Lena 14강

2012년 2월 4일 토요일

오후 3:19

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* 1. **Abstract**

In this Part 14, we will reverse a "real" application (that was first deliberately expired) to learn something about a little more difficult to patch target, using a rather unusual API choice.

We will also see an introduction to inline patching. For better comprehension and if you are a newbie, I advise you to first see the previous parts in this series before watching this movie.

The goal of this tutorial is to teach you something about a program's behaviour. In my search not to harm authors, I managed finding a rather old version of a program which hasn't been updated for 1.5 years.

I included the main executable only for the same reasons(not the install exe) for your research and to study the shown techniques. Taking a look in the specialized media, I also found this application to be "cracked" already.

Here, this application 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 DVD Menu Studio 1.1

I included the main executable only in this package for your research.

* 1. **Behaviour of the program**

The importance of decent study of the behaviour of your target has been emphasized on different occasions already…

… but I have deliberately expired this software. Hence, this one is different!

Almost no behaviour can be studied, exactly because we start with an expired program!

But I'll show you the little we can see….

The target was opened in PEiD and shows us this

The Krypto Analyzer plugin (Kanal) for PEiD has detected 3 crypto signatures.

Well, we'll see if these will hinder us in our work … ;)

INFO : of course they won't : crypto signatures are of use in registration schemes(see later)

So let's study what we can study !

Perhaps that can give us some info too?

There are 0 days left for trial because I set the clock 2 months forward

Hence, the "continue" (with the program) button is greyed out

The only thing we can do to let us continue is enter the registration code

INFO :

To be sure, I tried to result the clock 2 months back again : the program keeps the expired notice.

This means the program must keep the "expired" somewhere. That can be in the registry or in a file, etc.

Also this fact gives us already a hint for attacking the target. But I won't use this method today.

INFO :

Behind this registration nag is the program.

It is not visible here, but it is quite important that you notice that the program is built completely, but this nag doesn't let us reach the program without registering… I also noticed that the program is completely created BEFORE the nag is started !!! ( see later)

In fact. Although I said there is not much program behaviour… there is always quite a lot to study !!!

Ok. We know what we need to know.

Let's register the program!

* 1. Finding the patches

And fill in some bogus code.

Don't click Ok yet but go to Olly

By clicking in Olly

Pressing Ctrl-N brings up the Names

INFO :

In previous Part 13 in this series, I included a list with the most used API's for this situation. However, I will prove you here that there are also other relevant API's.

For example KillTimer in a time limited app

Press k on keyboard

See also Part 13 For more on this

:)

And place a BP on KillTimer

:)

And now go register the software

And click the program to bring up the registration box

We land here

Now let's have a look around

We land in the Badboy!

Remark this …

...and this

...and this

Scroll down to see what's after this

Well, it's more cmp …

So, see this

And here

Remember the last cmp

it's cmp EAX, 0B

Yep, cmp all over

Now go back where Olly broke and note …

!

...the most important of all EAX = 3 ==> BadBoy (remember this)

!

...and the cherish on the cake.

EAX = 4 ==> Goodboy (remember this for later)

Let's see where we jump from to reach the goodboy !!!

INFO :

I have deliberately not used the ARTeam OllyDbg.ini file to be able to show you once more how to use the "pane window" in Olly to find your way in the code. (No red/grey jumps are indicated).

BTW, you can also just check/uncheck these options(see options menu) to see the jumps or not instead of using ARTeam's ollydbg.ini file …

Rightclick and choose jump

Interesting! There is only one place that jumps to the goodboy!

And we land here. Mmmm, so this conditional jump decides on jumping to the goodboy or not.

Scroll up to see what sets the JNZ

Mmmm, more cmp

But in combination with the other info, it confirms that EAX must be equal to 4 to jump to the goodboy. Scroll further up

Ok. And do the same for this place to see again where is decided on jumping to the goodboy (or not)

And we land here. Mmmm, so this conditional jump too decides on jumping to the goodboy or not.

Scroll up to see what sets the JNZ

… "encore" ;)

Ok. And do the same for this place to see again where is decided on jumping to the goodboy (or not)

And we land here. Mmmm, so this conditional jump too decides on jumping to the goodboy or not.

Mmmmmmm

Scroll up to see more code

OK. Now let's study this here.

Aha, here is the start of the routine

That verifies registration

With all the comparing of EAX coming after that

This being the first compare with EAX deciding on jumping to the goodboy or not

Here, and here EAX is set

Mmmm, we could try to patch EAX here

See that the value for the pointer [EBP+8] sets EAX

Let's put BP's here

In the beginning of the routine

Doubleclick opcode to toggle BP

And here too

Because here is decided about EAX

RESUME :

So far, using the KillTimer API, I found the registration scheme. The registration scheme looks quite difficult at first, with lots of compares and conditional jumps.

But some logical studying of the scheme reveals that executing the goodboy is in compares with EAX being 4 (or not).

Building my way up with the jumps, I found the beginning of the routine and also the setting for EAX.

The goal now is to find out what happens when patching EAX here. Hence, let's rerun the registration to break here.

Now click here or F9

And see the ascii string = BadBoy we saw in the code :)) )

Re-run registration

And we land again in the beginning of the checking routine

Now, step over the code(F8)

And keep an eye on EAX

So, EAX equals 0c now and this can't be very good :)

Remember that EAX == 4 leads to the goodboy

Here it will all happen

See EAX

EAX will equal 3 in the next step

Remember, EAX equal to 3 leads to the BadBoy

I want to patch EAX to become 4

And I have 3 bytes to do that

Mmmmm, that's not so easy in 3 bytes because

Mov EAX, 4

Takes 5bytes

And also because EAX == 12f924 at this moment

Hence, let's inline patch the app!

* 1. Inline patching the software

INFO :

Although the technique I explain hereafter is often referred to as "code injection", strictly speaking, it is not.

The exact name for it is "inline patching". Code injection is a technique to introduce arbitrary code into a RUNNING computer process.

This can be done either locally or remotely through the internet. Locally means that an application writes arbitrary or wanted code into another application's address space so that when run, it appears as if the host application is responsible.

As we will see in later Parts in this series, "loaders" use a form of code injection. Stay tuned …

INFO :

Inline patching is a technique by which the programmer changes the code of a program on disk to enhance, to extend, to block, etc program's possibilities.

This technique is very often used when a software is packed. We will extensively come back to this later.

We have only three bytes here which is also not enough to jump to an inline patch. Scroll up first.

Problem is …. we are a bit out of space for a long jump

Never mind, let's look for some spare space at the end of the code

INFO :

The spare space I'm referring to, is called a code cave. A code cave is an unused block of memory(free bytes) that can be used to "inject" custom programming code to modify the behaviour of a program.

But : a long jump here will take two more bytes

And will overwrite the opcodes 53 and 56

INFO :

We can divide jumps according to the distance they jump and end up with two categories : short jumps and long jumps

The difference between them is that short jumps take only 2 bytes in opcodes but can only jump 80 bytes far.

All jumps beyond that are long jumps which take 5 bytes in opcodes!

So look carefully how to solve the matter

Do a binary copy to paste this code later like this

:)

And let's also make good use of our bookmark plugin to come back here later

:)

Now, I already looked for some available space but removed searching for it from this movie(to reduce the movie's size).

INFO :

We'll dig deeper in locating code caves in later parts in this series when studying inline patching in packed/protected code.

And I found some at 5E47D4

Go there now

:)

Remark the empty bytes here.

Doubleclick or press "spacebar" to assemble

And let's make EAX equal 4(remember the GoodBoy)

:)

Now indicate enough lines to perform the binary paste

All this starting at the next line

:)

And see the pasted lines here

Now assemble the jump back

Push <enter> to assemble

Notice that I'm assembling to jump back to the first line after the code that will be overwritten by the jump to the inline here

So the code is complete here

RESUME :

Because there is not enough free bytes to assemble MOV EAX, 4 in the code, I have done this by inline patching.

Hence, I have searched for some free bytes and found them here where I have assembled the patch for EAX and also the bytes that will be overwritten to jump here.

After that, the code will simply jump back and continue normal execution with one difference -----> EAX will be equal to 4 !!!

Of course, now I still need to assemble the jump to this inline patch in the code.

Place a bookmark here too Just in case :(

;)

And go back to the first bookmark

Scroll up to see more code

Like said, I still need to assemble the jump to the inline

Press enter

When arriving here, the code will …

...jump to our inline

...then execute the inline code(patching EAX into 4)

And jump back here to continue normal execution of the code.

Now we can remove the KillTimer BPs

So press Ctrl-N

Press k

:)

:)

And continue the registration ….

Are we on the way to success ???

Huh ????

But we land again at the beginning of the checking routine !!!

…

Think with me …

About what happened

Aha There is only one solution !!!

The tricky stuff is that registration changes the value of the pointer to EBP+8 ….

INFO :

Indeed, if you keep clicking "run", the code keeps coming back here

...and that is exactly the line we patched … and than the soft re-runs the …

...registration process with the new variable for EAX

… of course, here is the proof !!!

EAX equal to 0C

So, the program want to register with EAX = 0C

Mmm, but there was a cmp EAX, 0B

That passed everything…

Cloud that be the "I am registered" -message after EAX equal 4 is true ???

Mmmm, worth trying out ….

Goodboy message carried out.

Now let's change EAX to the hopefully "I am registered"

GoodBoy message carried out. Now let's change EAX to the hopefully "I am registered"

INFO :

Alfa numeric chars must always be proceeded by a zero

See the change here

And now continue registration with the GoodBoy being run already

Remove the BP

And give it a go

:)

So save the changes

…

(with EAX = 0B)

* 1. **Saving and testing the inline**

:)

And save under another name

Open the new program

And run it ….

Program registered!

In a program, there are always different methods to attack a target. And because this way is not reproducible without debugger, I took another look in this program.

First, I deleted this newly saved file and removed all BP's and opened the original file in Olly again to land ….

… here at EP again

* 1. **Finding the patches**

Run the program

Ok. The program runs. Now let's try the Call Stack approach.

Return to Olly

Pause Olly first

:)

번역 주)여기부터 Ctrl+r이 안되네. 마우스 방향을 좀 내려야겠다. 원래는

screen 3110 628 LeftClick screen 2106 1009 LeftClick screen 1560 20 LeftClick이었는데

screen 3110 676 LeftClick screen 2106 1009 LeftClick screen 1560 20 LeftClick로 바꿈.

I have already explained the Call Stack method before. See previous parts in this series for more info.

In Short, Olly tries to reconstruct the calls that lead to the last action from the program. In which …..

...the topmost is the last executed

...we can clearly see that the program ….

...calls the nag screen

But we want to find why and where this nag is created

Hence, we need to return earlier in the calls. Let's return far enough immediately...

So, not here in the creation

But here, hopefully in the decision on creating the nag or not

And we land here

Let's study the code a second and …

Mmmm, it seems we landed at the right place.

...if we click the exit button on the nag screen ????

Set a BP as visual aid first.

Let's find the beginning of this routine and see what hints we can find.

Scroll up

And breakpoint here too

Let's restart and break here to step this code to see if we can come up with something.

Bam … we break in the BP.

However, did you notice too that the program was already completed and ready to run?

It breaks here probably in the creation of the nag!

Now, take a good look around and what do we see ???

All timer specific API's

Mmmm, we could very well be in the "expired / not expired / registered?" settings.

Let's step the code.

That's positive.

We will run the KillTimer function first.

Yes, let's see in Win32.hlp

!

!

!

!

!

No need to explain this better!

I'm sure you understood all fine.

But let's see this in the code.

TimerID == 1

And the window's handle that installed this timer is the program's main window.

See the "DVD Menu Studio - [Untitled]"

Then, what is this compare for ?

Step F8 to see.

The pointer's value is zero!

Could this possibly mean "Am I registered" ???

Mmmm, we will jump the whole routine anyhow, either here

...or here

The only difference, is a different timer set ….

Mmm, and where will the long jump lead to ?

Well, let's just try it out.

Let's change the Z-Flag and not jump.

:)

And test it.

Bam. We break back in the beginning of the routine. So, this is to run the nag … or not.

Let's see it!

Bam. We break back in the beginning of the routine. So, this is to run the nag … or not.

Yes!

Program "registered". Study the necessary changes needed in the code

* 1. Patching and saving

I tested but removed it from this movie for size: this JNZ gets executed the second time we break.

So, this makes that the changes in the next JNZ will only get executed once.

Hence, no need to worry about that.

And let's see what the possibilities are.

INFO :

Whenever possible, I also like to change the deciding pointer's value because there is a big chance that the pointer also sets some other value along the way like for example to display a registered or unregistered string in the about box …..

INC

The pointer was zero.

It will be == 1 now.

번역 주) 왜 1로 바꾸지? 점프 안하기 위해서?

Ok. Save the patches and test.

* 1. Testing the patches

And so we land here again at EP from the newly saved software.

Let's run and test.

Everything is fine.

The nag was removed.

Because the soft is not crippled, it will keep working fine.

INFO :

Here, "not crippled" is used in the sense that all features are available in unregistered, I don't mean "crippled" like in "pure demo" of course.

A pure demo has not all options available in the code and can only be reversed if you add the missing code yourself :)

* 1. **Conclusion**

In this part 14, the primary goal was to study the possibilities of handling an expired program in a more complicated registration scheme.  
We also saw a short introduction in inline patching. I hope you understood everything fine and I also hope someone somewhere learned something from this.

See me back in part 15 ;)

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)