The CodeBreakers-Journal, Vol. 1, No. 2 (2004)
http://www.CodeBreakers-Journal.com

Unpacking by Code Injection

Author: E. Labir

Abstract

In this paper, we show how to gain insight information for a given target through code injection. Our attacks are totally stealth for most current anti-cracking technology and represent a real-life threat, the most relevant information we can retrieve is the following:

- *List of exceptions, handlers and related information.*
- *List of API calls to all* `DLLs` *(parameters, return codes,...).*
- *Full reconstruction of the Imports Table.*
- *Entry Point.*

Our methods are flexible and not difficult to implement, we outline the source code and provide a real-life example of how to analyse the log files.

Keywords: *Unpacking; Code Injection; defeating anti-cracking technology*

I. Introduction

Debugging a target can range from the extremely simple to the impossible. Well protected software uses to be rid of anti-debug tricks - mainly seh-based - and extremely obfuscated code, this all makes going through it an endless nightmare.

In Windows (Win9x onwards), a process lives inside its own address space. The address space is flat and contains all (mapped) `DLLs`, resources and stuff it needs. Windows also provides us the tools we need for injecting our own code and running it inside another process, [4]. In this article, we will use injected code to sniff lots of information from the target, *our methods bypass API redirection and all the standard anti-debug*.

The information we gain from injecting our code includes all calls (parameters and return codes too) to all `DLLs` mapped into the target's address space - we can also modify them. In particular, this provides the list of exceptions and their associated information (handlers, addresses, codes,...). With our system, we can easily retrieve the entry point (injecting a tracer) and, in some cases, even the stolen bytes. Even when the method fails, all these problems become very simplified.

The method needs ”some” human intervention, for analysing the log files, but saves up huge amounts of work. Detecting you are under this attack is not easy and it shall need redesign for many packers.

The subjects we deal with are:

1) (Section 1) Description of a packer.
2) (Section 2) How to inject and run our code in the target.
3) (Section 3) Hooking all API calls.
4) (Section 4) Interacting with the API calls (logging, `IAT` reconstruction).
5) (Section 5) Finding the Entry Point with an injected tracer.
6) (Section 6) How stealth is injected code?.
7) (Section 7) Conclusions, further research,...

Each section has code snippets (or pseudocode) and we also present examples as they appeared in our experiments.

II. Description of a packer.

A packer is a program aimed to prevent somebody else from examining how our program works or from modifying it. Typically, the entry point of the protected program is diverted to run the packer first. When it runs, the packer will carry out the following (main) steps:

- Decrypt the sections of the target (target = packed program).
- Rebuild its imports table.
- Jump to the actual entry point of the target.

The first step, sections decryption, isn't important for us. Normally, it doesn't use to be much of a problem to await until they are decrypted in memory and simply dump them to a file and glue them up together. However, points 2 and 3 are crucial.

The imports protection can be done with several degrees of sophistication, some of the implementations (in real-life packers) are quite weak and do not need at all our methods to be broken. We will always suppose that a strong protection has been applied to the IAT, meaning:

1) The Imports Table has been removed, the packer saves only (in a secure place) the hashes of the API names and their addresses at the IAT.
2) The algorithm is well obfuscated and has lots of anti-debug, anti-trace...
3) The packer doesn't use GetProcAddress. Instead, it implements its own algorithm to find the APIs at the exports table of the DLLs.
4) The IAT has been redirected (read below what this means).

Note: actually, point 3 doesn't affect us. However, a good protection should satisfy it.

It's amazing to see how many ”commercial” products don't comply neither with (1) nor with (3).

What about 4 (API redirection)?. Let's try to see what ”API redirection” means: Open any application, it's sure that it imports `kernel32.ExitProcess`. Now, look for a call to it, you will find something like the following:

```
call [XXXXXXXXh]     ; call to kernel32.ExitProcess
                     ; XXXXXXXXh inside the IAT
...                  ;
XXXXXXXXh: YYYYYYYYh ; address of
                     ; Kernel32.ExitProcess
```

The other most common possibility is to have a `call XXXXXXXXh` instead, the argument would be the same. The Windows loader ”sees” the imports table and fills the `IAT` with the addresses of the imported APIs. Packers destroy the information at the imports table of the protected program, therefore the `IAT` will have incorrect values when the packer yields control to the target. Then, the consequence is that the packer needs to fill the `IAT` of the target with the right values.

If you take a packed program, under a packer satisfying (4) - examples: Asprotect, Slovak Protector,...- the `IAT` looks like:

```
XXXXXXXXh: ZZZZZZZZh  ; ZZZZZZZZh is inside a
                      ; buffer dynamically
                      ; allocated by
                      ; the packer, so it will
                      ; not exist if you
                      ; remove the packer.

ZZZZZZZZh:  push ebp       ; (*)
        ror eax, 16h
        pushf
        popf
        mov ebp, esp        ; (*)
        call @@1
        db 68h
    @@1:
        add eax, 134h
        ....
        jmp ACTUAL_ENTRY_POINT_OF_API + k
```

The packer mixes the first instructions of the API with some garbage code (in the example, those instructions having a ”*” beside were in the original API code), it can include some trivial anti-debug trick too, and finally writes a jump (or something similar) to the entry point of the `API + k`, where `k` is the number of original instructions already run among the garbage code. This way, it is essentially impossible to find the actual API called through `ZZZZZZZZh` or to hook it.

The imports table is not a straightforward structure, going into details is out of the scope of this article, [2].

Some programs, called import rebuilders, are able to emulate a few instructions to search for the address in the `DLL` we jump to. Then, you can retrieve the API name from the exports table. However, import rebuilders are usually very limited in what they can sniff and will not overcome the latest packers (which, obviously, are tested against them before their release).

Let’s see what’s the matter with the entry point. As we commented, the packer - once it has done its job (decrypted the target, reconstruct the IAT,...)- has to give control to the protected application. This can be done running it as a new thread, with `kernel32.CreateThread`, or simply jumping to it in some exotic way. Running it as a new thread is not a good idea, the starting address of the thread will be too evident and so, we (again) suppose the worst case: The packer jumps to the entry point after a long time of well obfuscated and protected code, the jump is well hidden (self-modifying code, etc...).

Another problem one usually has to deal with is ”stolen bytes”: The packer takes a pre-defined set of `APIs`, a very common one is `GetModuleHandleA`, and does the following:

- Calls GetModuleHandleA and stores the return.
- Looks inside the protected app for patterns like:

```
call XXXXXXXXh
      ; call to
      ; kernel32.GetModuleHandleA
mov [handle], eax
```

Now, it removes the call and, on every startup, substitutes the `mov [handle], eax` by a simple `mov [handle], harcoded_value`, where `hardcoded_value` was returned before by the packer’s call to `GetModuleHandleA`.

If the cracker doesn’t detect this trick then the unpacked program will not run on all OS versions or will have another defect. These bytes the packer removes are called ”stolen bytes”. Stolen bytes of this kind, replacing a call by a hardcoded value, will also be easy to locate and reconstruct with our methods.

A full description of a packer is well out the scope of this article, please refer to [1] for a detailed explanation.

Packers are the preferred way to protect middle-price applications, virtually all shareware depends on their security. Thus, studying their advantages and disadvantages is an important field in **RCE** (Reverse Code Engineering).

III. Injecting and running your code

In this section, we outline how to inject and run our code into the target’s address space, see [4] for a full description. For Win9x see [3].

Our project will be divided into two parts, as in [4]. We will also have a carrier application, in charge of launching the target and injecting the code. We will call the injected code ”logger”, since this describes very accurately what it does.

The carrier needs to inject and run our logger before the target has a chance to start. Therefore, we need to create the target as `CREATE_SUSPENDED`. This way, the address space of the target will be initialised but the main thread will not be run until we call ResumeThread.

Thus, the carrier creates the target as `CREATE_SUSPENDED` and injects the logger inside its address space. Next, the carrier runs the injected code with `CreateRemoteThread` (Win2k onwards). The logger needs to do some preliminary work before the target is allowed to run. When everything is ready, the carrier runs the main thread of the target. Therefore, carrier and logger need to communicate in some way, the one we choose is through an event, which the logger sets to `TRUE` when the (packed) target can run.

Let’s outline how to carry out all the previous steps (see [win32.hlp] for a detailed reference on the `APIs` below):

```
; first, we create the event (choose a random name for your event)

      push offset zsEventName   ; name of event
      push FALSE                ; initial status = FALSE
      ...
      call CreateEventA

; Now, we create the target as CREATE_SUSPENDED.
; The call returns a handle to the created process we need for later.

    ...
    push CREATE_SUSPENDED
    ...
    push offset zsTarget
    call CreateProcessA

; The address space of the process has been initialised, but its
; primary thread is suspended. Now, we allocate some memory into
; the target to host our code.

    push PAGE_EXECUTE_READWRITE ; attributes for the allocated memory
    ...
    call VirtualAllocEx

; The return is the image base of the allocated memory. Finally, we can write to it, with
; kernel32.WriteProcessMemory, and run our code with kernel32.CreateRemoteThread.

    ...
    call WriteProcessMemory
    ...
    call CreateRemoteThread

; At this point, the logger is running while the main thread is suspended.
; The logger will change the status of the event
; we have created at some moment, we need to wait until then before to
; resume the main thread:

    push -1                 ; wait infinite time
    push dword ptr [hEvent] ; handle to the event, returned by CreateEventA
    call WaitForSingleObject

    ...
    call ResumeThread
    push 0
    call ExitProcess
```

As you see, injecting your code inside another process and running it is not too difficult. Note **the target is not being debugged**, therefore it can't complain about it.

IV. Hooking API calls

A. The problem and its relevance

As we briefly reviewed above, the packer will emulate the first k instructions of the `API` and will jump to the `(k+1)-th`. This k first instructions can be metamorphosed, for example, for `k=2` we might have:

```
; not changed entry point of
; kernel32.ExitProcess inside
; KERNEL32.DLL.

77E55CB5 kernel32.ExitProcess       push ebp
77E55CB6                            mov ebp,esp
77E55CB8                            push -1
 ; example of instructions you could
 ; find at the buffer:

            xchg eax, esp
            sub eax, 4
            jmp @@1
            db 68h
        @@1:xchg eax, esp
            mov dword ptr [esp], ebp
            push esp
            pop ebp
            push (77E55CB8+RANDOM_VALUE)
            sub dword ptr [esp], RANDOM_VALUE
            ret
```

They both do the same, however the second one is not easy to follow. The longer and more difficult to emulate is the buffer the harder is to reconstruct the imports table.

Now, observe that if you set a breakpoint on the entry point of `kernel32.ExitProcess` this will be easily bypassed by the packer. Of course, one can set this breakpoint later but then knowing the call parameters can be difficult (or impossible). Apart, your breakpoint can be fired up when is reached from another DLL-internal call.

Having the possibility of setting breakpoints on (suitable) places of the `APIs` is an important problem, control over their behaviour yields immediate control over the packer.

B. The approach

Let's suppose the `API` entry point of `kernel32.ExitProcess` was like this:

```
nop     ; 1
nop     ; 2
...     ; ...
nop     ; k
...     ; ...
nop     ;
jmp kernel32.ExitProcess_ActualStart
```

The packer would emulate the first k `nops` and would jump to the k+1. So, we can safely set our breakpoint at `jmp kernel32.ExitProcess_ActualStart`. Indeed, the original parameters have been preserved.

What we do, for a given `DLL` (say `kernel32`), is the following:

1) Get the image base of kernel32.
2) Take is `SizeOfImage` from the `PE-header`.
3) Change permissions over the whole image of kernel32 to `PAGE_EXECUTE_WRITECOPY`
4) Save the original Entry Points for all exports in the DLL, they can be found at `IMAGE_EXPORT_DIRECTORY.ED_AddressOfFunctions`.
5) `N` = number of APIs exported by `kernel32`, `IMAGE_EXPORT_DIRECTORY.ED_NumberOfFunctions`.
6) Divert each one of the APIs of `kernel32` to our buffer.
7) Restore permissions over the DLL (some packers use the DLLs to provoke exceptions writing to them).

Let's see how to redirect all APIs to our buffers in detail, note this is done before the packer runs.

Let's call the buffer we use DivBuffer. This buffer would have size `N*M`, where `M` is the maximum size of our buffers, what we do is:

```
for (i=0; i<N; ++i){
  Change the entry point
  of the i-th API to DivBuffer[i*M].
  Generate random garbage
  and write it to DivBuffer[i*M].
  Write some instructions, after the
  garbage, to save the value of i.
  Write a jump, after the previous
  instructions, to our hooker procedure.
}
```

In the algorithm, M is just a upper bound of our garbage size (`M = 30` is a good value for current packers). All garbage buffers we have created will lead to our hooker procedure, which is in charge of logging the calls and stuff. The hooker needs to be aware of the address we have come from, therefore is important to save the value of the index `i` before jumping to it.

Let's outline the hooker, it's quite a simple procedure: it saves registers, writes everything to a log file and then restores registers and jumps to the original API entry point (which we have saved). A hooker looks like the following:

```
hooker PROC

    mov eax, esp    ; save esp and ebp
    mov ecx, ebp
    pushad
     ...

; write the log file
;(API the packer called, parameters,...)

        ...

; compute the actual entry point
; of the API the packer wanted
; to call and overwrite the dword
; below, @ActualEP,  with it.

        ...

; restore registers

        popad

; this jumps to the actual entry
; point of the API (is a push/ret)

       db 68h
       ; first opcode of push XXXXXXXXh
    @ActualEP:  db 0,0,0,0
            ret

hooker ENDP
```

Observe that, at the very beginning, we have lost the value of both `eax` and `ecx`. It doesn't mind, `eax` and `ecx` are the "trash" registers for Windows, meaning they are not used by the `APIs`. Therefore, it's safe to use them internally (the rest of registers need to be preserved), this saves us some headaches because we can store there esp and `ebp`. Needless to say, the garbage code does need to preserve all registers but `eax` and `ecx`.

C. What happens when the packer calls an API?

The packer thinks it does the following:

1) Locate the image base of the API.
2) Look for it by name, normally comparing a hard-coded hash value of the name with the hashes of each API until it matches.
3) Emulate the k first instructions of the API (k can be randomly chosen each time).
4) Jump to the (k+1)-th API instruction.

It actually does:

1) Locate the image base of the API.
2) Look for it by name, normally comparing....
3) Emulate the k first instructions of the i-th garbage buffer (supposing it calls the i-th API) .
4) Jump to the (k+1)-th garbage instruction.

Finally, we can hook the API call at the entry point of our procedure "logger".

The packer doesn't see any difference but we have kept the call as if it was done without the protection.

D. Implementation hints

The following advises might help to easy up into coding your own logger:

- kernel32, user32 and advapi32 are loaded at the same image base in all processes running in the same OS. Therefore, the logger can inherit (virtually all) APIs it needs from the carrier.
- For testing your programs: there is no need starting by injecting your code in some other process. Before so, it's recommendable to allocate a buffer inside the carrier and to copy there the logger and play with them. This way you can do a minimum testing.
- When you finally inject your code in another process, you can set a breakpoint at the very beginning of the logger. This way, when you run it with CreateRemoteThread, it will crash and you will be given a chance to attach your debugger. In general, you can set an `int3` at any point of the logger you want to check.

- For the procedure writting to the log file:
  - Write the names of the APIs the target calls.
  - Do a small procedure passing from hexadecimal to a printable string (remember that offsets are given in endian order).
  - Use a shared file mapping for your log file, so you can see and save it at any moment without dumping from memory, this will let you to control execution.
  - Remember that file mappings can't be enlarged in memory, use a big one (1Mb).
- We recommend (as non-copyrighted target) Yoda's Crypter for the experiments, it redirects API calls and has a few exceptions/tricks you will have to bypass.

V. Logging and IAT reconstruction

After having seen how to hook all calls done by the packer (or the target) we now move on how to react to these calls, we deal with this subjects:

1) Logging API calls.
2) Altering their parameters or return values.
3) Provoking calls to get the (plaintext) API addresses.

The packer we used for this experiment is one of the hardest in the market. Of course, we will not reveal any information that could help into breaking it (all offsets and stuff has been altered).

Let's review, step by step, how to do it in real life:

A. Logging kernel32

The first you should always do is to log all calls directed to `kernel32`, only rarely a packer will call APIs from other DLLs (excluding `ADVAPI32`, which comes next).

In the log, you should have the API and where it was called from (the return is at `dword ptr [esp]`). Note that your API can be called from inside another APIs from the same (or other) DLL, so you should filter this calls in some way.

We did it as follows:

```
    mov ebx, dword ptr [esp]
    ; take return address
    shr ebx, 28
    ; keep only the most significative byte
    test ebx, ebx
    jnz Dont_Log_Me
```

This works because in WinNT `kernel32` always loads at `77E40000h`. There are much better ways of filtering them, matching against the loaded modules at the `PEB` is possibly the best, but this one works almost sure and is pretty simple. Let's see a log from the protected Notepad:

```
Logger started for DLL  KERNEL32.DLL,
target = Notepad.exe

VirtualAlloc From: 00B10024
   Param: Buffer size: 00000200 API return: 00A70000
VirtualAlloc From: 00B20101
   Param: Buffer size: 00001000 API return: 00A80000
LoadLibraryA From: 00A20BFE
   Param: ADVAPI32.DLL  <=== interesting!
VirtualFree  From: 00A2310B ...
GetLocalTime From: 00A45150  <=== interesting!
 ...
VirtualFree From: 00D01711 VirtualFree From:00D02224
```

Now, you have to read the APIs it has called and to pay attention on the most relevant ones. We've isolated `kernel32.GetLocalTime` because is typically used for 30-day time trials and the like. It's also interesting to note that the packer has loaded `ADVAPI32`, therefore you should now log all calls from `ADVAPI32`.

Just reading the logs will give you a precise idea of how it works.

B. Logging the parameters of some selected APIs

It would be totally pointless to start coding lots of assembler lines to decode the parameters of hundreds of APIs. Instead, we just choose a small subset of the called ones and handle them.

The parameters can be found at `esp+4`, `esp+8`,... therefore you only need to write them to the logger (beware in case they are pointers, `NULL` can crash your logger).

C. Modifying the result of the API calls

In this example the API we have chosen is `kernel32.GetLocalTime`. We want to hook it and to change its return to another value, a fixed date (so we are always registered). To do so, we need to know the value of the parameter `lpSystemTime` before to do the call, see [win32hlp], and to modify the structure it points to right after the call. Thus, the hooker, apart from writing the log, needs to save `lpSystemTime`. The following code shows how to carry out the complete process:

```
;---------------------------------------------------------------------------------
;                 hooking the return address from the call
;---------------------------------------------------------------------------------

; first, we save the bytes at the return address because we are
; going to overwrite them with a jump to our code

    cld                         ; clear direction flag
    mov esi, dword ptr [esp]    ; take return
    mov edi, offset your_buffer ;
    mov ecx, 6                  ; the size of a push/ret
    repnz stosb                 ;

; next we overwrite them with a push/ret leading to our code

    mov edi, dword ptr [esp]    ; take return address
    mov al, 68h                 ; write the push
    stosb                       ;
    lea eax, [My_GetLocalTime]  ; write the address of my procedure
    stosd                       ;
    mov al, 0C3h                ; write the ret
    stosb                       ;

; we also need to keep track of lpSystemTime

    mov eax, dword ptr [esp+4]      ; store the parameter lpSystemTime
    mov dword ptr [ebp+lpSystemTime], eax
```

Now, we can let the target to complete the call to `kernel32.GetLocalTime`, because it has been hooked to our code. Note that the target will have `Read / Write` permissions over all its sections and so you don't need to worry about it. Now, we have to change the return:

```
;---------------------------------------------------------------------------
;               Changing the return to our fake value
;---------------------------------------------------------------------------

; The target jumps here when hooked:

My_GetLocalTime PROC

    pushad
    pushf     ; not needed in this case but you might have to add it too

; compute the delta handle in ebp

    call @@1
@@1:pop ebp
    sub ebp, @@1

; modify the returned structure
```

```
    mov eax, dword ptr [ebp+lpSystemTime]   ; point to the SYSTEMTIME structure
    mov [eax.wYear], 2004
    mov [eax.wMonth], 4
    mov [eax.wDayOfWeek], 1
    mov [eax.wDay], 8

; write back the bytes at the return instruction

    cld
    lea esi, [ebp+your_buffer]
    lea edi, [ebp+API_ReturnAddress]    ; the value we had at esp at the hooker
    mov ecx, 6
    repnz stosb

; restore registers and flags and return
    popf
    popad
    ret

My_GetLocalTime ENDP
```

This will make to believe the packer we are still at 2004, April the 8th. Of course, logging the `API` return is as simple as storing eax and all the structures the `API` has used so far.

D. Dealing with the rest of DLLs

In this example, we saw that the packer loads ADVAPI32.DLL (the only examination of kernel32 gave us all loaded DLLs) . ADVAPI32 is a pretty important DLL, it contains the APIs we need to access to the registry, where packers use to save part of their registration information. We have two possibilities:

1) Generalise our algorithms: hook LoadLibraryA and, each time is called, see which DLL has been loaded and hook all its APIs too.
2) Simply run our current algorithm for the new DLL.

There is not much benefit on the first approach. In fact, we never needed it in our experiments.

E. IAT single address sniffing

Observe the following call done from the packer:

```
LoadLibraryA From: 00BA08BC Param: COMDLG32.DLL
```

A packer will never use APIs from COMDLG32.DLL, therefore this has to be done to reconstruct the target's IAT. Let's hook all calls done to APIs from COMDLG32.DLL and see what happens:

```
Logger started for DLL  COMDLG32
End of log file
```

An empty log file?, Why?. We have simply run the target application but we have still to compel it to call some API from COMDLG32. Now, we run the target but we also choose ”choose font” from the menu (you have to play with the target until you can see something interesting in the logs, this is why we suggested to create a shared file mapping as log file):

```
Logger started for DLL  COMDLG32
  CommDlgExtendedError Return Address: 01002E39
End of log file
```

Ok, got it. Let's consider the three most common possibilities for the current linkers, these are to link the API by:

- call dword ptr [IAT_ENTRY]: where IAT_ENTRY is an absolute address inside the IAT.
- call RELATIVE_IAT_ENTRY: Here, the return address + RELATIVE_IAT_ENTRY is inside the IAT (adjust for negative references).

In our case, Notepad.exe, this was:

```
01002E33  call dword ptr [10012AC]
          ; call we have logged
01002E39  test eax,eax
          ; return in our log
```

To distinguish both cases just do this:

```
    mov eax, Return_Address
    sub eax, 6

    cmp byte ptr [eax], 0FFh
    je First_IAT_Case

    inc eax
    cmp byte ptr [eax], 0E8h
    je Second_IAT_Case
```

This let's us to easily get any API we want: the logger, before to jump to the *original API entry point*, goes to the target and tries to sniff the IAT address corresponding to it. The method can fail, for example (Notepad.exe):

```
01006AEF   mov edi,dword ptr
           [KERNEL32.GetModuleHandleA]
01006AF5   call edi
```

We can try the value of edi on return, perhaps it has been preserved, and output is as our ”guessed” IAT address. Otherwise, we can restrict to the known cases as we did above (which will be enough in most cases, as we will see below).

This is an extract from our experiments:

```
lstrcmpW return address: 01001C8D
                         IAT address? 010010F0
lstrcpyW return address: 01001C9F
                         IAT address? 010010EC
lstrcatW return address: 01001D4F
                         IAT address? 010010DC
lstrcpyW return address: 010040ED
lstrlenW return address: 010040F6
lstrcpyW return address: 01004102
```

As you see, `lstrcmpW` has been correctly located (the same holds for `lstrcatW`). The values not having a ”guess” for the `IAT` correspond to the, mentioned above, `call edi` case. For the same application, `user32.dll` logs almost yields the full `IAT`.

It’s important to note that one can save the bytes before the return of the call and try to dissamble them later (there aren’t so many cases). With this, we bet we could obtain essentially the full IAT in most cases. However, as we are going to see, there is a much better way.

F. IAT full reconstruction

In the previous section, we have seen that to get a single API we only need to log the right DLL and to ”play” with the target. Therefore, we could, one by one, add all APIs until we have a full working application, *this would be pretty time-consuming.*

Full reconstruction of the IAT is much simpler than adding imports one by one, let’s see how to do it:

1) First, we need to retrieve one API from each `DLL` the target uses (this has to be done playing with it, but this time we only want one from each DLL and this can be achieved in a few minutes).
2) The piece of the IAT for each `DLL` is a zero terminated array like this one:
   ```
   offset 0:   ?????????
         ; dd immediately before the IAT
   offset 1:   DLL1_API1
         ; first API imported from this DLL
   offset 2:   DLL1_API2
         ; second API imported from this DLL
   offset 3:   ...
         ;
   offset N:   0
         ; null terminating dd
   ```
   The `dword` at offset 0 has unknown contents, it can be the `NULL` terminating the previous part of the `IAT` (for another `DLL`) or simply garbage.

   Our problem is that we know one of offset1, ..., `offsetN` but we need to get them all. Indeed, we can’t even assume that this array will have a `NULL` terminating `dword`, because the packer can have set there any other value to confuse us.
3)

The following algorithm solves this problem:

```
; input: eax = guessed IAT address
; ouput: reconstruction of the
; imports table for the DLL to
; which eax belongs to

xor ecx, ecx                ; counter

while ([eax+4*ecx] != 0)
{
  push ecx                    ; save ecx
  push eax                    ; save eax
  call dword ptr [eax+4*ecx]  ; Compel the target
                              ; to do the call.
                              ; This sends us to
                              ; the buffer created
                              ; by the packer to
                              ; emulate the first
                              ; instructions of
                              ; the call.

  get the API at our logger   ;
  restore the stack           ;

  pop eax                     ; restore eax, ecx
  pop ecx                     ;
  inc ecx                     ; next
}
```

This retrieves all addresses from `eax` onwards, finally we only have to move backwards until the previous zero.

Of course, when we do call `eax` for the offset having the interrogation marks we will crash the program (most likely). Thus, we need to set a `seh handler` to protect the execution of this algorithm.

G. Breakpoints on API calls and attaching your debugger

Usually, you will want to examine yourself from some call onwards. With our method, setting a breakpoint on an API call is totally straightforward. Let’s see a couple of ways to do so:

When the return address from an API call is a given one:

The logger *can't* simply take the return address, which was at [esp], and compare to a saved value:

```
    mov eax, dword ptr [esp]
    cmp eax, RETURN_FROM_GETLOCALTIME
```

Why not?. Because the packer will be running on a dynamically allocated buffer and, so,the image base can vary, leading to a different return address. Instead, we can take the least significative bytes of the return address and see whether they match or not:

```
    mov eax, dword ptr [esp]
    mov ebx, RETURN_ADDRESS_FOR_GETLOCALTIME
    shl eax, 16
    shl ebx, 16
    cmp eax, ebx
    je my_breakpoint
```

At the n-th time we call a given API:

We have the API which has been called at the hooker procedure, just compare it to a hardcoded value and keep a counter of the number of times it has been called. Job done.

There are many options for the breakpoint itself, the one we use displays a message warning about it and enters in an infinite loop, `here: jmp here`. Now, you can attach your debugger, pause the execution, `NOP` the `jmp` out and debug. The savings are spectacular.

VI. Logging all the Exceptions

Getting the list of exceptions provoked by the packer can also help. In particular, it helps if we want to attach our debugger and use our tracer, because we know it will not be killed by any exception. The best tutorial for understanding exceptions is ”Exception for assembler programmers, by Jeremy Gordon”, a must-read. We assume a minimum background on the subject. As is widely known, hooking `NTDLL.ZwContinue` will give us many exceptions, but not all the information we need. The problem becomes when we have `unwindings` or when a handler refuse to repair the exception and passes it to the following one. In this case, we need to reverse engineer a little bit in order to find the right breakpoint. The results are the following:

The easy part: `NTDLL.ZwContinue` has as first parameter a pointer to the context, as third one the exception code and the sixth is the address where the exception took place. Therefore, hooking `ntdll.ZwContinue` is enough for non-unwindings and non-refused exceptions.

The more difficult part (this was done for WinXP, translation for other OS versions will likely be trivial): we take `except32.exe`, open it with our debugger and choose ”handle exception in handler 1”. This will make the first two handlers to refuse to repair the exceptions. Now, press ”cause exception” and you are here:

```
00410608  div cl
0041060A  retn
```

We pass the exception to the handler, but stepping into (SHIFT+F7 in Olly). Now we see:

```
77F4109C  mov ebx,dword ptr [esp]
77F4109F  push ecx
77F410A0  push ebx
77F410A1  call ntdll.77F51763
77F410A6  or al,al
77F410A8  je short ntdll.77F410B6
77F410AA  pop ebx
77F410AB  pop ecx
77F410AC  push 0
77F410AE  push ecx
77F410AF  call ntdll.ZwContinue
```

Changing the return at `77F410A6` shows us that `ntdll.77F51763` returns `0` when the exception is not repaired. However, if we reach `ntdll.ZwContinue` we know we have omitted unwindings and the like, let's debug into `ntdll.77F51763`:

```
77F51763  push ebp
77F51764  mov ebp,esp
77F51766  sub esp,60
...
77F51771  call ntdll.77F51820
77F51776  test al,al
77F51778  jnz ntdll.77F806B9
```

This time, changing the return to `1` leads directly to `ntdll.ZwContinue`. By other hand, the return is the same both for refused and normal exceptions, therefore we don't step into it. The next interesting call is this one, which we need to debug into because otherwise we are at `ntdll.ZwContinue`:

```
77F517E8  push esi
77F517E9  call ntdll.77F7333F
...
```

With a bit more of patience we are here, this calls our exception handler (check yourself this is true for different instances):

```
77F7339B  mov ecx,dword ptr [ebp+18]
77F7339E  call ecx ; Except.0041080A
```

In summary, to log all the exceptions (WinXP) we need to hook `call ecx` and `ntdll.ZwContinue`.

Note: the first exception takes place before the target has started to run, it's indeed raised by the loader. Don't log it.

With this, we can easily log all the information we need about the exceptions.

VII. Entry Point location

Let's see the calls done by our packer, the last two ones from the extract above were:

```
...
VirtualFree From: 00D01711
VirtualFree From: 00D02224
```

This calls are obviously done by the packer (just in case we don't know it we can copy a few instructions from the return address onwards, it will be evident when they are obfuscated). Of course, the entry is after them.

In fact, we can check all our logs (`DLLs`, exceptions and others) and easily give a *lower bound*, as accurate as possible, to the entry point. The effort to find the Entry Point will always be drastically reduced.

A packer can be aware of this attack and group all exceptions and calls at the beginning. This wouldn't save us much work (anyway it would also be a good improvement, all the anti-debug based in the seh handler can be overcome for free). Dealing with the hard cases requires to inject a tracer.

A. Tracers

There are several ways of coding a tracer:

1) As part of a debugger.
2) Self-tracing code.
3) Code Emulator.

The first is not interesting for us, we are supposing our target to have heavy anti-debug tricks and the third one is out of the scope of this article. Let's pay attention on the second one:

Self-tracing code, already used in the old DOS times, is quite a tough anti-crack protection. Basically, all our code will be run with the trap flag set `ON`, meaning our seh handler will be called at each instruction.

The trap flag can be set as follows:

```
    pushf
    or dword ptr [esp], 100h
    popf
    nop     ; needed for some processors
```

This will produce an `EXCEPTION_SINGLE_STEP` and will call our seh handler. The seh handler has (kinda) `ring-0` access to the context, it can again modify the trap flag for the next instruction. Inside the seh handler you can set the trap flag as follows:

```
    push dword ptr [eax.cx_EFlags]
      ; eax points to the context
    or dword ptr [esp], 100h
    pop dword ptr [eax.cx_EFlags]
```

However, our seh handler is a bit more tricky. We need to:

1) Log all long "jmps".
2) Avoid anti-tracer tricks.

B. Log all long ”jmps”

When the trap flag is enabled we receive at cx_Eip the address of the next instruction to be run and in EXCEPTION_RECORD.ExceptionAddress the address where the exception has taken place. We only need to evaluate their difference:

```
    mov ebx, dword ptr
       [EXCEPTION_RECORD.ExceptionAddress]
    mov ecx, dword ptr
       [CONTEXT.cx_Eip]
    cmp ebx, ecx
    ja DontXchg
        xchg ebx, ecx

DontXchg:
    sub ebx, ecx
    cmp ebx, 0FFFFh
    jb DontLog
      ; don’t log jumps shorter than 0FFFFh
    call LogJmp
      ; log this long jump
```

This will log all jumps, next you simply have to have a look at the log file and decide yourself. For example, this is a real-life log:

```
VirtualFree API return: 00C01B74
   API return: 00000001
   <== last call we had available
Entry Point?  00090392
Entry Point?  00C01B74
Entry Point?  01006AE0
```

The right one is 01006AE0, easy to know if you see that 00090392 and 00C01B74 are at buffers previously allocated by the packer.

C. Avoid anti-tracer tricks

The tracer has to start after all exceptions and all calls done by the packer, this get’s rid of 99% of the tricks one can use to kill a tracer. In practice one has to examine the logs to decide what’s happened if his tracer gets killed, one can even attach the debugger if he needs so.

As an example, let’s see how to get rid of the rdtsc trick. rdtsc (read timestamp counter) is a semi-documented opcode which reads at edx:eax the current timestamp of the CPU, edx is the most significative part. For example, one can implement this trick as follows:

```
; read timestamp counter
    rdtsc
    push edx ; save most significative part

; loop to loose time
    mov ecx, 0FFFFh
next:
    xor eax, eax
    loopd next

; read again timestamp and compare
    rdtsc
    pop eax
    cmp eax, edx
    jne IAmTraced
```

When our program is traced it runs much more slowly (because we are running with the trap flag set). Overcoming this trick is quite easy, just do the next in your handler:

```
    mov ebx, dword ptr
        [EXCEPTION_RECORD.ExceptionAddress]

    cmp byte ptr [ebx], 0Fh
    jne NotRdtscOpcode

    cmp byte ptr [ebx+1], 31h
    jne NotRdtscOpcode

    ; if we are here is cos the current
    ; instruction has been an rdtsc.
    ; Mark this so we can change cx_Edx
    ; the next time the handler
    ; is called.

    mov dword ptr [IAmAtRDTSC], TRUE
```

In the next iteration we have to do:

```
    mov dword ptr [CONTEXT.cx_Edx],
        MY_CONSTANT_TIMESTAMP
    mov dword ptr [IAmAtRDTSC], FALSE; initialize
```

Don’t use MY_CONSTANT_TIMESTAMP = 0, it’s too evident.

In a more sophisticated version, one could count the number of instructions since the last rdtsc and decide whether cx_Edx has to be increased or not. This gets rid of virtually all rdtsc-like tricks. Other tricks would require other treatments.

D. How to install our tracer

There is still a detail we have to deal with, how to install our seh handler. After all exceptions of the packer have occurred no seh handler from it is needed, this means we can overwrite the final one so we catch all exceptions (due to the trap flag ON). To install our handler:

```
lea eax, [Tracer32_handler]
mov ebx, dword ptr fs:[0]
mov dword ptr [ebx+4], eax
```

In summary, our tracer will (slowly) run the packer logging all its `jmp's` until it reaches the entry point. Go to the log file and take it.

VIII. How stealth are these methods?

In this section we discuss some possible ways of detecting this intrusions and their weaknesses.

A. Threads enumeration

The target could enumerate all threads running in the system and check how many correspond to itself, this doesn't work. Note that the injected code, after the preliminary work has been done, is actually called by the target, meaning we can:

1) Allocate memory with `VirtualAllocEx`.
2) Take the entry point.
3) Store 1 memory page, or more, from the entry point onwards with `ReadProcessMemory`.
4) Save the context of the main thread with `GetThreadContext`.
5) Overwrite the entry point with our code, in this case the logger does the following tasks:
    a) Open the log file as a shared file mapping.
    b) Allocate memory for the garbage buffers (hooks).
    c) Hook `kernel32`, redirecting it to the allocated memory.
    d) Set an event to `TRUE` so the carrier knows it has finished.
    e) Await.
6) Run our code.
7) Stop our code with `SuspendThread`.
8) Write back the original code with `WriteProcessMemory`.
9) Restore the original context with `SetThreadContext`.
10) Resume the main thread of the packer with `ResumeThread`.

The only point here is to ensure the buffer to redirect the APIs will exist even if the main thread of the packer terminates. This can be done by creating this buffer as a named shared file mapping and opening it from the carrier. This way, the buffer will not be released until the carrier consents.

B. Locating the buffer where we have copied our code

Again, not a very good idea. Our code can be hidden inside the .reloc section of kernel32.... Indeed, we can monitor all calls to APIs like `CreateFileA` trying to open our file and change their return to `INVALID_HANDLE_VALUE`.

C. Code an emulator to jump over lots of instructions at the API beginning

Works but it can also be used for cracking purposes. If we are able to code an emulator which goes through many instructions, even some anti-debug trick, we can use it as Import Rebuilder.

D. Check the DLLs against the ones at the system directory

The packer needs to do some calls to retrieve this `DLLs`, next it will open it. This will be evident in our logs (log `NtCreateFile`).

E. Check the loaded DLLs against some hardcoded checksum

Works, but is not compatible for all Windows versions. The packer should be aware of any new DLL version, note that Windows makes available security patches every one month or so, some of them update the system `DLLs`.

F. Analyse the first instructions of the API to see if they are "API-like" ones

What about these ones?

```
push ebp
mov ebp,esp
push -1
```

```
    push dword ptr [ebp+4]
    push dword ptr [ebp+8]
```

We can also generate API-like instructions, note that all the previous ones can be inverted and so we can still easily log the input parameteres (in this case, `add esp,16` undo all computations).

G. The vector AddressOfFunctions has values that are outside the DLL image

Easy to defeat too. Link our logger with some instructions stored at the .reloc section of the `DLL`, `.reloc` is not needed once the `DLL` has been loaded in memory (there are more sophisticated methods, *almost absolutely undetectable*).

In summary: our method is quite stealth.

IX. Conclusions, further research

In this article, we have introduced a new method to gain insight information from heavily protected software. The method is very stealth and yields enough information to drastically reduce the time spent into cracking many hard targets.

The main drawbacks of our method are:

1) Needs human intervention to analyse the logs.
2) Needs to be aware of all tricks we used in the previous section.

Note that, since current anti-cracking software is not aware of these attacks, (2) will only be an inconvenient in the future.

Anti-cracking software should prevent from intrusions, thus, further research should move on how to prevent them. However, as we saw in the previous section, this is not an easy task.

Final remark: everything explained in this article can be coded in about two or three weeks (with ”moderate” effort).

References

[1] Havok, “Asprotected notepad” *Codebreakers-Journal*, First Issue 2004.
[2] Labir, E., “Adding imports by hand” *Codebreakers-Journal*, First Issue 2004.
[3] Natzgul, “How to access the memory of a process” *available at Fravia*.
[4] Kruse, T. “Processless Applications - Remote threads on Microsoft Windows 2000, XP and 2003” *Codebreakers-Journal*, First Issue 2004.