Lena 21강

2012년 2월 8일 수요일

오후 10:56

1. **Abstract**

In this Part 21, we will learn something about rebuilding imports in general and also while unpacking some UnpackMe's.

This will be an extremely important Part in this series : everything about unpacking a more advanced packer will be based on this.

Be sure to have understood this part before continuing in unpacking …. . For better comprehension and if you are a newbie, I advise you to first see all previous parts in this series before seeing this movie.

In my search not to harm authors, I chose these UnpackMe's because all are freeware packed, open source or deliberately chosen older versions of the packer/protector.

The goal of this tutorial is to teach you something about a program's behaviour.

Here, these UnpackMe's are only chosen because they are ideal ideal for this tutorial in reversing and they are targeted for educational purposes only.

I hope you will exploit your newly acquired knowledge in a positive way.

In this matter, I also want to refer to Part01.

As always: most of these writings are mine, others were collected from different sources.

**이것도 똑같음**

Set your screen resolution to 1152\*864 and press F11 to see the movie full screen !!!

Again, I have made this movie interactive. So, if you are a fast reader and you want to continue to the next screen, just click here on this invisible hotspot. You don't see it, but it IS there on text screens. Then the movie will skip the text and continue with the next screen. If something is not clear or goes too fast, you can always use the control buttons and the slider below on this screen.

He, try it out and click on the hotspot to skip this text and to go to the next screen now !!!

Click here as soon as you finished reading (on each screen!)

During the whole movie you can click this spot to leave immediately

1. **Tools and Target**

**이것도 똑같음**

The tools for today are : Ollydebug and… your brain.

The first can be obtained for free at

<http://www.ollydbg.de>

ImpRec can be downloaded from :

<http://www.cracklab.ru/download.php?action=get&n=MjI1>

Loadpe can be downloaded here :

<http://wasm.ru/tools/6/lordpe14.zip>

Unfortunately, no download for the brain ;)

All tools are freeware.

Todays targets are 4 UnpackMe's

I included them in this package for research.

Also included is RegisterMe.exe should you want to research the explained imports

1. **Some general info**

OVERVIEW :

There all different Microsoft Windows operating systems, and they all have different address for their API functions, because of different structured DLL's.

When an application starts, it has a list of all functions that aren't originally a part of the application.

These functions, called imports, are located in the operating systems DLL's, but the application doesn't know where. Every win32 executable application has an Import Address Table(IAT) residing inside the program.

The IAT is used as a lookup table when the application is calling a windows API function. So before starting, the windows loader has to find each address of each API that the program wants to call and "constructs" an IAT with them.

When the program is running and it wants to call an API, it simply looks in the IAT and thus finds immediately the address it needs to go in the DLL!

When an executable has been packed or protected the reverse engineer must recover the original executable file because a lot of packers/protectors destroy the IAT(while taking care of finding the API's for the program).

The import address table needs to be either rebuilt or fixed to allow for the executable to run properly. Import rebuilding is the reconstruction of the Import address table(IAT).

So far a short overview of the problem.

To understand all better and to be able to rebuild the IAT, it will be necessary to dig a little deeper in the theory.

However, I will only tell you here what's really necessary to understand and I'll skip those parts that are of no concert to us

INFO :

Now, let's start this over but look in it more in detail. First of all : when an executable is first loaded, the Window loader is responsible for reading in the files PE structure and loading the executable image into memory.

One of the other steps it takes i s to load all of the dlls that the application uses and map them into the process address space.

The executable also lists all of the functions it will require from each dll. Because the function addresses are not static, a mechanism was developed that allows for these variables to be changed without needing to alter all of the compiled code at runtime.

This was accomplished through the use of an import address table(IAT).

This is a table of function pointers filled in by the windows loader as the dlls are loaded.

When the application was first compiled, it was designed so that none of the API calls use direct hardcoded addresses but rather work through a function pointer.

Conventially this pointer table can be accessed in several ways. Either directly by a call[pointer address] or via a JMP thunk table.

By using the pointer table, the loader does not need to fixup all of the places in the code that want to use the API call, all it has to do is add the pointer to a single place in a table and its work is done.

INFO :

When it comes to packed executables, they almost invariably mess with the processes import table in order to make the executable smaller and make it harder for people to unpack and get running again.

Packed programs were still generated with standard compilers, and of course they were still designed to work this fixed mechanism (which is a very efficient way to handle the problem anyway).

If a packer has destroyed the default import table mechanism, that simply means that the packer/protector will have to figure out which dlls and functions to load and where to place the pointers so that the original program still operates as normal after it has done its decompression and restoration routines.

In order to understand how to restore a destroyed import table we will first have to start by understanding how the Import table is laid out and what work the Windows loader must do to parse it.

To make all this better understandable, I have opened a normal not packed exe in Olly and let's have a look at the normal structures.

We can clearly see all the calls to the API's in the code.

BTW, the exe that is opened here is the RegisterMe executable that I used for part03 in this series. I included it for if you want to research it.

This for example is the call to an API : it is the call to the function GetModuleHandleA which is in kernel32.dll

While this for example is a call to ExitProcess also in kernel32.dll

Now, what does this look like when assembling such a line?

It looks like every other normal call !!! How come Olly displays and knows it is a call to an API then ??? To find that out, just press <enter> to follow the call (== to see what code this call is calling)

Now press <enter>

… and we land in the jump(thunk) table. This is why Olly knows that the call before was a call to an API!

Info :

Each call to the API ExitProcess anywhere in this application will call this same address(0040120E) which makes it rather easy for the windows loader of course : only this JMP needs to point to the right address instead of changing every call anywhere to ExitProcess

And what does this look like when assembling ?

Right, it jumps to the value of a dword pointer.

Let's follow the pointer in the dump to find its value

And we land in the IAT with all the addresses for the API's in their respective dll's

In our example here: the API ExitProcess

Will make the code jump here to the value of the pointer for 00402004(mind the endians)

And we see that this is at 7C81CAA2 in my system(yes, in MY system)

Now, let's take an overall look in the IAT from this executable

There are two API's imported from kernel32.dll

Which are here in the IAT …

And they are nicely separated by a dword with all zeros

From the API's in the next dll

Which is User32.dll in our case

Which is then again separated by a dword filled with zero's from the next dll's API's.

However in our case, there are only two dll's imported, so, here, it is terminated by a dword filled with zero's

Also notice that the API addresses mostly start with 7(7XXXXXXX) but there are also some others(mind the endians !!!)

To resume it : each call in the code that wants to call an API, will call the respective address in the jump (thunk) table …

… which will point to the API's respective address in the IAT

Hence, there will be no problem at all to find the address in the dll

So far the point of view for a program. Let's now follow the point of view from the windows loader, which is also the path to follow for the reverse engineer.

And so, once again we must start in the header. The windows loader first reads the header of the program. Specifically for the construction of the IAT(I won't go in the other stuff the winloader does), the bytes at RVA 3C are read.

In this case, that will be at VA 40003C(ImageBase 400000). This is done because the import table's relative virtual offset (RVA) is stored in the PE Header at its value plus 80h. Important remark : the IAT is not the same thing as the import table !!!

This will become clearer in a second though.

INFO :

The imports table has all the information Windows needs to link the APIs for your program. The imports table has a very simple structure : there뭩 one header for each imported dll - there뭩 also an extra one, totally nulled, to mark its end - and each header contains all the information for one particular dll.

Tka eour case for example : if you import APIs from kernel32 and user32 you will find 3 headers, one for kernel32, a second for user32 and an extra one to mark the end of the imports table.

The windows loader read the information from each header, and uses this information to fill the IAT : the IAT is made up of the IATs for each dll.

This header for each dll is called IMPORT\_IMAGE\_DESCIPTOR, the word IMAGE? Recalls us thas all this stuff is done in memory and so all offsets are RVAs.

It has the following structure:

IMAGE\_IMPORT\_DESCRIPTOR :

OriginalFirstThunk

TimeDateStamp

ForwarderChain

Name

FirstThunk

INFO :

When the loader reads the IMAGE\_IMPORT\_DESCIPTOR, it first checks the DLL it refers to by examining tis name. Next, the loader loads this DLL and starts constructing the IAT.

Constructing the IAT is a bit tricky: first, the loader examines the OriginalFirstThunk, but this information is only used in case of trouble.

Next, for each name pointed to by the FirstThunk, it substitutes the pointer by the API's address.

If -for some reason- the API is not found, it goes to the OriginalFirstThunk and tries to get the information from there. If this final possibility still doesn't work then it crashes.

Therefore, in memory, all pointers in the FirstThunk contain addresses to the APIs from the current DLL instead of RVAs to the names of the APIs.

Note that the IAT construction is done AFTER the exe has been mapped in memory.

RESUME :

The loader read each API name in the FirstThunk and looks for its address. If it finds the address, then it substitutes the name by the address, otherwise it goes to the OriginalFirstThunk and tries again.

So, the OriginalFirstThunk is a "back-up" of the FirstThunk and is used in case of trouble.

The FirstThunk is an array of pointers to the names of the APIs we need to import.

If the loading process ran correctly, all the pointers into the FirstThunk have been overwritten by the addresses of the APIs and now all those addresses are called the IAT.

All calls from the program are redirected to the IAT. The addresses written as IAT by the loader can be:

1. The actual address of the API
2. A jump to the API
3. PUSH RVA API

INFO :

For a full CORRECT set up import table :

1. RVA and size of the Import Table need to be set into the data directory for the imports. Otherwise, windows is unable to find it and therefore the IAT won't be informed.
2. Declare each DLL with a IMAGE\_IMPORT\_DESCIPTOR and close the Import Table with a totally nulled one.
3. The IMAGE\_IMPORT\_DESCIPTOR has the OriginalFirstThunk, FirstThunk and Name well informed. The TimeDateStamp and ForwarderChain can be set to zero. The OriginalFirstThunk can be zero too.

I imagine all this is quite difficult to digest at once, but let's see it in this RegisterMe executable and it will become a lot clearer already.

:)

Go to RVA 3C (VA 40003C here)

INFO :

In EVERY exe : find the address for the import table at the value from VA 40003C + 80h (RVA 3C + 80h)

So, here, it is C0 + 80 == 140

Let's go there : scroll down.

The import table is at RVA 2050

INFO :

The "Import Table Address" means the address where to find the import table. DO NOT confuse this with the IAT, it is something completely different !!!

REMARK :

As you can see, finding the import table address is no problem in Olly : Olly really gives a wealth of info in the header and in fact, all you have to do, is look for "Import Table Address".

However, I think it is best to know the basics too, and understanding what you are doing makes you don't get lost outside Olly ;)

Let's find it.

And notice that this is right after the IAT in this case.

This looks like rubbish, but just analysis with the Analyse This!

Plugin. (See previous Parts)

The first part in the import table is the array of IMAGE\_IMPORT\_DESCIPTOR

An IMAGE\_IMPORT\_DESCIPTOR for each dll

An IMAGE\_IMPORT\_DESCIPTOR for each dll

And a closing IMAGE\_IMPORT\_DESCIPTOR

Each IMAGE\_IMPORT\_DESCIPTOR consists of five dwords

OriginalFirstThunk :

The first dword is the OriginalFirstThunk.

This gives the loader the information about where to find the names of the APIs to be imported from the current dll.

If now we go to IMAGE\_BASE + 2098, we find the API names to import(see further)

TimeDateStamp : useless for us, mostly all zero's

ForwardChain : useless for us, mostly all zero's

Name :

This is the RVA to the name of the DLL the IMAGE\_IMPORT\_DESCIPTOR corresponds to. In our example, we find "User32.dll" at 4021D8 (I'll scroll down in a second)

FirstThunk :

Points to the IAT where we can find all the addresses for all the imported functions.

(Not on disk, but once the executable is loaded in memory)

In our example, you can easily find all the address in the IAT for the imported API's from User32.dll, starting at 40200C

And the same can be reconstructed for the second dll in the same way

This import table is ended by a IMAGE\_IMPORT\_DESCIPTOR that is all zero's (5 dwords all zero's)

The import table in the dump window

The second part of the import table is the "arrays of dwords".

The arrays of dwords are pointed to by the OriginalFirstThunk of the IMAGE\_IMPORT\_DESCIPTORs.

Each dword of these arrays corresponds to an imported function. The arrays of dwords are separated by and ended by a dword filled with zeros.

This is the begin of the third and last part in the import table. This part continues further down. (See when I scroll down).

These are the strings of the imported functions and dll's. There is no conventional order : the dll names can be before or after the functions.

I said you would find the name "User32.dll at 4021D8 :)

See a nice and well-built IAT in the code. Study this for some time : two API's imported from kernel32.dll, separated by a dword with all zero's from APIs imported from suer32.dll and which is ended by a dword with all zero's.

See all the names from the imported functions, nicely ended by the dll name they reside in ….(remember, dll name can also be first)

All right, I hope this suffices to understand and remember everything you have to know about manually rebuilding the imports.

However, and that is the good news : we have some very good tools to rebuild imports in an automated way.

Indeed, rebuilding imports manually can be a long and annyoing thing, especially if the software is importing tons of API's from many dll's.

Using tools, we can mostly recover almost all API's and have only from time to time to do some manual work. With all this knowledge, let's see in some UnpackMe's how to proceed.

1. **The FSG2.0 UnpackMe**

And so we land here at the EP of the FSG UnpackMe. Remark that it is …. In the header!!!

FSG is typical unpacking by tracing. If you scroll a little down, you will already see the end of the unpacking stub'