Lena 18강

2012년 2월 5일 일요일

오후 4:28

1. **Abstract**

In this Part 18, we will reverse a small reverseme called "ReverseMe Tutorial" to learn something about diversion code, encryption/decryption, self modifying code and polymorphism in general.

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 small "ReverseMe Tutorial" ReverseMe in my search not to harm authors.

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.

As always, most of these writings are mine whilst others have been collected by me 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>

Unfortunately, no download for the brain ;)

Todays target is ReverseMe Tutorial.exe

I included it in this package for research.

1. **Behaviour of the program**

As always, it is extremely important to study your target well before attacking it. This will give you the clues to solve and attack the problem.

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

:)

All right, we need to remove this nag.

Remember the strings!

And this is the main program window once more explaining what needs to be done.

The ReverseMe exits.

All right, not much behaviour to study, but I suppose we know enough ?

:)

1. **Finding the patches**

:)

AH! Everything we need is here!

Here are the strings from the nag!

Looks like this will be an easy one.

Let's fine them in the code…

The nag we need to remove! This will be way too easy! Let's breakpoint the messagebox and break here.

Recognize Title and Text!!!

Let's already take a look in the code here.

What is this bizarre code?

INFO :

If you step this "data-like" code, Olly will warn you and ask if you really want to set a BP. Just answer "yes" to allow stepping.

Ok. Let's also take a look at the rest of the code. Scroll up.

Nmmm, strange….

Let's remove Olly's analysis to see the original commands

INFO :

Olly analysis this code as data.

We will see in a minute why …

:)

AH, this is already better but … this is still complete junk code!

Anyway, it's clear that Olly's analysis doesn't bring us anywhere.

We might as well remove Olly's automatic analysis at restart and manually analysis when necessary …

Let's do so.

:)

:)

:)

Scroll back down

AH! That's quite different and better readable then before readable then before removing the analysis!

Finally, let's run and break in the nag !!

Oops! But we didn't break in the breakpoints for the nag!!!!

How is that possible?

While we can clearly see the Title and Text for the messagebox !!!

So? What is this mystery?

The answer is very easy : this is diversion (decoy) code. It is specifically coded to distract the newbie reverser … Indeed, this piece of code is completely useless and is never run, which is clearly proven by the breakpoints.

INFO :

Sure I knew what was going to happen (after all, I coded the ReverseMe) but this dumb incident also proves that you need to know more then just searching text strings whilst also showing that one can not always blindly trust what one sees!

So what now? Are we hopelessly lost? Of course not. There are many ways to reach our goal. For example, setting a breakpoint in MessageBoxA using the commandbar plugin(see previous Parts in this series) and run the ReverseMe to break.  
Then press Alt-F9 to return to user code and to drop right in the action :)

But rather, let's find step by step what is really going on. Hence, let's restart the ReverseMe in Olly.

I've done that, but removed it from the movie for size till …

...we land here again at EP. We won't need these BPs any further, remove them.

Without Olly's auto analysis, the code looks quite better. Start exploring and step F8

INFO :

There exists also an option to "treat the analysis as code next time". Just look it up and experiment (right-click)

:)

:)

The return value will be the ImageBase value in EAX

:)

VA 401011 is set in EDI.

That is probably preparing something. See later.

Sounds interesting! See what's in the call pressing F7

And we land in the call where the start address of the code section is set in EAX.

Aha, then GetModuleHandleA was also "diversion code" :)

The value for EAX(which is the byte at 401000 at the moment) is XOR'ed with 5A

Next byte

Then EAX is compared with <JMP.&user.BeginPaint>

Which is 401218

And we loop if the address is lower

Perhaps this is bizarre?

See better what this is as follows

And see better what this is.

Let me also show you by assembling …

And notice that these are indeed equal!

Scroll up to verify what exactly is at 401218

Aha. The compare is in fact : "Have we reached the jump table yet ?"

Scroll back down

Right. Now let me resume. In this loop, the byte codes in the code section from VA 401000 till 401218 are XOR'ed with 5A.

This is called "Decrypting" because the bytes have previously been encrypted.

INFO :

Encryption/decryption is the process of obscuring/clarifying information to make it unaccessable without special knowledge.

Encryption can be used to ensure secrecy, but other techniques are still needed to verify the integrity and authenticity.

Encryption or software code obfuscation is used in software copy protection against reverse engineering, unauthorized application analysis and software piracy.

It is used in different encryption or obfuscating software.

INFO :

The XOR instruction here being the encrypting/decrypting command, is basic but very often used because it is very easily decrypted.

Just XOR the result with the same factor to obtain the start value again!

In the case that XOR is used as encrypting instruction, it is often referred to as an "encrypting XOR", in short: "enxor".

Let's watch what goes on ….

These are the byte codes in the beginning of the code section. Keep your eyes on this while I keep pressing F8 to see "live" what the decrypting does …

You can still see what the decrypting ahs changed (color) !!!

Let's run this loop completely setting a BP after the loop.

And notice that the code section (till VA 401218) has been decrypted !!!

Let's take a look at that piece of code and scroll up!

The "unreadable" code from before has been decrypted in perfectly readable mnemonics !

INFO :

The decrypted code is displayed in red because Olly has noticed the special treatment of the code.

Notice that if we had stepped over the call F8, hence executing the decryption at once, Olly would just display all this code as usual.

INFO :

The code section is normally not writeable. I suppose it is clear that this code section however is. You can verify that in Olly: see header for characteristics of the code section.

You can also verify this with all PE editing tools(LordPE, WPE, PE Tools, etc)

See also the previous Parts in this series.

The loop was run, so, return from this decrypting loop

Pressing F8

AH! Next is a call that calls in the decrypted code.

Let's also verify what's in this call. Press F7

And we land indeed in the decrypted code.

Take a look around and notice all these MOV instructions.

And also all the operations with EDI (remember from before that the program was probably preparing something when copying address 401011 in EDI)

INFO :

So that you understand immediately and so that you can follow this right away, I will already tell you that now follows a piece of self-modifying code.

INFO :

Self-modifying code is code that modifies itself on purpose. Self-modifying code was used in the early days of computers in order to save memory space, which was limited.

It was also used to implement subroutine calls and returns when the instruction set only provided simple branching or skipping instructions to vary the flow of control

INFO :

Self-modifying code was used to hide copy protection instructions in 1980s DOS based games.

The floppy disk drive access instruction 'int 0x13' would not appear in the executable program's image but it would be written into the executable's memory image after the program started executing.

Nowadays, self-modifying code is used by programs that do not want to reveal their presence, such as computer virus and some shellcodes.

번역 주)virii -> virus 같은데

Virii and shellcodes that use self-modifying code, also mostly do this in combination with polymorphic code.

Polymorphic virii are sometimes called primitive self-mutators. Modifying a piece of running code is also used in certain attacks, such as buffer overflows.

Due to all the above, it is clear that the reverser needs to know how to handle this type of obfuscation.

Step the code F8 to see what happens

Set EAX to zero

This line in short: replace the two bytes (word) starting at 401011 by 6A 00

After executing it, see that the previous bytes which were 33 c0(== XOR EAX, EAX) have effectively been replaced by 6A 00 (== PUSH 0) !!!!

REMARK :

Remember this. Instead of restarting one more time to point you again at this. We will need this later. You can clearly see here how this line of code has written the opcodes 6A 00 ….

In the line above to form the PUSH 0

Conclusion : the byte 6A at VA 401016 and the byte 00 at VA 401017 make the PUSH 0

I will remind you of this when patching the reverseme thus skipping the restarting to show you this again (for size of this movie)

In short: prepare the address for the next word, etc. Just step this piece of code F8 and see then what was achieved ….

The next line is completed!

Continue stepping F8

:)

Ah ! This CALL EDI will execute the lines at VA 401000. So, let's first analyze the code and take a look at the (self-)modified code.

Ono! The self-modifying code has written a messagebox in the code

But see all this bizarre stuff for title and text !!! Weird!

But let's continue stepping. Step IN CALL EDI to see what's in the call

And we land here. Do you already have an idea of what this code seems like ???

Perhaps you have understood this is another enxor ?

This time however, it is a part of the data section, starting from VA 403000

Till 403128

XOR'ing with B3(leading zero for alpa-numeric chars)

Let's also follow this in the dump window

:)

Keep an eye here while I press F8 to see the decryption in action :)

Have you seen the magic?

Now set BP to run the loop.

And look now what is here….

… and here !!!

Also notice that right after decrypting the data for the messagebox, the code will run the messagebox. Remove the BP and step the code F8 to see it …

This indeed is the nag we need to remove.

Study the code here.

And again, let's see the possibilities using Win32.hlp

:)

We have already seen the MessageBoxA API before. So, I will point you right away to the important argument which is …

...but if we make this parameter 1…

The owner will not be found and the messagebox will not be displayed !!!

Let's see it in the code

Conclusion : simply make the self-modifying code assemble PUSH 1 instead of PUSH 0 and the messagebox will be killed (Markus)

I asked for a two byte patch, let's see for another possibility :)

Study the code. Can we not jump the messagebox?

Ah! Remark that right after the nag, we will jump to 40106A to continue with the code. But what if we jump there BEFORE the nag?

That would mean, to make the self-modifying code NOT assemble PUSH 0 ….

...which is opcodes 6A 00 but …

...the JMP instead !!!

So, we will need to make the self-modifying code assemble bytes EB 57 (remember that !)

Ok. Keep this in mind together with the former info about the opcodes at VA's 401016 and 401017.

Let's continue first with the code to see if there are no more surprises :)

Notice once more that the jump I have just assembled, would indeed jump the nag.

INFO :

As always, there are still other possibilities to remove the nag. Using the byte at 401010 for example.

Step the jump

And notice that …. Here follows another piece of self-modifying code

And look how this self-modifying code has overwritten the jump I assembled.

Scroll down and continue stepping to see what is next.

Ah! End of the self-modifying code!

And we will execute the modified code immediately

Ok. Then we need to step F7(step IN call EDI) to see what will happen. Press F7

To land here again. Notice that Olly not analyzed this modified code yet, so, analyze first.

Ah! That makes it clear: the self-modifying code has prepared the main program!

Just step the code to see it happen.

Ok.

INFO :

The second piece of self-modifying code can not harm us. At that time, we should already have jumped the nag!

Continue stepping for more surprises?

Let's see what's in this call.

Step F7

To land in the original code again.

By now, you probably recognize this soft of code already? ;)

This time, it's an XOR'ing with 8Dh

Look here while I step F8

BP and run to see the result

Aha! The author wants to make it difficult for us! She destroys every trace of evidence!

REMARK :

I'm only showing you some basic ideas. I could have made it a lot more difficult.

Imagine all this in a really big program, decrypting here what is at the other end of the code or imagine a really complicated decrypting scheme instead of a simple enxor: there is much more complicated polymorphic code then this … etc etc

BTW, we will eventually do something on a more exotic protector in this series in this field. See me later when we deal with inlining in encrypted code ;)

INFO :

Polymorphic code is code that mutates while keeping the original algorithm intact. This technique is sometimes used by computer virii, shellcodes and computer worms to hide their presence.

Most anti-virus software and instruction detection systems attempt to locate malicious code by searching through the computer files and data packets send over a computer network.

If the security software finds patterns that correspond to known computer virii or worms, it takes appropriate step to neutralize the threat.

Polymorphic algorithms make it difficult for such software to locate the offending code as it constantly mutates.

INFO :

Encryption is the most commonly used method of achieving polymorphism in code.

However not all of the code can be encrypted as it would be completely unusable. A small portion of it is left unencrypted and used to jumpstart the encrypted software.

Anti-virus software targets this small unencrypted portion of code. Malicious programmers have sought to protect their polymorphic code from this strategy by rewriting the unencrypted decryption engine each time the virus or worm is propagated.

Sophisticated pattern analysis is used by anti-virus software to find underlying patterns within the different mutations of the decryption engine in hopes of reliably detecting such malware.

INFO :

Metamorphic code is code that can reprogram itself. Often, it does this by translating its own code into a temporary representation, and then back to normal code again.

This is used by some virii when they are about to infect new files, and the result is that their "children" will never look like themselves. The computer virii that use this technique do this in order to avoid the pattern recognition of anti-virus software: the actual algorithm does not change, but everything else might.

Metamorphic code is more effective than polymorphic code. This is because most anti-virus software will try to search for known virus-code even during the execution of the code.

Metamorphic code can also mean that a virus is capable of infecting executables from two or more different operating systems (such as Windows and Linux) or even different computer architectures. Often, the virus does this by carrying several virii with itself, so it is really a matter of several virii that have been joined together into a "supervirus".

INFO :

Alphanumeric code is a series of letters and numbers (hence the name) which are written in a form understandable and processable by a computer. One such alphanumeric code is ASCII

More specifically, in computer underground terminology alphanumeric code is machine code that is written so that is assembles into entirely readable ASCII characters such as "a"-"z", "A"-"Z", "1"-"9", "#", "!", "@", etc

Writing alphanumeric code requires a good understanding of a machine code encoding scheme for the specific architecture that the code is to be executed on.

This code is used with the intent of fooling applications, such as Web forms, into accepting code used for exploits. Typically, such exploits involve buffer overflows.

Sometimes, alphanumeric code could also be used when the programmer needs to write a program but does not have access to any compiler or assembler.

All that is needed to write an alphanumeric program is a text editor. And who doesn't own one of those?

INFO :

Shellcode is a relocatable piece of machine code used as the payload in the exploitation of a software bug which allows an unauthorized user to communicate with the computer via the operating system's command line as a result of exploiting a vulnerability in software running on the machine. Normally stored as a null terminated string, it cannot contain null characters.

A shellcode may be used as an exploit payload, providing command line access to a computer system with the privileges of the process that has been exploited.

To avoid detection by anti-instruction measures and to store more than one string, programmers often make use of self-decrypting code, polymorphic code and alphanumeric code.

Shellcodes can be stored in a process'memory space and subsequently executed as a result of the attacker gaining control of the program counter using vulnerabilities such as stack and heal-based buffer overflows, or format string attack.

There are various methods of controlling the program counter which vary between operating systems and processor architectures.

They include but are not limited to: overwriting the return address in a stack frame, overwriting exception handlers and Windows based shatter attacks.

INFO :

Machine code or machine language is a system of instructions and data directly understandable by a computer's central processing unit.

The "words" of a machine language are called instructions, each of which cause an elementary action by the CPU, such as reading from a memory location.

Instructions are patterns of bits with different patterns corresponding to different commands to the machine.

Every CPU model has its own machine code, or instruction set, although there is considerable overlap between some. If CPU A understands the full language of CPU B it is said that A is compatible with B.

CPU B may not be compatible with CPU A, as A may know a few codes that B does not.

Some machine languages give all their instructions the same number of bits, while the instruction length differs in others.

How the patterns are organized depends largely on the specification of the machine code.

Common to most is the division of an instruction into fields. Typically the value of one field (the opcode) specifies the exact operation (for example "add").

Other fields may give the type of the operands, their location, or their value directly (operands contained in an instruction are called immediate).

Some exotic instruction sets do not have an opcode field, only operand(s).

Other instruction sets lack any operand fields, such as NOSC.

After all this "theory", remember that we just re-encrypted the data section (partly).

Remove the BP and step the return to get back from the call to this encrypting routine

Another call.

Step in to see what's in this call

And we land here. I suppose this doesn't need win32.hlp to understand what will happen ;)

RESUME :

Ok. We finally know enough. The reverseme shows us some diversion (decoy) code, then it decrypts the code section, runs some self-modifying code to create the nag, then decrypts the data section for the strings, runs the nag, runs some more self-modifying code to create the main program, runs the main program and before exciting, destroys the data section by re-encrypting it with another enxor.

But let us end all this and kill the nag!

Restart the reverseme

1. **Patching the ReverseMe**

We land at the EP. Remark that the code was not analyzed by Olly, hence it's all readable mnemonics again.

Remember that VA's 401016 and 401017 will produce the PUSH 0 that we want to assemble to EB 57 (JMP 40106A). Right? But also remember that these bytes are encrypted at this moment. Right?

Well, with that knowledge, it's all simple and straightforward. Let's first see what these bytes are now.

Scroll up 401016 and 401017

Notice that it's the bytes 30 5A that we will need to change.

This can very easily be done by a simple enxor. Remember that the code section decryption was executed by XOR'ing with 5A?

Meanwhile, I have opened Crackers' Tool (but removed it from this movie for size).

XOR A,B --> C

XOR A,C --> B

XOR B,C --> A

EB XOR 5A = B1

B1 ---> byte at 401016

57 XOR 5A = 0D

0D ---> byte at 401017

RESUME :

Due to the decrypting, we need not to assemble EB 57 at VA's 401016/401017 but B1 0D. I hope this is clear?

:)

:)

:)

Selection 저장

:)

1. **Testing the patched ReverseMe**

We land in the EP of the patched reverseme. Because this may all be a little confusing in the beginning, I will step and comment a little. Just follow along

This is the decrypting call. Let's see the encrypted code first. Scroll up.

Right. See the saved patches. Now, step the decrypting call F8 to see the result of the decrypting

;)

See what the self-modifying code will assemble at 401016/401017 now !!!

So, we have executed the decrypting call. After this decrypting call is a call to execute this code, so, press F7 to land …

… here. Step the self-modifying code.

:=)

Step F7 to land ….

...here. This is the data decrypting.

:)

But instead of the nag, we will now jump the nag

...to the self-modifying code that creates the main window.

The nag was removed, the reverseme runs fine.

INFO :

I also mentioned the 1 byte patch solution, changing the PUSH 0 from the messagebox to PUSH 1.

You can try it out on your own if you want. You will find that you need to assemble byte 5A at VA 401039 into 5B.

BTW, this is definitely the best and easiest solution ;)

1. **Conclusion**

In this part 18, the primary goal was to learn about diversion code, encrypting & decrypting, self-modifying code and polymorphism in general while reversing a ReverseMe.

I hope you understood everything fine and I also hope someone somewhere learned something from this.

See me aback in part 19 ;)

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)