Reversing f0dder's CrackMe #2

by Cardenal Mendoza

February 26, 2000

1 Introduction

This essay is not for absolute beginners, although reversing the crackme is not very hard, but coding a keygen is. Heh, coding a 90% keygen is hard, but i guess coding a 100% keygen is impossible because you could only brutforce the last part of the crackme :(Apart from that it is a very nice crackme, but decide by yourself...

2 Tools used

1. SoftIce 4.05 for NT (I use Win 2000 for cracking)
2. Win32dasm (I guess IDA would be better - but I don't have it)
3. Ultraedit
4. f0dder's crackme #2 (of course ;)
5. some imagination or a great mathematical knowledge (for coding the keygen)

3 Reversing the crackme

First start the crackme and look what intresting stuff you can find. Well, since there is only one box to enter something, do that and press the bitmap. A nice message pops up telling you that the door is still locked. You stupid idiot, didn't you set a breakpoint on *GetWindowTextA* and *GetDlgItemTextA* first? Ok, put those breakpoints and click again... Again the message pops up, well, some smart people would now set a breakpoint on hmemcpy, but remember, I crack under Win 2000, there is no *hmemcpy*, so we must use another way. Again many people would say now, that we could use *memcpy*, but *memcpy* is not the same as *hmemcpy*, for cracking purposes memcpy is useless, trust me.
Well, I got some background information from f0dder on irc :) He said I should use the dead listing, at least he said this for #1, but I hope it will help us here, too. Just disassemble it in your preferred disassembler, I will use Win32dasm

since I don't have IDA. Anyway, if you did this, just trace a little bit through the code. Better print it completly (I hope you have a nice printer ;). First look at the imports. This are the two most important ones:

1. CreateFileA
2. GetDlgItemTextA

So what do this two imports tell us?

1. Its a keyfile protection (We should have read the readme.txt *g*)
2. It probably uses GetDlgItemTextA for getting the user input. (Hmm, ' we check this?)

I must admit that we could get the first information from the readme.txt, too, but who reads such txts? The second one is more intresting, how can it use GetDlgItemTextA without breaking on it? Perhaps f0dder thinks he is smart and only checks for the serial if a valid keyfile is found.
Now lets jump to programs entry point:

```
//*****************Program Entry Point**************
:004013F3      push 00000000
:004013F8      call dword ptr[00402014]
:004012FE      mov  dword ptr[0040334C], eax
:00401403      xor  ebx, ebx
:00401405      xor  eax, eax
:00401407      push eax
:00401408      push 00401320
:0040140D      push ebx
:0040140E      push 00000065
:00401413      push eax
:00401414      call dword ptr[0040202C]
:0040141A      push eax
:0040141B      call dword ptr[00402004]
:00401421      ret
```

What does us that part of the code tell? You should know this by yourself, if not, read iczelions win32asm tutorials. This is just the start of every Win32 dialog based application. Beside that it tells us nothing usefull, but it is always better to check, because the coder could check for the keyfile even before the call to *DialogBoxParamA*. Now lets scroll up again until we see the start of the code:

```
++++++++++++++++++++++++ASSEMBLY CODE LISTING+++++++++++++++++
//******************Start of Code in Object .text************
Program Entry Point=004013F3
```

```
:00401000      mov  edi, 004030A7
:00401005      mov  cl, 03
:00401007      dec  byte ptr[edi]
:00401009      dec  byte ptr[edi+01]
:0040100C      dec  byte ptr[edi+02]
:0040100F      dec  byte ptr[edi+03]
:00401012      add  edi, 00000004
:00401015      dec  cl
:00401017      jne  00401007
```

This part is also easy to understand. We see that 004030A7 is a string reference to 'tlfmfupo/lfz'. try to analyze it before you read on, its not that hard if you understand assembly. All it does is decreasing each byte in the string by one, lets do it on a paper, too. Yes, the result is 'skeleton.key'. This looks like the name of our keyfile. After this code, the location 004012C0 is called, lets take a look at it:

```
:004012C0      push 00000000
:004012C5      push 00000000
:004012CA      push 00000003
:004012CF      push 00000000
:004012D4      push 00000000
:004012D9      push 80000000
:004012DE      push eax
:004012DF      call dword ptr[0040200C]
:004012E5      cmp  eax, FFFFFFFF
:004012EA      jne  004012EE
:004012EC      xor  eax, eax
:004012EE      ret
```

As expected this code snippet does nothing more than opening the keyfile. If it is succesfully, it returns, if not, it sets eax=0 before returning. After returning the eip will be 00401021. I won't list every small piece of code, as I said you should print it by yourself. After returning, the program checks if the file is present (00401021), if not it jumps to 0040104C. Now trace into the following calls and see what happens there. I assume not to use SoftIce, yet, in order to see how useful a dead listing can be. Lets analyze the code, the call at 00401034 does nothing more than reading the whole file into memory and the call at 00401045 closes the file again. Notice the cmps/jmps after these two calls, if one of them fails, it jumps to 0040104C, setting a flag that the keyfile is not found. This is the reason we couldn't break on *GetDlgItemTextA* If you examine the call at 004012F7 you will notice that it reads 256 bytes, so our keyfile must be at least 256 bytes long. After those we could append something like "Keyfile generated by Cardenal Mendoza", but this is not an expensive app and I don't care if somebody steals a keygen for a crackme :P
Lets go reversing again. Write down the address of the keyfile in memory. Now

search through the dead listig for this address... ahh we are lucky, it occurs only two times, for the first time when the file is read into memory, for the second time at 00401134. The second location seems quite intresting. We should take a closer look at it:

```
:00401134      mov  esi, 004030B4
:00401139      mov  edi, 004031B4
:0040113E      xor  ecx, ecx
:00401140      mov  cl, 80
:00401142      mov  al, byte ptr[esi]
:00401144      cmp  al, 30
:00401146      jb   004012A8
:0040114C      cmp  al, 39
:0040114E      ja   00401158
:00401154      sub  al, 30
:00401156      jmp  0040116A
:00401158      cmp  al, 41
:0040115A      jb   004012A8
:00401160      cmp  al, 46
:00401162      ja   004012A8
:00401168      sub  al, 37
:0040116A      mov  ah, al
:0040116C      shl  ah, 04
:0040116F      mov  al, byte ptr[esi+01]
:00401172      cmp  al, 30
:00401174      jb   004012A8
:0040117A      cmp  al, 39
:0040117C      ja   00401186
:00401182      sub  al, 30
:00401184      jmp  00401198
:00401186      cmp  al, 41
:00401188      jb   004012A8
:0040118E      cmp  al, 46
:00401190      ja   004012A8
:00401196      sub  al, 37
:00401198      or   ah, al
:0040119A      mov  byte ptr[edi], ah
:0040119C      inc  esi
:0040119D      inc  esi
:0040119E      inc  edi
:0040119F      dec  cl
:004011A1      jne  00401142
:004011A3      ret
```

I suggest that either you have a good knowledge of that shift and logical operations or you simply look in SoftIce what happens. I did the second one because

I hate stuff like or and shl. I make it short, the keyfile is only valid, if it has only characters between '0' and '9' and 'A' and 'F'. Think about this, that are all possible chracters for hexadecimal numbers. If you analyze the code after the check you will know why. Just look what happens:

Keyfile:	A	2	D	1	F	9	E	1	7	F
Array:	A2	D1	F9	E1	7F	..	..	..	..	..

The code simply converts two ASCII values (for example 'A' and '2') to one byte (A2).
I guess this is clear. How should we go on? Starting to trace with SoftIce? Oh, no, just look at the first part of the code snippet: "Referenced by a Call...". Lets look where the code part is called from. There you should see 5 calls. In my opinion tgis looks like a protection =) I think we should look into the next call. Hehe, a reference to *GetDlgItemTextA*, now we know that we are right.

```
:004011A4      push 00000011
:004011A9      push 00403334
:004011AE      push 000003E8
:004011B3      push dword ptr[00403350]
:004011B9      call dword ptr[00402024]
:004011BF      cmp  eax, 00000010
:004011C4      ja   004012A8
:004011CA      je   004011E8
:004011CC      mov  esi, 00403334
:004011D1      mov  edi, 00403334
:004011D6      add  edi, eax
:004011D8      sub  eax, 00000010
:004011DD      neg  eax
:004011DF      mov  dl, byte ptr[esi]
:004011E1      mov  byte ptr[edi], dl
:004011E3      inc  esi
:004011E4      inc  edi
:004011E5      dec  eax
:004011E6      jne  004011DF
:004011E8      mov  esi, 00403334
:004011ED      mov  edi, 00403234
:004011F2      mov  ecx, 00000004
:004011F7      repz
:004011F8      movsd
:004011F9      mov  ecx, 00000070
:004011FE      mov  esi, 00403234
:00401203      mov  al, byte ptr[esi]
:00401205      xor  al, byte ptr[esi+01]
:00401208      mov  byte ptr[edi], al
:0040120A      inc  esi
```

```
:0040120B      inc  esi
:0040120C      dec  ecx
:0040120D      jne  00401203
:0040120F      ret
```

Slowly the dead listing begins to bore me, so let's make it fast ;)
As you can see, this part does nothing more than getting our serial, then checking if it is lower than 16 in length, and if yes, it appends the serial from the beginning until the serial is 16 characters long. Any serial which is longer is invalid. This is easy to see out of the dead listing. If not you should learn assembler. I won't explain all instructions, because I said this is not for the very beginner. If the serial is 16 byte long, it is xored in a special way. It xors the first and the second array element and writes the result into the 17th, then it xors the 3rd and the 4th to the 18th and so on. This is done until the serial-array is 128 bytes long. Now look into the next call, as you can easily see, it does nothing more than just xoring those two arrays, which are both 128 bytes long. I thunk this is even fr newbies easy to understand.
Ok, the next call is a important one, so lets take a deeper look at it:

```
:00401234      mov  esi, 004032B4
:00401239      mov  ecx, 00000080
:0040123E      mov  eax, dword ptr[0040309A]
:00401243      xor  al, byte ptr[esi]
:00401245      rol  eax, 1
:00401247      inc  esi
:00401248      dec  ecx
:00401249      jne  00401243
:0040124B      cmp  eax, dword ptr[0040309E]
:00401251      jne  004012B4
:00401253      ret
```

Well, I got heavy headaches because of this call when I coded a keygen for the crackme. The code itself should be clear. first eax is set to 0BADC0DEh and then it xors al with each byte of teh array and rols eax after every xor. Then after doing the whole loop eax is compared to the dword in 0040309E. Look in SoftIce to look whats in it. Yes, it is 0FCC5A375h. Hmm, that are all simply instructions. But now think how to reverse it. Of course it is easy to get a right start or ending value with a good array, but how to get the array with just two values? Well, figure out by yourself, you can always look at my keygen in appendix A. My keygen is very crappy and if I am honest, I now don't know why it works, erm, I know the prinziple but it was alot of try&error testing until I got it working. In appendix B there is a mail by +spath about a mathematical way to solve it, but I must admit that I am not so good in mathematics and I don't understand too much of it.
Code a keygen by yourself, you will see that it is senseless:(
If you analyze the next call you will see why. Did you notice that strange looking code at 00401064 before? We jump to it after the last call, which does nothing

more than xoring the code with our array. This means that we can't make a good keygen, we have to brute force a valid array first. But that is not my task, in my opinion a crackme must be crackable, bruteforcing is no cracking at all. f0dder should have called this a brute force me or at least write in to the readme that it is only brute force able, hehe, I didn't read it anyway ;) f0dder says that he wanted to show with this crackme that it is possible to code a protection which is not easily to crack. Sure, this is possible, but i knew it before. But apart from this last part cracking the crackme was fun.

4 Conclusions

So what did we learn from this crackme? I don't know what you learned, but I learned that dead listings can be very useful, at least if the protection is coded in assembly. Ok, I sometimes used SoftIce, for example when the keyfile is converted to the array. Decide by yourself what you learned from it. Always remember, this is my 3rd tutorial and my English is very bad :P If you have any questions mail me at Cardenal@gmx.net.

Please visit these webpages:

1. http://mendoza.tsx.org < my webpage
2. http://www.learn2crack.com < learn2crack
3. http://codex.cjb.net < the codex

5 Appendix I - My keygen source

```
; keymaker.asm
; I know this code is very crappy, but hey, it is only a keygen
; I hope you understand that I don't optimize it if it works ;)
; coded by Cardenal Mendoza [01/29/2000]
; compiles/runs without problems under Win 2000 and Masm
.386
.model flat, stdcall
option casemap:none
; Includes
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
MAINDLG equ 103
ID_EDITNAME equ 1000
ID_EDITSERIAL equ 1001
ID_STATIC equ -1
```

```
KeyGenDlgProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
MakeKey PROTO :DWORD
; The Data Section
.DATA
fmat        db   "%d",0
hInst       dd   0
KeyGenDlg   db   "MAINDLG",0
fname       db   "skeleton.key",0
fhandle     dd   0
xor_array   db   128 dup(0)
ser_array   db   128 dup(0)
fil1array   db   128 dup(0)
fil2array   db   256 dup(0)
bread       dd   0
bwrite      dd   0
; The uninitialized data
.DATA?
szName      db   250 dup(?)
szSerial    db   40  dup(?)
szTmp       db   6   dup(?)
DlgRect     RECT <>
DlgWidth    dd   0
DlgHeight   dd   0
DeskRect    RECT <>
; The code section
.CODE
main:
   invoke   GetModuleHandle, NULL
   mov      hInst, eax
   invoke   DialogBoxParam,hInst, ADDR KeyGenDlg, 0, ADDR KeyGenDlgProc, 0
   invoke   ExitProcess, eax
KeyGenDlgProc PROC hDlg:DWORD, wMsg:DWORD, wParam:DWORD, lParam:DWORD
   .IF wMsg==WM_INITDIALOG
      push  OFFSET DlgRect
      push  hDlg
      call  GetWindowRect
      call  GetDesktopWindow
      push  OFFSET DeskRect
      push  eax
      call  GetWindowRect
      push  0
      mov   eax, DlgRect.bottom
      sub   eax, DlgRect.top
      mov   DlgHeight, eax
      push  eax
      mov   eax, DlgRect.right
```

```
      sub   eax, DlgRect.left
      mov   DlgWidth, eax
      push  eax
      mov   eax, DeskRect.bottom
      sub   eax, DlgHeight
      shr   eax, 1
      push  eax
      mov   eax, DeskRect.right
      sub   eax, DlgWidth
      shr   eax, 1
      push  eax
      push  hDlg
      call  MoveWindow
   .ELSEIF wMsg == WM_DESTROY
      jmp   wmclose
   .ELSEIF wMsg == WM_CLOSE
      jmp   wmclose
   .ELSEIF wMsg == WM_COMMAND
      jmp   wmcommand
   .ENDIF
   xor      eax, eax
   ret
wmclose:
   invoke   EndDialog, hDlg, TRUE
   invoke   ExitProcess, NULL
   ret
wmcommand:
   cmp      word ptr[wParam], IDOK
   jne      notn
   invoke   MakeKey, hDlg
   jmp      rt
notn:
   cmp      wParam, IDOK
   je       wmclose
rt:
   ret
KeyGenDlgProc  ENDP
MakeKey PROC USES eax ebx ecx edx esi edi _hDlg:DWORD
   xor      ecx, ecx
   mov      ebx, 666h
   mov      esi, offset xor_array
rnd_lop:
   invokde  GetTickCount
   imul     eax, ebx
   rol      eax, 2
   mov      dword ptr[esi], eax
```

```
   add      esi, 4
   inc      ecx
   rol      ebx, 5
   add      ebx, eax
   cmp      eax, 24
   jl       rnd_lop
   mov      edi, offset xor_array
   mov      ecx, 96
   mov      eax, 0BADC0DEh
encr2_lop:
   xor      al, byte ptr[edi]
   rol      eax, 1
   inc      edi
   dec      ecx
   jnz      encr2_lop
   xor      al, 075h
   mov      byte ptr[esi], al
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      edi, offset xor_array
   mov      ecx, 104
   mov      eax, 0BADC0DEh
encr1_lop:
   xor      al, byte ptr[edi]
   rol      eax, 1
   inc      edi
   dec      ecx
   jnz      encr1_lop
   xor      al, 0FCh
   mov      byte ptr[esi], al
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
```

```
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
rol     eax, 8
xor     al, 0C5h
mov     byte ptr[esi], al
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
rol     eax, 8
xor     al, 0A3h
mov     byte ptr[esi], al
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
mov     byte ptr[esi], 00
inc     esi
```

```
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   inc      esi
   mov      byte ptr[esi], 00
   rol      eax, 8
   xor      al, 075h
   mov      byte ptr[esi], al
   inc      esi
   xor      eax, eax
   xor      esi, esi
   xor      edi, edi
   xor      edx, edx
   mov      esi, offset ser_array
   mov      edi, offset xor_array
cp_lop:
   mov      dl, byte ptr[edi]
dada:
   cmp      dl, 48
   jg       dudu
   add      dl, 9
   jmp      dada
dudu:
   cmp      dl, 57
   jl       huhu
   sub      dl, 9
   jmp      dudu
huhu:
   mov      byte ptr[esi], dl
   inc      eax
   inc      edi
   inc      esi
   cmp      eax, 16
   jnz      cp_lop
   mov      edi, offset ser_array
   mov      ecx, 70h
xorma:
   mov      al, byte ptr[edi]
   xor      al, byte ptr[edi+1]
   mov      byte ptr[esi], al
   inc      esi
   inc      edi
   dec      ecx
   jnz      xorma
   xor      eax, eax
   xor      ebx, ebx
```

```
   xor      ecx, ecx
   xor      edx, edx
   xor      esi, esi
   xor      edi, edi
   mov      eax, offset fil1array
   mov      ebx, offset ser_array
   mov      ecx, offset xor_array
   mov      esi, 80h
make_file:
   mov      dl, byte ptr[ecx]
   xor      dl, byte ptr[ebx]
   mov      byte ptr[eax], dl
   inc      ecx
   inc      ebx
   inc      eax
   dec      esi
   jnz      make_file
   xor      eax, eax
   xor      ebx, ebx
   xor      ecx, ecx
   xor      edx, edx
   xor      esi, esi
   xor      edi, edi
   mov      edx, 10
   mov      esi, offset fil1array
   mov      edi, offset fil2array
   mov      ecx, 80h
lala:
   mov      al, byte ptr[esi]
   mov      bl, al
   and      al, 0fh
   and      bl, 0f0h
   cmp      al, 0Ah
   jl       zuzu
   add      al, 37h
   jmp      iuiu
zuzu:
   add      al, 30h
iuiu:
   ror      ebx, 4
   cmp      bl, 0Ah
   jl       zuzu2
   add      bl, 37h
   jmp      iuiu2
zuzu2:
   add      bl, 30h
```

```
iuiu2:
   mov      byte ptr[edi], bl
   inc      edi
   mov      byte ptr[edi], al
   inc      edi
   inc      esi
   dec      ecx
   jnz      lala
   invoke   CreateFileA, ADDR fname, GENERIC_WRITE + GENERIC_READ,
            NULL, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
   mov      fhandle, eax
   invoke   WriteFile, fhandle, ADDR fil2array, 256, ADDR bread, 0
   invoke   CloseHandle, fhandle
   mov      eax, offset ser_array
   mov      byte ptr[eax+16], 0
   invoke   SetDlgItemTextA, _hDlg, ID_EDITSERIAL, ADDR ser_array
exit_proc:
   mov      eax, 1
   ret
MakeKey     ENDP
end main

; keymaker.rc
#include "\masm32\include\resource.h"
#define  ID_KeyGenDlg  103
#define  ID_EDITNAME   1000
#define  ID_EDITSERIAL 1001
#define  ID_STATIC     -1

MAINDLG DIALOGEX DISCARDABLE 0, 0, 160, 75
STYLE DS_3DLOOK | DS_CENTER | WS_POPUP | WS_VISIBLE | WS_CAPTION |
      WS_SYSMENU | WS_EX_CLIENTEDGE | WS_DLGFRAME
CAPTION "*KeyMaker* for f0dders crkme #2*"
FONT 8, "MS Sans Serif"
BEGIN
   DEFPUSHBUTTON "&Randomize!",IDOK, 10, 10, 60, 12
   EDITTEXT ID_EDITSERIAL, 10, 40, 140, 12, ES_READONLY | ES_CENTER
   GROUPBOX "Serial:", ID_STATIC, 5, 30, 150, 25
   CTEXT "Coded by Cardenal Mendoza [01/29/2000]", ID_STATIC, 5,
         60, 150, 10, SS_SUNKEN | WS_DISABLED
END

; makefile
NAME = keymaker
OBJS = $(NAME).obj
RES  = $(NAME).res
```

```
$(NAME).EXE:$(OBJS) $(RES)
link /SUBSYSTEM:WINDOWS,4.0 $(OBJS) $(RES)

.asm.obj:
ml /c /coff $(NAME).asm

.rc.res:
rc $(NAME).rc
```

6 Appendix II - +spath's mail

Hi mendo,
Ok, heret's how I see the problem:
from what I understood you have something like this:

```
   mov   esi, offset StartArray
Eloop:
   xor   al, byte ptr[esi]
   rol   eax, 1
   inc   esi
   cmp   esi, offset(StartArray+127)
   jle   ELoop
```

The corresponding decrypt loop would be

```
   mov   esi, offset(StartArray+127)
Dloop:
   xor   al, byte ptr[esi]
   ror   eax, 1
   dec   esi
   cmp   esi, offset StartArray
   jge   DLoop
```

Now let me define some notations:
C is the resulting dword, C(n) is bit n of this dword
P is the starting dword (BADC0DE ?), P(n) is bit n of this dword
T(n)(k) is bit k of the temporary value in EAX after decryption round n
K(x)(y) is bit y of byte x of the array
+ is XOR
You know C and P, and want to generate plenty of K values; let's say you fill the array with all random values, the you start decrypting; for the first round you have 32 equations:
T(1)(0) = C(1) + K(127)(1)
T(1)(1) = C(2) + K(127)(2)
..

T(1)(6) = C(7) + K(127)(7)
T(1)(7) = C(8)
..
T(1)(30) = C(31)
T(1)(31) = C(0) + K(127)(0)
for any round n you have (A):
T(n)(0) = T(n-1)(1) + K(128-n)(1)
T(n)(1) = T(n-1)(2) + K(128-n)(2)
..
T(n)(6) = T(n-1)(7) + K(128-n)(7)
T(n)(7) = T(n-1)(8)
..
T(n)(30) = T(n-1)(31)
T(n)(31) = T(n-1)(0) + K(128-n)(0)
with T0 = C and T(128) = P, so that at last you have:
P(0) = T(128)(0) = T(127)(1) + K(0)(1)
P(1) = T(128)(1) = T(127)(2) + K(0)(2)
..
but using (A) we can extend P(0) as
P(0) = T(127)(1) + K(0)(1)
= T(126)(2) + K(1)(2) + K(0)(1)
= T(125)(3) + K(2)(3) + K(1)(2) + K(0)(1)
= ...
note that some parts will be easier, when the bit is in the range 8-31, for instance:
P(7) = T(127)(8)
= T(126)(9)
= ...
= T(104)(31)
= T(103)(0) + K(24)(0)
So basically any bit of P can be written as a combination of T(103) and key bytes from K(0) to K(24); you can expand these equations a bit more until you get 32 equations describing each bit of P from T(96) and K(0) ... K(31). If you decrypt with random K bytes until T(96), you have 32 equations with 32 unknown variables.
> From here you can solve it using a gaussion pivot or a martix inversion; a good idea would be to first use gaussian elimination to obtain a triangular matrix, so that you can easily calculate the determinant, which should give you constraints on some K bits. You can get infos about these methods at http://www.geog.ubc.ca/numeric/labs/lab3/lab3/lab3.html, this case is just sligthly different since you work in Z/2Z.
regards, Spath.