

Reverse Engineering KW's EasyCrack
By Rossignol [April, 2006].

Technical Consultation by KW.

Text Revision: 1.05

ENGLISH VERSION

Disclaimer and Terms of Use

PLEASE READ THIS DISCLAIMER CAREFULLY! FAILURE TO COMPLY WITH THIS DISCLAIMER REVOKES ANY RIGHT TO USE THE MATERIAL CONTAINED WITHIN THIS TEXT!

The content in this text is provided for informational purposes only and on an “as is” basis. Rossignol [henceforth: “the author”] does not warrant the accuracy, correctness, reliability, comprehensiveness, or currency of any content. The author disclaims all warranties, express or implied, regarding any content, and further disclaims liability for any consequences from the use or misuse of any content.

Preface and Target Audience

This text was written with beginners in mind. In order for beginners to fully benefit from the information provided within this work, a detailed explanation on many topics is required – an explanation which might seem redundant to people with slightly more advanced knowledge on the topic. The knowledge required to work with the material in this text is the minimal understanding of the 80x86 Assembly Language instructions, and the minimal mastery of the C++ programming language. As this text was written for beginners, let me warn you, advanced programmers, right from the start – you've nothing to look here for. I would also like to stress my opposition to software piracy – please, never practice your skills on real-world commercial software. People work with a great effort in order to create quality software products, and they deserve your respect to their work and copyrights - this is not a vain disclamation statement; I truly mean what I write.

Prerequisites

In order to work on the material in this text, some software will be required:

- OllyDbg v1.10 (<http://www.ollydbg.de/>).
- IDA – Interactive Disassembler – Freeware v4.3 (<http://gd.tuwien.ac.at/pc/simtelnet/winxp/progmisc/freeida43.exe>).
- A C++ Language Compiler.

© 2006, Text Written by Rossignol.

Reverse Engineering the Key Checking Algorithm

"Computers are composed of nothing more than logic gates stretched out to the horizon in a vast numerical irrigation system." - Stan Augarten

In order to begin reverse-engineering this target, perform the following steps:

- 1) Open OllyDbg.
- 2) Select the "File" menu, and select "Open". Open the EasyCrack.exe file.
- 3) Hit F9 in order to execute the debugged program. Input a name and a serial number of your choice, and click "Done". Unless you're unrealistically lucky in your guessing, you will get a window informing you that you've entered an erroneous serial number, in a very arrogant manner, that is. Close EasyCrack and return to OllyDbg's window.
- 4) Press ALT+F1 in order to open the Command Line window. The Command Line window allows you to toggle breakpoints on various elements, amongst which are Win32 API function calls. In the Command Line window, input "bpx GetDlgItemTextA", and press Enter. The GetDlgItemTextA Win32 API function is responsible for fetching the string data out of dialog boxes. That's exactly what we want due to the fact that EasyCrack has to fetch the Name string in order to perform computations on it. "bpx" means "BreakPoint on Execute", that is exactly what we want, since our demand is breaking on the call to GetDlgItemTextA.
- 5) The "Intermodular Calls" window should appear on your screen. This screen shows the addresses where the GetDlgItemTextA Win32 API function calls are made. You can close this sub window, as well as the Command Line sub window. Hit F9 in order to run EasyCrack, input any name and any serial number, and finally click on "Done".

6) OllyDbg should land on address 0x004010E1, as follows:

004010E1	FF15 18204000	CALL DWORD PTR DS:[<&USER32.GetDlgItemTextA	GetDlgItemTextA
004010E2	3508	TEST EAX, EAX	
004010E9	75 00	JNZ SHORT EasyCrac.004010F8	
004010EB	50	PUSH EAX	
004010EC	68 A4234000	PUSH EasyCrac.004023A4	ASCII "PFFT."
004010F1	68 84234000	PUSH EasyCrac.004023A4	ASCII "You might want to enter a name?"
004010F6	EB 1F	JMP SHORT EasyCrac.00401117	
004010F8	33F6	XOR ESI, ESI	
004010FA	56	PUSH ESI	IsSigned => FALSE
004010FB	56	PUSH ESI	pSuccess => NULL
004010FC	68 E9030000	PUSH 3E9	ControlID = 3E9 (1001.)
00401101	57	PUSH EDI	hWnd
00401102	FF15 14204000	CALL DWORD PTR DS:[<&USER32.GetDlgItemInt	GetDlgItemInt
00401108	3BC6	CMPL EAX, ESI	
0040110A	75 14	JNZ SHORT EasyCrac.00401120	
0040110C	56	PUSH ESI	
0040110D	68 A4234000	PUSH EasyCrac.004023A4	ASCII "PFFT."
00401112	68 40234000	PUSH EasyCrac.004023A4	ASCII "That's not a proper serial.. Use a normal number be
00401117	57	PUSH EDI	hOwner
00401118	FF15 10204000	CALL DWORD PTR DS:[<&USER32.MessageBoxA	MessageBoxA
0040111E	EB A9	JMP SHORT EasyCrac.004010C9	
00401120	50	PUSH EAX	
00401121	8D45 CC	LEA EAX, DWORD PTR SS:[EBP-34]	
00401124	50	PUSH EAX	
00401125	E8 D6FEFFFF	CALL EasyCrac.00401000	
0040112A	85C0	TEST EAX, EAX	
0040112C	59	POP ECX	
0040112D	59	POP ECX	
0040112E	56	PUSH ESI	
0040112F	74 0C	JE SHORT EasyCrac.0040113D	
00401131	68 38234000	PUSH EasyCrac.00402338	ASCII "RIGHT"
00401136	68 2C234000	PUSH EasyCrac.0040232C	ASCII "You got it!"
00401138	EB DA	JMP SHORT EasyCrac.00401117	
0040113D	A1 B4234000	MOV EAX, DWORD PTR DS:[4023B4]	
00401142	68 24234000	PUSH EasyCrac.00402324	Title = "WRONG"
00401147	6A 08	PUSH 08	
00401149	59	POP ECX	
0040114A	99	CDD	
0040114B	F7F9	IDIV ECX	
0040114D	FF3495 28204000	PUSH DWORD PTR DS:[EDX*4+402028]	Text
00401154	57	PUSH EDI	hOwner
00401155	FF15 10204000	CALL DWORD PTR DS:[<&USER32.MessageBoxA	MessageBoxA
00401158	FF05 B4234000	INC DWORD PTR DS:[4023B4]	
00401161	833D B4234000	CMPL DWORD PTR DS:[4023B4], 08	
00401166	75 00	JNZ EasyCrac.004010C9	
0040116F	56	PUSH ESI	
00401170	57	PUSH EDI	
00401175	E9 4EFFFFFF	JMP EasyCrac.004010C3	
00401178	8B45 08	MOV EAX, DWORD PTR SS:[EBP+8]	Case 110 (WPL_INITDIALOG) of switch 0040108C
00401179	A3 AC234000	MOV DWORD PTR DS:[4023AC], EAX	

Illustration #1: OllyDbg's display right after the break.

Let us inspect what's going on in Illustration #1. First, OllyDbg broke on the call to the Win32 API function, as per our demand. When the GetDlgItemTextA function executes, it stores the length of the string in the dialog into the EAX register. In order for EasyCrack to verify the length of the name, that is, whether we've entered a name at all, it TESTs EAX with itself. The TEST operation logically ANDs its operands, and updates the flags register according to the result of the AND operation, without saving the result into the first operand:

Invariant: EAX = 0 (name string length is zero).
 TEST EAX, EAX → AND 0, 0 → ZF (Zero Flag) = 1

Invariant: EAX > 0 (name string length is greater than zero).
 TEST EAX, EAX → AND LEN, (LEN | LEN > 0) → ZF = 0

As evident above, the Zero Flag will be set to 1 only if the length of the name string is zero; hence, no name was entered. After the Zero Flag was updated, a Conditional Jump of the JNZ (Jump If Not Zero) type is performed. The JNZ instruction checks the value located in the Zero Flag; that is, if ZF ≠ 0, the jump is performed, otherwise the jump is ignored and the execution flow continues to the next line of code.

- 7) Press F8 (Step Over) until you reach address 0x00401102. Here we hit another function call. This time, the GetDlgItemInt Win32 API function is used to fetch the string entered in the serial number field as an integer. Following the function call, there's another verification check, intended to check whether number we've entered falls within the valid range. The check is performed by using the CMP (compare instruction). The CMP instruction is in fact analogous to the subtraction (SUB) instruction with only one difference; it does not update the left operand with the result of the operation, it merely updates the flags register according to the result of the operation:

Invariants: EAX = 0 (zero or no number at all entered).
 ESI = 0 (hard coded value).
CMP EAX, ESI → SUB 0, 0 → ZF = 1

Invariants: EAX > 0 (a value greater than zero entered¹).
 ESI = 0 (hard coded value).
CMP EAX, ESI → SUB (VAL | VAL > 0), 0 → ZF = 0

Using this technique, EasyCrack checks whether the entered serial number falls within the valid number range. The valid number range is the group of natural numbers starting from 1 to $2^{32}-1$ (0x00000001 – 0xFFFFFFFF, which would fit exactly in a 32bits wide register)². If you will attempt to enter a number larger than $2^{32}-1$ (Like 2^{32} , hence 4294967296, the serial number will be recognized as an erroneous one due to an overflow; a 32bits wide register can not hold values larger than 0xFFFFFFFF, and 2^{32} would be equal to 0x100000000 in the hexadecimal radix).

- 8) Press F8 a few times until you reach address 0x00401125. At this address you will see a function call. Let us step into this function by pressing F7. Now, press F8 further until you reach address 0x0040100F. Here we can see a function call to the strlenA Win32 API function. The strlenA function returns (stores in the EAX register) the length of the string passed to it, this way, EasyCrack checks how long the entered name was in order to set the number of iterations to be performed on it in the next phase. Press F8 once more and you'll see that the value in the EAX register is being MOVED to the EDI register. The MOV instruction takes two operands and copies the value from the second operand into the first operand. Naturally, the length value of the name string is now also contained in the EDI register.

¹ Entering a negative numerical value is impossible in this case, since EasyCrack does not allow entering any non-decimal number characters into the serial number textbox.

² One of the error message boxes in EasyCrack erroneously states that the valid range is the natural numbers between 1 and 2^{32} . This information is false. The correct valid range is the natural numbers between 1 and $2^{32}-1$, inclusive.

- 9) So far, we have used OllyDbg to find out where in the executable should we be on the lookout for the serial number verification routine. At this stage, I suggest that you open IDA, and load EasyCrack's executable file into it by clicking on the "File" menu, selecting "Open", selecting EasyCrack.exe. In the preceding window, select "Portable executable for IBM PC (PE) (pe.ldw)" option, and click "OK". After the file opens, click on the "Jump" menu and select "Jump to address" (or simply press on the "G" button on your keyboard from anywhere in IDA). In the Jump to address dialog box, input 00401017. The IDA View window should then look like the following:

```

.text:0040100A      mov     esi, [ebp+lpString]
.text:0040100B      push    edi
.text:0040100C      push    esi
.text:0040100D      call   ds:strlen@0 ; lpString
.text:00401010      mov     edi, eax
.text:00401011      xor     edx, edx
.text:00401012      test    edi, edi
.text:00401013      jle     short loc_401047
.text:0040101D      loc_40101D: ; CODE XREF: sub_401000+3F1j
.text:0040101D      movsx   ecx, byte ptr [edx+esi]
.text:0040101E      add     [ebp+var_4], ecx
.text:0040101F      mov     [ebp+var_8], ecx
.text:00401020      rol     [ebp+var_4], 1
.text:00401021      mov     eax, ecx
.text:00401022      imul    eax, [ebp+var_4]
.text:00401023      mov     [ebp+var_4], eax
.text:00401024      mov     eax, [ebp+var_8]
.text:00401025      add     [ebp+var_4], eax
.text:00401026      xor     [ebp+var_4], ecx
.text:00401027      inc     edx
.text:00401028      cmp     edx, edi
.text:00401029      jl      short loc_40101D
.text:0040102A      cmp     [ebp+var_4], 0
.text:0040102B      jnz     short loc_401063
.text:0040102C      loc_401047: ; CODE XREF: sub_401000+10Fj
.text:0040102C      push    0
.text:0040102D      push    offset Caption ; lpCaption
.text:0040102E      push    offset Text ; lpText
.text:0040102F      push    ds:hWnd ; hWnd
.text:00401030      call   ds:MessageBox@0
.text:00401031      xor     eax, eax
.text:00401032      jmp     short loc_40107F
.text:00401033      loc_401063: ; CODE XREF: sub_401000+45Fj
.text:00401033      xor     [ebp+arg_4], 1337C0DEh
.text:00401034      sub     [ebp+arg_4], 00A0C0DE5h
.text:00401035      mov     eax, [ebp+var_4]
.text:00401036      not     [ebp+arg_4]
.text:00401037      xor     eax, [ebp+arg_4]
.text:00401038      neg     eax
.text:00401039      sbb     eax, eax
.text:0040103A      inc     eax
.text:0040103B      loc_40107F: ; CODE XREF: sub_401000+61Fj

```

Illustration #2: IDA's View window, I have marked strategic areas in red.

As apparent in Illustration #2, we are now at the same spot that we left OllyDbg at. It's noteworthy that we use IDA just for the convenience of analysis. The next instruction of concern is located at address 0x00401017:

```
.text:00401017      xor     edx, edx
```

Here, EasyCrack resets the EDX register to zero, in order to use it as an iteration counter. XORing a value by itself always results in zero. The assembly XOR operation XORs the first operand with the second, and stores the resulting value into the first operand.

Let us inspect the subsequent two lines:

```
.text:00401019      test     edi, edi
.text:0040101B      jle      short loc_401047
```

Once again we encounter the TEST instruction. This time we test whether EDI > 0, and if the condition yields true, we jump. The JLE instruction means “Jump If Lower or Equal”. In our case, it will jump to address 0x00401047 if the Zero Flag (ZF) is equal to 1 or if the Sign Flag (SF) is not equal to the Overflow Flag (OF). In summation, this is yet another check in order to see whether the serial number conforms to the required format. Next, the real crux of the algorithm begins.

Now, we’re going to inspect the loop which is responsible for computing what I call a “reference value”, which EasyCrack will use in order to verify the correctness of the entered serial number. Consider the following code (located at offset 0x0040101D):

```
.text:0040101D      movsx    ecx, byte ptr [edx+esi]
.text:00401021      add      [ebp+var_4], ecx
.text:00401024      mov      [ebp+var_8], ecx
.text:00401027      rol      [ebp+var_4], 1
.text:0040102A      mov      eax, ecx
.text:0040102C      imul     eax, [ebp+var_4]
.text:00401030      mov      [ebp+var_4], eax
.text:00401033      mov      eax, [ebp+var_8]
.text:00401036      add      [ebp+var_4], eax
.text:00401039      xor      [ebp+var_4], ecx
.text:0040103C      inc      edx
.text:0040103D      cmp      edx, edi
.text:0040103F      jl       short loc_40101D
```

This code could be easily replicated with a simple C++ do-while loop; let's look at how such loop would look like:

```

int EDX = 0;           //xor     edx, edx

do
{
    //Implementation ...
    ++EDX;             //inc     edx
} while (EDX < EDI);    //cmp     edx, edi
                        //jnl     short loc_40101D

```

First we set our iteration counter to zero, as done in offset 0x00401017, we then go through one function iteration and increment the counter as per the INC (increment) instruction at address 0x0040103C. We then verify whether the condition check yields true and decide whether to proceed looping accordingly.

Now that we know the structure of our loop, we can make it an even neater for loop:

```

for (int EDX = 0; EDI < EDX; ++EDX) { //Implementation ... }

```

Now it's the time to fill that loop with implementation. I will now explain line by line how to do it:

Put the ASCII value of the EDXth character in the name string into ECX (Character array start address: EBX):

```

movsx ecx, byte ptr [edx+esi]
Pseudo-code: ECX = str[EDX]

```

Add the ASCII value of the ongoing character to the value in VAR4:

```

add [ebp+var_4], ecx
Pseudo-code: VAR4 = VAR4 + ECX

```

Put the ASCII value of the ongoing character into VAR8:

```

mov [ebp+var_8], ecx
Pseudo-code: VAR8 = ECX

```

Rotate the value inside VAR4 by one bit to the left:

```

rol [ebp+var_4], 1
Pseudo-code: rol(VAR4, 1)

```

Put the ASCII value of the ongoing character into the EAX register:

```

mov eax, ecx
Pseudo-code: EAX = ECX

```

Multiply the sum value in VAR4 with the ASCII value of the ongoing character, store the result in EAX:

```
imul  eax, [ebp+var_4]  
Pseudo-code: EAX = EAX • VAR4
```

Put the result of the multiplication into VAR4:

```
mov   [ebp+var_4], eax  
Pseudo-code: VAR4 = EAX
```

Put the ASCII value of the ongoing character into VAR8:

```
mov   eax, [ebp+var_8]  
Pseudo-code: EAX = VAR8
```

Add the ASCII value of the ongoing character to the result of the above multiplication, store the result into VAR4:

```
add   [ebp+var_4], eax  
Pseudo-code: VAR4 = VAR4 + EAX
```

XOR the value in VAR4 with the ASCII value of the ongoing character:

```
xor   [ebp+var_4], ecx  
Pseudo-code: VAR4 = VAR4 ^ ECX
```

This whole scroll can be written neater this way:

$$\text{VAR4} = (((\text{str}[\text{EDX}] \cdot \text{rol}(\text{VAR4} + \text{str}[\text{EDX}])) + \text{str}[\text{EDX}]) \wedge \text{str}[\text{EDX}])$$

Developing a C++ Key Generation Function

So far, our function would look like this:

```
unsigned long GenNumForName (std::string x)
{
    unsigned long j = 0;  // Accumulator

    for (size_t i = 0; i < x.size(); ++i)
    {
        j = (((x[i] * rol(j + x[i])) + x[i]) ^ x[i]);
    }

    //return TO_BE_FILLED_IN_LATER;
}
```

The C++ language does not contain a built-in bitwise-rotate-left function, so we'll have to implement one. There are two general approaches to achieve this; the first (and easiest) approach is using Inline Assembly:

```
unsigned long rol (unsigned long iNum, int n = 1)
{
    __asm {
        mov eax, iNum
        mov ecx, n
        rol eax, cl
        mov iNum, eax
    }
}
```

First, we create an `__asm { }` block in order to tell the C++ compiler that Inline Assembly will be used. Then we MOV the value in the *iNum* actual parameter into EAX, we move the value in the *n* actual parameter to ECX (moving particularly to ECX is essential for ROLing, otherwise the compiler will issue an Improper Operand Type error). We then ROL the value located in EAX with the value located in the L.O. (Low Order) byte of the L.O. word of ECX (Hence: CL). Then, we MOV the result back into *iNum* (Notice that in this particular case, there's no need in passing the *iNum* parameter by reference).

The above solution is not optimal due to the fact that it virtually kills portability, since it will only work on Intel 80386+ based platforms. We could achieve the same functionality using the C++ syntax, as follows:

```
unsigned long rol (unsigned long iNum, int n = 1)
{
    return (iNum << n) | (iNum >> (8 * sizeof(iNum) - n));
}
```

This function simulates the behavior of a bitwise ROL. It first computes the resulting value for SHL (iNum, n), which will shift the whole DWORD *n* positions to the left (Casting away the former MSBs, and inserting zero as the new LSBs). It will then OR the whole expression with the new LSBs, which will cause it to be appended onto the new value, which is the exact modulus operandi of the ROL instruction¹.

Having said this, so far we have the following code in our possession:

```
unsigned long rol (unsigned long iNum, int n = 1)
{
    return (iNum << n) | (iNum >> (8 * sizeof(iNum) - n));
}

unsigned long GenNumForName (std::string x)
{
    unsigned long j = 0; // Accumulator

    for (size_t i = 0; i < x.size(); ++i)
    {
        j = (((x[i] * rol(j + x[i])) + x[i]) ^ x[i]);
    }

    //return TO_BE_FILLED_IN_LATER;
}
```

However, there's still some more work to be done. Have a look at the code at address 0x00401041:

```
.text:00401041      cmp     [ebp+var_4], 0
.text:00401045      jnz     short loc_401063
```

This turns to be just another verification check in order to verify the compliance of the entered serial number with the expected formatting. Hence, if after the loop the cumulative value in VAR4 will be equal to zero, an error message will be displayed.

¹ Although the equation of $[\text{rol}(x, 1) = 2x]$ is generally accepted, it is not applicable in this case due to overflowing.

Now, let's have a look at address 0x00401063:

.text:00401063	xor	[ebp+arg_4], 1337CODEh
.text:0040106A	sub	[ebp+arg_4], 0BADCODE5h
.text:00401071	mov	eax, [ebp+var_4]
.text:00401074	not	[ebp+arg_4]
.text:00401077	xor	xor eax, [ebp+arg_4]
.text:0040107A	neg	eax
.text:0040107C	sbb	eax, eax
.text:0040107E	inc	eax

What we have to do here, is simply to fill in the return value as follows:

```
return (((~j) + 0xBADC0DE5) ^ 0x1337CODE);
```

Here we just reverse the order of the relevant events in the above procedure in order to generate the serial number. Let us now put this whole puzzle together:

```
//----- genalgo.h -----
#ifndef GENALGO_H_GUARD
#define GENALGO_H_GUARD

unsigned long rol (unsigned long, int = 1);
unsigned long GenNumForName (std::string);

#endif
//-----

//----- genalgo.cpp -----
#include <iostream>

unsigned long rol (unsigned long iNum, int n = 1)
{
    return (iNum << n) | (iNum >> (8 * sizeof(iNum) - n));
}

unsigned long GenNumForName (std::string x)
{
    unsigned long j = 0;

    for (size_t i = 0; i < x.size(); ++i)
    {
        j = (((x[i] * rol(j + x[i])) + x[i]) ^ x[i]);
    }

    return (((~j) + 0xBADC0DE5) ^ 0x1337CODE);
}
//-----
```

All you have to do now is write a stub for the GenNumForName function, #include "genalgo.h", and test the key generator (You can use the stub I've prepared in advance. The stub and all the other source code files relevant to this text are located in the Source directory inside this work's archive).

Calculating a Serial Number Manually

I have realised that you might want to see how the calculation is performed on paper. Here is the whole path one has to go through in order to compute the serial number for the name “Rossignol”:

- 1) First, we infer the ASCII characters for the name “Rossignol” as follows:

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]
Name	R	o	s	s	i	g	n	o	l
ASCII (Decimal)	82	111	115	115	105	103	110	111	108

- 2) Given the fact that our reference processing loop iterates `x.size()` times, in this case, it'll loop 9 times. Let's put the name, step by step, manually, through the loop:

Accumulator: $j = 0$ (Initial value)

Formula: $j = ((x[i] \cdot \text{rol}(j + x[i], 1)) + x[i]) \wedge x[i]$

$$\text{Step \#1 (x[0]): } j = ((82 \cdot 164) + 82) \wedge 82 = 13,448$$

$$\text{Step \#2 (x[1]): } j = ((111 \cdot 27,118) + 111) \wedge 111 = 3,010,254$$

$$\text{Step \#3 (x[2]): } j = ((115 \cdot 6,020,738) + 115) \wedge 115 = 692,384,938$$

$$\text{Step \#4 (x[3]): } j = ((115 \cdot 1,384,770,106) + 115) \wedge 115 = 334,772,466$$

$$\text{Step \#5 (x[4]): } j = ((105 \cdot 669,545,142) + 105) \wedge 105 = 1,582,763,366$$

$$\text{Step \#6 (x[5]): } j = ((103 \cdot 3,165,526,938) + 103) \wedge 103 = 3,926,727,482$$

$$\text{Step \#7 (x[6]): } j = ((110 \cdot 3,558,487,889) + 110) \wedge 110 = 591,643,986$$

$$\text{Step \#8 (x[7]): } j = ((111 \cdot 1,183,288,194) + 111) \wedge 111 = 2,495,970,722$$

$$\text{Step \#9 (x[8]): } j = ((108 \cdot 696,974,365) + 108) \wedge 108 = 2,258,787,524$$

IMPORTANT: Some steps are subjected to an overflow and thus, the resulting values are truncated. I marked the equal-sign of the steps in which an overflow occurs in blue.

3) Now we shall compute the return value:

Accumulator: $j = 2,258,787,524$

Formula: $\text{result} = (((\sim j) + 0xBADC0DE5) \wedge 0x1337C0DE)$

The decimal value for 0xBADC0DE5 is 3,134,983,653.

The decimal value for 0x1337C0DE is 322,420,958.

Setp #1: $\sim 2,258,787,524 = 1,632,484,623$

Step #2: $1,632,484,623 + 3,134,983,653 =$
Overflow (value truncated) $= 876,196,128$

Step #3: $876,196,128 \wedge 322,420,958 = 655,258,110$

The serial number for the name "Rossignol" is 655258110.

Concluding Comments

In this text, you've had an opportunity to learn how to reverse-engineer a simple serial number verification algorithm. If you are completely new to reverse-engineering and feel a bit lost – let me cast away your fears at once. The more you practice – the more experience you gain. If you feel inconfident with any topics discussed within this work, please feel free to drop by at #cracking4newbies on EFNet, IRC, and ask your questions.

Thanks and Acknowledgements

This text would've not been possible without the invaluable help from the following individuals (by pseudonyms, ordered alphabetically):

BoRO, _death, dila, fornix, Junior, KW, Ousir, _teh, upb

Last but not least, I'd like to thank (and dedicate this work to) a beloved woman, S. – without you, it'd all make no sense.

Rossignol

© 2006, Text Written by Rossignol.
Technical Consultation by KW.
Reviewing by BoRO, KW.

THE USE OF THIS DOCUMENT IS BOUND TO THE TERMS AND CONDITIONS IN THE INCLUDED DISCLAIMER. FAILURE TO COMPLY WITH THE DISCLAIMER REVOKES ANY RIGHT TO USE THE MATERIAL CONTAINED WITHIN THIS TEXT.

Revision 1.05; Updated On April 11th, 2006.