Cracking Oxford Advanced Learner's Dictionary (CD-COPS 1.8)

by

macilaci

Introduction

There‘s always some inroduction. This time it is about a CD protection. In general I don‘t like them. Most of these copy – schemes are defeated in times when the burners are burning RAW mode. One may think, that there‘s no effective way to protect their products.

Tools used

CDCops when not using Icedump disables softice’s keyboard and inside code is a link of checksums, so using breakpoint on execution makes things difficult. Advices: bpm, bpr, bpx on API+some bytes.

Essays: mclallo‘s and Laptonic‘s both concerning CDCops protection in earlier vesions
Disassemblers: IDA, WDasm
Debuggers: WinICE, Icedump
Other tools: Wdump 95
Assembler: Masm

These and many other tools can be found on various sites. You can use search engine to find them (www.google.com, www.altavista.com).

The essay

I doubt You will find this CD on some warez sites, since it is a educational CD. I just wanted to make a copy of this disc and I realised that copy protection is not working with my copy. I begun searching the web: found some silly information, that this protection measures the angle between first and last sector on the CD, does the encryption test and refuses or accepts the CD. Some people claim that these CD‘s are copyable by certain CD-R brands like Kodak Gold. I haven‘t tried this possibility. So my goal was to make this application working. The fig.1 shows the rejected copy. We will talk abou the machine code later.

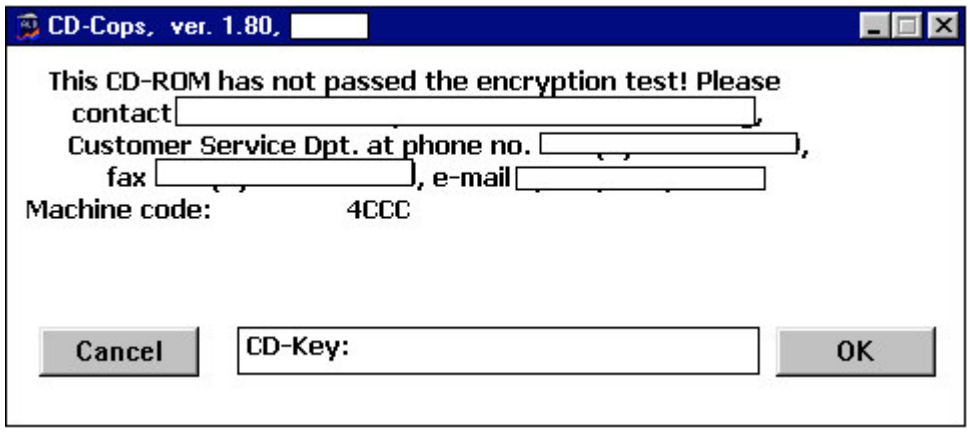

Fig.1

Concerning the above mentioned essays I looked at the executables and my surprise was that the qz_ executable on the CD was an visual basic application. This could mean, that the program each time it runs, it decrypts itself with the given key. This key has nothing to do with the entered product number. The CD product number or whatever call they it is just a number for a given CD to pass the encryption test.

Eleven encrypted sections, who wants more?

Looking at the main loader executable within Wdasm I wondered how are times changing. At the version 1.8 there are eleven references to CD-Cops Ord3 call. Each call has behind himself one encrypted section of data.

With a bunch of bpr's (see softice's manual) I started looking at these bytes. This data section is decrypted by the dll with rotating key, which is not depending on the data itself. Each section has two checksums, first is the checksum of the data itself, second is the checksum of the checksum. The checksums are coputed many times within the dll and the main executable, so patching the executable will result in an ugly crash within INT31. The execution simply jumps through some calls and operations to be made. Using the DPMI services also makes life difficult on the emulator like VMWare or VirtualPC.

So patching the executable isn't good way, even when all to do is to patch few bytes inside the main file as you will see below.

The Registry Story

As I tried to figure out more on this program I was experimenting with the program. Just doing simple CTRL+D within the CD-ROM measuring resulted into a running program. I wondered how could be this possible. Of course you'll have to enter a key that starts the 'encryption test'. Simple delay between reading from CD-ROM and the main program tricked to run the program. After that the application stores its information within registry in these keys (in the HKLM too):

```
HKEY_CLASSES_ROOT\OXFORD___ALD002OU_2241000
@="AABBCCDD/AABBCCDD"

HKEY_CLASSES_ROOT\OXFORD___ALD002OU_2241000.CRC
@="06B8BE09“
```

As it turns out, the crc key is not life important, so the application runs without it. The HKLM keys are just in case of data lost or something like that. The most important is the first mentioned key. I tried it to copy to another machine, but that simply refused the key as in fig.1 – the CD passed not the encryption test. So this key must be machine dependent.

Dumping and code understanding

First we have to look at the executable – it is a 16-bit application. There aren't many sixteen bit dumpers for Windows 9x, so I personally tried the Wdump v2.10 (fig.2). Use of this dumper is quite easy. It allocates memory space and associates this memory space to file. Using Softice's move (m) command it is possible to move code pieces to this memory space and then save it to file. Beware: when starting your debuggnig session, always start Icedump because of anti-Softice code disabling keyboard.

Fig.2

The first reference to CDCops is at 0001:0fb7:

```
cseg01:0FB7                 call    CDCOPS_3
cseg01:0FBC                 db  6Dh
cseg01:0FBD                 db  24h
cseg01:0FBE                 db    9 ;
```

So lets do a break on the 0001:0fb7. First do a break on the window‘s procedure at 0001:0e30.
Tip: Use winice‘s log capabilities to locate the segment 01 of the executable within memory space.
Example (from winice history log):

```
WINICE: Load16  Sel=555F Seg=0001 Mod=WINASM   - here we go
WINICE: Load16  Sel=53E7 Seg=0002 Mod=WINASM
WINICE: StartDLL  CSIP=4DEF:0B6F Mod=CDCOPS
Break due to BPMB #555F:00000FDC X DR3   - I‘ve already set this breakpoint
  MSR LastBranchFromIp=00000A62
    MSR LastBranchToIp=00000A6A
:?1192-fb7                                  -how long is our section?
000001DB  0000000475  "□Ű"
:? 13c8-fb7
00000411  0000001041  "¯ł"                  -better get more
:m 555f:0fb7 l 411 030:82f0e000             -move it to Wdump‘s memory
space
:d 0030:82f0e000                        -let me see if it is there
```

Bold marked string is what I wrote in winice. So we have a new file with 411 bytes inside. Start up hexeditor and paste that code into appropriate space within the executable. When done, start up IDA and look around.

So this way I‘ve got four new sections inside my executable. Next we will explore the registry values. Doing a bpx on **shell!regqueryvalue** function will show us how many times it reads registry. I explored the parameters and values and searched for our favourite registry key. At the second time when the breakpoint occurs, you will get to this routine:

```
cseg01:131E                push    26Dh
cseg01:1321                push    ds
cseg01:1322                push    3E5Eh
cseg01:1325                push    ds
cseg01:1326                push    2B8h
cseg01:1329                call    dword ptr ds:0EA6h ;shell!regqueryvalue
cseg01:132D                or      ax, dx
cseg01:132F                jnz     loc_0_13C8
cseg01:1333                mov     ah, 0FFh
cseg01:1335                mov     si, 3E5Eh
cseg01:1338                mov     di, ds
cseg01:133A                mov     es, di
cseg01:133C                mov     di, 3E4Ah
cseg01:133F                inc     ah
cseg01:1341                cld
cseg01:1342                lodsb                   ;load the string
cseg01:1343                cmp     al, 2Fh ; '/'    ;look for slash
cseg01:1345                jz      loc_0_1338
cseg01:1347                stosb              ;store it to new location
cseg01:1348                or      al, al
cseg01:134A                jnz     loc_0_1342
cseg01:134C                or      ah, ah
cseg01:134E                jz      loc_0_13C0
cseg01:1350                mov     si, 3E4Ah
cseg01:1353                mov     cx, 8
cseg01:1356                lodsb
cseg01:1357                cmp     al, 61h ; 'a'
cseg01:1359                jb      loc_0_135D
.
.
.
cseg01:136D                shl     edx, 4     ;get the hex string to edx
cseg01:1371                or      dl, al
cseg01:1373                loop    loc_0_1356       ;got it all?
cseg01:1375                lodsb
cseg01:1376                or      al, al
cseg01:1378                jnz     loc_0_13C8
cseg01:137A                mov     cx, 10h    ;we will do it ten times
cseg01:137D                xor     ax, ax
cseg01:137F                xor     bx, bx     ;zero ax and bx
cseg01:1381                shr     edx, 1     ;shift right edx
cseg01:1384                rcr     bx, 1      ;rotate through carry flag
cseg01:1386                shr     edx, 1     ;shift right edx
cseg01:1389                rcl     ax, 1      ;rotate through carry flag
cseg01:138B                loop    loc_0_1381  ;next man
cseg01:138D                sub     ax, ds:3DFFh  ;sutract with key1
cseg01:1391                xor     ax, ds:2B6h   ;xor with key2
cseg01:1395                add     bx, ds:3DFFh
cseg01:1399                xor     bx, ds:2B6h
cseg01:139D                cmp     ax, bx          ; is that code valid?
cseg01:139F                jnz     loc_0_13C8       ;if no then jump
cseg01:13A1                mov     word ptr ds:3E13h, 3 ; yes, it is...
cseg01:13A7                mov     ds:187Fh, al ;store the computed
machine code low byte
```

```
cseg01:13AA                 mov     ds:1881h, ah ;store the computed
machine code high byte
cseg01:13AE                 xor     al, al       ;keep execution
cseg01:13B0                 mov     ds:1883h, al
```

This subroutine computes the machinecode key (my was 4CCC) from the registry value. The compare key is necessary to be sure that the key was computed using a given algorithm (rotate through carry). Computed machine code is then stored for latter use by comparation routine. With bpm on the above addresses we will get to the second jump where the machine code is compared to the real machine code:

```
4D07:2550  A07F18            MOV       AL,[187F] ;get low byte
4D07:2553  8A268118          MOV       AH,[1881] ;get high byte
4D07:2557  3B062A3E          CMP       AX,[3E2A] ;compare to real code
4D07:255B  0F849400          JZ        25F3      ;if good then jump
4D07:255F  B104              MOV       CL,04     ;bad guy
4D07:2561  E9D700            JMP       263B
```

In the program are now two jumps deciding whether the program is running on the good machine or not. When the first is not set and the second is set, the program continues running no matter what registry values are inside windows. I was trying to modify the jumps doing a bunch of bprs over the encrypted code, but a lot of checksums and security code gave me a better idea of defeating this protection scheme. Looking after the 3e2a memory area lead me to this routine:

```
cseg01:355C                 push    es
cseg01:355D                 cld
cseg01:355E                 mov     ax, 2
cseg01:3561                 mov     bx, 0FFFFh
cseg01:3564                 int     31h  ; DPMI Services   ax=func xxxxh
cseg01:3564                          ; SEGMENT TO DESCRIPTOR
cseg01:3564                          ; BX = real mode segment
cseg01:3564                          ; Return: CF set on error
                                   ; CF clear if successful, AX = selector
                                   corresponding to real mode segment (64K
                                   limit)
cseg01:3566                 jb      loc_0_4240
cseg01:356A                 mov     es, ax          ; read bios date
cseg01:356C                 mov     cx, 5
cseg01:356F                 mov     si, 5
cseg01:3572                 mov     dx, 5873h       ;set some initial value
cseg01:3575                 mov     ax, es:[si]     ;get the date string
cseg01:3578                 inc     si
cseg01:3579                 inc     si
cseg01:357A                 xor     dx, ax
cseg01:357C                 shr     dx, 1
cseg01:357E                 add     dx, ax
cseg01:3580                 loop    loc_0_3575      ;compute machinecode
cseg01:3582                 and     dx, 0FEFEh
cseg01:3586                 mov     word_429_3E2A, dx ;store at 3e2a
cseg01:358A                 pop     es
cseg01:358B                 retn
```

Taking look at 356a on the es:[si] address told me that the machine code was computed from bios date. So modifying the bios date would solve the problem...

BIOS Flash or what?

So the above things gave me the idea to compute the machine code and store it in the registry. Below I will provide the source code for this utility (Compiled with masm32 as a win32 *command line* utility):

```
; ########################################################################

      .386
      .model flat, stdcall
      option casemap :none   ; case sensitive

; ########################################################################

      include \masm32\include\windows.inc

      include \masm32\include\user32.inc
      include \masm32\include\kernel32.inc
      include \masm32\include\masm32.inc
      include \masm32\include\advapi32.inc

      includelib \masm32\lib\advapi32.lib
      includelib \masm32\lib\user32.lib
      includelib \masm32\lib\kernel32.lib
      includelib \masm32\lib\masm32.lib

    ; ------------
    ; Local macros - used from masm32 examples
    ; ------------
      print MACRO Quoted_Text:VARARG
        LOCAL Txt
          .data
            Txt db Quoted_Text,0
          .code
        invoke StdOut,ADDR Txt
      ENDM

      input MACRO Quoted_Prompt_Text:VARARG
        LOCAL Txt
        LOCAL Buffer
          .data
            Txt db Quoted_Prompt_Text,0
            Buffer db 128 dup(?)
          .code
        invoke StdOut,ADDR Txt
        invoke StdIn,ADDR Buffer,LENGTHOF Buffer
        mov eax, offset Buffer
      ENDM

      cls MACRO
        invoke ClearScreen
      ENDM

      Main   PROTO

; ########################################################################

    .data
      key1        dw 0BBB5h
      key2        dw 0C7DAh
      Buffer2 db 128 dup(0)
      Regkey      db "OXFORD___ALD002OU_2241000\",0
      Fixed       db " Your computer is now ok ",0
 ; ########################################################################

    .code

    start:
      invoke Main
      invoke ExitProcess,0
```

```
; ########################################################################
sub_0_1412 proc near                ;part of converting routine from edx to ascII string
                and     al, 0Fh
                add     al, 90h
                daa
                adc     al, 40h
                daa
                stosb
                retn
sub_0_1412 endp


sub_0_140A proc near                ;part of converting routine from edx to ascII string
                push    ax
                shr     al, 4
                call    sub_0_1412
                pop     ax
                and     al, 0Fh
                add     al, 90h
                daa
                adc     al, 40h
                daa
                stosb
                retn
sub_0_140A endp

sub_0_1403 proc near                ;part of converting routine from edx to ascII string
              xchg    al, ah
              call    sub_0_140A
              xchg    al, ah
              push    ax
              shr     al, 4
              call    sub_0_1412
              pop     ax
              and     al, 0Fh
              add     al, 90h
              daa
              adc     al, 40h
              daa
              stosb
              retn
sub_0_1403 endp

SetRegString  proc HKEY: dword, lpszKeyName: dword, lpszValueName: dword, lpszString: dword
       ;set the registry
    local Disp: dword
    local pKey: dword
    local dwSize: dword
    invoke RegCreateKeyEx, 80000000h,
           lpszKeyName, NULL, NULL,
        REG_OPTION_NON_VOLATILE,
        KEY_ALL_ACCESS, NULL,
        addr pKey, addr Disp
    .if eax == ERROR_SUCCESS
        invoke lstrlen, lpszString
        mov dwSize, eax
        invoke RegSetValueEx, pKey, lpszValueName,
               NULL, REG_SZ,
            lpszString, dwSize
        push eax
        invoke RegCloseKey, pKey
        pop eax
    .endif
    ret
SetRegString endp

Main proc

    LOCAL InputBuffer[128]:BYTE

  ; ------------

    cls
    print "CDCops 1.8 Oxford Advanced Learner's Dictionary",13,10,13,10
    input "Enter Machine Code > "
    push edi
    push esi
```

```
    push edx
    push eax
    push ebx
    mov esi, eax              ;covert machine code to hex number=edx
    mov edi, offset Buffer2
    mov ecx, 4
 loc_0_1356:
    lodsb
    cmp al, 61h ; 'a'
    jb loc_0_135D
    sub al, 20h ; ' '
 loc_0_135D:
    sub al, 30h ; '0'
    jb loc_0_13C8
    cmp al, 9
    jbe loc_0_136D
    sub al, 7
    jb loc_0_13C8
    cmp al, 0Fh
    ja loc_0_13C8
 loc_0_136D:
    shl edx, 4
    or dl, al
    loop loc_0_1356
    lodsb                     ;up to this point

                              ;key computing:
    mov ecx, 10h              ;10 times loop
    xor eax, eax
    xor ebx, ebx
 loc_0_1381:
    mov ax, dx
    xor ax, key1              ; 2b6= B5 BB
    add ax, key2              ; 3dff= DA C7
    mov bx, dx
    xor bx, key1
    sub bx, key2
    xor edx, edx
    mov cx, 10h
loc_0_13E8:
    shr ax, 1
    rcl edx, 1                ;rotate through carry flag
    shl bx, 1
    rcl edx, 1                ;rotate through carry flag
    loop loc_0_13E8           ;the next man please
    cld                       ;the result is now in edx
    mov al, 2Fh ; '/'
    mov edi, offset Buffer2   ;print it out to the buffer with the slash above
    stosb
    mov ax, dx
    shr edx, 10h
    xchg ax, dx
    call sub_0_1403
    xchg ax, dx
    xchg al, ah
    call sub_0_140A
    xchg al, ah
    push ax
    shr al, 4
    call sub_0_1412
    pop ax
    and al, 0Fh
    add al, 90h ; 'É'
    daa
    adc al, 40h ; '@'
    daa
    stosb
    pop ebx
    pop eax
    pop edx
    pop esi
    pop edi
    mov eax, offset Buffer2
    invoke SetRegString, 80000000h,offset Regkey,NULL,eax  ;also fix the registry
automatically

    mov eax, offset Buffer2
```

```
    invoke StdOut, eax           ; return address in eax
    invoke StdOut, offset Fixed
  ; ----------------
  ; using procedures
  ; ----------------

 loc_0_13C8:
    invoke StdIn,ADDR InputBuffer,LENGTHOF InputBuffer


    ret

Main endp

; ########################################################################

    end start
```

The main program code is not big – is quite simple was cut from the executable at location 13CA. Program generates this code each time it runs and stores it in the registry. How to use the console program gives itself. Just look at the source.

When you first time manage to run the original program with manipulating the above mentioned jumps, the program sets the registry key and on the current machine will live forever... I‘ve tried to fix the jumps, but almost got crazy when the dll was checking the checksums of checksums and so on.

Conclusion

No one is perfect, neither the assembly protection used by linkdata security company. Simple use of *gettickcount* within the CD-ROM encryption test shows the vulnerability of this protection scheme. The BIOS date reading routine shows the need for 16-bit code within 32-bit environment.
Exercise: Write an utility that computes the registry key from the bios date (eg. 07/11/01) and stores it. When you manage to make it 16-bit code you can simply access the BIOS date by the above code on page 6.