# Tutor3:

Welcome to tutorial 3...
In this tutorial I will try to recapitulate the things we have learned in the first two tutorials...
Also we will be handling a few real world applications...

**Recapitulation:**

What did we learn in the previous tutorials?
That cracking is a way which lets you make programs on your computer behave as you want it to behave ... and not as someone else wants...
That there are several steps to go through while cracking each program which are:

- Watching the standard behavior of the program.
- Disassembling the program.
- Finding Strings in the disassembly that we recognize from phase 1
- Finding the protection routine
- 'Understanding' the jumping mechanism of the protection
- Changing the original program, so that it always behaves as you want it to
- run it and feel the power you have ... The power knowledge gives you :-)

These are all steps which you have to go through... over and over again ... Until you get bored...
Untill you can dream of the sequence of key-strokes you have to perform while patching an exe..
Until you start feeling the protection routine ... It is the only way to become a good cracker...

**The Target:**

So for a little exercise, let's crack a few programs:)
The target is High Logic's Font Creator Program. (http://www.high-logic.com/download.html)

**Step 1: Run the program:**

Run it ...
You will see a cool splash....
then a dialog box ... It says: "This product is not registered"
Telling you how and where to register.... Lower right panel says 'Unregistered'....
That is interesting of course.

Now let's Check out for registration options....
Under help, you can select register...
Try to enter a fake name and serial ... look what it does....
"Registration Failed: Invalid Password ".
That is the exact message we get....
We know enough :-)
Close the program....

**Step 2: Disassemble the program:**

Make a copy of fcp.exe, and call it fcp.org (original)...
Fire up your W32Dasm and open FCP.ORG... It takes a while to decompile this baby...
When W32Dasm is done with disassembling, open the string references list...
(Refs-> String Data Refs or the "strn Ref" button)
Let's search for familiar strings, like: This product is unregistered, Registration Failed,
Thank you for registering....

Indeed, when we scroll down the alphabetically arranged string references, we come across the string: Registration Failed.... Go stand on it ... and double click it...
Now you can close the string references window....

**Step 3: Analyzing the protection routine.... / Understanding the jumping Mechanism...**

When we scroll up a bit, we see another string: "Thank you for registering..." right above this we see a call at line 0048AD23 ... Right after this call, we see a jne command.... go stand on it, and execute the jump ... You see that it lands right on the line which says you are unregistered. Now press the [ret jmp] button to return to the previous place.... We see that if the code does not jump, it thanks you for registering. If it jumps, it shows you, you entered an invalid serial...

We found the code that is executed so we see the dialog box that pops up when we type in an invalid serial.... This code also shows there is a dialog box that pops up when a valid serial is entered.... We want this piece of code to behave as following: Show registered when valid serial is typed and show registered when invalid serial is typed...

Now let's take a closer look at the disassembly....

```
////////////////////// Code snip ////////////////////////////
ADDRESS        MACHINE CODE              ASSEMBLER INSTRUCTIONS

:0048AD23     E8FC90F7FF              call 00403E24
:0048AD28     7524                    jne 0048AD4E
:0048AD2A     6A00                    push 00000000
:0048AD2C     668B0D9CAD4800          mov cx, word ptr [0048AD9C]
:0048AD33     B202                    mov dl, 02

* Possible StringData Ref from Code Obj ->"Thank you for registering the "
                                        ->"Font Creator Program."
                                          |
:0048AD35     B8A8AD4800              mov eax, 0048ADA8
:0048AD3A     E8D97EFBFF              call 00442C18
:0048AD3F     8B45FC                  mov eax, dword ptr [ebp-04]
:0048AD42     C7805001000001000000    mov dword ptr [ebx+00000150], 00000001
:0048AD4C     EB22                    jmp 0048AD70

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
    |:0048AD28(C)
    |
:0048AD4E 6A00                    push 00000000
:0048AD50 668B0D9CAD4800          mov cx, word ptr [0048AD9C]
:0048AD57 B201                    mov dl, 01
* Possible StringData Ref from Code Obj ->"Registration failed: Invalid Password"
                                   |
0048AD59 B8E4AD4800              mov eax, 0048ADE4
////////////////////// Code snip ////////////////////////////
```

A translation that is the same looks like this...

---

```
call IsSerialValid
jne   serial_not_valid
call MessageBox_Thanks_for_registering
jmp   End

serial_not_valid:
call MessageBox_Serial_Invalid

End:
```

---

In plain human language this would be like:

If Serial Is Not Valid, Tell Serial Is Invalid,
else Thank You For Registering...

Since we want it to Thank us for registering always, we make sure the jne Serial_Not_Valid jump is never executed .....
when we do this te pseudocode will look like this:

---

```
call IsSerialValid
XXX   xxxxxxxxxxxxx                    ; serial_not_valid removed ...
call MessageBox_Thanks_for_registering
jmp   End

serial_not_valid:
call MessageBox_Serial_Invalid

 End:
```

---

The call at 0048AD23 is executed, after this, there will never be a chance to jump to the invalid serial messagebox :-)

**Step 4: Changing the original program...**

Same as before: Load FCP.exe in HIEW.... Press enter twice so you get into disassembly mode. push F5 to enter the address ... and when you are there, change the JNE into two NOP commands Now press F9 to save and F10 to exit....

**Step 5: Run the program, and register with any name and serial :-))**
Do you feel it? Do you feel the power it gives you?

---

Extra !

When we were looking through the string references list, we also saw the other unregistered string we had paid attention to ... The "This product is not registered" string we saw in the dialog box that pops up at startup.... Since we are here to learn, and we don't want to miss any opportunity, let's take a closer look at it....

Locate the string again in the String references in hiew...
"This product is not registered." Double click it and you will land in the code below....

```
////////////////////// Code snip ////////////////////////////

ADDRESS        MACHINE CODE          ASSEMBLER INSTRUCTIONS

:00476565      E836DFFAFF          call 004244A0
:0047656A      A10CB54B00          mov eax, dword ptr [004BB50C]
:0047656F      83781800            cmp dword ptr [eax+18], 00000000
:00476573      753B                jne 004765B0

* Possible StringData Ref from Code Obj ->"This product is not registered."
```

```
                                    |
:00476575      BADC694700           mov edx, 004769DC
:0047657A      8B45FC               mov eax, dword ptr [ebp-04]
:0047657D      8B800C020000         mov eax, dword ptr [eax+0000020C]
:00476583      E818DFFAFF           call 004244A0

* Possible StringData Ref from Code Obj ->"Registration fee is US $20 for "
                                        ->"a single license."
                                    |
:00476588      BA046A4700           mov edx, 00476A04
:0047658D      8B45FC               mov eax, dword ptr [ebp-04]
:00476590      8B8010020000         mov eax, dword ptr [eax+00000210]
:00476596      E805DFFAFF           call 004244A0

* Possible StringData Ref from Code Obj ->"Read the helpfile for more
                                      |
:0047659B      BA406A4700           mov edx, 00476A40
:004765A0      8B45FC               mov eax, dword ptr [ebp-04]
:004765A3      8B8024020000         mov eax, dword ptr [eax+00000224]
:004765A9      E8F2DEFAFF           call 004244A0
:004765AE      EB13                 jmp 004765C3

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00476573(C)
|
* Possible StringData Ref from Code Obj ->"This product is registered to:"
                                    |
:004765B0      BA706A4700           mov edx, 00476A70

//////////////////// Code snip ////////////////////////////
```

"Play" again with the jumps and you will see what happens with the jumps...
When you are done jumping and looking a bit at the disassembly, you should be able to see that the disassembly can be brought back to the simple pseudo code below....

---

| CODE | COMMENT |
|---|---|
| Call Is_Program_Registered | |
| cmp memory location , 0000 | ; compare memory location to zero |
| jne Program_Is_Registered | ; jump to program is registered label |
| | |
| Print_Program_Is_Not_Registered | |
| Print_Registration_Is_20$_Only | |
| Print_Read_Help_for_More_Info | |
| jmp Finish | ; jump to where the finish label is |
| | |
| Program_Is_Registered: | ; this is just a label / an address ... |
| Print_Program_IS_Registered | |
| Print_Registered_to | |
| | |
| Finish: | ; this is the finish label ... |

---

This pseudo-code is a lot more readable for us newbies ;-)
Well... what happens is: there is again a check ... If the program is registered...
After that it compares a memory location to zero.... hmmm.. Why would it do something like this? We'll see that in a second ... Then it jumps if this memory location is not equal to zero, and it doesn't jump when it is zero...
What would this mean for us crackers? Yup... this memory location should *always* be unequal to zero :-) That means it should contain a value like 1, 2 or 1000...

not the value 0....

The source would have looked something like this:

```
//////////////////// Code snip ////////////////////////

    BOOL programregistered;          // a boolean variable ...
                                     // these variables contain a true or a
                                     // false value ... ( a 0 or a 1 )

    programregistered = IsProgramRegistered();      // a function call ...
                                     // the function returns its value to
                                     // the boolean variable programregistered

    if (programregistered = TRUE)
       DoNotNag() ;
    else
        DoNag();

//////////////////// Code snip ////////////////////////
```

This piece of code shows us that programs can set a memory location ( the programregistered bool variable is a memory location) to determine the state of your registration....
That is what happens in this piece of code ... And also what happens in the real program...

Now we do understand that the memory location above is equal to 1 if you are registered, and is equal to zero if you are not registered.... To crack this piece of code = to make it always think it is registered, we could change the memory location to 1, so the program always thinks we are registered.... Is this very easy to do? Well.... Not now :-)
I will show it to you in tutor 3...

The easiest would be to modify the "jne Program_Is_Registered" in such a way that the program always thinks we are registered... No matter what the call Is_Program_Registered puts in memory location that has to be equal to something NOT zero...
The easiest way to do this would be to change the jne into a jump instruction.
And this is a piece of cake :-))

Open the program in hiew; Go to the right offset in the program.... Change the jne into a jmp.... What happened?

```
ADDRESS    MACHINE CODE                 ASSEMBLER INSTRUCTIONS

ORIGINAL

00476573: 753B                          jne      .0004765B0   -------- (1)
00476575: BADC694700                    mov       edx,0004769DC ;" Gi_"
0047657A: 8B45FC                        mov       eax,[ebp][-0004]
0047657D: 8B800C020000                  mov       eax,[eax][00000020C]
00476583: E818DFFAFF                    call     .0004244A0   -------- (2)
00476588: BA046A4700                    mov       edx,000476A04 ;" Gj "

PATCHED

00476573: E938000000                    jmp      .0004765B0   -------- (1)
00476578: 47                            inc       edi
00476579: 008B45FC8B80                  add       [ebx][0808BFC45],cl
0047657F: 0C02                          or        al,002 ;" "
00476581: 0000                          add       [eax],al
00476583: E818DFFAFF                    call     .0004244A0   -------- (2)
00476588: BA046A4700                    mov       edx,000476A04 ;" Gj "
```

---

```
The jmp hiew assembled, looks a bit weird ... It needed 3 extra bytes....

753B               jne      .0004765B0 <<-- jne          (2 bytes)(75 3B)
E938000000         jmp      .0004765B0 <<-- assembled jmp(5 bytes)(E9 38 00 00 00 )
```

That is not so nice :-( Now we altered some code we did not want to alter at all...
The mov edx, 0004769DC line of code is gone.... And something else has come in place...
Well ... Actually there is no need to worry....
The code we altered here will *never* be executed.... We made sure that it would never be ;-)
We made sure that anything that comes to line 476573 jumps *DIRECTLY* to line 4765B0 :-))
so the code at the lines between 476578 and 4765B0 are never executed ... Those are the lines
which tell us we are not registered :-)

There may be several things you haven't fully understood but that's not important here, you will
as we do more tutorials together.

---

**What we did in this tutorial?**

- We saw that there are several standard steps that you will have to go through while cracking.
  - -- watching a program in action
  - -- disassembling the program and looking for familiar strings
  - -- finding and analyzing the protection routine of the program
  - -- Modifying the program so it does the thing you want.
- While going through these steps we make extensively use of our Tools: Hiew and WDasm...
  - -- Disassembling and Finding string references. Simulating the jumps which take place during execution of the program.
  - -- Finding the corresponding address in Hiew, assembling assembler instructions in Hiew.
- We also saw that programs can keep a boolean value in memory to determine if you are registered or not.