Lena 17강

2012년 2월 5일 일요일

오전 1:54

1. **Abstract**

In this Part 17, we will reverse a keygenme to learn something about the basics of keygenning.

Keygenning is considered as high-level-reversing, the top of the bill (together with other things like crypto, etc).

Because this series is intended for newbies, and especially also for those without programming background, I'll restrict this first tutorial to selfkeygenning.

Keygenning will be dealt with in Part36. For better comprehension and if you are a newbie, I advise you to first see AL 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 "Selfkeygenme" in my search not to harm somebody. Here, this keygenme 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 ;)

See also Cracker's Tool included in part02.

Todays target is Keygenme.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 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 keygenme and we are here at EP.

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

:)

Well, at least we know now what needs to be done.

INFO :

A keygen (truncation of "key generator", also known as "keymaker") is a small program that will generate a key or serial/registration number for a piece of software.

The advantage of using a keygen or just finding a serial(keyfishing) as opposed to finding a serial on the web, is that the generated key will be unique, (as it is generated on the spot using the same method as the software vendor) and not traceable or in use by the legitimate owner.

Note: Some "keygens" actually cycle through known serial numbers as opposed to using real-time algorithmic key generation.

Another advantage of using keygens is to prevent(legitimate) registration key theft, especially when the software is used in public computers.

This is done by installing the software using generated keys, instead of legitimate ones, and only using legitimate keys when the software (or their author) requested one (for example, when requesting an upgrade to a newer version).

This is especially true, since most applications do little to protect the key from theft: most applications store the key without encryption, and some even display the key in the About Box, whilst almost no applications use a mechanism to prevent key theft(for example, by using software activation or hardware fingerprinting).

Either way, when a legit key is stolen and distributed over the Internet, the stolen key should quickly be invalidated by the software author (thus annulling the license).

INFO :

Keys generated with a key generator may not work with software or games that are used online for either multiplayer gaming, or for getting software updates.

This is because in the online version, software developers sometimes set a limit to only accept keys that they know were printed on the discs during manufacturing(or issued during online registration), and deny keys that weren't (The other keys generated with a keygen).

This is because the algorithm to produce the keys allows a massive number of keys to be generated and accepted in the offline version, but the online version has a more limited number of keys accepted. But there is some chance that a keygen will produce a valid key. Keygens are often preferred to cracks because cracks might introduce new bugs into an application.

These are usually written by cryptology enthusiasts. Many kengens are released by warez groups and can contain background music, usually a tracker file in the chiptune style, and sometimes even extensive artwork which can be in ASCII Art. Most of the time an .nfo file is contained with the keygen.

Keygenning can also be used to make a legal application which creates cryptographic keys, or generate valid keys for crackmes.

And eventually, a selfkeygenme is the program itself which is reversed to be its own keygenerator.

Two text boxes available.

That will mean our registration name and a serial probably?

Let's try it out.

Oops?

Debugger detected?

Let's see that in the code …

1. **Finding the patches**

INFO :

An often used API to get the text from a textbox is GetDlgItemTextA(and also GetWindowTextA).

I already explained in previous tuts how to breakpoint on API's and I also already mentioned the commandbar plugin for this.

Let's use this plugin today.

Olly gives us hints about what we can use

And also what exactly is needed

But this API is what I'm after.

Set a breakpoint in this API (immediately in User32.dll) by pressing <enter>

Right. No remarks from Olly

Hence the BP is set.

You can verify here if you want.

Let's register again to see if we break in our BP in Olly

Indeed. We break in the API in User32.dll

INFO :

There are two textboxes in the keygenme. So, press a second time "run" to get the input in the second textbox to break here again.

Right. We break again in the BP.

To run till code, use the shortcut keys "Alt-F9".

And we land here.

Right after the second GetDlgItemTextA got executed.

:)

:)

Let's see this in the code

The API GetDlgItemTextA has read (a maximum of)

1Ah == 26d bytes

In the textbox with id 106d

(see Reshacker)

106d == 6Ah

And has copied them in a buffer at 403038

From the window with handle 205B0

It is clear that this was to gather the name from the textbox

And the second GetDlgItemTextA has collected the serial from the textbox with id 107d == 6Bh

When returning, the number of bytes read is in EAX

Which is 5 bytes for the serial I used(47806)

For your convenience, I'll step the code while commenting where necessary.

Did GetDlgItemTextA find input?

Step F8

Jump to the messagebox to ask for input if there was none

Else, jump past the messagebox

To execute the API lstlenA to "retrieve" the length of the name.

Take a look in Win32.hlp if you want, it's selfexplaining

The length of the name is set in EAX

And then, we come to the serious matters : this looks like calculation for the serial.

Let's look …

INFO :

XOR

Syntax:

|  |  |  |
| --- | --- | --- |
| XOR | Dest | Src |

The XOR instruction connects two values using logical exclusive OR (OR uses inclusive OR). This instruction clears the O-Flag and the C-Flag and can set the Z-Flag. To understand XOR better, consider those two binary values:

1001010110

0101001101

1100011011

If you OR them, the result is 1100011011 When two bits on top of each other are equal, the resulting bit is 0. Else the resulting bit is 1.

You can use calc.exe to calculate XOR. The most often seen use of XOR is XOR, EAX, EAX? This will set EAX to 0, because when you XOR a value with itself, the result is always 0.

----> ESI is prepared for the calculation routine (== set to 0)

INFO :

MOV(Move)

Syntax:

|  |  |  |
| --- | --- | --- |
| MOV | Dest | Src |

This is an easy to understand instruction. MOV copies the value from src to dest and src stays what it was before.

There are some variants of MOV:

|  |  |  |
| --- | --- | --- |
| MOVS/MOVSB/MOVSW/MOVSD | EDI | ESI |

Those variants copy the byte/word/dword ESI points to, to the space EDI points to.

MOVSX: MOVSX expands Byte or Word operands to Word or Dword size and keeps the sign of the value.

MOVZX: MOVZX expands Byte or Word operands to Word or Dword size and fills the rest of the space with 0.

The value 1 is set in EAX. Together with ECX, these will serve as a counter : "How many chars of the name are finished and how many still to go ?"

The name that was copied in the buffer at VA 403038 by GetDlgItemTextA

In copied in EDX

The first char of the name which is in byte [1+403037] is copied in DL

INFO :

AND(Logical "And")

Syntax:

|  |  |  |
| --- | --- | --- |
| AND | Destination | Source |

The AND instruction uses a logical AND on two values. This instruction \*will\* clear the O-Flag and the C-Flag and can set the Z-Flag. To understand AND better, consider those two binary values:

1001010110

0101001101

0001000100

If you AND them, the result is 0001000100

When two 1 stand below each other, the result is of this bit is 1, if not : the result is 0.

Does this info make i t clear for you that this AND operation here will "cut" everything "higher" then DL to zero ??

번역 주)DH -> zero잖아. 왜 DL이야?

For example : if EDX is 4F7825A3 before this instruction, then it will be 000000A3 after the operation.

Or see here now

And after the operation

INFO :

You have probably understood this operation

AND register, 0FF

is often used to remain with the low byte only in a register

INFO :

I already mentioned before that an alfa numeric char needs to be proceeded by a zero. For example AND EDX,FF will NOT be understood and also be refused by Olly.

EDX(which holds only the value for the 1st char in our case) is copied in EBX to do some operations

INFO :

IMUL(Inter Multiplication)

Syntax:

|  |  |
| --- | --- |
| IMUL | Value |

|  |  |  |  |
| --- | --- | --- | --- |
| IMUL | Dest | Value | Value |

|  |  |  |
| --- | --- | --- |
| IMUL | Dest | Value |

IMUL multiplies either EAX with value (IMUL value) or it multiplies two values and puts them into a destination register (IMUL dest, value, value) or it multiplies a register with a value (IMUL dest, value).

If the multiplication result is too big to fit into the destination register, the O/C flags are set. The Z flag can be set too.

In our case, the value for EBX and EDX are equal. So, in fact the result is [EBX]

INFO :

ADD (Addition)

Syntax:

|  |  |  |
| --- | --- | --- |
| ADD | Destination | Source |

The ADD instruction adds a value to a register or a memory address.

This instruction can set the Z-Flag, the O-Flag and the C-Flag (and some others, which are not needed for reversing).

So, the result from the proceeding instructions is copied in ESI

But it is not finished yet because the value from the char is again copied in EBX

INFO :

SAR(Shift Arithmetic Right)

Syntax:

|  |  |  |
| --- | --- | --- |
| SAR | Dest | Count |

Shifts the destination to the right by "count" bits with the current sign bit replicated in the leftmost bit. The Carry Flag contains the last bit shifted out.

Modifies flags: CF OF PF SF ZF

See it as "divide by 2" for count 1

Just see what it means for the first char

6Ch becomes 36h

Adding 3 to EBX

Multiplying EBX by EDX and set the result in EBX

INFO :

SUB (Subtraction)

Syntax:

|  |  |  |
| --- | --- | --- |
| SUB | Dest | Src |

Sub is the opposite of the ADD command. It subtracts the value of src from the value of dest and stores the result in dest.

SUB can set the Z/O/C flags.

Subtract EDX from EBX and set the result in EBX

Add again add the result in EBX to ESI

And at last, add ESI with ESI

(or multiply ESI by 2)

INFO :

INC (Increment)

Syntax:

|  |  |
| --- | --- |
| INC | Register |

INC is the opposite of the DEC instruction, it increases values by 1.

INC can set the Z/O flags.

Increment EAX will prepare the next char in DL after the loop

INFO :

DEC(Decrement)

Syntax:

|  |  |
| --- | --- |
| Dec | Something |

Dec is used to decrease a value(that is: value = value-1)

Dec can be used in the following ways:

Dec EAX == decrease EAX

Dec [EAX] == decrease the dword that is stored at [EAX]

Dec [401000] == decrease the dword that is stored at [401000]

Dec [EAX+401000] == decrease the dword that is stored at [EAX+401000]

The dec instruction can set the Z/O flags if the result fits.

How many chars still to go?

The JNZ serves as a loop as long as ECX is not zero (as long as there are remaining chars in the name)

Now, scroll up to see better

And step F8

ECX is decremented to 6

So, we loop. Just look.

Let's not continue this loop all 7 times. Set BP and run to see what comes next.

Press F9 to run till BP

And scroll down

Study the code.

Here is already the compare that will decide if we jump to the badboy or not

And what exactly is compared?

In ESI is the name that was recalculated to 003EF552

Which is compared with … the serial that we input !!!

(Not in ascii but in hex of course)

MIND THE ENDIANS(see previous parts in this series but I'll explain endianness further in this Part)

Let's continue stepping to see it happen

The compare is not equal, hence we will jump

… the Goodboy

…to the Badboy

AH! And see what the badboy is!!

So, the debugger was not detected, it is the normal badboy message!! ;)

Just a trick from the author :=)

INFO :

I coded this only this way to tell you not to believe everything right away but to verify it !

:)

I left the BP set, so, let's break in the compare again and see if we can register this keygenme

And we end up in the same compare of course.

Now, think with me : we need to make the serial to be 003EF552. You can verify in Cracker's tool what serial this is. (See later for the conversion).

The goal however is to make a keygen, a selfkeygen to be more specific. Well, that is not so difficult.

This can easily be done by using the badboy messagebox. We need to inject some code first. Just follow along. Step till JNZ.

And see that we would jump to the Badboy. Now, scroll down to find a code cave

Well, here is some free space. Let's use it.

RESUME :

So far, I have stepped the code from the registration scheme and found that a key is made from the name input. This key is then compared to the serial I input.

If these don't match ---> Badboy. With this knowledge, now I want to assemble the code in a way that the key is displayed in the Badboy's messagebox. Hence, I will need to assemble an easy inline and some changes in the code.

Let's start by assembling the inline

Do you understand what I have assembled?

Let me explain : I am using the free bytes that I saw here in the dump window

And who are in the data section which is writeable

So, what I have assembled will write the calculated serial in 00439000

Remember, ESI holds the calculated serial

In short : this code will write the calculated key in 439000

Let's assemble the next

Indeed, we will jump to the Badboy message to display the serial. Of course, I still need to assemble the jump to this inline. Let's do that.

Notice that I have assembled this JNZ to jump to the

:)

And to end, I still need to make the program write the right serial in the messagebox instead of the Badboy text.

Let's see how we can do that(using win32.hlp)

:)

INFO :

I suppose you know how to work with win32.hlp meanwhile. I'll make this one short because it's all selfexplaining.

Hence, I'll point you immediately to the important lines

See that this address points to the text that will be written in the messagebox

Now, suppose that we would change this address, and assemble it to point to the key that our inline writes at VA 439000 ???

Well, let me just show you what would happen then !!!

:)  
And the selfkeygen is created!

But first, let me repeat in short what will happen. We are here, after the key from the name is calculated.

Here, I have assembled to jump to the inline in case of a wrong user input

...where the right calculated key is written in 439000. Then we jump to the Badboy

...Where I changed the code so that the right serial is displayed in the messagebox

Test all this by clicking run.

시스템 생성 대체 텍스트: 00401333
oe4e!334
49
75D3
3B3드383i4eO0
754드
6p00
6난8a344Oe0
68B894406B
6R00
ES옛）00e0O0
印i3
6R00
6982〕44000
6800904300
'ROB
ES8e0e.000
EB!5
「F7S:4
FF7510
「「’驛eC
FF7S08
EeSF600000
C9
'JHZ
C『IP
JNZ
PUSH
PUSH
PUSH
PUSH
이룁L
J"P
PUSH
PUSH
PUSH
PU딤H
CnLL
J"P
PUSH
PUSH
PUSH
EC닝
SㅐORTK.,9．사늘．ee4el3eg
ESI.D닒ORDPTRDS.t4e3138]
SHORTK.ygef們’.004el383
L1
Kㅎ，g에＂e.OO4e3462
K.,g轍N..004B3488
l쫴
<JHP.各USERSZ.he'''9e8。뷰R》
SHORTKe,9翩He.oe4O1366
'1
^
0040133E
oe4e1340
0040194드
oe4OI긍4R
0040134C
004013「l
OO4el3S3
0e4el355
oe4O13야놈
OO4O195F
넌j낙'-'긺꽈현숩
O。댜Uld。b
0e40!365
0e401368
0040136E
00401371
oe4e13컨추
Oe4O1379
004013万놉
0048137。
0040137F
oe4e13여）
oe4OI383
004el389
00401388
0040138C
0040138D
oe4o!38F
oe4013EF
0040!3cn
L:104019)1
0e40노콰:
凸p4乃맨-
'1ㄴ',:.'
（尸（,■뿌《「'《）7＜구
00401396
0040!397
0e40!3)8
004013닷9
004013앞경
0040!396
OO4e13앗〕
(J껨l
l
St,te· N〔야이＂〔R더》나K)DpL
TitLe=''KeyGentenal51 "
T．닝t·"Th.t'',‘功t.(5.1「)k헹9.n騰no닐r
ho니ne'=NLjLL
"e'’。9.B애R
K.,gehH..Oe4a3462
K.,genne.ee439eOO
「壑，나=
111零L슨,
h〔OKIt1B--RPPL戚）Dnㄴ
'Ye,6enlena151 "
"ULL
C卜ㅒ
－·bo
e
r9
·j
…·흡
·슝
"Ue
·．L
「I盼r納=鞭왠
lrar駟쬔b닒（
―炅쁜뺘鞋
'U합＇·Ln0O닐rrOC
8.)
빠蟾HD
【'Tutset「k이‘이쿵鮮뇰',。la'$:
n珌coBeR
《JHP.各USER32.he'$ages。Hn>
하괌ORTK.,9예＂..OO4elS7D
t아刃．4]
t야첩G.3]
t아砲．2]
Ca1000
33c0
PUSHtRR6.l」
CnLL<J"P.各USER32.Def닒요ndo낚Pr。CR>
LEnUE
RET"노．
0O
Ke炸enlenal기
因
띠··
Butperhapsyㅇudoubtit?
Let'sverify!
n밍 8
뱀‘r구C
各즙
騙湍痲赫綸
V、
훨r
disPlaysthe
COrreCtSerial
〔건’빽므변
DBoe
DB00
DBoe
DB00
DBoe
929Open빕自IU-Uoa긴aUapa딩eo‘긷숍pa룅dU凸p麴U
CCSEeaOO':,只－80^pO0800apa맬麴UO什깊
v··…갼＞…뇨＞……＞··^
·■■■·■■■■■■■■■■■ ·．．■·■■．．···．■·．

The messagebox displays the correct serial !!!

But perhaps you doubt it?

Let's verify!

We can verify this easily with a small tool that I included in a previous part.

Remember that the serial for my name is 3EF552

This is the serial MIND THE ENDIANS

INFO :

When a sequence of small units is used to form a larger ordinal value, convention must establish the order in which those smaller units are placed.

This cloud be considered similar to the situation in different written languages, where some are written left to right, while some others(such as Arabic and Hebrew) are written right to left.

The decimal numbering is big-endian when written using numbers, starting at the left with the highest order magnitude and progressing to smaller order magnitudes to the right.

For example, the number 1234 starts with the thousands (in this case: one thousand) and continues through the hundreds (2) and tens (3) to units (4).

Let's see in computers: there seem to be no significant advantages in using one method of endianness over the other, and both have remained common.

There even exists a third endianness : beginning in the middle!

Let's look deeper into this …

INFO :

Exist : little-endians, big-endians and middle-endians. In our case, little-endians are most important because this is the method used by Intel X-86 processors.

When some computers store a 32-bit integer value in memory, for example 4A3B2C1D at address 100, they store the bytes within the address range 100 through 103 in the following order for little-endian :

|  |  |
| --- | --- |
| (address) | 100 101 102 103 |
| (bytecode) | 1D 2C 3B 4A |

That is, least significant ("littlest") byte (also known as LSB) first. In other words, endianness does not denote what the value ends with when stored in memory, but rather which end it begins with.

For your info only, big-endians would mean the following :

|  |  |
| --- | --- |
| (address) | 100 101 102 103 |
| (bytecode) | 4A 3B 2C 1D |

And middle-endians would mean the following :

|  |  |
| --- | --- |
| (address) | 100 101 102 103 |
| (bytecode) | 3B 4A 1D 2C |

Or alternatively :

|  |  |
| --- | --- |
| (address) | 100 101 102 103 |
| (bytecode) | 2C 1D 4A 3B |

I hope this clears that up.

Let's see what the serial for my name look like

:)

:)

;)

And let's see what the keygenme thinks

:)

We break in the breakpoint.

But notice that the serial I input matches the calculated key!

REMARK : notice that the key that is written here dates from the previous running of the inline.

NOTICE THE LITTLE-ENDIANS

Step F8

The keys are equal, hence we don't jump to the inline

Well, we did that already !!!

REMARKS :

In this keygenme, I have shown you how to use the Badboy message to make a selfkeygen. There are many more ways and possibilities, but this is the easiest for starting.

For example : one can also make a selfkeygen that displays the right serial in the textbox which makes it possible to copy and paste.

However, this already requires the adding of the API SetDLgItemTextA in the soft plus a lot more changes in the code.

To reduce the size of this movie, I won't show how to save the changes to file to create the (self)keygen.

However, I have included a selfkeygenme in the package for research. You can use it to find a serial for your name and try it out in the keygenme.

1. **Conclusion**

In this part 17, the primary goal was to study the making a (self)keygen. I want to emphasize however that here is only demonstrated a technique.

This technique is certainly not useable for every program. Each registration scheme differs, and so does also the making of a (self)keygen.

There still is a lot to tell about keygenning. In the shown example, only the name is recalculated. In most real programs, also the serial is calculated into a key before comparing.

But I hope this can be a good start for more. I also hope you understood everything fine and that someone somewhere learned something from this.

See me back in part 18 ;)

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)