Lena 15강

2012년 2월 4일 토요일

오후 7:04

1. **Abstract**

In this Part 15, we will reverse a small reverseme called "ReverseMe Nags" to learn something more about inline patching using a variable pointer.

I'm assuming you have seen and understood Part14 in this series and will base some of my thoughts on it as this will be an extension of previous tutorial.

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.

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

I coded this little ReverseMe Nags in my search not to harm somebody. Clicking the Register button registers the ReverseMe Nags so that the closing nag doesn't appear.

The goal of the reverser is to remove both nags in the code! Here, this reverseme is only chosen because it is ideal for this tutorial in reversing and it is 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 Part 1.

**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>

Unfortunately, no download for the brain ;)

Todays target is ReverseMe Nags.exe I included it in this package for research.

I included is also a patcher so that you can easily study the patched soft in case you should want to do that.

Also included are the shown plugins for OllyDbg : nonameo and bookmarks.

I'm also mentioning Wark/WPE which can be downloaded at :

<http://www.zero-g.it/RE/exetools/Wark13.rar>

All above mentioned are free available tools

1. **Behaviour of the program**

As always, it is extremely important to study your target well before attacking it. This may give you extra hints in how to solve the problem.

So, let's do that together in Olly first. I have already opened the reverseme and we are here at EP.

Keep your mouse pointer here and click whenever you are ready reading.

Run the ReverseMe

Ok, first shows the starting nag we have to remove.

Let's wait some seconds till the nag is gone …

...and suddenly, BAM, the main program window appears, telling us that if we click the "Register" button, the program is in registered status, thus avoiding the closing nag.

However, if we don't register, we get a closing nag as follows …

Mmmm, ok, right.

Now, the goal is to eliminate the nags in the code, and NOT by clicking the "Register" button. The closing nag disappears after a few seconds and then the program exits

Mmmm, let's restart and take a better look in the code.

1. Finding the patches

Run the Reverseme

And the starting nag appears again. Now, be fast and give Olly the focus by clicking in Olly….

And now press F12 to pause Olly.

INFO :

You can of course also do this by simply clicking the pause button in Olly ;)

We can now use the Call Stack to see what calls were called so far.  
This is an excellent way to discover some useful stuff. Just follow along.

INFO :

See previous Parts in this series if you don't know what the Call Stack or its use are.

Ok.

The starting nag is hindering us a little, but we can see enough already, indeed, we can see more then enough LOL ;)

Mmmm, the nag is called here …..

….and is created here

Let's go there and see this in the code by doubleclicking this line

This is where the nag is created (in MFC42.dll)

Scroll up to get a better overview

Oops !!!

Hehe, what more could we need?

This JE could simply jump passed the creation of the nag !!!

...which is indeed here, remember.

Ok.

Let's not go too fast and study exactly what goes on. So, place a BP, restart and run till BP.

BAM, and we break in the BP. Step F8 and see it all happen…

BTW, the JE does NOT jump !!!

Yep.

That's the starting nag. So, we should have jumped in the JE to avoid the nag.

Meanwhile, the nag gone and Olly automatically pauses when returning to the main code. Now, click run again to see what's up now …

BAM, we break again in the same BP.

This time however, it's to create the main program. Just step F8 and see.  
BTW, remark that the JE does NOT jump either.

Right.

Now, if we had jumped in the JE … the main window would not have been created ….

Mmmm, that's complicating things huh !!!

If we would make this JE jump always … the main program is not gonna be created !!!

But never mind, let's continue …

Olly pauses again

BAM, and we break again in the BP.

This time, it is for creation of the closing nag… Step F8 again.

The nag disappears and Olly pauses again.

Mmmm, now, let's run till the end, restart and then do some thinking ;)

1. **Inline patching the target**

And so, we return to our BP.

Now, let's think this over. What have we learned so far? This JE may NOT jump for the creation of the main window, but should jump for the nags to avoid them.

So, the JE may not jump the second time, right ???

Well, that's simple enough. Let's inject some code to do that. But first, we need a pointer whose value we can increment each time we pass this JE. Mmmm, ok, let's find a free byte.

BTW, I always do that in the data section, thus avoiding having to make the code section writeable.

INFO :

The program's header (see Part 3 !!!) contains the characteristics for all sections. These hold flags for executable code, writeable, readable, initialized data, uninitialized data, … The code section is normally NOT writeable.

You can however easily change this in the header under the characteristics in the section table (with Olly or easier with any PE Editor like LordPE, PE Tools, Wark, Stud PE etc etc)

INFO :

An example for making the code section writeable in Wark/WPE : open the file in WPE. Click "Sections", rightclick the section you want to change and choose "Edit section header" and edit the characteristics.

The section flags are clearly shown in WPE.

Wark/WPE can be downloaded here :

[www.zero-g.it/RE/exetools/Wark13.rar](http://www.zero-g.it/RE/exetools/Wark13.rar)

But remember that I was about to look for a free byte in the data section NOT to have to make the code section writeable ;)

Note that I'm also making a bookmark here, so as to be able to return here without having to search through the code. See the bookmark plugin in this package. (Thanks to the author)

Right. Here, I have found a free byte in the data section. So, this is the byte I will use for my "counter".

Each time the JE in the code section is executed, I will increment this byte's value by one. Notice that the value is zero now. Also notice its VA : 00445E80.

BTW, to make sure there is no interference with the program itself, first place a HW BP on write here and run the program.

If the program breaks --> search another free byte! I have verified it, but I've cut it from this movie to reduce its size.

Everything went just fine.

BTW, if you don't understand, just see further and everything will become clear (I hope)

INFO :

Remember the part in this series where we spoke about the different breakpoints.

We will discuss the mentioned technique better in later parts in this series.

I only want to mention here that I've done the necessary research to make sure there is no problem using this particular byte.

See more on this later!!!

We need to make the jump here to the code cave where we will place the injected code.

Scroll up to see better.

Ok. This JE takes 2 bytes, a far jump however will take 5 !!! So, the jump will overwrite the next line ….

Indeed, note down this line, so we can correct it also in our injected code.

And let's go searching for our code cave

Mmmm, this looks ok …

Here is some free space. Enough for the 'code injection' and right at the end of the code. Remember the VA, we will need it.

Making a bookmark here too…

And return to the JE to assemble the jump to this place

JMP 437d62

Press <enter> to assemble

Notice that the LEA … line is indeed overwritten

:)

번역주) NonaWrite???뭐지?

Writing inlines is particularly easy using the nonameo plugin because you can write the inline patches here to put them in our code cave which can be located anywhere. Just follow along.

BTW, I will later explain making manual inline patches on different occasions when dealing with packers/protectors.

No harm done if this is all a little complicated at first.

So, this first line indicates where we will assemble the patch. Obligatory to note this way.

BTW, this line has to be closed by a colon. I forgot it here but will see and correct it later. (See further)

It means I will assemble these patches at VA 00437D62

Now, don't go screaming about this first line of code here. Indeed, the

INC BYTE PTR DS:[445E80]

Is an ugly piece of code.

It should in fact be substituted by

PUSH EAX

MOV EAX, BYTE PTR DS:[445E80]

INC EAX

MOV BYTE PTR DS:[445E80], EAX

POP EAX

Which does the same. But yeah. This is what reversing did to me. Always looking for the smallest code to place in some forgotten code cave somewhere ;)

Now, in short, let's see what this code does :

INC byte ptr[445E80]

Will increment the value of this byte by one(remember, it's the byte in the data section).

This happens each time this code gets executed.(See further)

This byte is then compared with 2.

It will only be 2 the second time, when coming back here for the creation of the main window, right?

If it's not 2, it will jump till ……

… here, thus jumping over the creation of the nag.

But, if it equals 2, then this overwritten code gets executed ….

… to continue jumping ...

… here to create the main program ….

Oops, and then I noticed that I forgot the colon in the first line

Yep, that's it!

Now, assemble this code

Let's see the assembled inline

Ok. The code got assembled in the right spot. Now, save all the changes and test the app.

1. **Saving and testing the inline**

Save all changes to file

:)

:)

Setting a BP will make it possible to follow the executed code in the inline First run till BP

Step F8 and follow what happens.

The first step will jump to the inline

:)

;)

I'm sure you can understand what will happen. Just follow along….

Jumping

Scroll up. You'll see the nag won't get executed …

...because that would have been here

:)

So, no starting nag!

Instead of the nag, we break here again.

I hope you understand what happened.

Now, continue and just follow along.

Step the jump F8

And we land again in the inline

;)

This time, JNZ is not executed.

Step F8

The code runs the overwritten code

And we jump back to execute the main program

Press run!

The main program is executed!

We land back in the jump to the inline

The closing nag is gone too!

Mission succeeded.

In this part 15, the primary goal was to study some more about inline patching using a (variable) pointer. I hope you understood everything fine and I also hope someone somewhere learned something from this.

See me back in part 16 ;)

The other parts in this series are available at

<http://tinyurl.com/27dzdn> (tuts4you)

<http://tinyurl.com/r89zq> (SnD FileZ)

<http://tinyurl.com/l6srv> (fixdown)

Regards to all and especially to you for taking the time to look at this tutorial.

Lena151 (2006, updated 2007)