Essentials Measurement and Tuning

- Techniques James Reinders,” *Interface*, 2005.
- [12] P. J. Drongowski, “An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyzer,” pp. 1–20, 2008.
 - [13] O. Yuschuk, “OllyDbg.” [Online]. Available: <http://www.ollydbg.de/>.
 - [14] “ImpREC - aldeid.” [Online]. Available: <https://www.aldeid.com/wiki/ImpREC>.
 - [15] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monroe, “SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security,” *2019 2019 IEEE Symp. Secur. Priv.*, pp. 345–363, 2019.
 - [16] “UPX Source Code.” [Online]. Available: <https://github.com/upx/upx>.
 - [17] “Intel® Microarchitecture Code Named Sandy Bridge Events.” [Online]. Available: <https://download.01.org/perfmon/index/snb.html>.
 - [18] “Uncover the hidden relationships in big data with Eureka | Nutonian, Inc.” [Online]. Available: <https://www.nutonian.com/>.
 - [19] “oberhumer.com: UCL data compression library.” [Online]. Available: <http://www.oberhumer.com/opensource/ucl/>.
 - [20] “UPX LZMA SDK.” [Online]. Available: <https://github.com/upx/upx-lzma-sdk>.
 - [21] “UPX stub-tools.” [Online]. Available: <https://github.com/upx/upx-stubtools/releases>.

BIOGRAPHICAL INFORMATION

This report belongs to Jay Mayank Patel. Jay has obtained a Bachelor of Engineering degree in Computer Engineering, Master of Technology in Information Technology and Master of Science in Computer Science and he has successfully defended his master's thesis in Information Security under the supervision of Dr. Jiang Ming.

Jay is interested in pursuing his career in the field of Information Security, Cyber Security, and Software Security. He has worked on several projects in Information Security area involving static and dynamic analysis tools, cloud services, malware analysis and various open source platforms such as Kali Linux, IDA Pro, IDS/IPS, Wireshark, etc. Jay was appointed as a Grader for the Spring 2018 and Fall 2018 semester and had successfully conducted the labs along with the CTF competition as a part of Information Security (CSE 5380).

Jay explored his expertise in the field of Information Security by working on the research project "Ransomware Early Stage Detection Using Machine Learning on Hardware Performance Counters" in Summer 2018 which improved his thought process and gave an insight into his research "On The Feasibility Of Malware Unpacking With Hardware Performance Counters".

Jay is willing to pursue his career in the area of his major and looking for a full-time opportunity in the field of Cyber Security to utilize her skills and R&D knowledge of Malware Analysis and Detection in the corporate world.