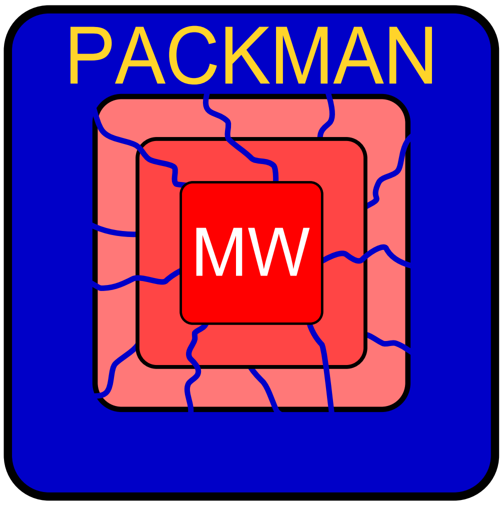
PACKMAN: Packed Malware Analyzer

Author: Enes Göktaş

Email: enes.goktas@vu.nl

23 April 2014



Supervised by:
PhD student Remco Vermeulen
Professor Herbert Bos

Vrije Universiteit Amsterdam - Computer Science Department
De Boelelaan 1081 - 1081 HV Amsterdam - The Netherlands

Contents

Problem Description

Malware developers may pack their malware to prevent detection by an anti-virus software or to thwart reverse engineering. A packed malware can be compressed, obfuscated or encrypted. Each packed malware contains an unpack routine, that jumps to the unpacked piece of code once it is done with unpacking. To make the analysis of malware even more difficult, malware may contain multiple unpacking layers. Each layer contains an unpack routine except the last one which is the malware itself. The difficulty of analyzing packed malware rises even more if malware developers use multiple different packers.

A malware developer is also likely to pack its malware with some customized packer to make its malware unique, hard to analyze, and to hide or remove known signatures. Unfortunately, custom packed malware increases the required amount of manual analysis. This work aims at uncovering this type of malware, where the unpacking layers are more or less independent from each other. In the rest of this report, we will consider the unpacking layers as ‘levels’.

One of the main obstacles of manually analyzing packed malware is that it is very time consuming. This report will describe the Packed Malware Analyzer (PACKMAN) that aims to ease the job of a malware analyst. The next sections will describe PACKMAN in more detail, show some experiment results and discuss possible future work.

PACKMAN: Packed Malware Analyzer

PACKMAN is developed by using the PIN Dynamic Binary Instrumentation framework[1]. The exhaustive framework has lots of functionalities and possibilities to instrument Linux and Windows binaries. This enables analysts to write advanced dynamic analysis tools for binaries running on the x86 (i.e. 32 bit) and x86-64 (i.e. 64 bit) instruction-set architectures. PACKMAN is built to work on windows based 32-bit executables.

The main features of PACKMAN are as follows:

- detect levels through tainting,
- dump detected levels,
- save library calls,
- keep an execution log file,
- analyze multi-threaded executables,
- analyze child processes.

The following subsections will discuss these features respectively.

Detect Levels

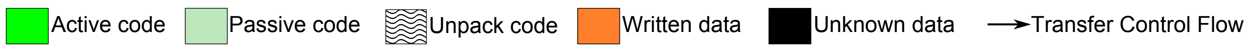
Essentialy, PACKMAN makes use of the fact that packed malware has to write the unpacked code to the memory after which it has to transfer control flow to the written code. Therefore, during the analysis of a packed malware, PACKMAN taints every written location. Once the packed malware transfers the control flow to a tainted location, PACKMAN considers it as the packed malware entering a new level (see Fig. 1). To realize this, PACKMAN analyzes every executed instruction. See below for the pseudocode of PACKMAN's core operation:

```
analyze_instruction(current_ins):
    # stopcondition for the automatic unpacking
    if is_call_to_sensitive_function(current_ins):
        reached_malware_core()    # current level is malware core

    if is_tainted(current_ins):
        new_level_detected()
    else if is_write(current_ins):
        taint_destination(current_ins)
```

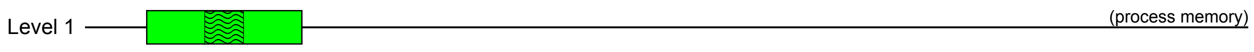
Because the activities of the unpack routines are different from the malicious activities of the core malware, we expect that the set of used library functions is also different. Therefore we believe that a stopcondition for PACKMAN could be based on this information. However stopping the detection of layers based on a single hit to a certain function would be a naive solution. A better solution would be to trigger the stopcondition based on more sensitive function calls or a certain execution pattern.

Regarding the detection of new levels, we experienced that just requiring the execution of one tainted instruction to enter a new level would also not be the best solution. For example, consider the case where the malware updates some instruction in the running level. As a result the updated instruction will become tainted. Whenever the tainted instruction is

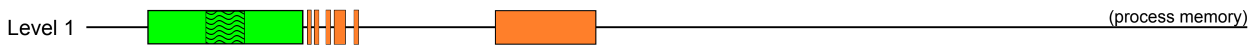
Figure 1: This series of figures illustrates an abstract view of an unpacking process of a packed malware by means of PACKMAN. The malware consists of two unpacking levels. The third level contains the original malware itself.



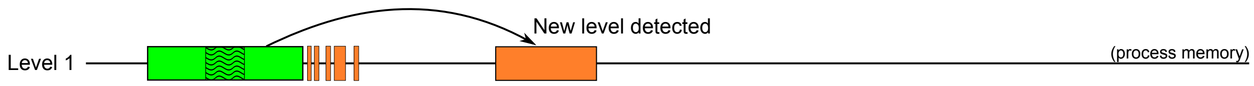
(a) Legend for the figures below.



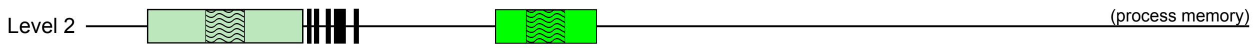
(b) Initial state. A packed malware is loaded to memory.



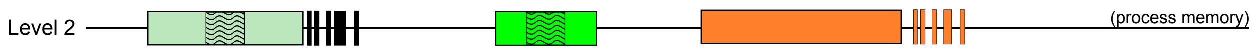
(c) The running code, including the unpacking routine, has written data to the memory. The written locations have been tainted as potential "unpacked data", which are the orange memory chunks.



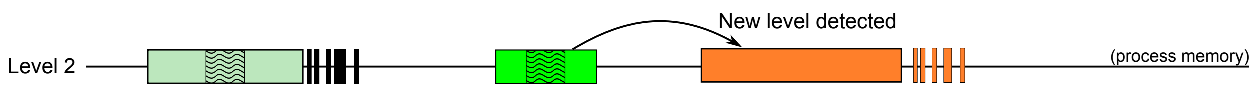
(d) Whenever a tainted memory location is executed, a new level is detected, where (1) the contiguous pages that contain the unpacked code are dumped from the memory to the disk as one file, and (2) all the taints are zeroed for the next level.



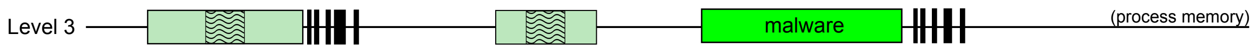
(e) Level 2 starts executing.



(f) Again, all the written data are tainted.



(g) Control is transfered to a location that was written in the current level. Hence, a new level is detected.



(h) The unpacking levels are peeled off from the packed malware and the core of the packed malware is reached. PACKMAN stops either when it detects the execution of a sensitive function, or when the malware stops itself.

executed, the naive solution will consider it as entering a new level. However, the tainted instruction is not the beginning of a new level. Therefore, to handle this case, PACKMAN requires that at least 5 consecutive executed instructions are tainted to detect a new level.

The tainting is done as follows. In each level, PACKMAN collects the memory locations that are written by the program. This is done with the `SaveWrite()` function in the tool. The function is called before every instruction that writes to the memory.

To check for new levels, the `CheckForNewLevel()` function is executed for every executed instruction. This function checks whether the addresses occupied by the instruction's opcode and its operand(s) are in the written memory region. If the execution of 5 consecutive instructions are in the written memory region, then PACKMAN assumes that a new level is detected.

The datastructure that is used for detecting levels is:

```
typedef struct MemBlock {        //Represents a memory region
        uint begin;              //Begin address
        uint end;                //End address
        MemBlock * next;         //Pointer to the next Memory Block
} MemBlock;

typedef struct Level {
        uint id;                 //Level's number
        uint oep;                //Level's Original Entry Point
        MemBlock * write_list;   //Written addresses
        ProcRef * procref_list;  //Saved library calls (Procedure Refs)
        Level * next;            //Pointer to the next level
} Level;
```

The written memory locations are collected in the write_list attribute of the Level structure. The written locations are represented with the MemBlock structure. A MemBlock structure stores the begin and end addresses and a pointer to the next MemBlock. Whenever there are contiguous memory blocks, these are merged to keep the memory usage by PACKMAN low.

As every software has a starting point in the code, which is called the Original Entry Point(OEP), the levels in packed malware also have a starting point. At the detection of new levels, the first executed tainted instruction is saved as the OEP of the new level.

The ProcRef attributes will be discussed in the Save Library Calls section.

Dump Levels

Another feature of PACKMAN is dumping the levels to Portable Executable(PE)[6] formatted files such that it can be further analyzed with other analysis tools (e.g., IDA Pro[4]).

PACKMAN dumps a PE formatted file at the beginning of every new level. To find the start of the PE file in the memory, PACKMAN utilizes two of the characteristics of loaded PE files, which are (1) first two bytes of a PE file contain the characters 'M' and 'Z' respectively, and (2) the PE files are page aligned. So, PACKMAN looks for the 'MZ' characters that are page aligned. This is done through a backward search starting at the OEP of the new level.

However our experiments have shown that not every level necessarily has a PE header. The 'MZ' magic characters are then missing, which disrupts the search to the base address of the level. Also the dumping becomes more difficult as the dumped file must have a PE header to define it as a PE file. There are a few problems that have to be tackled in the absence of a PE header: (1) searching the base of the level, (2) dumping the level as a PE file, and (3) finding the end of the level.

The difficulty of the first problem is that when PACKMAN performs a search to the base of the level, reading unallocated memory locations will produce access violation errors. However, by tracking memory allocations, PACKMAN is able to circumvent the reading of unallocated memory locations. The base of a level is defined as the base of an allocated memory region that contains the new level. Tracking memory allocations also eases the third problem. The end of a level is defined as the end of the allocated memory region that contains the new level.

To enable further analysis of the levels with other tools, it is required that the dumped file has a PE header to be recognized as a PE file. PACKMAN adds a "custom" PE header to levels that lack a PE header. It does this by carefully setting the magic characters, the size attributes and the OEP of the level.

The dumped file is put in the output directory of the analysis within the `out` directory of PACKMAN. The name of the output directory is the concatenation of the analysis timestamp and the name of the malware and the name of the dumped file is the concatenation of the process ID, the thread ID that dumped the file, and the number of the level to which the dumped file belongs. Initially the packed malware starts in level 1.

In the example below, "malwareX" has been analyzed on April 20th, 2014 at 12:15 and the thread with ID 0 in the initial process (i.e. indicated with the ID 0) has dumped the second level of the malware.

```
2014-04-20_12-15_malwareX                         <- output directory
 - dump_level_2_procID_0_threadID_0._exe_         <- dumped PE file name
```

Save Library Calls

To ease the reconsruction of an Import Table when dumping a level, PACKMAN stores the made library calls[1]. Determining whether a call instruction is targeting a library function or not is done by verifying that the target of the call instruction is within a memory region occupied by a library – PACKMAN tracks every library load and stores the occupied memory regions – and that the address of the call instruction itself is not within any of these stored regions. This solution is able to capture library calls made from unpacked levels.

The library function calls are saved per level in a ProcRef(i.e. Procedure Reference) list:

```
typedef struct ProcRef {
        uint proc;      //Address of the library procedure        (DST)
        uint ref;       //Address of the indirect memory location (SRC)
        ProcRef * next; //Pointer to the next ProcRef structure
} ProcRef;
```

Multithreaded Instrumentation

Because packed malware may utilize threads during the execution, PACKMAN handles threads[2] by assigning a Thread Local Storage[5] variable to each thread. This is especially required and useful when several aspects of the malware have to be tracked separately for each thread. The structure of the Thread Local Storage of the threads is as follows:

[1]The reconstruction feature itself is not yet implemented in PACKMAN. This feature is left as a future work.

```
class thread_data_t
{
        UINT32 written_instr_execution_count;
        MemBlock * temp_level_oep;
        UINT32 temp_long_jump;
        UINT32 addr_latest_exec_instr;
        CALLED_PROC calledProc;
}
```

The first three properties are used for managing the new level detection. `written_instr_execution_count` counts the number of consecutive execution of tainted instructions. As mentioned before, whenever this reaches 5, PACKMAN considers it as the detection of a new level. The second property is for temporarily saving the entry point of the 'potential' new level. If a new level is detected, `temp_level_oep` will be set as its OEP. The third property, `temp_long_jump`, is for saving the address of the instruction that jumped to the 'potential' new level. This information would be especially useful during the manual analysis. You then know what instruction made the entrance to the next level.

The fourth property, `addr_latest_exec_instr`, tracks what the previously executed instruction of the thread was, which is required to be able to set `temp_long_jump`. PACKMAN hooks certain functions to log more detailed information about them, e.g. argument and return values of VirtualAlloc invocations. To ensureand to verify in the hooks that this only happens at calls from the malware, PACKMAN stores the name of the called library function in `calledProc`.

Child Process Instrumentation

When a malware has child processes it is preferred that PACKMAN also analyzes those processes created by the malware. For example a packed malware may write the unpacked malware somewhere on the disk and start it afterwards as a separate process. We experienced a similar case where the malware copied itself to another location, removed the original one and ran the copied version. Furthermore, the malware performed this action after it had unpacked it a several times.

Fortunately, the PIN framework already provided an automatic way to propagate analysis tools to child processes[3]. However, PACKMAN intervenes this propagation to setup up the required parameters for the PACKMAN instance that will analyze the child process.

Analysis Logging

During the analysis of a malware, PACKMAN keeps a log file. These log files can be used to get a better insight in what the malware has been doing in general. The information is logged to the output directory of the analysis. An example of the names of the logfiles is as follows:

```
tool_procID_0.log
tool_procID_3820_parentID_0.log
```

The first file name in the listing above belongs to the initial process of the executable being analyzed. The second file name belongs to a child process. Because the name of the logfile has to be set before the process runs and the process ID is unknown beforehand, the logfile of the initial process always has procID 0. Since the process id of the child processes are known before they are started running, their corresponding process ID can be set in their logfile name when PACKMAN intervenes the propagation of the analysis from the parent process to the child process.

The logfiles consist respectively of 4 parts: information about the executable, loading of the executable, logged information during the execution of the executable, tearing down of the executable.

The *first part* looks like

```
Pin 2.13 kit 61147
 Instrumenting PE file: C:\{path_to}\{malware_name}
 Loaded PE file at [0x00400000..0x00457fff](0x00058000)
 Its Entry Point is 0x0045005d
```

It tells you which executable is being analyzed with what version of the PIN Binary Instrumentation Tool. Also it tells you where the executable is loaded and what its OEP is. This could potentially reveal the location of any dropped executables by the malware to the disk.

The *second part* of the logfiles consists information about the loading of the executable. The occupied memory region by the loaded modules is shown between square brackets. Additionally the size of the executable is shown between parantheses. Before an executable is started in Windows, three libraries are always loaded. These are KERNELBASE.dll, kernel32.dll and ntdll.dll. For an example of the second part, see the following:

```
EXEMEMORY  :: insert memory chunk [0x00400000..0x00457fff](0x00058000)
DLLMGMT    :: load at [0x75730000..0x75779fff] C:\{path_to}\KERNELBASE.dll
DLLMGMT    :: load at [0x76ce0000..0x76db3fff] C:\{path_to}\kernel32.dll
DLLMGMT    :: load at [0x773c0000..0x774fbfff] C:\{path_to}\ntdll.dll
```

The keywords at the beginning of the above lines describe what type of logging information is contained in the corresponding line. We will refer to these keywords as ”log types” in the rest of this report.

The *third part* of the logfiles contains the logged data during the execution of the malware after the initilization of the process (i.e. the second part mentioned above). To increase the legibility we have defined several log types that describe the logged lines. Below we will discuss each of the log types and give an example as how the lines corresponding to a log type would look like in the logfiles.

The *the last part* of the logfiles shows some information about the unloading of the analyzed executable. This part is printed when the executable or the analysis quits. An example of this part is as follows:

```
DLLMGMT    :: remove at [0x75fc0000..0x76c08fff] C:\{path_to}\SHELL32.dll
...
DLLMGMT    :: remove at [0x773c0000..0x774fbfff] C:\{path_to}\ntdll.dll
===============================================
Current level = 3
===============================================
```

First, all of the libraries are unloaded. Then there is the possibility to print some statistics about the analyzed executable. Currently only the highest level reached in the packed malware is printed in the statistics part.

The logged lines in the third part of the logfiles are structured as follows:

```
[T:{thread_id}|L:{level_nr}] {log_type} :: {description}
```

This structure shows which thread logged the line and at what level it was at the moment of logging.

Log Type: LT_NEWLEVEL

When PACKMAN is about to detect a new level it prints the following lines:

```
[T:0|L:1] *NEWLEVEL*:: a potential new level detected
[T:0|L:1] *NEWLEVEL*:: #1 executed a written instruction @0x03552d80
[T:0|L:1] *NEWLEVEL*:: #2 executed a written instruction @0x03552d81
[T:0|L:1] *NEWLEVEL*:: #3 executed a written instruction @0x03552d83
[T:0|L:1] *NEWLEVEL*:: #4 executed a written instruction @0x03552d86
[T:0|L:1] *NEWLEVEL*:: #5 executed a written instruction @0x03552d8d
[T:0|L:2] *NEWLEVEL*:: detected a new level! its entry point is 0x03552d80
[T:0|L:2] *NEWLEVEL*:: instruction that jumped to the new level is at \
        0x00401858
[T:0|L:2] *NEWLEVEL*:: new level is in memory chunk \
        [0x03510000..0x03555fff](0x00046000) and has *NO* PE file header
[T:0|L:2] *NEWLEVEL*:: dumped new level to file \
        C:\{path_to}\dump_level_1_procID_0_threadID_0_custom._exe_
```

Once there is a contiguous execution of 5 written instructions, the dumping phase starts. PACKMAN first prints some information about the new level (i.e. its OEP, instruction that jumped to the new level, size of the new level)and then it logs the path of the dumped file.

However, false positives will be logged as follows:

```
[T:0|L:1] *NEWLEVEL*:: a potential new level detected
[T:0|L:1] *NEWLEVEL*:: #1 executed a written instruction @0x00456ac7
[T:0|L:1] *NEWLEVEL*:: detection of new level failed
```

In this example, the instruction at `0x00456ac7` was modified, got tainted and had consequently triggered the detection of a new level when executed. Because the successive instruction was not tainted, it broke the heuristic and stopped the level detection process.

Log Type: LT_LIBCALL

Calls to library functions are logged as:

```
[T:0|L:1] LIBCALL     :: @0x000624cf call dword ptr [0x31144]=0x76d327fd \
         kernel32.dll::CreateThread
```

This shows at which memory location the call was made and how library functoin was invoked. In this example an indirect call instruction `call dword ptr [0x31144]` at `0x000624cf` invoked the `CreateThread` function locating at `0x76d327fd`.

Log Type: LT_CALLDETAILS

Certain library function, e.g. CreateThread that contains the thread function address as the third argument, may be of special interest to malware analysists. An example of a logged line for such a function is:

```
[T:0|L:1] CALLDETAILS  :: CreateThreadBefore; number of threads started \
        by exe=1; thread Entry Point=0x000623ee
[T:0|L:1] CALLDETAILS  :: CreateThreadAfter; done
```

Log Type: LT_DIRECTCALL

Usually the calls to library functions from the malware code happen through indirect call instructions. However, during the experiments we noticed that there might also be direct control-flow from malware code to the library functions. An example of a direct control flow:

```
[T:0|L:1] DIRECTCALL  :: @0x6fff03c5 jmp 0x77404bc5 \
       ntdll.dll::NtCreateUserProcess
```

Log Type: LT_EXEMEMORY

PACKMAN saves memory regions allocated by the malware. These are logged as:

```
[T:0|L:1] EXEMEMORY    :: insert memory chunk [0x034b0000..0x034f2fff] \
       (0x00043000)
```

Log Type: LT_DLLMGMT

Loading and unloading of libraries are logged as follows:

```
[T:0|L:1] DLLMGMT      :: load at [0x76980000..0x76a2bfff] \
       C:\{path_to}\msvcrt.dll
[T:0|L:1] DLLMGMT      :: remove at [0x76980000..0x76a2bfff] \
       C:\{path_to}\msvcrt.dll
```

Log Type: LT_THREADMGMT

PACKMAN logs the beginning and ending of threads as follows:

```
[T:5|L:1] THREADMGMT  :: new thread with id 5 started; new #threads is 3
[T:5|L:1] THREADMGMT  :: thread with id 5 stopped; new #threads is 2
```

Log Type: LT_CHILDPROC

When a child process is started, PACKMAN logs the path to the started executable and the id of the child process:

```
[T:0|L:3] CHILDPROC   :: started app C:\{path_to}\ajefuz.exe with \
       process id 3192
```

Log Type: LT_ERROR

Errors or unexpected cases are logged as follows:

```
[T:0|L:1] !! ERROR !! :: {error message}
```

Experiments

We developed PACKMAN with PIN Binary Instrumentation Tool version "Pin 2.13 kit 61147" in Windows 7 (32 bit). During the development of PACKMAN we mainly used a simple UPX-packed Hello World program. Later we used other packed malware to test and further improve PACKMAN. In this section we will discuss our the experiments with the packed Hello World program, the packed ZeusP2P malware, and the packed Virut malware.

Table 1: Measurements of the execution times of the analyzed executables. The second column (i.e. Clean Run) is the execution time without the pin framework. The third column indicates the execution time of the executable with the pin framework but without the instructions instrumented. The last column indicates the execution time with the instructions instrumented.

Packed exec.	**Clean Run**	**Tainting Disabled**	**Tainting Enabled**
Hello World	200 ms	3,495 ms	21,971 ms
Zeus P2P	8,951 ms	20,816 ms	155,165 ms
Virut	3,801 ms	7,520 ms	270,843 ms

Table 2: Performance degradation. The numbers show how many times the execution time increases when the type of analysis is changed from the one to the other in Table 1.

Packed exec.	**Clean Run -> Tainting Disabled**	**Tainting Disabled -> Tainting Enabled**	**Clean Run -> Tainting Enabled**
Hello World	17.5x	6.3x	109.9x
Zeus P2P	2.3x	7.5x	17.3x
Virut	2.0x	36.0x	71.3x

Packed Hello World

The analysis of the packed Hello World took about 22 seconds with tainting enabled. This is 110 times slower than when the executable is run without PACKMAN, which took 0.2 seconds to execute. The high performance degradation is the result of the clean run having a very low execution time.

The packed Hello World was packed with UPX and contained two levels, namely the unpacking level and the level that contains the original Hello World program. PACKMAN successfully dumped the second level:

```
[T:0|L:1] *NEWLEVEL*:: a potential new level detected
[T:0|L:1] *NEWLEVEL*:: #1 executed a written instruction @0x012913c0
[T:0|L:1] *NEWLEVEL*:: #2 executed a written instruction @0x01298620
[T:0|L:1] *NEWLEVEL*:: #3 executed a written instruction @0x01298622
[T:0|L:1] *NEWLEVEL*:: #4 executed a written instruction @0x01298623
[T:0|L:1] *NEWLEVEL*:: #5 executed a written instruction @0x01298625
[T:0|L:2] *NEWLEVEL*:: detected a new level! its entry point is 0x012913c0
[T:0|L:2] *NEWLEVEL*:: instruction that jumped to the new level is at \
       0x012aa9b9
[T:0|L:2] *NEWLEVEL*:: new level is in memory chunk \
       [0x01280000..0x012abfff](0x0002c000) and has a PE file header
[T:0|L:2] *NEWLEVEL*:: dumped new level to file \
       C:\{path_to}\dump_level_2_procID_0_threadID_0._exe_
```

Figure 2 shows the disassembly of the original Hello World program and of the dumped version in Ida pro. As expected the disassembly is the same. The only two differences are the addresses and the "Exchange" variable not being recognized by Ida pro in the dumped version of the hello world executable.

Packed ZeusP2P

The analysis of the ZeusP2P malware with tainting enabled took about 155 seconds. This is 17 times slower than when the executable is run without PACKMAN. The execution time of the malware without PACKMAN took about 9 seconds. This performance degradation is much less than the performance degradation of the Hello World analysis, which is caused by the malware itself having a much higher execution time than the Hello World program.

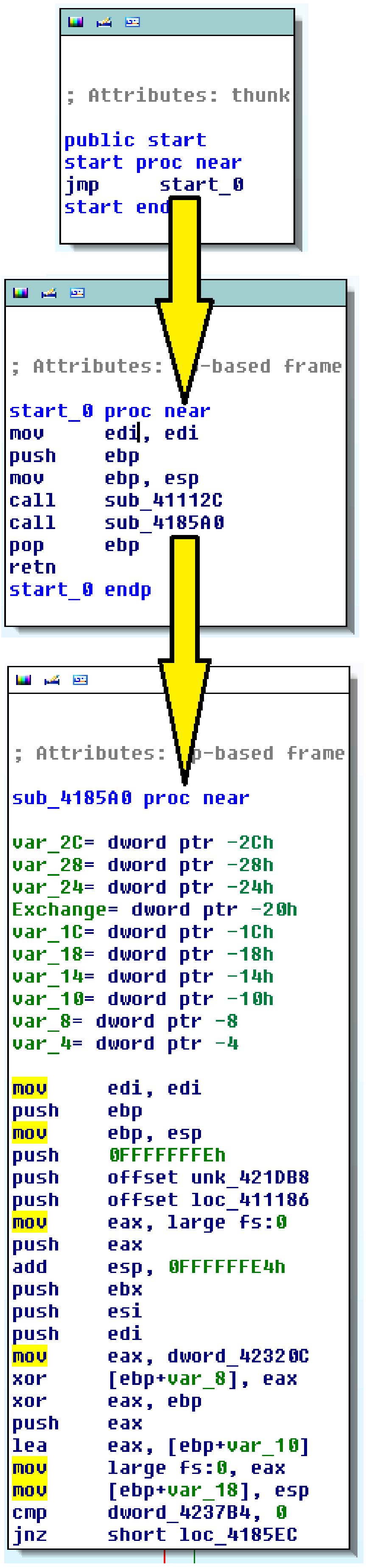
The output directory of the analysis of ZeusP2P with PACKMAN looks as follows:

```
2014-04-22_16-45_zeus_04193500185c6d063d909ecfec03448a
- tool_procID_0.LOG
- dump_level_2_procID_0_threadID_0_custom._exe_
- dump_level_3_procID_0_threadID_0._exe_
- dump_level_4_procID_0_threadID_0._exe_
- tool_procID_3324_parentID_0.LOG
- dump_level_2_procID_3324_threadID_0_custom._exe_
- dump_level_3_procID_3324_threadID_0._exe_
- tool_procID_1480_parentID_0.LOG
```

We can see in the output directory that the malware had two child processes, namely one with ID 3324 and one with ID 1480. The main process consisted of 4 levels and the process with ID 3324 consisted of 3 levels. The latter process is started in level 4 of the main process. With some further investigation we found that the main process copied the malware to another location on the disk, removed the original file and started the copied malware, which is the process with ID 3324:

```
[T:0|L:4] CHILDPROC   :: started app \
       C:\Users\pmat\AppData\Roaming\Kaacy\usogyp.exe with process id 3324
```

The interesting part is that until level 3 the execution of the main process and process of the copied malware is the same. The execution between the processes differ in level 3. As far as we noticed in the log files, the difference between the two is that the copied malware goes through the process list and injects data in several of the processes. We believe that the malware checks in level 3 what the context of the executed malware is and depending on the context it enters either level 4 and copies the malware to another location, or it starts injecting data to different processes.

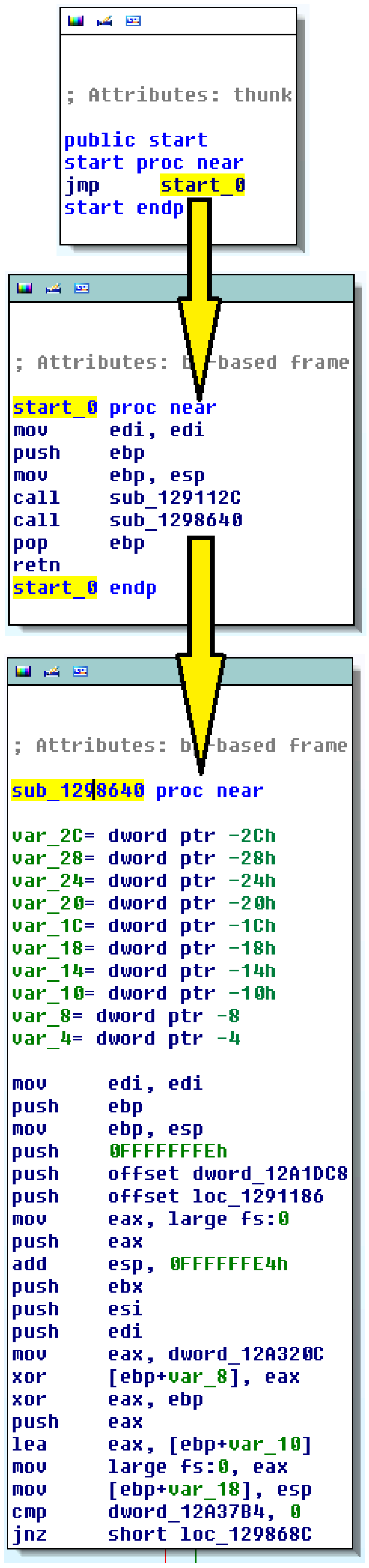


```
; Attributes: thunk

public start
start proc near
jmp     start_0
start end
```

```
; Attributes: -based frame

start_0 proc near
mov     edi, edi
push    ebp
mov     ebp, esp
call    sub_41112C
call    sub_4185A0
pop     ebp
retn
start_0 endp
```

```
; Attributes: p-based frame

sub_4185A0 proc near

var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
Exchange= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

mov     edi, edi
push    ebp
mov     ebp, esp
push    0FFFFFFFEh
push    offset unk_421DB8
push    offset loc_411186
mov     eax, large fs:0
push    eax
add     esp, 0FFFFFFE4h
push    ebx
push    esi
push    edi
mov     eax, dword_42320C
xor     [ebp+var_8], eax
xor     eax, ebp
push    eax
lea     eax, [ebp+var_10]
mov     large fs:0, eax
mov     [ebp+var_18], esp
cmp     dword_4237B4, 0
jnz     short loc_4185EC
```

(a) Original Hello World program.



```
; Attributes: thunk

public start
start proc near
jmp     start_0
start endp
```

```
; Attributes: b -based frame

start_0 proc near
mov     edi, edi
push    ebp
mov     ebp, esp
call    sub_129112C
call    sub_1298640
pop     ebp
retn
start_0 endp
```

```
; Attributes: b -based frame

sub_1298640 proc near

var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

mov     edi, edi
push    ebp
mov     ebp, esp
push    0FFFFFFFEh
push    offset dword_12A1DC8
push    offset loc_1291186
mov     eax, large fs:0
push    eax
add     esp, 0FFFFFFE4h
push    ebx
push    esi
push    edi
mov     eax, dword_12A320C
xor     [ebp+var_8], eax
xor     eax, ebp
push    eax
lea     eax, [ebp+var_10]
mov     large fs:0, eax
mov     [ebp+var_18], esp
cmp     dword_12A37B4, 0
jnz     short loc_129868C
```

(b) Dumped Hello World program.

Figure 2: Part of the disassembly of the original and the dumped Hello World in IDA Pro

For example, two of the processes that are injected with data are of explorer.exe and of taskhost.exe:

```
[T:0|L:3] LIBCALL     :: @0x00431258 call dword ptr [0x40112c]= \
        0x7782859f kernel32.dll::WriteProcessMemory
[T:0|L:3] CALLDETAILS :: WriteProcessMemoryBefore; \
        target process handle=0x174; \
        target process id=0x0x76c; \
        target process image name=C:\Windows\explorer.exe
[T:0|L:3] CALLDETAILS :: WriteProcessMemoryBefore; \
        destination in target process=0x031a0000; \
        buffer in this process=[0x03c60590..0x03ca658f]; \
        size=0x00046000
[T:0|L:3] CALLDETAILS :: WriteProcessMemoryAfter; done
```

and

```
[T:0|L:3] LIBCALL     :: @0x00431258 call dword ptr [0x40112c]= \
        0x7782859f kernel32.dll::WriteProcessMemory
[T:0|L:3] CALLDETAILS :: WriteProcessMemoryBefore; \
        target process handle=0x174; \
        target process id=0x5ac; \
        target process image name=C:\Windows\System32\taskhost.exe
[T:0|L:3] CALLDETAILS :: WriteProcessMemoryBefore; \
        destination in target process=0x00340000; \
        buffer in this process=[0x03c60590..0x03ca658f]; \
        size=0x00046000
[T:0|L:3] CALLDETAILS :: WriteProcessMemoryAfter; done
```

In Figure 3, that depicts level 4, we can see the path of the copied malware.

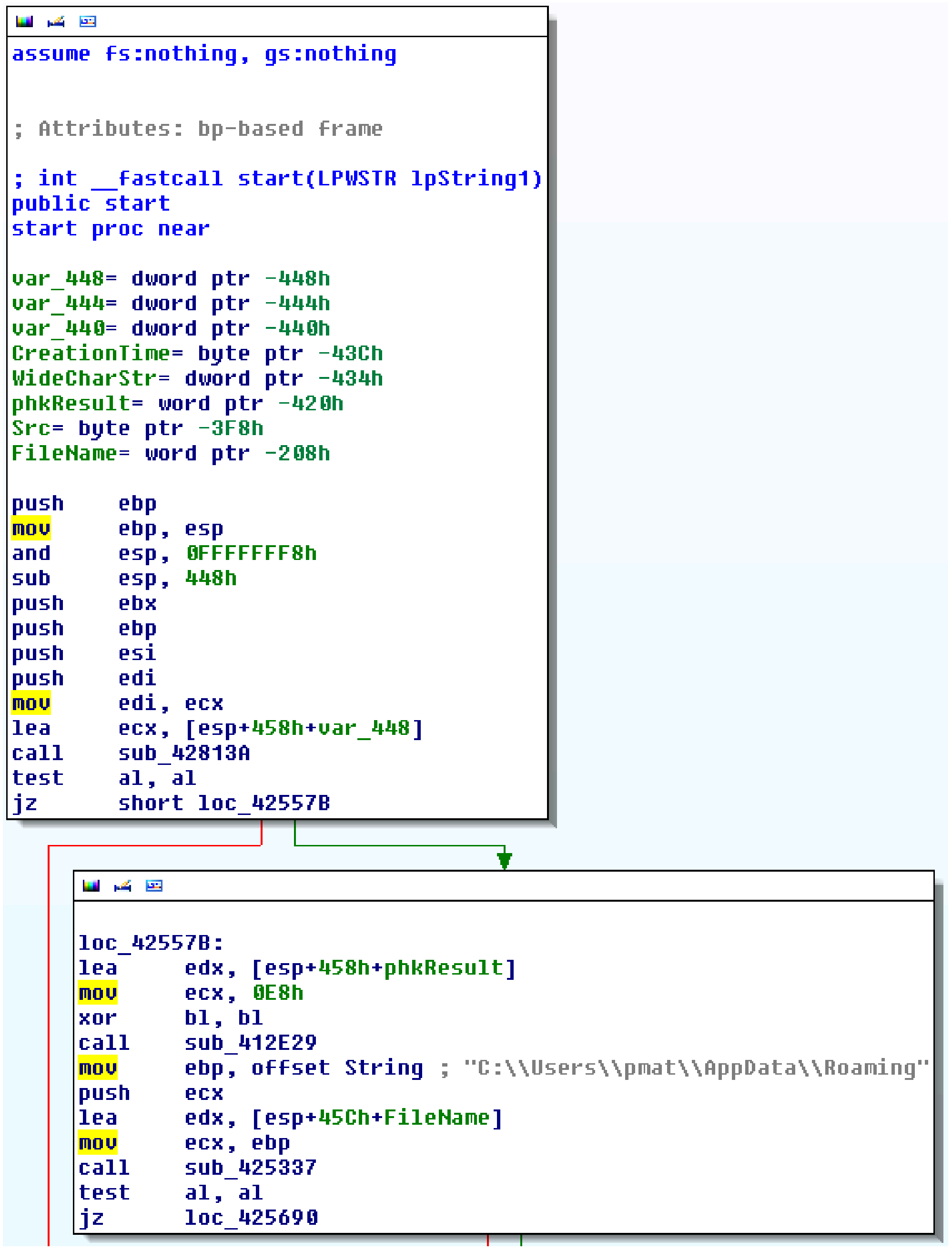
Packed Virut

The analysis of the Virut malware with tainting enabled took about 271 seconds. This is 71 times slower than when the executable is run without PACKMAN, which took about 4 seconds to execute. This performance degradation is notably much higher than the performance degradation of the analysis of ZeusP2P. This is probably to the additional obfuscation layer that the Virut sample has, see Figure 4. This obfuscation leads to more instructions being executed and as a result has an additional stress on the instrumentation by PACKMAN.

The output directory after the analysis of the Virut malware with PACKMAN had the following structure:

```
2014-04-22_17-01_virut_5b7301d13defd76c2597165ad1e6993b
- dump_level_2_procID_0_threadID_0._exe_
- dump_level_3_procID_0_threadID_0_custom._exe_
- dump_level_4_procID_0_threadID_0._exe_
```

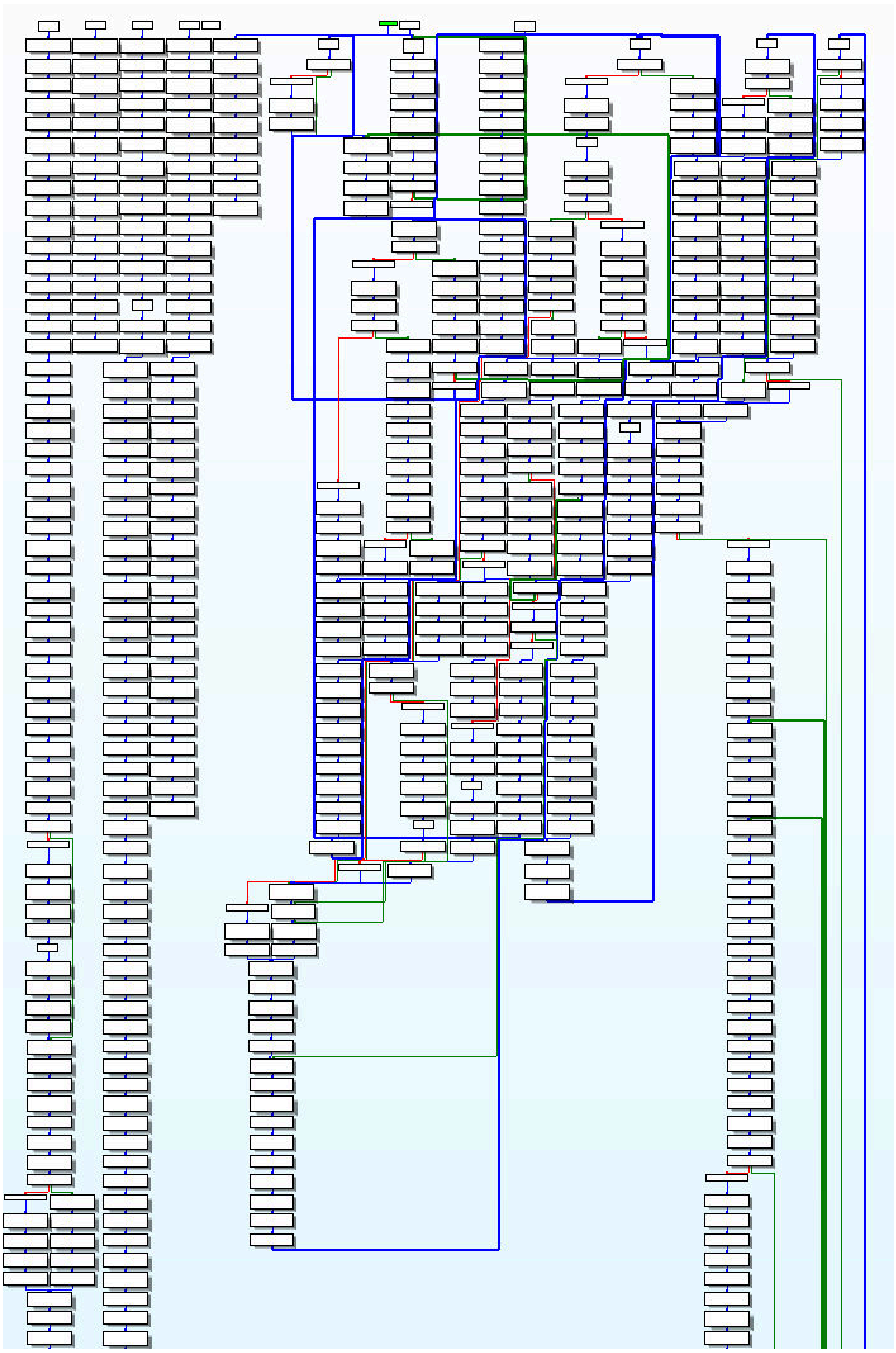
Figure 3: Disassembly of level 4 that was dumped by the main process. In the lower basic block we can see the path of the copied malware.



```
assume fs:nothing, gs:nothing


; Attributes: bp-based frame

; int __fastcall start(LPWSTR lpString1)
public start
start proc near

var_448= dword ptr -448h
var_444= dword ptr -444h
var_440= dword ptr -440h
CreationTime= byte ptr -43Ch
WideCharStr= dword ptr -434h
phkResult= word ptr -420h
Src= byte ptr -3F8h
FileName= word ptr -208h

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
sub     esp, 448h
push    ebx
push    ebp
push    esi
push    edi
mov     edi, ecx
lea     ecx, [esp+458h+var_448]
call    sub_42813A
test    al, al
jz      short loc_42557B
```

```

loc_42557B:
lea     edx, [esp+458h+phkResult]
mov     ecx, 0E8h
xor     bl, bl
call    sub_412E29
mov     ebp, offset String ; "C:\\Users\\pmat\\AppData\\Roaming"
push    ecx
lea     edx, [esp+45Ch+FileName]
mov     ecx, ebp
call    sub_425337
test    al, al
jz      loc_425690
```

Figure 4: A disassembled function of the dumped third level in IDA pro. The function is obfuscated by splitting instructions into more basic blocks and then scattering the resulting basic blocks in the memory. The function starts at the tiny green box at the top.



Future Work

We believe that there are still more improvements possible to PACKMAN. In this section we will list the ideas about PACKMAN we came up with, which are left as a future work.

- The project could be set up as an open source project such that other developers and experienced malware analysts could contribute to the PACKMAN's development.
- Currently the dumped PE files are non-executable and are only usable in static analysis. An interesting research would be making the unpacked levels stand-alone executables.
- Reconstructing the Import Address Table in the dumped PE files. This would improve the static analysis as you would then, for example, know what functions the call instructions are targeting. We believe that this improvement is also one of the requirements for making the dumped files executable.
- Adding interactive options to PACKMAN could make the malware analysis more practical.
- Research ways of detecting whether the reached level is the core of the malware or not.
- Investigate ways to improve the runtime performance of PACKMAN.

Conclusion

In this report we have discussed the PACKMAN tool, which is built with the PIN binary instrumentation tool of Intel. PACKMAN unveils the inner layers of a packed malware. To accomplish this, it taint all written memory and detects a new level of execution when the instruction pointer starts executing tainted memory. Moreover, PACKMAN is applicable to malware that utilizes multiple threads and that make use of child processes. We have tested the tool on a simple UPX-packed Hello World program and two real packed malware, namely ZeusP2P and Virut. The main goal of PACKMAN is to ease the analysis of packed malware and eventually to substitute as much parts of the manual analysis performed by malware analysts as possible.

References

[1] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*, Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005, pp. 190-200.

[2] Pin User Guide, *Instrumenting Threaded Applications*, https://software.intel.com/sites/landingpage/pintool/docs/62141/Pin/html/index.html#MallocMT

[3] Pin User Guide, *Instrumenting Child Processes*, https://software.intel.com/sites/landingpage/pintool/docs/62141/Pin/html/index.html#FollowChild

[4] Hex-Rays, *Interactive DisAssembler (IDA) Pro*, https://www.hex-rays.com/products/ida/

[5] Wikipedia, *Thread Local Storage*, http://en.wikipedia.org/wiki/Thread-local_storage

[6] Wikipedia, *Portable Executable*, http://en.wikipedia.org/wiki/Portable_Executable

[7] Wikipedia, *Zeus (Trojan horse)*, http://en.wikipedia.org/wiki/Zeus_%28Trojan_horse%29

[8] Wikipedia, *Virut*, http://en.wikipedia.org/wiki/Virut