Solutions for adrianbn's lincrackme3
Author: hasherezade

Content:

Intro

I. Analysis

1. Getting around antidebugging tricks

2. Finding the key verification algorithm

3. Understanding the algorithm

II. Keygenning

Intro

Presented solution describes how to solve this crackme without remote debugging.
Used:

-IDA Demo 5.7 (Windows)
-remote Linux account with gdb installed
-hexeditor

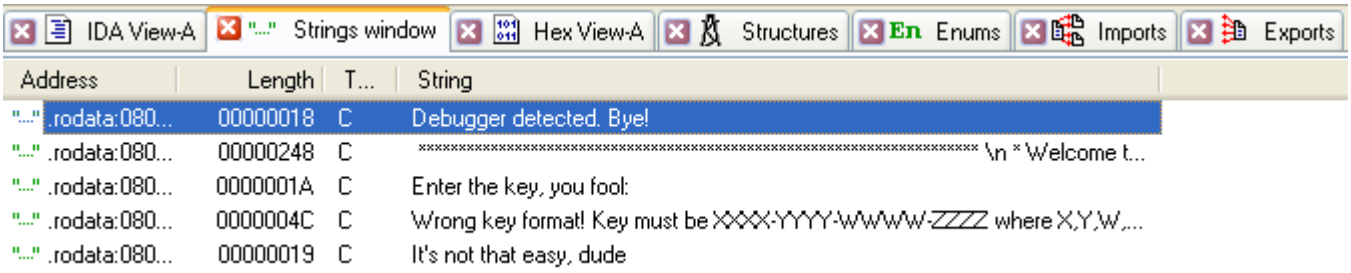
I. Analysis

1. Getting around antidebugging tricks

This crackme is an ELF file, so first I copied it on the Linux account and deployed. If I run it (without debugger) and fill with some dummy data, it shows the message "`Wrong key! I've seen monkeys smarter than you…`" ☺
Then I tried to run it via gdb:

```
$ gdb ./lincrackme3-32

(gdb) r
Starting program: /home/rez/Desktop/lincrackme3/lincrackme3-32
(no debugging symbols found)
Debugger detected. Bye!

Program exited with code 0377.
(gdb)
```

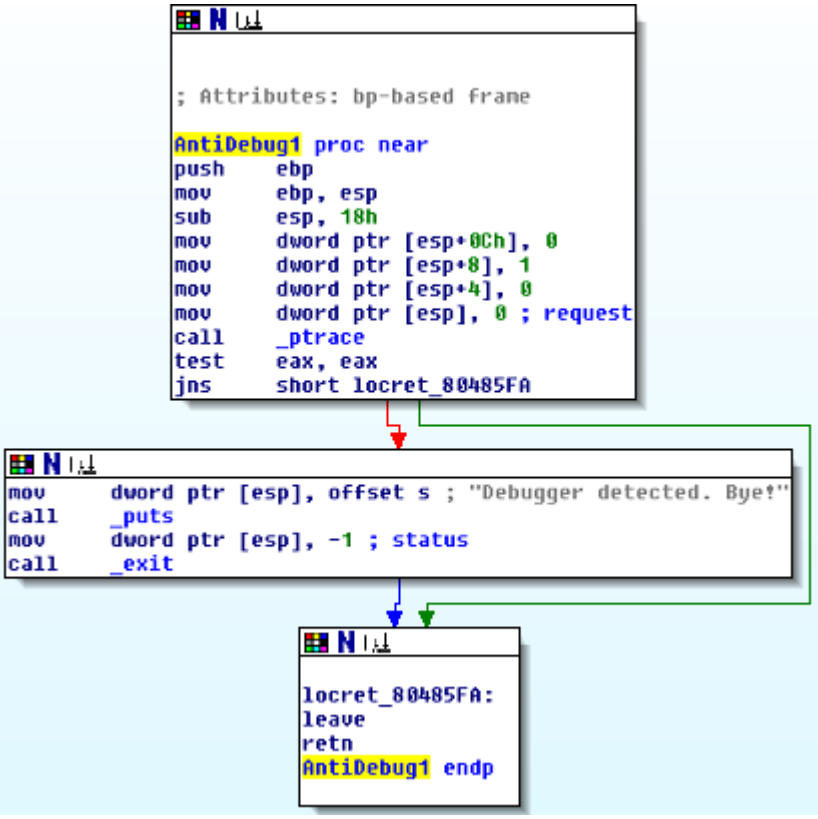
OK, as we know there are some antidebugging tricks…
To see the internal structure of the program I used IDA (because it generates nice graphs). After opening the program in IDA I looked into "strings", to see, where the above string is used (to reach easily the antidebugging procedure). Fortunately, it's not encrypted…



Additionally, what I noticed is the fact, that the BadBoy message (the one about monkeys☺) is not listed with other strings…. Nor I can see any GoodBoy message… So I guess it will be generated during the program execution…
But for now, let's focus on the string about debugger detection…
I clicked on it and what I see is:

```
.rodata:08048AA0 s               db 'Debugger detected. Bye!',0 ; DATA XREF: sub_80485B4+2E↑o
.rodata:08048AA0                                                ; sub_80485FC+16↑o
```

Means, we have two references to this string (two antidebugging procedures?). Let's see them one by one.
This is the first reference (I named it AntiDebug1):



It used the antidebugging technique basing on the **ptrace**.

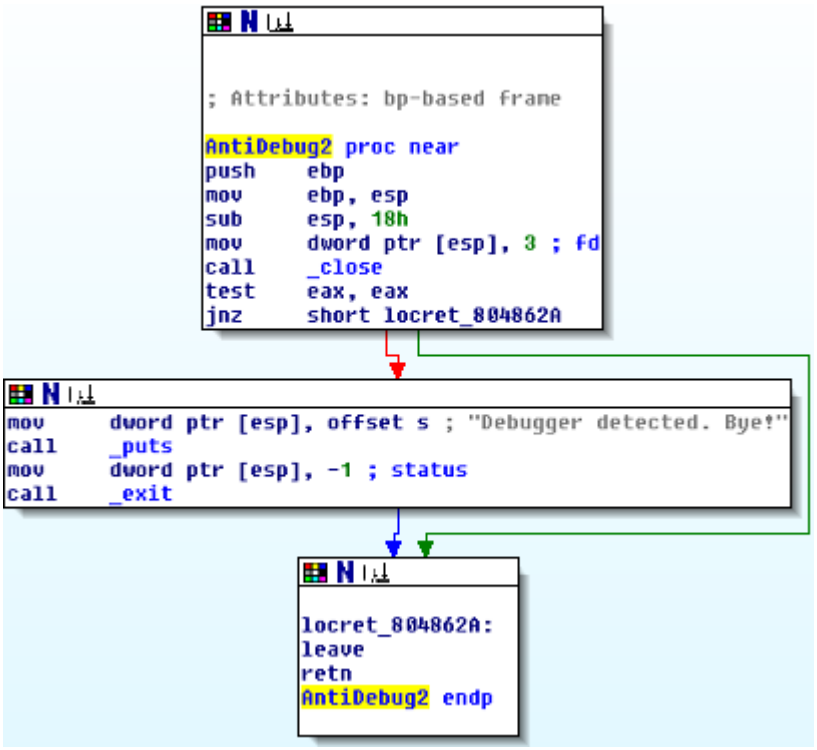
`if (ptrace(PTRACE_TRACEME,0,1,0)<0)` → BadBoy

How it works? The definition of this function looks in a following way:

```
int ptrace(int request, int pid, int addr, int data);

RESULT: 0 when everything went fine,
        -1 in case of error;
```

The error occurs, when the process is already traced – it happens in case if it runs under a debugger.

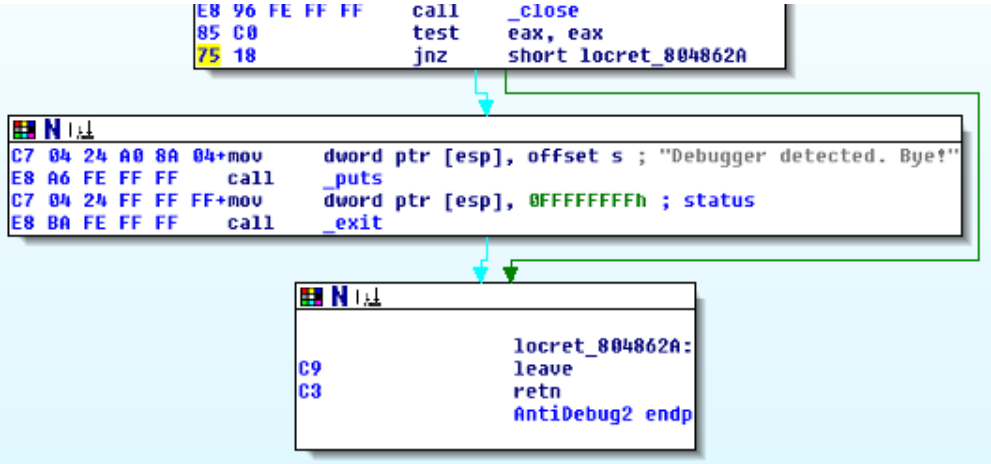
And here's the second (I named it AntiDebug2):



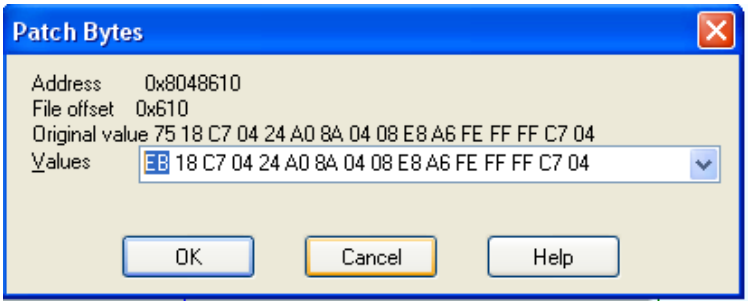
`If(close(3) == 0)` → BadBoy
This trick is based on the fact that gdb opens 3 additional file descriptors (3,4,5) for a debugged process.
The definition of the called function looks in a following way:
`int close(int fildes);`
"Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned"

So, in simple words – if function returns 0, it means that the descriptor we are trying to close was open before. Means, gdb opened it – implies, the process is under debugger.

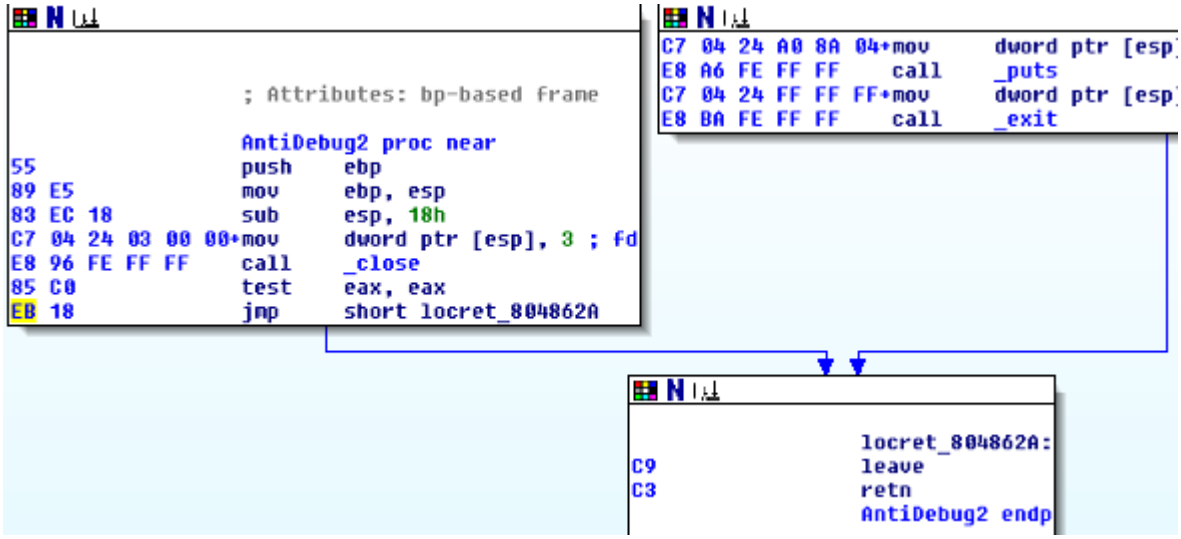
I decided to disable the antidebugging functions by simple patching. Just changing conditional jumps into the unconditional jumps should help. For both cases it looks nearly same, that's why I am showing only one example.
I used a patching tool of IDA (to create a dif file) and hexeditor (to apply changes on the binary).



Unconditional short jump is denoted by the opcode EB.



That's how the function looks after patching. As we see, it skips the part indicating, whether debugger has been detected.



Unfortunately, in IDA I cannot save this changes to binary. So, I produced the dif file, and changed by hand the bytes in given offsets in hexeditor. This is the content of the dif:

```
This difference file is created by The Interactive Disassembler

lincrackme3
000005E0: 79 EB
00000610: 75 EB
```

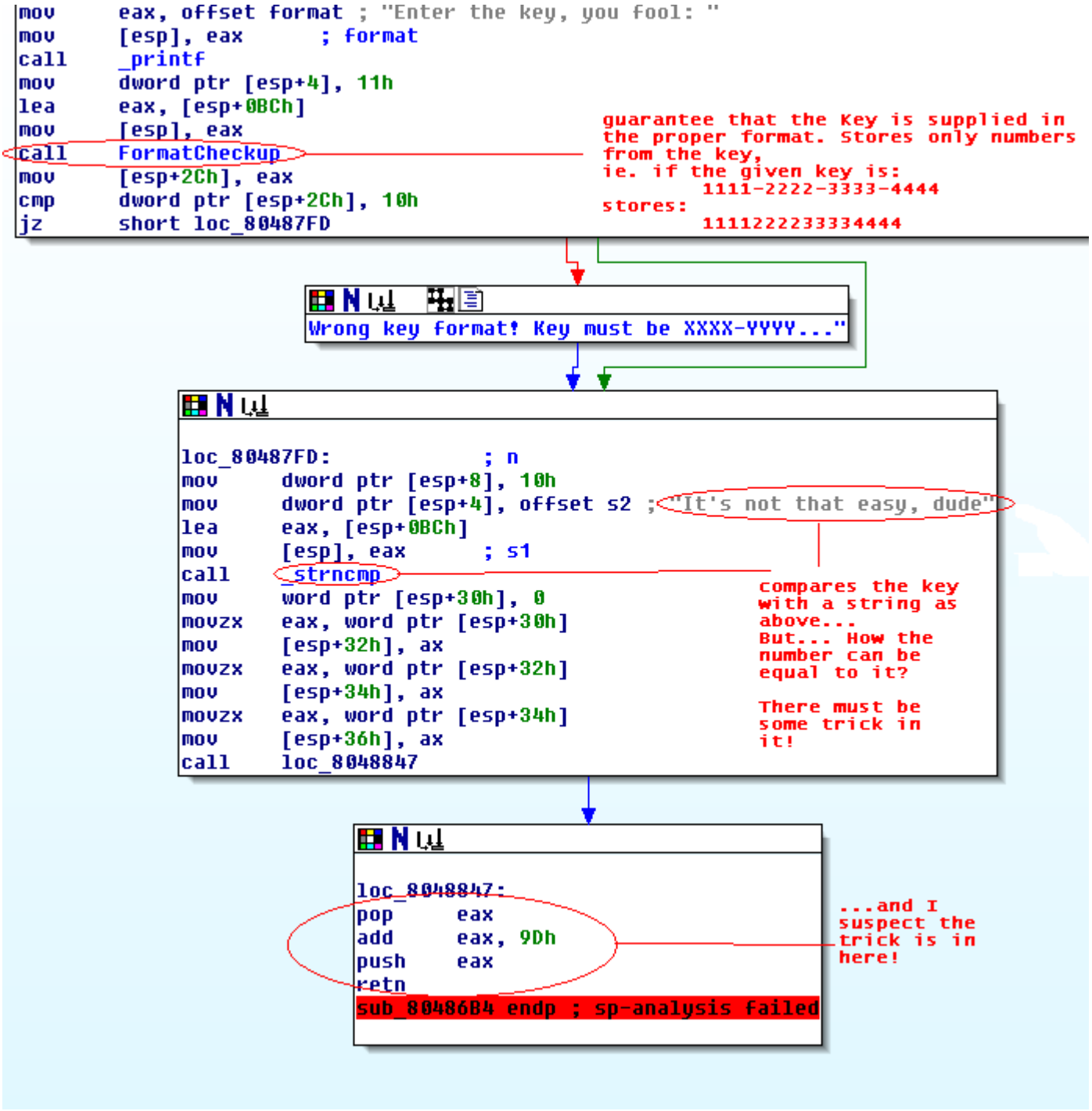
OK, let's try to run the patched version under GDB!

```
(gdb) r
Starting program: /home/hasherezade/l3/lincrackme3-32
(no debugging symbols found)
 ************************************************************************
 * Welcome to lincrackme3! This is a simple linux crackme. The goal     *
 * is to understand the key checking method and write your own keygen   *
 * for it. Keep an eye on antidebugging techniques, because thats the   *
 * reason for this crackmes, knowing them and the way to bypass them.   *
 * Key format: XXXX-YYYY-WWWW-ZZZZ with all numbers. Good luck!         *
 * @author: Adrian adrianbn[at]gmail.com http://securityetalii.es       *
 * ************************************************************************

Enter the key, you fool: 1111-2222-3333-4444
Wrong key! I've seen monkeys smarter than you...

Program exited with code 01.
```

It works! ☺
So, we can go further…

2. Finding the key verification algorithm

To find the procedure where the key is verified, I decided to follow the string “Enter the key, you fool”. It guided me to the following procedure (below the result of my analysis):

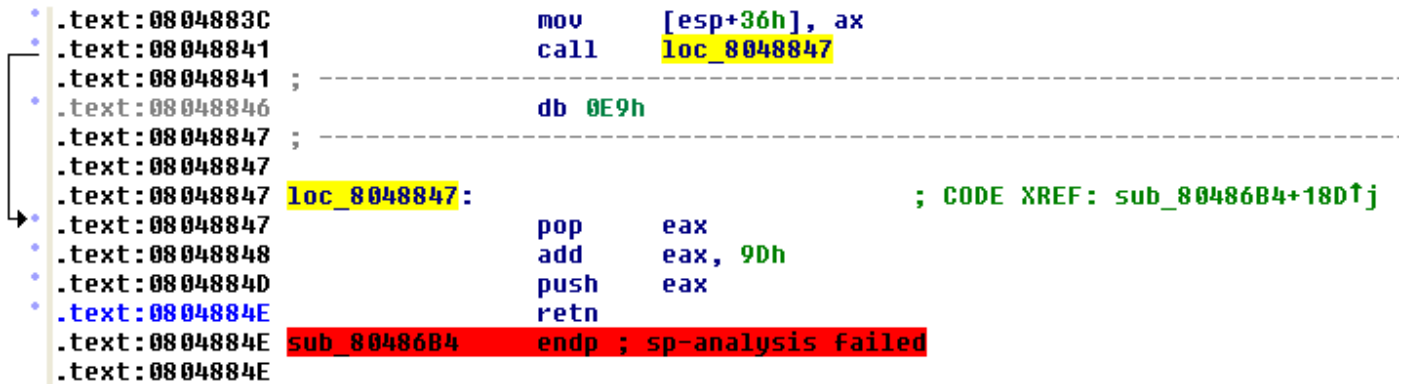


Why I think that **loc_8048847** contain the trick?

I'm just describing:

- When the call is made, the address of the next instruction (which should be executed after the call returns) is pushed on the stack
- we makes call to loc_8048847, and then pushing the address of the next instruction to EAX… Modifying it and pushing again…
- when the function returns, it does not actually go to the next instruction which was originally remembered, but to the overwritten value!

In this case, the offset of the next instruction after **0x8048841** is **0x8048846** (this value will be pushed on the stack)**.** We can see it more clearly on the text view. As we can also notice, this offset refers to some rubbish value, used just for code obfuscation.



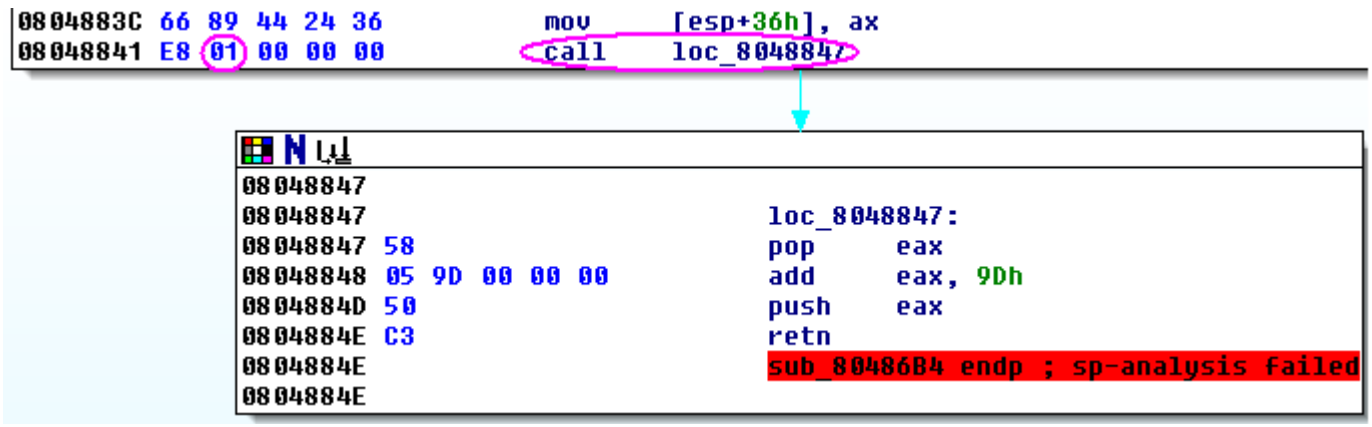
IDA don't know where the flow goes after the **ret,** so it cannot do the static analysis for us… That's why I used gdb at this point.
I set the breakpoint on the address **0x8048847**, and then, instruction by instruction I examined where it follows…

```
(gdb) Breakpoint 1, 0x08048847 in ?? ()
(gdb) info registers
eax            0x0  0
...
 (gdb) ni
0x08048848 in ?? ()
(gdb) info registers
eax            0x8048846    ← as suspected, pointer to next instruction goes to EAX
...
(gdb) ni
0x0804884d in ?? ()
(gdb) info registers
eax            0x80488e3    ← EAX = 0x8048846 + 0x9d
...
(gdb) ni
0x080488e3 in ?? ()  ← as we see, the execution really followed to this line
Let's take a look around this address…
(gdb) disas 0x80488e3 0x80488e3+20
Dump of assembler code from 0x80488e3 to 0x8048915:
0x080488e3 <exit@plt+1023>:      mov    0x28(%esp),%ebx
0x080488e7 <exit@plt+1027>:      call   *%ebx        ← another strange call… now the address in EBX
...
Let's make one step forward and then see what is in EBX…
(gdb) ni
0x080488e7 in ?? ()
(gdb) info registers
eax            0x80488e3   134514915
ecx            0xfffffff8  -24
```

```
edx            0x49 73
ebx            0x80485b4   134514100 ← where it points?

(gdb) disas 0x080485b4 0x080485b4+60
Dump of assembler code from 0x80485b4 to 0x80485f0:
0x080485b4 <exit@plt+208>: push   %ebp
0x080485b5 <exit@plt+209>: mov    %esp,%ebp
0x080485b7 <exit@plt+211>: sub    $0x18,%esp
0x080485ba <exit@plt+214>: movl   $0x0,0xc(%esp)
0x080485c2 <exit@plt+222>: movl   $0x1,0x8(%esp)
0x080485ca <exit@plt+230>: movl   $0x0,0x4(%esp)
0x080485d2 <exit@plt+238>: movl   $0x0,(%esp)
0x080485d9 <exit@plt+245>: call   0x8048464 <ptrace@plt>
0x080485de <exit@plt+250>: test   %eax,%eax
0x080485e0 <exit@plt+252>: jmp    0x80485fa <exit@plt+278> ← do you remember this offset?
0x080485e2 <exit@plt+254>: movl   $0x8048aa0,(%esp)          (5E0 in a raw file)…
0x080485e9 <exit@plt+261>: call   0x80484c4 <puts@plt>       it's the antidebugging function we patched!
0x080485ee <exit@plt+266>: movl   $0xffffffff,(%esp)
End of assembler dump.
...
Let's go further on… After this procedure finishes, execution goes to:
0x080488e9 in ?? ()
Again we can take a look around:
(gdb) disas 0x080488e9 0x080488e9+20
Dump of assembler code from 0x80488e9 to 0x80488fd:
0x080488e9 <exit@plt+1029>:      jmp    0x8048850 <exit@plt+876>
And that's how the function where we jump looks… (I pasted just a beginning)
(gdb) disas 0x08048850 0x08048850+0x60
Dump of assembler code from 0x8048850 to 0x80488b0:
0x08048850 <exit@plt+876>: mov    %eax,%eax
0x08048852 <exit@plt+878>: movl   $0x0,0x24(%esp)
0x0804885a <exit@plt+886>: jmp    0x80488d6 <exit@plt+1010>
0x0804885c <exit@plt+888>: mov    0x24(%esp),%eax
0x08048860 <exit@plt+892>: movzbl 0xbc(%esp,%eax,1),%eax
0x08048868 <exit@plt+900>: cbtw
0x0804886a <exit@plt+902>: add    0x36(%esp),%ax
0x0804886f <exit@plt+907>: sub    $0x30,%eax
0x08048872 <exit@plt+910>: mov    %ax,0x36(%esp)
0x08048877 <exit@plt+915>: mov    0x24(%esp),%eax
0x0804887b <exit@plt+919>: add    $0x4,%eax
0x0804887e <exit@plt+922>: movzbl 0xbc(%esp,%eax,1),%eax
0x08048886 <exit@plt+930>: cbtw
0x08048888 <exit@plt+932>: add    0x34(%esp),%ax
0x0804888d <exit@plt+937>: sub    $0x30,%eax
0x08048890 <exit@plt+940>: mov    %ax,0x34(%esp)
0x08048895 <exit@plt+945>: mov    0x24(%esp),%eax
0x08048899 <exit@plt+949>: add    $0x8,%eax
0x0804889c <exit@plt+952>: movzbl 0xbc(%esp,%eax,1),%eax
0x080488a4 <exit@plt+960>: cbtw
0x080488a6 <exit@plt+962>: add    0x32(%esp),%ax
0x080488ab <exit@plt+967>: sub    $0x30,%eax
0x080488ae <exit@plt+970>: mov    %ax,0x32(%esp)
End of assembler dump.
```

After following with the flow of this function, we can see that it process the key we supplied… It seems that this is the key verification function we are searching for! It will be nice to see this function in IDA in form of a graph… To achieve it I did one trick. I patched the program in such a way, that instead of letting it jump to this strange obfuscated part, I moved the flow directly to the KeyVerification procedure.
More details below:

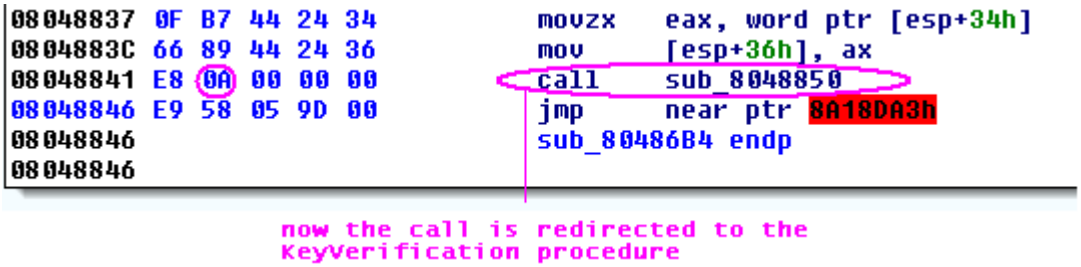
Before patching:



E8 01 00 00 00 – means, that we make a call to the instruction with offset **0x00000001**(mind endians!).
In simple words it means, we make a call which is one byte after the current instruction (we skip this obfuscating byte, which was visible on the text view). As it is shown, it means jump to **0x8048847**. Why?
0x8048841 – current line
0x5 – length of instruction (in bytes)
0x1 – offset
0x8048841+0x5+0x1 = 0x8048847 ☺

Now we want to redirect a call to **0x8048850**, so:
0x8048850 – (0x8048841+0x5) = 0x0a ☺

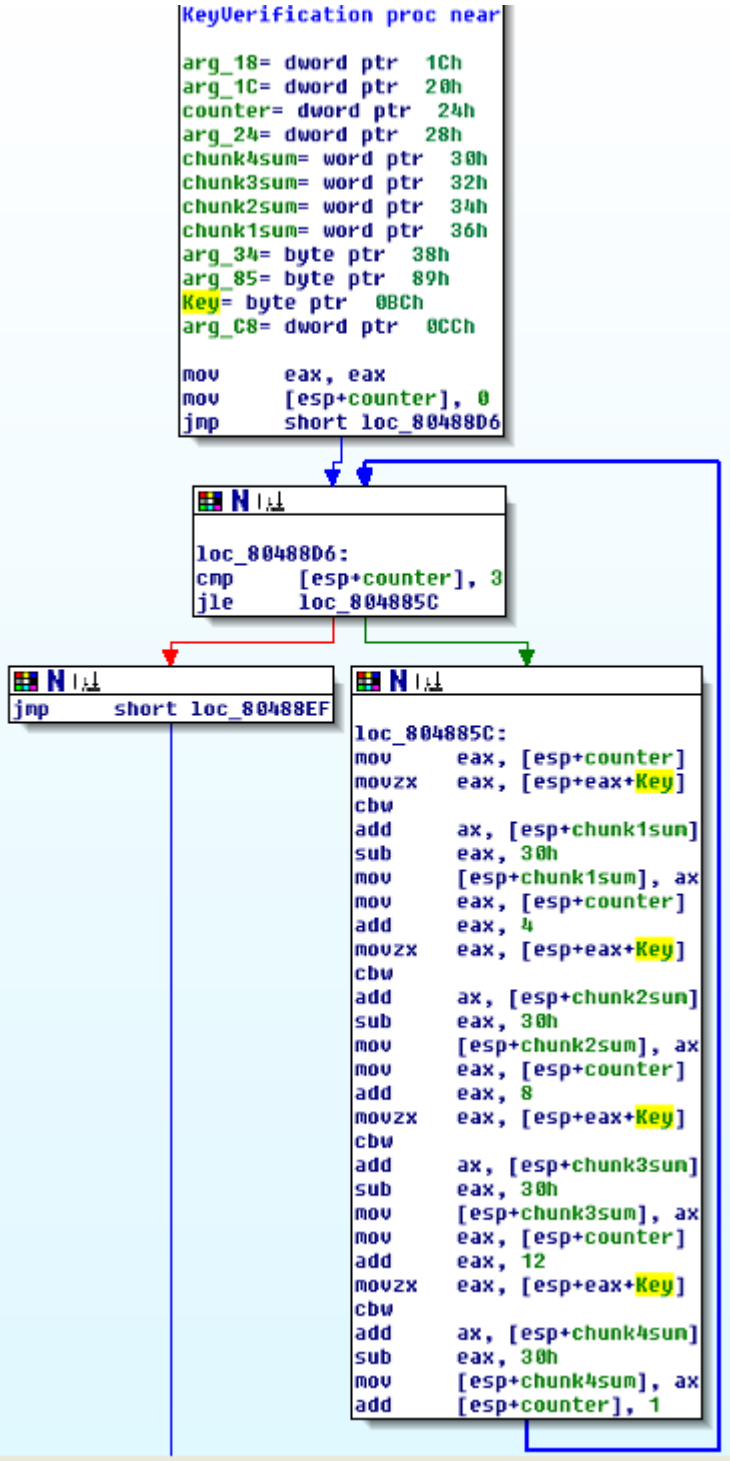
After patching:



Now we can click it and IDA will generate for us a beautiful and easy to understand graph ☺

3. Understanding the algorithm

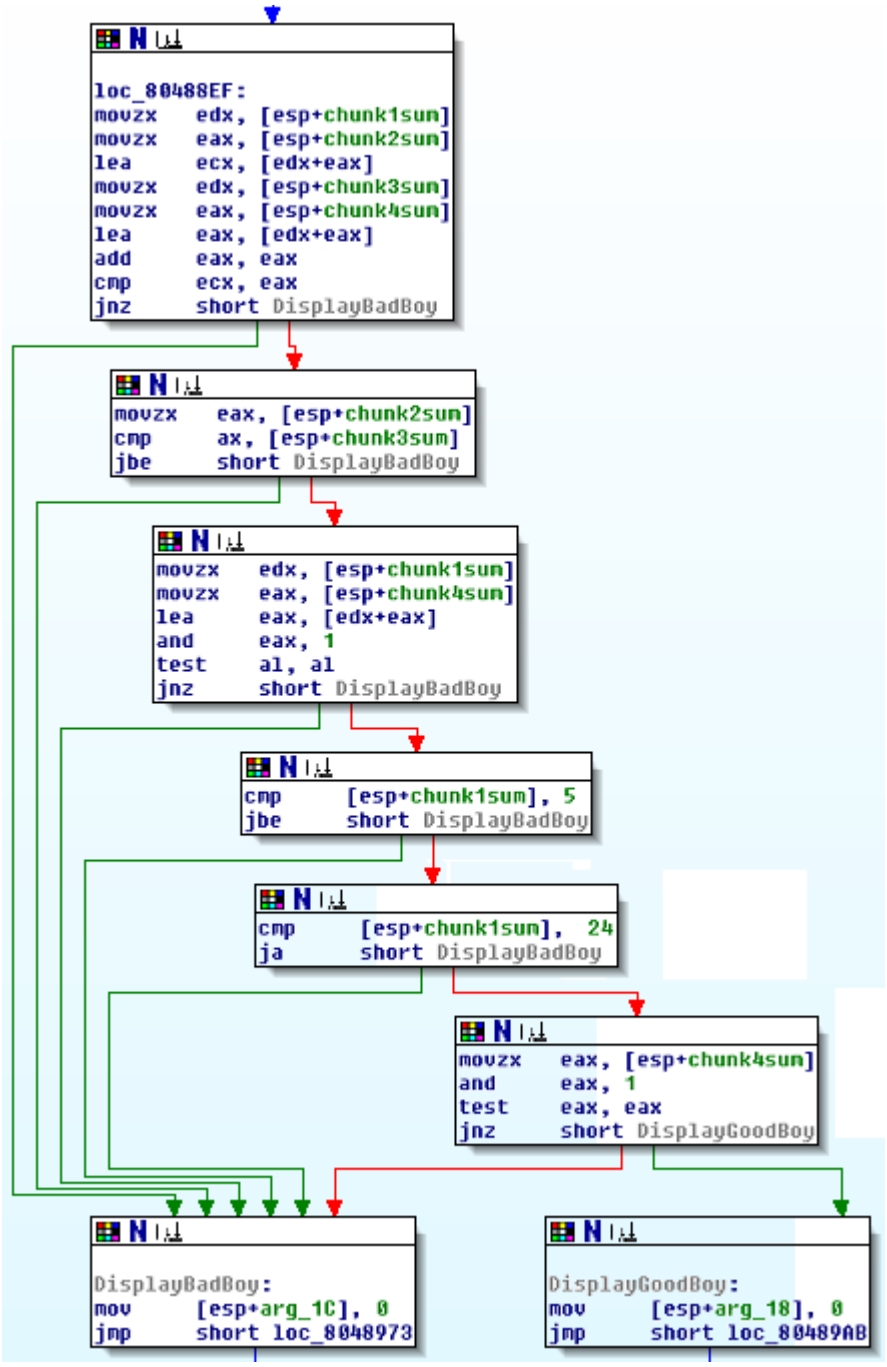
After observing the flow in gdb and reading the graph generated by IDA, I realized that in the first part of the procedure, sums of key's chunks are calculated and stored in some variables




```
chunk1sum=Key[0]-'0'+ Key[1]-'0'+ Key[2]-'0'+ Key[3]-'0';
chunk2sum=Key[4]-'0'+ Key[5]-'0'+ Key[6]-'0'+ Key[7]-'0';
chunk3sum=Key[8]-'0'+ Key[9]-'0'+ Key[10]-'0'+ Key[11]-'0';
chunk4sum=Key[12]-'0'+ Key[13]-'0'+ Key[14]-'0'+ Key[15]-'0';
```

I think this part don't need more detailed explanation ☺

Real verification goes on in the next part. There it is decided, whether to go to a GoodBoy or to a BadBoy ☺
Let's take a look:



GoodBoy is displayed, when the sums of chunks satisfy the following conditions (to shorten the notation I will call the the **sum of the first chunk** - ***sum1*** and so on…)

- **sum4** must be odd
- 5< **sum1**<=24
- **sum4+sum1** – must be even
  - *it means, **sum1** must be odd*
- **sum>sum3**
  - *so, **sum2** > 0 and **sum3**<36*
- **sum1+sum2**==2*(**sum3+sum4**)
  - *all the sums are integers, so **sum2** must be odd as well, so that sum1+sum2 can be divisible by 2*

Additionally:

- Any sum must be a number between 0 (0+0+0+0) and 36 (9+9+9+9)…

II. Keygenning

Having the conditions gathered, we are able to write our own key verifier.
Below there is an example in C++. I used it to verify numbers generated by keygen.

```
bool checkup(int Sum1,int Sum2,int Sum3,int Sum4){
    if(Sum1>36||Sum2>36||Sum3>36||Sum4>36) return false;
    if(Sum1<0||Sum2<0||Sum3<0||Sum4<0) return false;
    //conditions from the keyCheck procedure:
    if(Sum1+Sum2!=(2*(Sum3+Sum4)) ) return false;
    if(Sum2<=Sum3) return false;
    if((Sum1+Sum4)%2 != 0) return false;
    if(Sum1<=5||Sum1>24) return false;
    if (Sum4 %2 == 0) return false;
    return true;
}
```

My idea of generating keygen was based on the bottom-up approach. First I generated sums and then I split each of them into their ingredients.
Let's put together what we now about particular sums:

Sum1	Sum2	Sum3	Sum4
• odd • [7;23] *(we can reject 6 and 24 from [6; 24], because they are even…)*	• odd • [1; 35] *(we can reject 36 from [1; 36], because it's even…)*	[0; ***Sum2*** -1]	• odd • [0; 35] *(we can reject 36 from [1; 36], because it's even…)*

Moreover we have the equation:
sum1+sum2=2*(**sum3+sum4**)

sum1+sum2=**2*sum3**+2*sum4
2*sum3=(sum1+sum2)-2*sum4 *// conditions guarantee that the right side is even*
sum3= (sum1+sum2)/2 - sum4 *//so, we can divide by 2 freely…* ☺

As we see, one more constraint is required: **sum4 >= (sum1+sum2)/2**

The most independence we have in choosing sum1 and sum2, so let's start from generating them:

```
int getSum1(){
    int s1=0;
    while((s1=(rand()%(23-7))+7)%2==0)
    ;
    return s1;
}

int getSum2(){
    int s2=0;
    while((s2=(rand()%(35-1))+1)%2==0)
    ;
    return s2;
}
```

Basing on those values and the known constraints we can generate Sum4:

```
int getSum4(int Sum1,int Sum2){
    int c=((Sum1+Sum2)/2)+1;
    int s4=0;
    while((s4=(rand()%c))%2==0)
    ;
    return s4;
}
```

The whole sum generating function can look like this:

```
bool generateSums(){
    Sum1=getSum1();
    Sum2=getSum2();
    do{
        Sum4=getSum4(Sum1,Sum2);
        Sum3=(Sum1+Sum2)/2-Sum4;
    }while(Sum2<=Sum3);
    return checkup(Sum1, Sum2, Sum3, Sum4);
}
```

Sum3 and Sum4 are interdependent, and Sum3 is in the relationship with Sum2. Because of the random factor involved, I made here a loop, so that we can be sure that we got values satisfying the conditions. At the end there is additional checkup to make sure that the result is valid.

Having those 4 sums, it's very easy to generate several valid keys out of the single set. To split a sum into a chunk of 4 values, I created following function:

```
void fillChunk(char buf[4],int sum){
    for(int i=0;i<4;i++)
        buf[i]='0';
    while(sum>0){
        int i=rand()%4;
        if(buf[i]<'9'){
            buf[i]++;
            sum--;
        }
    }
}
```

I hope now everything is clear ☺
Full source code of the keygen is included in the package, as well as it's compiled versions, for windows and for Linux.
Thanks for reading!