NOTE: ED!SON is not a PC cracker but he's tutorial is great
for newbies. ED!SON is a United Cracking Force cracker. Windows Cracking
Tutorial v1.o Page 1 of 7
==========
ED!SON's Windows 95 Cracking Tutorial v1.oo
========== CONTENTS
========
1. Introduction to Windows cracking
2. Quick introduction to SoftICE/Win 2.oo 3. Finding registration codes
3.1 Task Lock 3.00 - A simple registration number only protection
3.2 Command Line 95 - Easy name/code registration
4. Making a keymaker for Command Line 95
5. How PUSH and CALL and things really work when the program call a function
6. About Visual Basic programs APPENDIX
========
A. Making SoftICE load symbols
B. Syntax for the functions
C. Where to obtain the softwares
D. Contacting me 1. INTRUDUCTION TO WINDOWS CRACKING
==========
Cracking a Windows program is most often more simple than a program running in Dos. In Windows, it's hard to hide anything from anyone who really looks for information, as long as Windows own functions are used. The first (and often only) tool you need is SoftICE/Win 2.oo, a powerful
debugger from NuMega. Some people find it hard to use, but I will tell you how to do efficient debugging with it, and I hope you'll understand me :-) I have made an Appendix (A) with some SoftICE/Win 2.oo info you should read.
I never had any problems installing SoftICE, so if you have, I'll have to refer to the manual. URLs to all software you need are in Appendix C. - ED!SON, edison@ccnux.utm.my 2. QUICK INTRODUCTION TO SOFTICE/WIN 2.OO
==========
This should be a fairly bad view of the SoftICE screen layout:

```
|--------------------|
|     Registers      |  Use 'R' to edit
|--------------------|
|    Data Window     |  Use 'D' to view an address, 'E' to edit
|--------------------|
|    Code Window     |  Use 'U' to view an address, 'A' to insert asm code
|--------------------|
|   Command Window   |  Type commands and read output here
|--------------------| Other important keys are (in the default key layout):
'H'/F1   - On-line help
F5/Ctr+D - Run
F8       - Step into functions
F10      - Step over functions
F11      - Step out of function
```

3. FINDING REGISTRATION CODES
==========
This is probably the easiest way to practice, to get a shareware program and try to register it.

3.1 Task Lock 3.00 - A simple registration number only protection
==========

This is a simple protection, only a code, that doesn't change. 3.1.1

Examining the program
==========
Is it 16 or 32 bit? Where do I enter registration information? Does the help give me any clue on how the registration works? Go and have a find out before you continue!
....You should be checking now!...Are you checking?...Have you checked?...

OK, now you know it's a 32-bit Windows 95 application, and that you can register the program by entering a single Registration Number in a dialog box that appears when choosing the menu "Register|Register...". You also know, by reading in help, that there are two types of registration: Individual and Site License. So most probable there will be TWO checks for valid codes.

3.1.2 Trap the code routine

The codes are usually entered in normal Windows Edit boxes. To check the code, the program must read the contents of the Edit box with one of these functions:

16-bit	32-bit
GetWindowText	GetWindowTextA, GetWindowTextW
GetDlgItemText	GetDlgItemTextA, GetDlgItemTextW

The last letter of the 32-bit functions tell if the function uses one-byte or double-byte strings. Double-byte code is RARE.
Maybe you got my idea... "If I only could break on GetWindowText" And - you can! But first you must be sure that these symbols are loaded by SoftICE. (See Appendix A) To set up a "trap" (really called breakpoint) in SoftICE, you first enter the
debugger with Ctrl+D, then use the command BPX followed by the name of the function or a memory address. And Task Lock is 32-bit so let's put a breakpoint on GetWindowTextA.
If that doesn't work, we can try the others.

Type like this in SoftICE:

```
:bpx getwindowtexta
```

If you get an error message like "No LDT", make sure you don't run any other applications in the background. I've noticed that Norton Commander/Dos disturbs this function. You can check if you got any breakpoint by listing breakpoints:

```
:bl
```

This would give something like:

```
00)   BPX USER32!GetWindowTextA C=01
```

To get out of SoftICE, you press Ctrl+D again Well, anyway, you have set your breakpoint
that will trap any call to GetWindowTextA. Now let's try to enter some value in the registration number
field and press OK... You press OK... and you just get a stupid message box telling you your code was wrong. So it wasn't GetWindowTextA... Let's try GetDlgItemTextA.

First we erase the old breakpoint:

```
:bc 0
```

(0 means number of breakpoint in the breakpoint list)
And now set the new one:

```
:bpx getdlgitemtexta  Let's try again...
```

3.1.3 In the debugger

Wow! It worked! You're now inside SoftICE, at the place where the function GetDlgItemTextA starts. To jump to wherever it was called from, press the key F11. You are now inside SGLSET.EXE, if you're not sure, look on the line between the code and the command window, it should look something like this:

```
----------SGLSET!.text+1B13----------
```

You can also disable the breakpoint now by doing this:

:bd 0 To enable it later if you want to run again do like this then:

:be 0 The first line in the code window looks like this:

```
CALL    [USER32!GetDlgItemTextA]
```

To see the lines above, press Ctrl+Up arrow a few times, until you see the lines below. If you don't know anything about Assembler, I have added comments to the lines.

```
        RET                                     ; End of function
        PUSH    EBP                             ; Beginning of function
        MOV     EBP, ESP                        ; ...
        SUB     ESP, 0000009C                   ; ...
        PUSH    ESI                             ; ...
    >   LEA     EAX, [EBP-34]                   ; EAX = EBP-34
        PUSH    EDI                             ; ...
        MOVE    ESI, ECX                        ; ...
        PUSH    32                              ; Save: Maximum size of string
    >   PUSH    EAX                             ; Save: Address of text buffer
        PUSH    000003F4                        ; Save: Identifier of control
        PUSH    DWORD PTR [ESI+1C]              ; Save: Handle of dialog box
        CALL    [USER32!GetDlgItemTextA]        ; Get the text
```

The PUSH commands means save the values for later use. I have marked the important lines with a '>' char. By looking at this we know that the address to the text buffer was stored in EAX, and that EAX was EBP-34. So now we look at EBP-34 like this:

:d ebp-3 You should now be able to read what you entered if you look in the data window.

Now we have to find a place where your code is matched with a working one. So we step through the program one instruction at a time with F10 until we find something about EBP-34... You don't step for very long until this code pops up:

```
    >   LEA     EAX, [EBP+FFFFFF64]             ; EAX = EBP-9C
        LEA     ECX, [EBP-34]                   ; ECX = EBP-34
        PUSH    EAX                             ; Save: EAX
        PUSH    ECX                             ; Save: ECX
    >   CALL    00403DD0                        ; Call a function
        ADD     ESP, 08                         ; Erase saved information
        TEST    EAX, EAX                        ; Check function return
        JNZ     00402BC0                        ; Jump if not zero
```

To me, this looks directly like a string compare function. They work like this:
* Input two strings, return zero if equal, otherwise non-zero And why should the program compare a string with the one you entered? To see if it was valid! (As you probably already have figured out) Well, what is hiding behind the address [EBP+FFFFFF64] then? SoftICE doesn't handle negative numbers very well, so to find out the real value of this you do this calculation:

100000000 - FFFFFF64 = 9C You can do the calculation in SoftICE like this:

:? 0-FFFFFF64

The number 100000000 is too big for SoftICE, but on the other hand it gives the same result.
And now... time to look what hides behind EBP-9C... Do like this:
:d ebp-9c

The data window will now show a long row of numbers - the code! But remember what I said earlier... two types of registration gives two codes... so after you've written down the code you got, we continue stepping width F10... We come to this piece of code:

```
    >   LEA     EAX, [EBP-68]                   ; EAX = EBP-68
        LEA     ECX, [EBP-34]                   ; ECX = EBP-34
        PUSH    EAX                             ; Save: EAX
        PUSH    ECX                             ; Save: ECX
    >   CALL    00403DD0                        ; Call the function again
        ADD     ESP, 08                         ; Erase saved information
        TEST    EAX, EAX                        ; Check function return
        JNZ     00402BFF                        ; Jump if not zero
```

And what can you find at the address EBP-68? Well... another registration code!

```
:d ebp-68
```

That's it... I hope everything worked!

3.2 Command Line 95 - Easy name/code registration, and we make a keymaker

This is a nice sample program, with a very easy code calculation.

3.1.1 Examining the program

You examine the program and you see it's a 32-bit application, demanding Name and Code in the registration dialog.
So let's start!

3.1.2 Trap the code routine

We do as with TaskLock - we set breakpoints. We can set breakpoints on both of the two most probable functions: GetWindowTextA and GetDlgItemTextA. Press Ctrl+D to make SoftICE show up, and then:

```
:bpx getwindowtexta

:bpx getdlgitemtexta
```

Now go to the registration dialog, and enter a name and some number (an integer is the most usual code). I wrote like this, and pressed OK...

```
 Name:   ED!SON '96         Code:   12345
```

The program stopped at GetDlgItemTextA. Just like with TaskLock, we press F11 to return to the calling function. We scroll upwards with Ctrl+Up and the call looks like this:

```
MOV     ESI, [ESP+0C]
PUSH    1E                              ; Maximum length
PUSH    0040A680                        ; Address to buffer
PUSH    000003ED                        ; Control handle
PUSH    ESI                             ; Dialog handle
CALL    [User32!GetDlgItemTextA]
```

The number 40A680 looks interesting to us, so we check that address:

```
:d 40a680
```

And what shows up in the data window, if not the name we entered. Well, we look below the above piece of code, and it says:

```
        PUSH    00                              ; (not interesting)
        PUSH    00                              ; (not interesting)
        PUSH    000003F6                        ; Control handle
        MOV     EDI, 0040A680                   ; Save address to buffer
        PUSH    ESI                             ; Dialog handle
        CALL    [User32!GetDlgItemInt]
```

GetDlgItemInt is similar to GetDlgItemText, but it returns an integer from the text box. It is returned in EAX, so we step past these instructions, and look in the registers window... For me it says:

```
        EAX=00003039 And what is hex 3039? Type:

        :? 3039
```

And you get this:

```
        00003039  0000012345  "09"
        ^ hex     ^ dec       ^ ascii And, as you can see (and already had
```

guessed), it shows the code you wrote.
Ok,what now? Let's look at the code that follows, first the return code is saved:

```
        MOV     [0040A548], EAX                 ; Save return code
        MOV     EDX, EAX                        ; Put return code in DX too
```

3.1.3 Calculating the code
==========

Then the code is calculated!

```
        MOV     ECX, FFFFFFFF                   ; These rows calc string length
        SUB     EAX, EAX                        ; .
        REPNZ SCASB                             ; .
        NOT     ECX                             ; .
        DEC     ECX                             ; ECX now contains the length
        MOVSX   EAX, BYTE PTR [0040A680]        ; Get byte at 40A680
        IMUL    ECX, EAX                        ; ECX = ECX * EAX
        SHL     ECX, 0A                         ; Shift left 0A steps
        ADD     ECX, 0002F8CC                   ; Add 2f8cc to the result
        MOV     [0040A664], ECX And validated...
        CMP     ECX, EDX                        ; Compare codes
        JZ      00402DA6                        ; If equal, jump...
```

When you have stepped to the comparison of the codes, you can check what your code REALLY should have been:

```
         :? ecx
```

For me this gave:

```
        000DC0CC  0000901324
```

This means that the right code for me is 901324. So press F5 or Ctrl+D to let it run, and try again, with the RIGHT code, but in decimal form. It will work!

4. MAKING A KEYMAKER FOR COMMAND LINE 95
==========

We look at the calculation of the code above, and translate it to C. We make this very simple description of how the code is calculated:

```
    code = ( (uppercase_first_char * length_of_string) << 0x0A) + 0x2f8cc
```

; Note (1): One thing not to forget is that all chars are converted to uppercase when you enter them in the text box, so we have to do the same.

Note (2): "<< 0x0A" means "multiply with 2^10" A whole program in C could look like this:

```
#include
#include
        int main() {
                unsigned long code;
                unsigned char buffer[0x1e];
printf("CommandLine95 Keymaker by ED!SON '96\n");
                printf("Enter name:   ");
                gets(buffer);                   strupr(buffer);
                code = ( ((unsigned long)buffer[0] *
                        (unsigned long)strlen(buffer))
                        << 0x0A) + 0x2f8cc;
               printf("Your code is: %lu", code);                  return 0;
} Enjoy!
```

4. HOW PUSH AND CALL AND THINGS REALLY WORK WHEN THE PROGRAM CALL A FUNCTION

We look at this piece of code from TaskLock again:

```
        PUSH    32                              ; Save: Maximum size of string
        PUSH    EAX                             ; Save: Address of text buffer
        PUSH    000003F4                        ; Save: Identifier of control
        PUSH    DWORD PTR [ESI+1C]              ; Save: Handle of dialog box
CALL    [USER32!GetDlgItemTextA]        ; Get the text If you call this from a C
program, the call would look like this:   GetDlgItemTextA(hwndDlg, 0x3F4,
buffer, 0x32);
                  ^ [ESI+1C]      ^ EAX PUSH stores data on something called the
stack. This results in that each PUSH
put a new piece of data on top of the stack, and the function then checks what
is lying on the stack and use it to do whatever it's supposed to.
```

5. ABOUT VISUAL BASIC PROGRAMS

The Visual Basic .EXE files are not real, compiled EXEs. They just contain code to call VBRUNxxx.DLL, which reads data from the EXE to run the program. This is also why Visual Basic programs are so slow. And when the EXE files are not real, you can't disassemble them, you just find the call to the DLL and a lot of garbage, and when you debug, you end up in the DLL too.
The solution is a decompiler. There is a decompiler for Visual Basic 2 & 3, made by someone called DoDi. It is shareware and available on the net. (See Appendix C) In Windows 95, there are Visual Basic version 4 32-bit apps, and for them there is no decompiler I know of, although I wish there was.
Note: No real (or bright) programmer makes programs in Basic.

A. MAKING SOFTICE LOAD SYMBOLS

To check if SoftICE has loaded the symbols for GetWindowText, you enter SoftICE by pressing Ctrl+D and then write like this: :exp getwindowtext
If you don't get all the GetWindowText functions listed, you need to edit \SIW95\WINICE.DAT by removing the comment chars (';') from some of the 'exp=' lines that follows this text: "Examples of export symbols that can be included for chicago" near the end of the file. You can remove comment chars from all of the lines, or to save memory, on just these files: kernel32.dll, user32.dll, gdi32.dll, which are the most important ones.
When you're ready editing, you'll have to reboot the computer to make it work.

B. Syntax for the functions

It's always much easier to understand the function calls we talk about when you have the declarations, so here we go: int GetWindowText(int windowhandle, char *buffer, int maxlen);
int GetDlgItemText(int dialoghandle, int controlid, char *buffer, int maxlen);
int GetDlgItemInt(int dialoghandle, int controlid, int *flag, int type); For a more detailed description of the functions, check a Windows/Win32 programming reference.

C. WHERE TO OBTAIN THE SOFTWARES
==========
CRACKING TOOLS
 SoftICE/Win 2.oo: http://www.geocities.com/SoHo/2680/cracking.html
 VB Decompiler: ftp://ftp.sn.no/user/balchen/vb/decompiler/ SAMPLE PROGRAMS
 TaskLock: http://users.aol.com/Sajernigan/sgllck30.zip CommandLine
95: ftp://ftp.winsite.com/pub/pc/win95/miscutil/cline95.zip

D. CONTACTING ME
==========
On IRC (EFNet): In #Ucf2000, #Cracking
By e-mail: edison@ccnux.utm.my
On my homepage: http://www.geocities.com/SoHo/2680/cracking.html debugger allows you to break execution of a program, and step through the code instruction by instruction.

