Construction of keygenerators (a step by step tutorial)
For beginners

AUTHOR: Quantico
EDITOR : Wordpad (just to be different for a change, I like using colours too :) (increase the margins to full screen)
AUDIENCE : Beginner

TARGETS :	1) HTMASC 2.2	(http://www.bitenbyte.com)	(file is ~ 500kb)
	2) J-Write 2.3	(http://www.demon.co.uk/mwa-soft/)	(file is ~ 460kb)
	3) Crypto 3.1	(http://www.execpc.com/~sbd/)	(file is ~ 120kb)

SOLUTIONS : Keygenerators ofcuz :)

TOOLS :
1) Our beloved softice to see the code 'real time' (any version, I use 3.2)
2) Wdasm for looking at large code snippets (I suppose any version would do)
3) Regedit & notepad (to unregister the programs :)
4) Turbo pascal, tasm and tlink, and a knowledge of a 'little' programming in pascal and asm.
5) Patience
6) A 2 litre bottle of Pepsi. Drink as required. :)

INTRODUCTION

Even though there are many tutorials on writing keygens out there nowadays, I still feel there is a need to teach the very beginners how to do them, else how can we learn and become more advanced ? Many of the keygen tutorials are aimed at the intermediate or advanced crackers in which the protection scheme in the chosen program is well beyond the capabilities of most newbies. There have been some good tutorials however, for the newbies, written by guys like CoRN2, pain and Vizion. Since I have only recently begun to make keygens myself, the information contained in this tutorial won't be too advanced.

Anyway, if you have any queries about the programs' protections, my bad coding or anything else, feel free to ask me or any other op about it on IRC in #cracking4newbies, I think most of them will help.....that's what we're there for.

NOTES

Programs' protections get *slightly* more complex through the tutorial so don't skip to number 3 straight away, take your time and understand the other two first. They are much easier. If I tell you the 3rd app uses internal data lookup tables, don't be put off, its the most interesting ;)

Feel free to write the keygens in whatever language you want to. I can only code a little in pascal and asm so that's what I will write the keygen's with. I hope I can explain the protection for each app well enough so you can go straight and write it with C or whatever you want.

This essay does not cover how to set up softice, place breakpoints or anything else like that, it is assumed you have some useful knowledge of softice and the assembler language. If not, then go and get Ed!son's, exact's and CoRN2's softice tutorials and learn them well.

One thing I cannot emphasize enough, to build keygens you must always understand COMPLETELY the protection of your target.
Little bits of code in a target may seem insignificant during run-time but they may have an enormous effect on the end code.
Then you have to reproduce what the program does to the data you enter to make a keygen.

Make sure you have softice set up according to how many other tutorials tell you with symbols from user32.dll loaded. For this tutorial you will only need 2 regular breakpoints: GetDlgItemTextA and Hmemcpy.

LETS BEGIN

I have a certain method I use when attempting to make a keygen.

1) Find your REAL code (if possible) in the program to see the format of it (numbers ? letters ? dashes ?).
2) Test the real code to make sure it isnt a temp code or something else nasty which programmers might do.
3) Go through the program code step by step, understanding it, writing down notes and values of registers (eax etc.).
4) Write the keygen in whatever language is most suitable.
5) TEST the beast

TARGET 1 : HTMASC 2.2

As you should always do, run the program and see what sort of protection it is. Then enter your name and a fake reg number, enter softice, place your breakpoints.

```
:bpx getwindowtexta  ( for 32bit program)
:bpx getdlgitemtexta
```

and return back to HTMASC with ctrl-d. Press OK and......nothing happens apart from the stupid box saying your code is wrong. So it wasn't either of those two. When that happens, set a breakpoint on hmemcpy and try again.

(NOTE: you can tell this is a Borland app (Delphi I think) by looking the the OK & Cancel buttons, they do not look like normal ones. Borland apps do not usually use getwindowtexta or getdlgitemtexta but they ALWAYS use hmemcpy so it is safe to breakpoint on it.)

Softice should have snapped this time and if you press ctrl-d to try and let it read the other box with hmemcpy again, it doesn't snap. So, press OK again and stay in softice. Press F12 (7 times) until you get back to the HTMASC code and then use F10 to step past 6 RETs. You are now back in the main HTMASC code where all the interesting stuff happens.

```
:00460B0D E8FE8FFBFF   call 00419B10                ;  you land back after this call
:00460B12 8B45F8           mov eax, dword ptr [ebp-08]    ;  put our name into eax

* Possible StringData Ref from Code Obj ->"2.0 Registration valid for minor updates"

:00460B15 BA440C4600       mov edx, 00460C44   ; put the above string into edx
:00460B1A E855EBFEFF       call 0044F674            ; interesting call
:00460B1F 8BF0                   mov esi, eax
:00460B21 3BF7                   cmp esi, edi                 ; and what might this be ?
:00460B23 0F85C1000000   jne 00460BEA
```

The important lines are highlighted in red. I will do this throughout the tutorial.

Just before the call to 00460C44, it moves our name and another string into eax and edi respectively. So that call probably contains the serial calculation and there is a CMP almost straight after the call. Remember what I said earlier, find your real code first if you can to see what format it is in. Step past the call and onto the line with the cmp. Type '? edi' and you will see your code. Type '? esi' and you will see another number. Lets not patch the jump. Instead, disable your breakpoints, exit softice, type the number into the reg code box and press OK. If you did it correctly, the program should now be registered to you.

So we know the registration code is a simple number, no characters, dashes or other symbols. Time to make the keygen.

To save you a while of looking for how to unregister the program, I'll tell ya. Go to your windows directory and edit the file called Htmasc32.ini. Delete the registration section at the bottom of the file. And restart the program.

Repeat the above steps until you get to the important call and use 't' to trace into it.

```
====== cut a bit of rubbish to save space =====

:0044F684 55                 push ebp
:0044F685 68F6F64400         push 0044F6F6
:0044F68A 64FF30             push dword ptr fs:[eax]
:0044F68D 648920             mov dword ptr fs:[eax], esp
:0044F690 BB0F020000         mov ebx, 0000020F             ; ebx = 20F hex or 527 dec
:0044F695 8BD6               mov edx, esi                  ;  copy name into edx
:0044F697 8D45F8             lea eax, dword ptr [ebp-08]
:0044F69A E80140FBFF         call 004036A0                 ;  unimportant ( I hope :)
:0044F69F 8D45F8             lea eax, dword ptr [ebp-08]
:0044F6A2 8BD7               mov edx, edi                  ; put '2.0 Registration' string in edx
:0044F6A4 E8DF40FBFF         call 00403788                 ;  joins name and other string
:0044F6A9 8B45F8             mov eax, dword ptr [ebp-08]   ;  put the new string in eax
:0044F6AC 8D55FC             lea edx, dword ptr [ebp-04]
:0044F6AF E8446BFBFF         call 004061F8                 ;  convert string to upper case (see blue
writing)
:0044F6B4 8B45FC             mov eax, dword ptr [ebp-04]   ; put it in eax
:0044F6B7 E8C440FBFF         call 00403780                 ; get length of string and store in eax
:0044F6BC 84C0               test al, al                   ; was length zero?
:0044F6BE 761B               jbe 0044F6DB                  ; if it was then clear off
:0044F6C0 B201               mov dl, 01

* Referenced by a (C)onditional Jump at Address:  (just the loop jumping back)
|:0044F6D9

:0044F6C2 33C9               xor ecx, ecx                       ; clear ecx
:0044F6C4 8ACA               mov cl, dl                         ; low edx (dl) is used as a counter
:0044F6C6 8B75FC             mov esi, dword ptr [ebp-04]        ; put long string in esi (uppercased)
:0044F6C9 0FB64C0EFF         movzx ecx, byte ptr [esi+ecx-01]   ; get byte+counter out of it
:0044F6CE 03D9               add ebx, ecx                       ; add it to ebx
:0044F6D0 81C3D2040000       add ebx, 000004D2                  ; add 4D2 (1234) to it
:0044F6D6 42                 inc edx                            ; increase counter
:0044F6D7 FEC8               dec al                             ; length-1
:0044F6D9 75E7               jne 0044F6C2                       ; end of string
                                                                ;  if not, go back.
..............
:0044F6F5 C3                 ret
```

Once the above loop has finished, the program soon RETurns to where we first entered the call after moving ebx (the proper code) into eax. I left out a little code at the end of the call as it is unimportant and I have described above basically what it does.

AFTER CALL RETURN:

```
:00460B1A E855EBFEFF      call 0044F674
:00460B1F 8BF0            mov esi, eax      ; we come back here
:00460B21 3BF7            cmp esi, edi      ; compare good/bad codes
:00460B23 0F85C1000000    jne 00460BEA   ; go away bad_guy if not equal
```

Here is a summary of the protection scheme:

1) Join your name with the string "2.0 Registration valid for minor updates"
2) Convert all the characters to upper case (if they are lower case, -20 from their ascii code)
3) Set your code variable to 527 or hex 20F
3) Add the code for each digit to the variable followed by 1234 or hex 4D2

Here is my pascal source. Feel free to do it in asm or C or whatever, preferably not cobol ;)

```
program HTM_keygen;
uses crt;

var
  count                    : integer;  { only need small numbers }
  code                     : longint;  { need bigger number here }
  ch                       : char;
  name, dummystr           : string;

begin;
Writeln('HTMASC keygen by Quantico Mex/C4N');
Write('Enter your name:  ');
readln(name);                           { um....read the name!}
dummystr := '2.0 Registration valid for minor updates';
dummystr := name + dummystr;            {join the name and the dummy string}
code := 527;                            { this is what ebx starts at in program}
for count := 1 to length(dummystr) do
  begin
   ch      := char(dummystr[count]);    { get 1 digit of string at a time }
   ch      := Upcase(ch);               { make sure it is upper case }
   code := code + Ord(ch);              { add the number to code }
   code := code + 1234;                 { add 1234 to code }
end;
writeln;
writeln('Your code is:  ', code);       { write the reg code }
End.
```

This code just consists of a loop that goes through all the digits of the joined string, adds the ascii code for each digit to a variable called code followed by 1234. When it exits the loop it writes the code to the screen and exits.

NOTE ABOUT UPPERCASE:

Remember this ?

```
:0044F6AF E8446BFBFF     call 004061F8                  ;  convert string to upper case (see blue)
:0044F6B4 8B45FC         mov eax, dword ptr [ebp-04]  ; put it in eax
```

I'll explain this call just so you understand it if you ever meet it again. It is a common procedure for converting strings to uppercase. This is part of the call.

```
:00406219 8A02           mov al, byte ptr [edx] ; byte of string
:0040621B 3C61           cmp al, 61      ; is it already upcase?
:0040621D 7206           jb 00406225     ; if so then jump _____
:0040621F 3C7A           cmp al, 7A      ; higher than z?       |
:00406221 7702                  ja 00406225     ; if so then jump -----|
:00406223 2C20                   sub al, 20      ; convert to lower case|
|
                                                                        |
* Referenced by a (C)onditional Jump at Addresses: ---------------------|
|:0040621D(C), :00406221(C)
|
:00406225 8806                  mov byte ptr [esi], al
:00406227 42                   inc edx  ; increase edx to get next char
:00406228 46                   inc esi  ;
:00406229 4B                    dec ebx  ; decrease string length counter
:0040622A 85DB                     test ebx, ebx ; reach end of string ?
:0040622C 75EB                    jnz 00406219  ; if not, jump back for more
```

This tests each character to see if it is lower than 'a' in which case it is already an upper case letter or a symbol, jumps to get the next letter of it is or if not then tests the upper limit and if it is between these two numbers, 61h and 7Ah then it is a lower case letter so subtract 20h from it to convert it to an upper case letter. (check your ascii chart to see for yourself). I just explain this here as MANY programs use it to convert to uppercase and I know some newbies who get confused when they see something like this, they think it is part of the main protection routine.

That's it for the first app. Lets move on.

TARGET 2 : J-Write 2.3

As before, run the program, check the error message when you fill in crap registration information and then place your regular breakpoints getwindowtexta and getdlgitemtexta. You will find once again that they don't work so bpx on hmemcpy again. (I actually find the hardest part of writing tutorials is getting the user to where the protection lies, especially when using hmemcpy so please bear with my poor explanations.) Once you press OK, and softice snaps, ctrl-d twice to let the program read the contents of the other two boxes. Then press F12 until you return to the j-write code, then F10 to step past the RETs until you reach this location.

```
:004649D9 E88E9FFBFF        call 0041E96C                    ; you come back after this call
:004649DE 8B45F4             mov eax, dword ptr [ebp-0C]  ; put name in eax
:004649E1 8D4DF8             lea ecx, dword ptr [ebp-08]
:004649E4 5A                  pop edx
:004649E5 E80AFFFFFF         call 004648F4                    ; must be codegen routine
:004649EA 8B45F8              mov eax, dword ptr [ebp-08]
:004649ED 50                  push eax                          ; push good code
:004649EE 8D55FC             lea edx, dword ptr [ebp-04]
:004649F1 8B83F8010000     mov eax, dword ptr [ebx+000001F8]
:004649F7 E8709FFBFF        call 0041E96C
:004649FC 8B55FC             mov edx, dword ptr [ebp-04]
:004649FF 58                  pop eax
:00464A00 E84BF3F9FF        call 00403D50
:00464A05 7415                je 00464A1C                    ;  good guy jump
```

Ok, lets not worry about the code generating routine the first time and find out our code instead. Since there is no jump after the call to '004648F4' we can presume that the compare is NOT done in that call so it is safe to step past. Once you step past the mov eax, dword ptr [ebp-08] instruction, do a 'd eax' and you will see a code. So pop back out of softice and test it. Sure enough it works.

Think about the format of the code you got. A-BCDEFG. 1 digit before the dash and 6 after it, some zero's will be there just after the dash if you didn't enter a very long name. The digit before the dash is the same as the first digit of the code you entered, isn't it? It is maybe a coincidence but as we are crackers, we are supposed to presume things, it is better to presume something like this and be proved wrong than to not presume it and never know (in my own opinion :). The other numbers appear to be ones just generated from the code-making routine.

Just to make sure, you can repeat the process with a code like A12345 instead of whatever you used the first time. You will see that the 'A' DOES appear as the first digit of your code. We were right.

We don't need to unregister this time as the program lets us enter our info again anyway. Repeat the above process until you get to line 00469E5, the one we said was the code-making routine, and trace into it with 't'.

```
* Referenced by a CALL at Address:
|:004649E5
|
:004648F4 53                    push ebx
:004648F5 56                    push esi
:004648F6 57                    push edi
:004648F7 55                    push ebp
:004648F8 83C4F0               add esp, FFFFFFF0
:004648FB 8BE9                  mov ebp, ecx
:004648FD 8BDA                   mov ebx, edx
:004648FF 8BF0                  mov esi, eax                        ; put name in esi
:00464901 BF01000000       mov edi, 00000001                ; edi = 1
:00464906 33C0                  xor eax, eax                        ; clear eax
:00464908 A330284700        mov dword ptr [00472830], eax  ; clear memory location
:0046490D 8BC3                   mov eax, ebx                          ; al = first digit of code you wrote
:0046490F E8C4FFFFFF      call 004648D8                      ; first important call
:00464914 EB0A                   jmp 00464920 >------------->---------------->-
                                                                                  |
* Referenced by a (C)onditional Jump at Address:                           |
|:00464929(C)                                                                     |
                                                                                  |
:00464916 8A443EFF           mov al, byte ptr [esi+edi-01]     ; byte of name + count
:0046491A E8B9FFFFFF       call 004648D8                       ; second important call
:0046491F 47                       inc edi                                    ; increase counter
                                                                                  |
* Referenced by a (U)nconditional Jump at Address: <----------------<--
|:00464914(U)

:00464920 8BC6                  mov eax, esi
:00464922 E819F3F9FF      call 00403C40
:00464927 3BF8                   cmp edi, eax                     ; compare namelength and counter
:00464929 7EEB                   jle 00464916                     ; did we reach the end of name?
:0046492B 55                        push ebp
:0046492C 885C2404          mov byte ptr [esp+04], bl
:00464930 C644240802      mov [esp+08], 02

* Referenced by a (C)onditional Jump at Address:
|:004648CE(C)
|
:00464935 A130284700         mov eax, dword ptr [00472830]  ; put the value from the previous call into
eax
....................
```

The code between here and the return just converts the number into a string for the compare with the code you entered and not a lot else, apart from adding the correct number of zeros after the '-' to make sure the end of the code is 6 digits. We do not need to worry too much about it.

```
....................
:0046496A C3                    ret
```

It is obviously the loop between :0046490D & :00464929 which is calculating something with the name we entered. We can't see this from the surface so, like the previous app, we need to look inside that call to 004648D8.

```
* Referenced by a CALL at Addresses:
|:0046490F   , :0046491A
|
:004648D8 8BC8                  mov ecx, eax                    ; ecx = eax
:004648DA 33C0                  xor eax, eax                    ; eax = 0
:004648DC 8AC1                  mov al, cl                      ; put digit in al
:004648DE 69C0F7000000        imul eax, 000000F7              ; eax = al * F7h
:004648E4 B917000000          mov ecx, 00000017               ; ecx = 17h
:004648E9 99                    cdq
:004648EA F7F9                  idiv ecx                        ; eax = eax/17h
:004648EC 010530284700       add dword ptr [00472830], eax  ; add to memory location
:004648F2 C3                    ret                             ; RETurn ! duh ;)
```

Cool, a small routine, not much explaining to do :)

The first call to this routine takes the first digit of the code you entered, multiplies it by F7 (247d) then divides it by 17h (23d). It does nothing with the remainder of the division (which is stored in edx I think) and only takes the whole value from eax and adds it to the memory location which was cleared at line :00464908.

The second time it is called, from 0046491A, each letter of your name is taken (mov al, byte ptr [esi+edi-01]) and inputted to this call where the same thing happens, it multpilies it, divides it, then adds the result to the memory location.

Notice that the app does NOT do ANYTHING with the company which you entered. Quite a few apps are like this, though some are not, as you will see in Crypto, app number 3.

SUMMARY OF PROTECTION

1) Take the first digit of your input code and perform the manipulations on it, add the result to the store and keep the first digit for the first digit of your 'real' code.
2) Perform the same manipulations on each letter of your name, adding the result to the store each time. This is the last part of the code.
3) Make up the code. The first letter of the code you entered followed by a dash, followed by a few zeros and then the number calculated from parts 1&2. The last part of the code after the dash must be 6 digits. We can do it like this : Once we have made up the number from parts 1&2, we can convert it into a string and check how many zeros we need using something like :

```
numzero := 6 - length(code);
```

Then we can write the correct number of zeros, simple ! (I am sure there is a better way to do it, perhaps some good coder can help out :)

Here is the source of the keygen.

```
PROGRAM JW_keygen;
```

```
VAR
count, temp, temp1  : integer;
result                  : longint;
name, strresult       : string;
digit1                  : char;

BEGIN

 Writeln('J-Write 2.3 - Keygen by Quantico MEX/C4N');
 Write('Enter your name : ');
 Readln(name);
 Write('Enter a single digit for the first digit of your code : ');
 Readln(digit1);

 temp := (Ord(digit1)*$F7) DIV $17;                    { calculate the result of the first digit bit }

 FOR count := 1 TO length(name) DO BEGIN
  temp1  := byte(name[count]);
  temp1  := (temp1*$F7) DIV $17;                    { make up the larger number }
  result := result + temp1;
end;

result := result + temp;                                { add the first part (digit1) }

 Write('Your code is ', digit1, '-');                     { write the first part of the code }

Str(result, strresult);                                 { convert the second part to a string }

 FOR count :=1 to 6-length(strresult) DO BEGIN
   Write('0');                                        { write the correct number of zeros }
 end;

 Writeln(strresult);                                   { write the second part }

END.                                                      { finish }
```

That's that one done......NEXT!

TARGET 3 : Crypto v3.1

Ok, install and run the program, check it out and then head for the registration dialog. You will notice again that the boxes require a name and a company. Fill some crap in and this time a bpx getdlgitemtexta gets us where we want to be. Let it read the contents of all 3 boxes, then get back to the program code.

```
:0040CA11 FFD5              call ebp
:0040CA13 8D442410          lea eax, dword ptr [esp+10]
:0040CA17 50                push eax                        ; our code
:0040CA18 E823360000        call 00410040                   ; string2decimal
:0040CA1D 83C404            add esp, 00000004
:0040CA20 8BE8              mov ebp, eax                    ; ebp = our code in decimal now
```

```
* Possible StringData Ref from Data Obj ->"Gregory Braun"   ; hmmmmm !!!!!

:0040CA22 683CAD4100      push 0041AD3C
:0040CA27 56              push esi                  ; our name !!!!!

* Reference To: KERNEL32.lstrcmpA, Ord:0290h

:0040CA28 FF1570A44200   Call dword ptr [0042A470]       ; DOH !!!!!
:0040CA2E 85C0            test eax, eax
:0040CA30 7524            jne 0040CA56                   ; DOH !!!!!!!

* Possible StringData Ref from Data Obj ->"Software by Design"

:0040CA32 68C8AD4100      push 0041ADC8
:0040CA37 57              push edi                  ; our company

* Reference To: KERNEL32.lstrcmpA, Ord:0290h

:0040CA38 FF1570A44200    Call dword ptr [0042A470]
:0040CA3E 85C0            test eax, eax
:0040CA40 7514            jne 0040CA56
:0040CA42 81FD8D030000   cmp ebp, 0000038D          ; does our code = 909d ?
:0040CA48 750C            jne 0040CA56              ; if not then go to calculations to get real code
```

Heh, sorry about that long code snippet, I just think it is quite funny, and definately stupid that the author, Gregory Braun, has hardcoded his name - company - and a reg code in the app which is all compared to what we entered right here ! Perhaps you will learn something from this, Gregory dear fellow.

In case you do not understand the above code, the author is comparing what we entered to his own information. If it does not match, then the program jumps to 0040CA56 where the calculations begin to calculate the correct code for our name.

```
* Referenced by a conditional Jump at Addresses:
|:0040CA30(C), :0040CA40(C), :0040CA48(C)        ; come here if not the author :)
|
:0040CA56 57              push edi               ; push company
:0040CA57 56              push esi               ; push name
:0040CA58 E8C3110000     call 0040DC20           ; make the code
:0040CA5D 83C408         add esp, 00000008      ; correction
:0040CA60 3BC5           cmp eax, ebp            ; compare good code & our code
:0040CA62 741E           je 0040CA82
```

You do not have to worry about stepping past the call and checking the codes this time, it works anyway, we will get straight to the point. Just by the way, my code is 3394259702, yours should be something similar.

```
* Referenced by a CALL at Addresses:
|:00406CC6   , :0040C89B   , :0040CA4C   , :0040CA58         ; called from a few places !

:0040DC20 8B442404            mov eax, dword ptr [esp+04]          ; name
:0040DC24 56                  push esi                             ; push name
:0040DC25 8B355CA84100      mov esi, dword ptr [0041A85C]      ; esi = C69AA96C
:0040DC2B 50                  push eax                             ; push name
:0040DC2C 81CE78030000   or esi, 00000378                      ; or the starting number
:0040DC32 E8B9DEFFFF       call 0040BAF0                          ; 1st call
:0040DC37 83C404             add esp, 00000004
```

```
:0040DC3A 03F0              add esi, eax                        ; add the result to esi
:0040DC3C 8B44240C          mov eax, dword ptr [esp+0C]         ; eax = company
:0040DC40 50                push eax                            ; push company
:0040DC41 E8AADEFFFF        call 0040BAF0                       ; 2nd call
:0040DC46 83C404            add esp, 00000004
:0040DC49 03C6              add eax, esi                        ; add previous number to result of call
:0040DC4B 5E                pop esi                             ; restore esi
:0040DC4C C3                ret                                 ; go back for compare
```

NOTE : In some of the other apps at the authors site, he uses different starting numbers in esi like ABADDEED, BEDABABE other amusing ones.........funny guy. I bet he thought no-one would ever know what he put here :) Oh well, I will email him and tell him to impove his protection scheme. I chose this app because it is more interesting than the others as it is about encryption. Seems strange to me that an author who can code an app like this to do with encryption can not use it in his own protection scheme !

Anyway, the program is performing the same manipulations on both our name and our company in the calls to 0040BAF0 and is adding the results to the starting number or'd with 378h. This code isn't hard to understand, lets see if the manipulations are more difficult.

```
:0040BAF0 53                push ebx
:0040BAF1 56                push esi
:0040BAF2 8B74240C          mov esi, dword ptr [esp+0C]   ;  store some things
:0040BAF6 57                push edi
:0040BAF7 55                push ebp
:0040BAF8 33FF              xor edi, edi
:0040BAFA 56                push esi                      ; push input for lstrlen call

* Reference To: KERNEL32.lstrlenA, Ord:029Ch

:0040BAFB FF15C8A44200      Call dword ptr [0042A4C8]
:0040BB01 85F6              test esi, esi                 ; did the sucker enter nothing ?
:0040BB03 7432              je 0040BB37                   ; then clear off
:0040BB05 85C0              test eax, eax                 ; lstrlen return
:0040BB07 742E              je 0040BB37                   ; clear off if 0
:0040BB09 B900000000        mov ecx, 00000000
:0040BB0E 7E27              jle 0040BB37

* Referenced by a (C)onditional Jump at Address:
|:0040BB35(C)

:0040BB10 0FBE9C0818BE4100   movsx ebx, byte ptr [eax+ecx+0041BE18]
```

This is getting something from a memory location. Do a 'd 0041BE18' and you will see a long string.

```
#serB&nzlmfM1/5(!sd$Mq.{s]+sFjtKpzSdtzoXqmb^Al@dv:s?x/
```

This is the first datatable, we call it DataTable1. Remember eax is the length of name and ecx is the counter of the loop. Therefore it is moving the [namelength+count] byte of the DataTable1 into ebx.

```
:0040BB18 0FBE2C31          movsx ebp, byte ptr [ecx+esi]     ; byte of name[count]
:0040BB1C 8D5101            lea edx, dword ptr [ecx+01]       ; counter increase
:0040BB1F 0FAFDD            imul ebx, ebp                     ; ebx =
(NameByte*DataTable1Byte)
:0040BB22 0FBE8950BE4100    movsx ecx, byte ptr [ecx+0041BE50]
```

Here is the second datatable, DataTable2, do a 'd 0041BE50' and you will see :

```
lb!pz*ls;rnllf$vi^Axpe)rx5aic&9/2m5lsi4@0dmZw94cmqpfhw
```

Ecx is the counter so it is the datatable[count] byte which is moved into ecx.

```
:0040BB29 0FAFD9        imul ebx, ecx     ; ebx * DataTable2[count]
:0040BB2C 0FAFDA        imul ebx, edx     ; ebx * (count+1)
:0040BB2F 03FB          add edi, ebx      ; store in edi
:0040BB31 8BCA          mov ecx, edx      ; loopcount+1
:0040BB33 3BD0          cmp edx, eax      ; end of input ?
:0040BB35 7CD9          jl 0040BB10       ; go for more

* Referenced by a (C)onditional Jump at Addresses:
|:0040BB03(C), :0040BB07(C), :0040BB0E(C)

:0040BB37 8BC7       mov eax, edi         ; move final result into eax
:0040BB39 5D         pop ebp
:0040BB3A 5F         pop edi
:0040BB3B 5E         pop esi
:0040BB3C 5B         pop ebx
:0040BB3D C3         ret
```

This is done with both the name and the company which we entered and after the call returns, the total value of all the manipulations are stored in eax for the compare with our code (in decimal).

SUMMARY OF PROTECTION :

1) Start the CodeStore at C69AA96Ch then OR it with 378h
2) Take byte from name and multiply it with byte(DataTable1[namelength+count])
3) Multiply the result with byte(DataTable2[count])
4) Multiply that result with loopcount+1 and store result
5) Repeat this for each letter of the name then add final summation to a variable
6) Repeat steps 2-5 for the company string.

This time we will do the keygen in assembler, it isn't too difficult and the code is good code for ripping anyway :) If you were going to do this in pascal, the main routine would look something like :

```
Begin
dummystr1 := '#serB&nzlmfM1/5(!sd$Mq.{s]+sFjtKpzSdtzoXqmb^Al@dv:s?x/';
dummystr2 := 'lb!pz*ls;rnllf$vi^Axpe)rx5aic&9/2m5lsi4@0dmZw94cmqpfhw';

for count :=  1 to length(input) do
  begin
   counter1  := byte(input[count]);
   counter2  := counter1 * byte(dummystr1[count+Length(input)]);
   counter2  := counter2 * byte(dummystr2[count]);
   counter2  := counter2 * count;
   realcode  := realcode + counter2;
end
End;
```

Here is the assembler source.

```
; compile with tasm crypto.asm
; tlink /t /3 crypto.obj

.model  tiny
.386
Org 100h
.data

INTRONAME     db 13,10,'Crypto v3.1 - Key Generator by Quantico [mEX/c4N]',13,10
              db 13,10,'Enter your name : ','$',13,10

COMPANY       db 13,10,'Enter your company : ','$',13,10

DUMBO         db 13,10,'You must enter something...',13,10,'$'


STORENAME     db 18h, 19h dup(0)
STORECOMP     db 18h, 19h dup(0)

THEIRCODE     db 13,10,'Your code is : '
STORECODE     db 10 dup(0),13,10,'$'

DATATABLE1    db '#serB&nz|mfM1/5(!sd$Mq.{s]+sFjtKpzSdtzoXqmb^Al@dv:s?x/'
DATATABLE2    db '|b!pz*ls;rn||f$vi^Axpe)rx5aic&9/2m5lsi4@0dmZw94cmqpfhw'

Convert_Digs  db '0123456789ABCDEF'

.code
.startup

MAIN PROC    NEAR

    MOV     AH, 09h
    MOV     DX, OFFSET INTRONAME
    INT     21h                     ; show the lovely intro and
                                    ; ask for our name
    MOV     AH, 0Ah
    MOV     DX, OFFSET STORENAME
    INT     21h                             ; get what they typed
    CMP     BYTE PTR [STORENAME+1], 0  ; did they enter nothing ?
    JE      DUMB                            ; then tell them

    MOV     AH, 09h
    MOV     DX, OFFSET COMPANY              ; ask for company
    INT     21h

    MOV     AH, 0Ah
    MOV     DX, OFFSET STORECOMP
    INT     21h                             ; get input
    CMP     BYTE PTR [STORECOMP+1], 0  ; enter nothing ?
    JE      DUMB                            ; tell them

    CALL    MAKEKEY                         ; the main procedure

    MOV     AH, 09h
    MOV     DX, OFFSET THEIRCODE            ; show them the code
    INT      21h
    JMP    FINISH                              ; go to quit

DUMB:
    MOV    AH, 09h
    MOV    DX, OFFSET DUMBO                    ; tell them to enter something
    INT    21h

FINISH:
    MOV    AH, 4Ch
    INT    21h                                 ; quit program

MAIN ENDP

MAKEKEY  PROC   NEAR

    MOV         ESI, 0C69AA96Ch          ; esi = C69AA96C
    OR          ESI, 000000378h
    PUSH        ESI                      ; save result for use later
    LEA         ESI, STORENAME+2         ; esi = name
    MOVSX       EAX, BYTE PTR [ESI-1]    ; eax = namelength
    CALL        NEXTSTAGE                ; make first calculations
    POP         ESI                      ; restore esi
    ADD         ESI, EAX                 ; then add the call result
    PUSH        ESI                      ; save it again
    LEA         ESI, STORECOMP+2         ; esi = company
    MOVSX       EAX, BYTE PTR [ESI-1]    ; eax = companylength
    CALL        NEXTSTAGE                ; make second calculations
    POP         ESI                      ; restore esi
    ADD         EAX, ESI                 ; add both parts

    XOR         EBX, EBX                 ; clear ebx
    XOR         EDX, EDX                 ; clear edx
    MOV         EDI, OFFSET STORECODE    ; place to put the string of the code
    MOV         ECX, 10d                 ; number base 10
    CALL        convert_num              ; number2string so we can print it
    RET                                  ; return
MAKEKEY ENDP

NEXTSTAGE  PROC   NEAR
    XOR        EDI, EDI
    XOR        ECX, ECX
GETMORE:
    MOVSX    EBX, BYTE PTR [EAX+ECX+DATATABLE1]
    MOVSX    EBP, BYTE PTR [ECX+ESI]
    LEA        EDX, DWORD PTR [ECX+01]
    IMUL       EBX, EBP
    MOVSX    ECX, BYTE PTR [ECX+DATATABLE2]
    IMUL       EBX, ECX
    IMUL       EBX, EDX
    ADD        EDI, EBX
    MOV        ECX, EDX
    CMP        EDX, EAX                  ; end of input ?
    JL         GETMORE                   ; if not, go for more
    MOV        EAX, EDI                  ; eax = call result
    RET
NEXTSTAGE  ENDP

Convert_Num proc near
        pushf
        pushAD

        sub     esp, 4
        mov     ebp,esp

        cld
        mov     esi, edi
        push    esi

;--- loop for each digit

        sub     bh, bh
        mov     dword ptr [ebp], eax            ;save low word
        mov     dword ptr [ebp+4], edx          ;save high word
        sub     esi, esi                        ;count digits

Connum1:
        inc     esi
        mov     eax, dword ptr [ebp+4]          ;high word of value
        sub     edx, edx                        ;clear for divide
        div     ecx                             ;divide, DX gets remainder
        mov     dword ptr [ebp+4],eax           ;save quotient (new high word)

        mov     eax, dword ptr [ebp]            ;low word of value
        div     ecx                             ;divide, DX gets remainder
                                                ;  (the digit)
        mov     dword ptr [ebp], eax            ;save quotient (new low word)

        mov     bl, dl
        mov     al, byte ptr [Convert_Digs+ebx]  ;get the digit
        stosb                                    ;store

        cmp     dword ptr [ebp], 0              ;check if low word zero
        jne     Connum1                         ;jump if not
        cmp     dword ptr [ebp+4], 0            ;check if high word zero
        jne     Connum1                         ;jump if not

        sub     al, al
        stosb                                   ;store the terminator

;--- reverse digits

        pop     ecx                             ;restore start of string
        xchg    ecx, esi
        shr     ecx, 1                          ;number of reverses
        jz      Connum3                         ;jump if none

        xchg    edi, esi
        sub     esi, 2                          ;point to last digit
```

```
Connum2 :
        mov     al, byte ptr [edi]             ;load front character
        xchg    al, byte ptr [esi]            ;swap with end character
        stosb                                  ;store new front character
        dec     esi                            ;back up
        loopd   Connum2                       ;loop back for each digit

;--- finished

Connum3  :
        add     esp, 4

        popad
        popf
        ret
 endp           ;Convert_Num

END MAIN
```

That's all for this time folks, I hope you enjoyed reading this and I hope you learned something from it. If you have, then I am satisfied. If you have any queries or anything else, you can e-mail me at Quantico@postmaster.co.uk and I will try to reply but if you do not get an answer, it might be because I am away for most of the summer, socialising is more fun than sitting in front of a computer :) See you next time in the intermediate - advanced keygen tutorial.......if I can ever crack some difficult apps ;)
Later,
Quantico -=[mEX/c4N]=-

http://www.mexelite.jjcom.com

GREETS.

Vizion, bisoux, Rudeboy, +MaLaTTia, pain, +HalVar, |Fresh|, GIJ, t00nie (good to have you back man :), madmax!, mpbaer, +Yoshi, Baser, CrueHead, egis'98, DASavant, CoRN2, f0ssil, odin, kidlat, Norway, Quine, snipes, Vucoet, Tin, nIabI, JosephCo, all others in #cracking4newbies.

DISCLAIMER

The information contained in this text is legal 'as is' but I can in no way be held responsible for illegal use of this material or any damage caused. Be careful :)

"The Lord is my rock, and my fortress, and my deliverer; my God, my strength, in whom I will trust...."
Psalms 18:2