Implementing your own generic unpacker

Julien Lenoir

12/10/2015

Abstract

As it was the case 10 years ago, packed malwares are still a real problem for large scale analysis. Many papers on this subject exist and developping yet another generic unpacker is not something new. Driven by the need to have an open-source implementation of this kind of tool and by our curiosity we decided to implement our own. After unsuccessful attempts to implemented an existing architecture, we focused on improving it and propose ours in this paper. This is also a small journey inside Windows internals related to exception handling and memory management.

Contents

1 Introduction

1.1 Context

As part of a research activity on a malware classification framework we have encountered problems with packed executables. Indeed, a significant number of malwares are packed to defeat anti-virus detection. Applying classification algorithms on packed malware is problematic since some features would be extracted from the packer code rather from the malware code. The need for a generic unpacker arose naturally. We needed a generic tool able to unpack a large number of "off the shelf" packers as long as homemade ones. More precisely, the constraints on our tool are:

- It should work inside a virtual machine, on popular virtualization technology.
- It must be as stealthy as possible.
- It must rebuild a valide PE for static analysis.

We decided to implement the unpacking engine described in MutantX-S [10] which is based on Justin [4]. Such approaches target *simple packers*, that is to say packers that do *not* use code virtualization and unpack fully their payload prior to executing it. Recent studies have shown that the majority of malwares is still packed by those kind of simple packers[2]. So this approach should still be relevant.

1.2 Related work

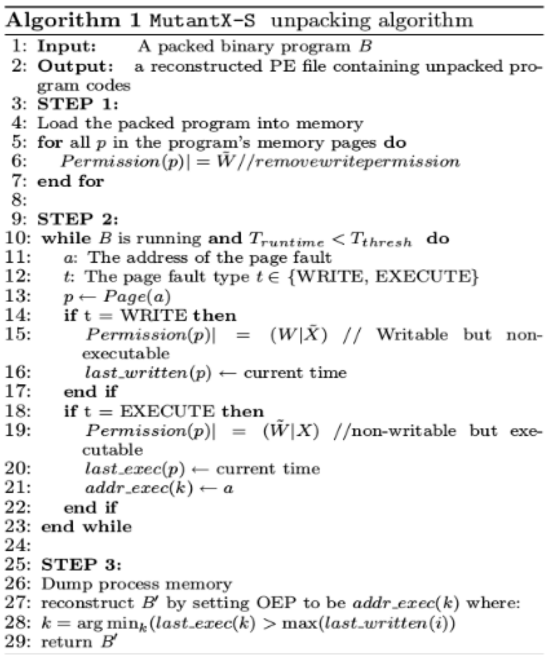
The fact that we developped our own generic unpacker is nothing really new. Many papers on the subject have been released so far: MutantX-S[10], Justin[4], Ether[9], Omniunpack[8], Renovo[11] and more recently Packer Attacker[3]. Unfortunately, they did not match all our constraints. Moreover, except for Packer Attacker which has been released recently, they are not open source.

One of the interest of this paper resides in the implementation details of such a tool on a *recent* operating system, i.e. Windows 7 32 bits. We think that anyone interested in developping this kind of tool for malware classification or manual analysis might be interested. Moreover, it is also a small journey inside Windows memory management and exception handling mechanisms.

1.3 Generic unpacking

All generic unpackers focus on one goal: finding when the original payload starts its execution. We call this point the OEP (Original Entry Point) of the unpacked program. That is to say the point where the packer has finished unpacking the original program and transfers control to it. Our tool is targeting packers we could qualify as *simple*, that is to say packers that fully decrypt and/or uncompress the original code in memory before transfering control to it. A lot of very popular COTS packers and custom/homemade packers used by malware authors to defeat ad-hoc unpackers fall into this category. Packers that use code virtualization or that mix unpacking layers and payload code are not supported.

We tried to implement an algorithm which is well described in MutantX-S[10] and Justin[4] papers (interested readers are encouraged to read MutantX-S paper for algorithm demonstration). The following figure , extracted from the paper, details all the algorithm's steps. The general idea here is to run the packed program, monitor WRITE and EXECUTE memory accesses performed by the process during the unpacking phase and then apply an algorithm on this trace to figure out the OEP.



Algorithm 1 `MutantX-S` unpacking algorithm

```
Input:    A packed binary program $B$
Output:   a reconstructed PE file containing unpacked program codes
STEP 1:
Load the packed program into memory
for all $p$ in the program's memory pages do
    $Permission(p)| = \bar{W} //removewritepermission$
end for

STEP 2:
while $B$ is running and $T_{runtime} < T_{thresh}$ do
    $a$: The address of the page fault
    $t$: The page fault type $t \in$ {WRITE, EXECUTE}
    $p \leftarrow Page(a)$
    if t = WRITE then
        $Permission(p)| = (W|\bar{X})$ // Writable but non-executable
        $last\_written(p) \leftarrow$ current time
    end if
    if t = EXECUTE then
        $Permission(p)| = (\bar{W}|X)$ //non-writable but executable
        $last\_exec(p) \leftarrow$ current time
        $addr\_exec(k) \leftarrow a$
    end if
end while

STEP 3:
Dump process memory
reconstruct $B'$ by setting OEP to be $addr\_exec(k)$ where:
$k = \arg\min_k(last\_exec(k) > \max(last\_written(i))$
return $B'$
```

To find the OEP we will consider only the set of memory pages that are both written and executed. We will then find the time *t*, when the last write is performed to the set of pages. pages. The OEP is then the first execution that happens after *t*.

This can be explained in simple words. As the original code will be decrypted/uncompressed by the packer code, then executed, the OEP is necessarily located in the is written-then-executed set. Moreover, if we consider that the original code is fully unpacked before it gets executed, then the OEP is the first execution in a written-then-executed page that happens after the last write (end of unpacking) in this written-then-executed page set.

Although this algorithm is quite simple in theory, implementing it efficiently on a working operating system is quite tricky. To do so, we must be able to :

- change memory protection on the fly,
- catch faults induced by those modifications,
- prevent the monitored program from changing back its memory protection,
- handle memory layout change (allocations, dll loading, etc.),
- not disturb the monitored program execution.

We will see how all these constraints can be satisfied, but we first need a deep understanding of Windows 7 memory management and exception handling mechanisms.

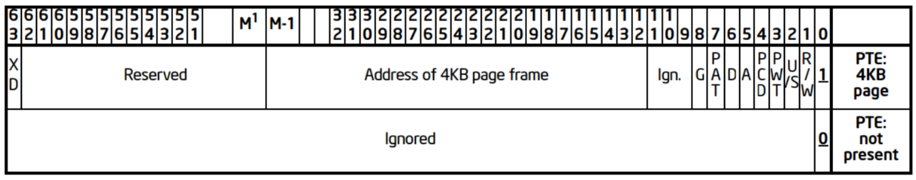
2 Inside Windows 7

This section describes the key Windows internal system mechanisms involved in memory management and exception handling of Windows 7. Expert readers will notice that there are a number of details omitted for simplicity purpose. This section was written by reading Windows Internals[7], browsing ReactOS[6] source code and studying Windows code (kernel and userland). This chapter applies to Windows 7 32bits in PAE (Physical Address Extension) mode. Nevertheless, some of those key Windows internal mechanisms probably also apply to newer versions of the operating system.

2.1 PTEs and address translation

On x86-64, when paging and protected mode are active, virtual addresses are translated to physical addresses using PDEs (page directory entries) and PTEs (page table entries). Complete documentation on virtual to physical address translation can be found in intel developper's manual ([5]).

PTE entries characterize the way a 4KB page of virtual memory can be accessed at the lowest level of the CPU. On PAE systems, PTE entries are 64bits long, the following figure shows their layout:



Intel documentation contains the full description of all bits of those PTEs, we will not provide details about all of them. Nevertheless somes of them are interesting because we will use them in our tool:

- **bit 0** (present): if 0 PTE is invalid, no translation is possible. Any access to the corresponding virtual address causes a PF (page fault). If 1, a physical page for this virtual address exists.
- **bit 1** (R/W): 0 means virtual address is not writable, 1 means it is.
- **bits 9 to 11**: ignored bits.
- **bits 12 to 62**: physical memory address.
- **bit 63** (XD): 1 means the virtual address is not executable, 0 means it is executable.

From a hardware perspective a PTE can only be present or not, for the Windows operating system there is a larger variety of PTE types.

Indeed, Windows uses PTE's reserved bits of both valid and invalid PTEs for its own purpose. Those internal OS mechanisms are enforced by the memory manager fault handler (*MmAccessFault*, described later). Examples of invalid PTEs for the processor but nevertheless meaningful for the kernel are:

- **Page File**: the physical page for a given virtual address is not in memory. It has to be retrieved from paging file.
- **Demand zero**: no physical page is associated with this virtual address. Upon first access, a physical page full of zeroes must provided.

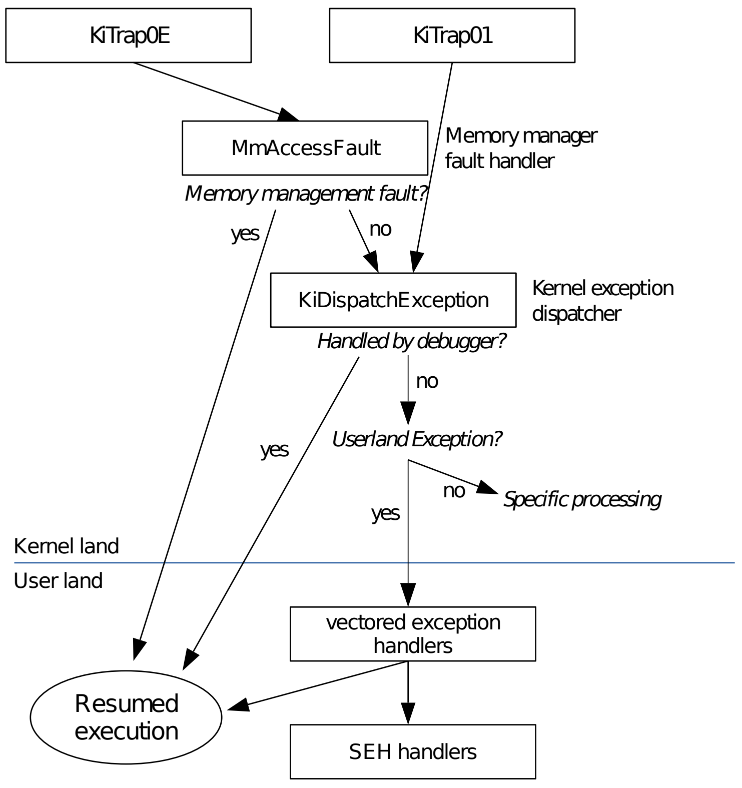
- **Prototype PTE**: used for memory shared between different processes.

Windows makes also use of reserved bits of valid PTEs for copy-on-write. For example:

- **bit 9**: when set, tells the memory manager to perform copy on write, i.e. duplication of physical page on next write access.
- **bit 11**: when set, tells memory manager that copy-on-write (physical page duplication) is already done.

2.2 Page faults and exceptions

When an access violation occurs, due either to an invalid PTE or to a protection bits violation, the processor triggers a page fault exception 0xE. The processor then transfers control to the Windows kernel by calling the page fault handler *KiTrap0E*. When *KiTrap0E* is called the kernel is attached to the process causing the fault. The following figure shows the *KiTrap0E* code path:



While operating system triggers page faults very often, only a few of them are actual access violations of userland processes. *KiTrap0E* quickly call the memory manager fault handler (*MmAccessFault*). This function is complex since it implements most of the low level operating system memory management logic. We will not explain its whole behaviour, however a few points are worth describing. It has the following prototype:

```
MmAccessFault(FaultStatus, VirtualAddress, PreviousMode, pTrapFrame);
```

- FaultStatus: a bitfield set by the processor indicating the type of fault. Common fault status are 0 for read access, 1 for write access and 8 for execute access.
- VirtualAddress: the virtual address (userland or kernel) that caused the fault.
- PreviousMode: execution mode of processor when the fault happened. 0 for kernel mode and 1 for user mode.
- pTrapFrame: a pointer to the KTRAP_FRAME structure containing the execution context of the process at fault time.

It returns a NTSTATUS code:

- STATUS_SUCCESS: process execution is resumed using the possibly modified input KTRAP_FRAME structure.
- Error status code: exception handling continues.

If an invalid access was made on a special PTE set up by the memory manager on purpose (see 2.1) like a *demand zero* page, *MmAccessFault* performs the required actions like bringing up a zero-allocated physical page and resuming process execution.

On the other hand, if the access causing the page fault is not induced by an internal memory mechanism, in other words *MmAccessFault* does not need to handle the exception, it is forwarded to the exception handling chain. Prior to forwarding exceptions to userland handlers[1], the kernel ensures that it can actually build the required exceptions structures on the top of the userland thread's stack. However, if for any reason the kernel does not manage to build those structures on the stack, the userland process is killed.

In the end registered userland exception handlers are called inside the process, starting with the *VectoredExceptionHandlers*.

2.3 Process virtual address space

In the Windows world, userland virtual addresses can be either commited, free or reserved for further commitment. Every reserved or commited memory page belong to a region of pages, this region corresponds to a set of contiguous memory pages. All pieces of information relevant to userland memory pages are accessible through the the MEMORY_BASIC_INFORMATION structure:

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;
  PVOID  AllocationBase;
  DWORD  AllocationProtect;
  SIZE_T RegionSize;
  DWORD  State;
  DWORD  Protect;
  DWORD  Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Where *AllocationBase*, *RegionSize* and *AllocationProtect* refer to the memory region a given page belongs to. The *State* flags defines if the memory is free, reserved of committed. *Protect* defines the protection on this memory page i.e.

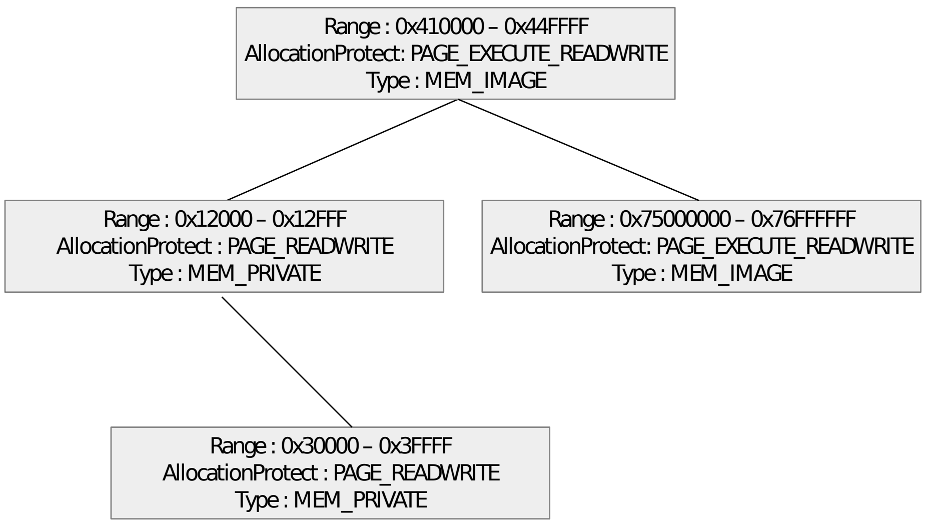
[1]Entry point: *KiUserExceptionDispatcher*

a combination of readable, writable and executable (32 distinct combinations are available).

Userland processes can interact with their virtual address space. They can allocate or de-allocate memory regions and map or unmap sections (memory mapping of files or shared memory) throught system calls like:

- NtAllocateVirtualMemory
- NtProtectVirtualMemory
- NtQueryVirtualMemory
- NtFreeVirtualMemory
- NtMapViewOfSection
- NtUnmapViewOfSection

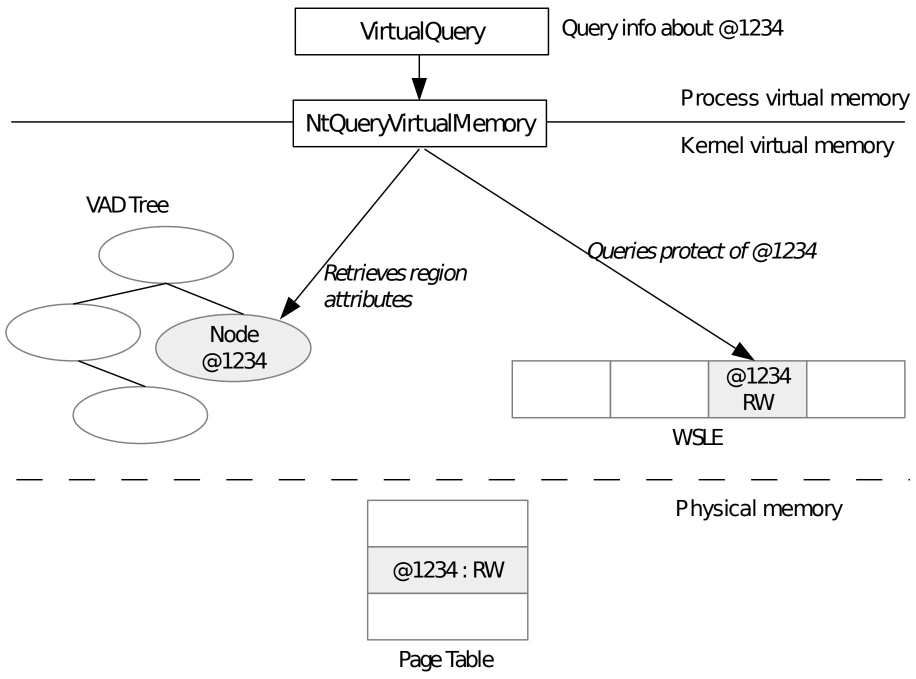
For each process on the system, Windows maintains a structure which represents its virtual address space. This structure is called the VAD (Virtual Address Descriptor), it is a binary tree where each node is a region, a range of contiguous memory pages, of the process address space.


Range : 0x410000 – 0x44FFFF
AllocationProtect: PAGE_EXECUTE_READWRITE
Type : MEM_IMAGE
Range : 0x12000 – 0x12FFF
AllocationProtect : PAGE_READWRITE
Type : MEM_PRIVATE
Range : 0x75000000 – 0x76FFFFFF
AllocationProtect: PAGE_EXECUTE_READWRITE
Type : MEM_IMAGE
Range : 0x30000 – 0x3FFFF
AllocationProtect : PAGE_READWRITE
Type : MEM_PRIVATE

The kernel parses this memory structure very often, almost on every operation on processes' address space. Every process on the system has its own VAD, storing properties (like *Type*, size, copy-on-write, etc.) of all the memory regions. It also implicitely tells whether a given address is free or not : if no region contains a given virtual address, it means that it is free.

The *Protect* flags of virtual memory pages are stored either in the WSLE (working set list entry), for memory with valid PTEs, or in PTE itself in case of virtual address with invalid PTE.

To better understand how those pieces are all put together, let us look at how the *NtQueryVirtualMemory* system call works. It is used for example when a processes wants to retrieve the MEMORY_BASIC_INFORMATION on a committed virtual address:



When a process queries information about its memory space, the *NtQueryVirtualMemory* system call is reached. It first parses the VAD tree to see if the provided virtual address belongs to a memory region. If it does not it means that is address is free. If it does, it will retrieve the protection information of this memory address in the WLSE array. What's important to notice is this syscall never looks at the protection bits of the corresponding PTE in physical memory.

2.4 System call input sanitization

Userland parameters provided to system calls are sanitized using an exception based mechanism. This is the reason why we decribe it a bit, as we will need to take this into account when designing our unpacker. Every system call is reponsible for sanitizing the inputs it takes. This sanitization is done by two functions: *ProbeForRead* and *ProbeForWrite*. *ProbeForWrite* basically actually writes the whole buffer to ensure it is writable, its pseudo code is very close to this:

```
NTSTATUS ProbeForWrite(BYTE * outbuffer, SIZE_T out_size)
{
        SIZE_T i;
        BYTE c;

        for (i=0; i<out_size;i++)
        {
                c = outbuffer[i];
                outbuffer[i] = c;
        }
}
```

If a buffer provided to *ProbeForRead* or *ProbeForWrite* is not readable or writable, respectively, a kernelmode exception is raised. This exception is caught by the calling system call which immediately returns to userland with an error code.

```
NTSTATUS dummy_syscall(void * outbuffer, size_t out_size)
{
        try
        {
                ProbeForWrite(outbuffer,out_size);
        }
        __except( EXCEPTION_EXECUTE_HANDLER )
        {
                return GetExceptionCode();
        }

        //syscall stuff...
        return result;
}
```

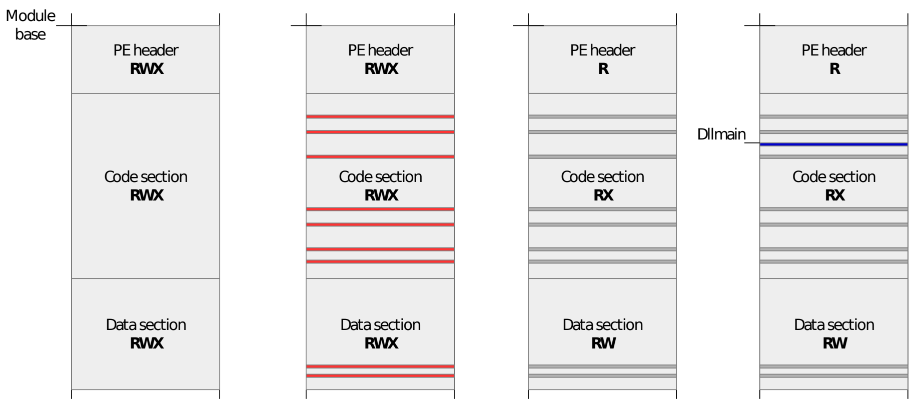
We will see in the next chapter why this has an impact on our design choices.

2.5 Userland image loader

The Windows library loader is called when a process needs to load a dynamic link library (DLL). This can happen before the original OEP for static dependencies of after the OEP for dynamic dependencies. In either case, the dll loader is called from userland. It is implemented inside *ntdll.dll*. When a process needs to load a DLL, the loader does the following things

1. ensure the DLL is not already loaded
2. map the DLL in memory
3. load the DLL's dependencies
4. patch the DLL's relocations
5. call the DLL entrypoint, *dllmain*

The following schema shows the different steps, from left to right:



First the loader maps the dll in memory. It then sets all PE sections memory protection to read/write/executable and patches all the relocations if the dll is subject to address space layout randomization. Once relocations have been patched, it sets the PE section memory protection to the appropriate value and finally calls *dllmain*.

The loader uses two internal memory structures:

- *RTL_CRITICAL_SECTION*: a lock for thread synchronization
- *LDR_DATA_TABLE_ENTRY*: a linked list to maintain the list of currently loaded modules

This is very interesting for us because, at any time, we can determine:

- If the loader is active of not (currently loading a DLL)
- Which thread of the process is loading the DLL
- Which DLL is begin loaded
- If the currently loaded DLL is a dependency of another DLL being loaded (recursion)

In the next section we'll see how this can be used to improve unpacking rate.

3 Unpacker architecture

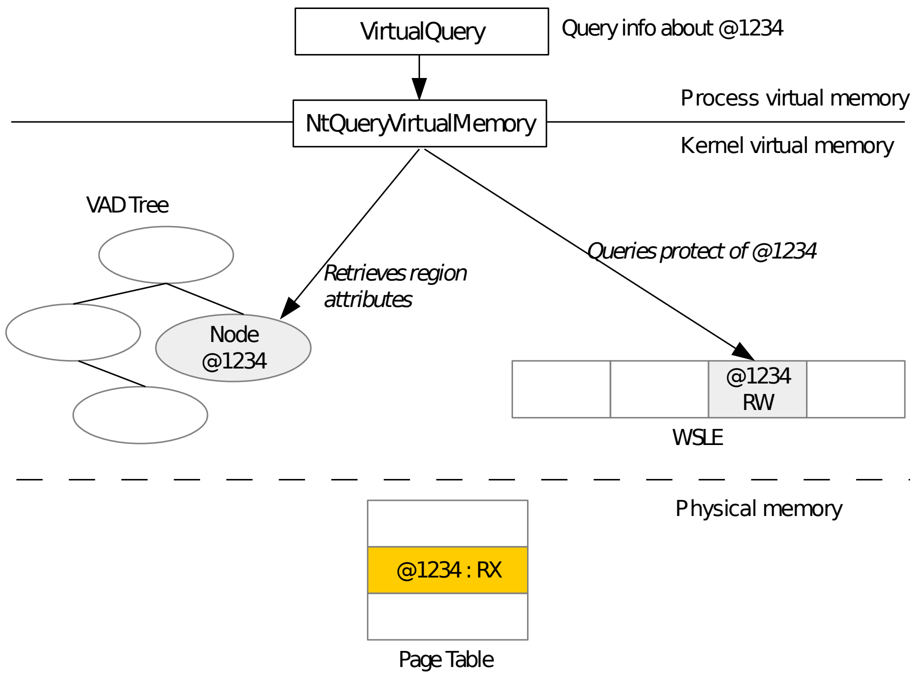
One of our constraint is to build a tool that can run inside a virtualized windows without being dependent to any particular virtualization technology. That is why we decided to build a tool running inside the windows environnement. Of course it has drawbacks but to us it is the best tradeoff. Our tool has two components : a kernel driver and a userland process in python.

Contrary to other generic unpackers like PackerAttacker[3], MutantX-S[10] or Justin[4] we decided to catch exceptions in the kernel rather than doing it in the vectored exception handlers of the process (in userland). Although using userland exception handlers is way easier to implement, there are good reasons not to use them. The first reason is: the unpacked progress can detect it has suspicious handlers regitered, unwanted foreign code running in its virtual address space. Another and most important reason : there are real world situations where such implementation leads to unpacked program failure.

A good example of such unpacking failure is the use of system calls by the packer code. While the target process runs inside our instrumented environment its virtual memory protection are changed. If the target process performs a system call and provides to the kernel an address which is meant to be writable but which is not (due to our instrumentation), the system call fails and so does the unpacking code. The only way to address such issues is to catch exception in kernel land.

3.1 Memory manager and PTE un-synchronization

In the previous chapter we described briefly the way the memory manager works and how the *NtQueryVirtualMemory* system call works. Recall that one of our main constraints is to be as stealthy as possible. Which means that, at all times, the monitored process must not be able to notice that is is being unpacked. This obviously means that when a monitored process call *NtQueryVirtualMemory* system call it must get a result coherent with what the process address space would be if the process was not monitored. Given the complexity of the structures used by the memory manager (VAD, etc.), maintaining a *packer view* of the process would be a very complex work. Instead we decided to leave all the structures of the memory manager untouched. We are creating an un-synchronization between the internal structures of the memory manager and the actual PTE entries. Changing the protection of a memory page in the PTE from RW to RX is as easy as changing only two bits in the PTE entries (see 2.1) ! As it is a low level modification, provided that we flush the TLB cache, further access to the virtual address associated with this PTE will trigger a fault that we can catch. Moreover, it is nearly impossible for a monitored process to detect it bacause even memory system calls like *NtQueryVirtualMemory* do not examine the content of the PTE. As we can see on the following scheme :



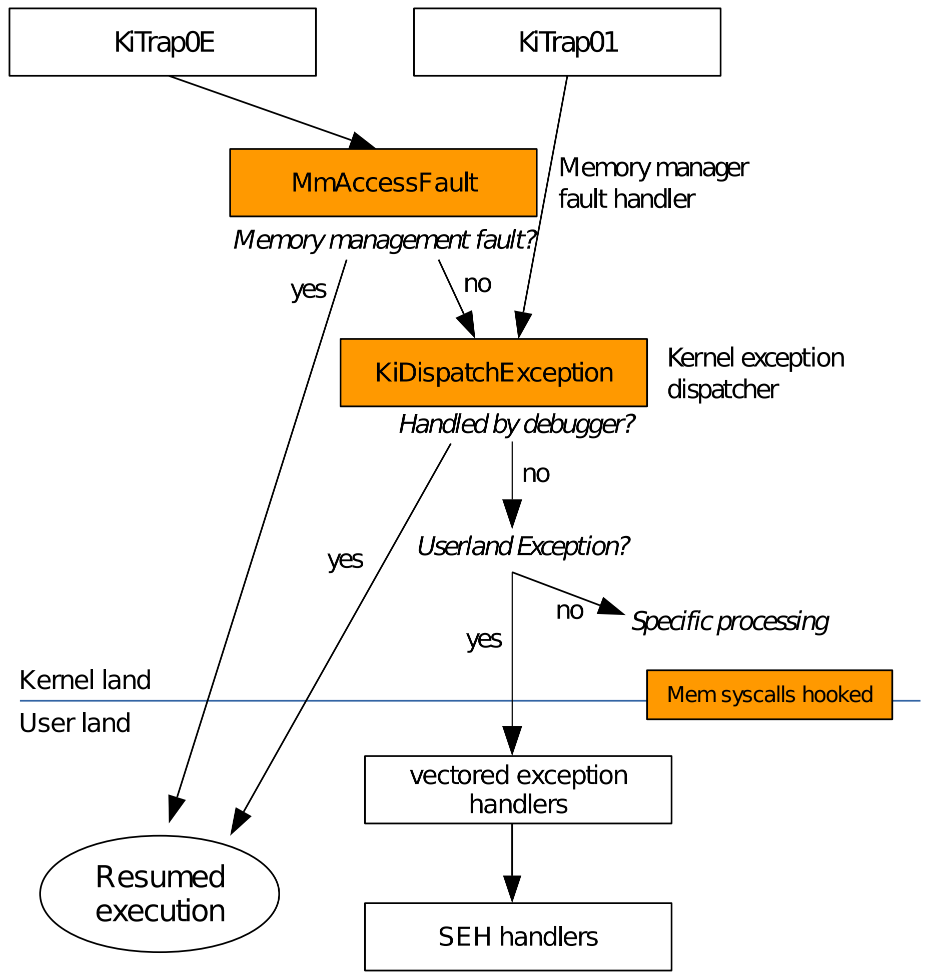
However, the memory manager re-synchronize PTEs and its internal structures on specific events :

- The page fault handler, MmAccessFault, is called.
- Memory manager structures are updated by a memory system call (NtProtectVirtualMemory for example)

To avoid untimely PTE updates we have to catch exceptions early in the exception chain : before the memory manager does.

3.2 Exception chain hooking

When the memory manager fault handler is called, low level PTE's and higher level kernel memory manager structures get re-synchronized. This is something we don't want because we then might loose track of packer activity. In order to avoid such re-synchronization we need to catch exceptions before the memory manager is called. We have to catch exceptions very early on the exception chain, the following scheme shows where we set up our hooks (in orange) :



Two kernel functions are hooked :

- *MmAccessFault* : The memory manager fault handler. This way we can catch execption before the real *MmAccessFault* get executed.
- *KiDispatchExcpetion* : The kernel exception dispatcher, in order to be on the int01 path. This way we can single-step both the kernel and userland processes.

3.3 System call hooking

System calls related to virtual memory management have to be filtered to avoid the malware from modifying its own memory. In order to do this, we hook the following kernel system calls :

- *NtAllocateVirtualMemory*
- *NtMapViewOfSection*
- *NtProtectVirtualMemory*

We are currently hooking only system calls related to allocation or protection of memory regions. It could be usefull to hook more system calls but this list is sufficient to get interesting results.

To hook allocation related system calls like *NtAllocateVirtualMemory* or *NtMapViewOfSection* the pseudo-code is simple :

```
NTSTATUS syscall_hook(void * ptr, size_t size)
{
  result = syscall(ptr, size);
  if (result == STATUS_SUCCESS)
  {
    for page in pages(ptr, size)
    {
      set_low_level_protection(ptr, EXECUTE_NO_WRITE);
    }
  }

  return result;
}
```

The memory manager will set its internal kernel structures to the appropriate values, thus enforcing the *Packer view* of the memory. We just have to change the memory allocation of newly created pages at physical level.

The hook of *NtProtectVirtualMemory* is a bit more complicated. We also have to let the original system call execute to let it change the memory manager internal structures accordingly. While this system call executes it re-synchronizes the memory manager structures and the low level PTEs, but this is something we don't want. The only way to do it well is parsing the low level PTE prior to orginal system call execution, to memorize PTE protection values and re-apply these values after system call execution. In this case the pseudo-code is a bit more complicated :

```
NTSTATUS syscall_hook(void * ptr, size_t size)
{
  //get low level protection for each page
  for page in pages(ptr, size)
  {
    mem[page] = get_low_level_protection(ptr)
  }

  //system call re-synchronize memory views
  result = syscall(ptr, size);
  if (result != STATUS_SUCCESS)
    return result;

  //put back the protection
  for page in pages(ptr, size)
  {
    set_low_level_protection(ptr,mem[page]);
  }

  return result;
}
```

3.4 Fine tune unpacking algorithm

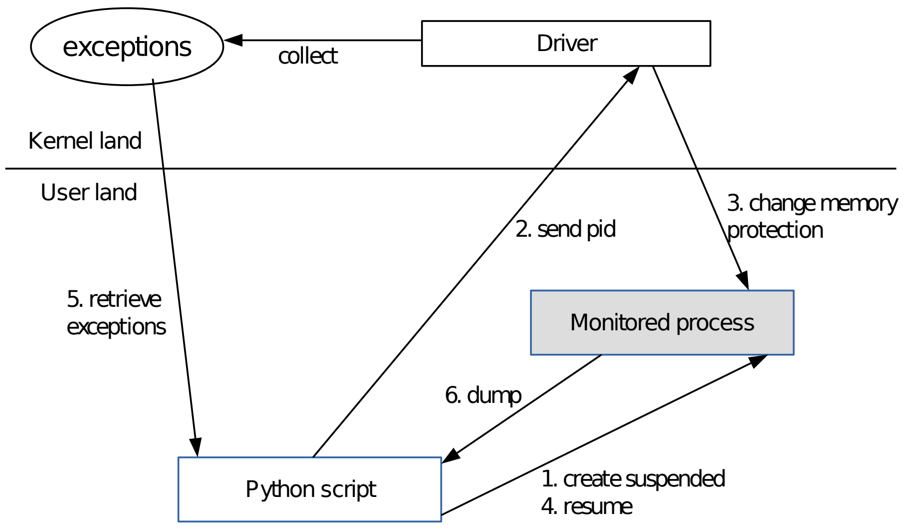
In the previous section 2.5 we described how the userland loader works. During our experiments we realised that the loader behaviour was disturbing the unpacking algorithm. Indeed, the algorithm described in section 1.3 is very sensitive to dynamic code generation. The way the loader maps a DLL, patches relocations and then executes code in the DLL (dllmain) misleads the unpacking algorithm which interprets this as dynamic code generation and part of the unpacking process. If a packed program loads a library dynamically after its OEP, the unpacking algorithm then computes an invalid OEP and so unpacking fails.

To work around this issue we decided to log the loader state on every exception in order to determine if the exception is generated by the loading of a library or not. This way we were able to significantly improve the unpacking rate.

Although the loader artifact can be an inconvienient, it can also be an opportunity. Indeed, if instead of filtering out exceptions induced by the loader we keep only those exceptions, we are able to build an unpacker for DLLs.

3.5 Overall architecture

The following scheme shows the overall architecture of the tool :



Our tool, *Gunpack* has two components : a driver which is responsible for hooking the kernel and logging exceptions, and a userland component responsible for launching and dumping the target process.

The unpacking process is performed in various steps :

- **step 1** : The userland component creates an instance of the target process in suspended state
- **step 2** : Pid of the target process is sent to the driver.
- **step 3** : The driver then parses the target process memory to change its memory protection in the PTEs. It also ensures that the whole process

memory is present in physical memory by triggering copy-on-write and on-demand access.

- **step 4** : Userland component resumes the target process which starts its execution. Along its execution the driver catches exceptions, logs them and performs necessary actions such as page protection flipping at low level.
- **step 5** : Once execution is over, the userland component computes the OEP
- **step 6** : Finally the target process is dumped

To generate a clean dump of the process we use a third party library call Scylla [1]. We also use this library to rebuild the process imports if it is possible as a best effort approach.

4 Results

So far our tool is still under development and it has not been tested enough, or at least not as much as we would like. We have tested it successfully against homemade packed samples. Just to give an quick evaluation of the tool's efficiency we also performed a small campaign against still popular COTS packers (see [2]).

Building a good set of packed sample is a very complicated task. Indeed, there is a large numbers of packers, each of them has many versions and they provide various options (compression, anti-debug, etc.). So we used two set:

- A set of non-malicious programs packed with various packers. We put in this set packed samples from the *Tuz4you* website as long as packed samples we generated with the actual packer.
- Use a set of *real life* malwares from *VirusTotal* sorted by packer.

In either case we had to unpack all of them manually to ensure that the OEP of the output binary is the right one. This explains why our set is not very large. Even though it is quite small it gives a good overview of our tool's efficiency. Our criteria of successfull unpack is very restrictive. Samples for which the whole original code is decompressed/decrypted but which entry point is not a the exact OEP are considered unpacking failures.

As we are trying to rebuild imports to generate a working PE for best effort approach, we distinguish two results:

- Unpacked successfully : the program was unpacked and the generated PE entrypoin points to the original program entry point (OEP). The generated PE can be used for static analysis as long as classification.
- Working PE : unpacked successfully and imports rebuild successfully. The unpacked program should run exactly as the original program. Can be used for dynamic analysis.

Here are the results of our first test set:

Packer	Unpacked successfully	Working PE
UPX (3.91)	Yes	Yes
MPRESS (2.19)	Yes	No
PeCompact (2.X)	Yes	Yes/No
NsPack (2.4 to 3.7)	Yes	Yes
Aspack (2.2)	Yes	Yes
Asprotect	Yes	No
Armadillo	No	No
VMProtect	No	No

As we can see our tool works well on simple and very popular COTS packers. It fails on more complex packers that use virtualization but this was expected. Our second test set was composed of 20 distinct samples of packed malwares from virus total. Here are the results:

Packer	Valid PE	Valid OEP found	Unpacked PE runs
UPX	13	12 (˜90%)	2(˜15%)
Aspack	12	9 (˜75%)	3(˜25%)
NSpack	15	9 (˜60%)	5(˜30%)
PeCompact	14	10 (˜91%)	4(˜29%)
Upack	15	13 (˜86%)	4 (˜26%)
fsg	10	7 (˜70%)	2(˜20%)
exe32pack	6	4 (˜66%)	0(˜0%)

We can see that unpacking rate is not 100% even against packers that were successfully unpacked in the first test set. Even though it can seem surprising, it is due to properties of the unpacking algorithm which is very sensitive to dynamic code generation. The results of unpacking depends also on what the packed program does, for example if it generates code at runtime, unpacking will fail.

In order to address this issue we could restrict the scope of OEP search by applying heuristics to determine when the program finishes unpacking, like other tools do. This is something we will implement in the next version of our tool.

References

[1] NtQuery bla. *Scylla.* https://github.com/NtQuery/Scylla.

[2] Xabier Ugarte-Pedrero , Davide Balzarotti , Igor Santos , Pablo G. Bringas. *SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers.* IEEE Symposium on Security and Privacy, 2015.

[3] Nick Cano. *Packer Attacker.* https://github.com/BromiumLabs/PackerAttacker, 2015.

[4] Fanglu Guo , Peter Ferrie , Kent Griffin , Tzi cker Chiueh. *A Study of the Packer Problem and Its Solutions.* RAID, 2008.

[5] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual.* http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[6] ReactOS developpers. *ReactOS project.* https://www.reactos.org/.

[7] Mark E. Russinovich , David A. Solomon , Alex Ionescu. *Windows Internals Book 6th edition.* Microsoft Press, 2012.

[8] Lorenzo Martignoni , Mihai Christodorescu , Somesh Jha. *OmniUnpack: Fast, Generic, and Safe Unpacking of Malware.* Computer Security Applications Conference, 2007.

[9] Artem Dinaburg , Paul Royal , Monirul Sharif , Wenke Lee. *Ether: Malware Analysis via Hardware Virtualization Extensions.* CCS, 2008.

[10] Xin Hu , Sandeep Bhatkar , Kent Griffin , Kang G. Shin. *MutantX-S: Scalable Malware Clustering Based on Static Features.* USENIX Annual Technical Conference, 2013.

[11] Min Gyung Kang , Pongsin Poosankam, Heng Yin. *Renovo: A Hidden Code Extractor for Packed Executables.* Workshop on Recurring Malcode, 2007.