TwistedTux's First keygenMe

The description for this crackme is:

```
My first KeygenMe so don't be rude :)
I had fun to write it and i think it's pretty easy.
Rules: no patching, write a keygen.
```

The crackme was written in C/C++ and compiled for *nix. It is rated *2 - Needs a little brain (or luck)*.

Starting Point

I used IDA Pro to disassemble the crackme. The binary uses the default `_start` label as the entry point identifier::

```
.text:08048390                 public start
.text:08048390 start           proc near
.text:08048390                 xor     ebp, ebp
.text:08048392                 pop     esi
.text:08048393                 mov     ecx, esp
.text:08048395                 and     esp, 0FFFFFFF0h
.text:08048398                 push    eax
.text:08048399                 push    esp
.text:0804839A                 push    edx
.text:0804839B                 push    offset sub_8048730
.text:080483A0                 push    offset sub_8048740
.text:080483A5                 push    ecx
.text:080483A6                 push    esi
.text:080483A7                 push    offset sub_8048450
.text:080483AC                 call    ___libc_start_main
.text:080483B1                 hlt
.text:080483B1 start           endp
```

The purpose of this snippet is to prepare all arguments for `__libc_start_main`, which performs all the necessary initialization before the `main()` routine of our C/C++-code is called. The prototype of `__libc_start_main` is::

```
int __libc_start_main(
    int (*main) (int, char * *, char * *), /* address of main */
    int argc, /* number of arguments - argc */
    char **ubp_av, /* unbounded pointer to arguments - argv */
    ..., /* other arguments that we don't care about
    )
```

So we know that the `main` routine starts at offset 0x0848450::

```
.text:08048450 main            proc near               ; DATA XREF: start+17o
.text:08048450
.text:08048450 argc            = dword ptr  8
.text:08048450 argv            = dword ptr  0Ch
.text:08048450
.text:08048450                 push    ebp
.text:08048451                 mov     ebp, esp
.text:08048453                 mov     eax, [ebp+argc]
.text:08048456                 cmp     eax, 3
.text:0804845B                 jnb     short loc_8048485 ; branch if argc >= 3
.text:0804845D                 mov     esi, [ebp+argv]
.text:08048460                 mov     edi, [esi]
```

```
.text:08048462                 sub     esp, 8
.text:08048468                 mov     esi, offset format ; "Utilisation : %s <pseudo> <clef>
.text:0804846D                 mov     [esp], esi      ; format
.text:08048470                 mov     [esp+4], edi
.text:08048474                 call    _printf
```

I manually renamed the routine as `main` and gave the two function arguments the conventional names:

- `argc` - the number of arguments
- `argv` - the pointer to the arguments, where `argv[0]` is the name of the binary, and `argv[1]` is the first parameter passed to the crackme.

The snippet checks if there are at least three command line arguments (including the binary name). If not, it prints the usage statement:

```
int main(int argc, char *argv[])
{
    if( argc < 3 )
        printf("Utilisation : %s <pseudo> <clef>\n");
    else
        goto loc_8048485;
}
```

If the program is called with both `pseudo` and `clef` arguments then we continue with offset `loc_808485`.

Length of the Key (Clef)

Here`s offset `0x08048485`:

```
.text:08048485 loc_8048485:                            ; CODE XREF: main+Bj
.text:08048485                 sub     esp, 0Ch
.text:0804848B                 mov     ebx, [ebp+argv] ; ebx = argv
.text:0804848E                 mov     edx, ebx
.text:08048490                 add     edx, 4          ; edx = &argv[1]
.text:08048496                 mov     edx, [edx]      ; edx = argv[1] (= pseudo)
.text:08048498                 mov     [esp+4], edx    ; pseudo --> [esp+4]
.text:0804849C                 mov     edx, ebx        ; edx = argv
.text:0804849E                 add     edx, 8          ; edx = &argv[2]
.text:080484A4                 mov     edx, [edx]      ; edx = argv[2] (= clef)
.text:080484A6                 mov     [esp+8], edx    ; clef --> [esp+8]
.text:080484AA                 mov     esi, [esp+8]    ; esi = clef
.text:080484AE                 push    esi             ; s
.text:080484AF                 call    _strlen         ; eax = strlen(clef)
.text:080484B4                 add     esp, 4          ; caller cleanup
.text:080484BA                 cmp     eax, 6          ; strlen(clef) == 6
.text:080484BF                 jnz     short loc_8048479
```

The snippet first creates a stack frame of 3 bytes, of which 2 bytes are used to store the command line parameters:

- the first parameter, `pseudo`, is stored at `[esp+4]`
- the second parameter, `clef`, is stored at `[esp+8]`

The snippet then calculates the length of `clef` with a call to the C library `strlen`, and checks if the length is 6 - if it isn't we jump to `loc_8048479`::

```
.text:08048479 loc_8048479:                            ; CODE XREF: main+6Fj
.text:08048479                                         ; main+9Bj ...
.text:08048479                 mov     eax, 1
.text:0804847E                 mov     ebx, 1          ; status
.text:08048483                 int     80h             ; LINUX - sys_exit
```

This causes the program to exit with exit code 1, I therefore renamed this location as `exit` in IDA Pro. Exiting at this point probably means that the `pseudo`/`clef` combination is invalid, and we know that `clef` **needs to have 6 letters**.

Key's First Character

If the length of `clef` is 6 we continue here:

```
.text:080484C1                 mov     edi, [esp+4]    ; edi = pseudo
.text:080484C5                 push    edi             ; s
.text:080484C6                 call    _strlen         ; strlen(pseudo)
.text:080484CB                 add     esp, 4          ; caller cleanup
.text:080484D1                 mov     [esp], eax      ; pseudo_length --> [esp]
.text:080484D4                 push    eax
.text:080484D5                 call    func_1          ; func_1(pseudo_length)
.text:080484DA                 mov     ebx, offset hardcoded_str ; "A-CHRDw87lNS0E9B2TibgpnMV
.text:080484DF                 add     ebx, eax        ; ebx = &hardcoded_str[func_1_rv]
.text:080484E1                 mov     dl, [ebx]       ; dl = hardcoded_str[func_1_rv]
.text:080484E3                 mov     edi, [esp+8]    ; edi = clef
.text:080484E7                 mov     dh, [edi]       ; dh = clef[0]
.text:080484E9                 cmp     dl, dh          ; clef[0] == hardcoded_str[func_1_rv]
.text:080484EB                 jnz     short exit      ; if not equal then exit
```

The snippet translates to:

```
char hardcoded[] = "A-CHRDw87lNS0E9B2TibgpnMVys5XzvtOGJcYLU+4mjW6fxqZeF3Qa1rPhdKIouk";
char* pseudo = argv[1];
char* clef = argv[2];
int pseudo_length = strlen(pseudo);
int func_1_rv = func_1(pseudo_length);
if( clef[0] != hardcoded_str[func_1_rv] )
    exit(1); // we failed
```

So the we know that **the first character of** `clef` **needs to be at position** `func_1_rv` **into** `hardcoded_str`**, where** `func_1_rv` **is the return value of** `func_1`::

```
.text:080485CD func_1          proc near               ; CODE XREF: main+85p
.text:080485CD
.text:080485CD pseudo_length   = dword ptr  8
.text:080485CD
.text:080485CD                 push    ebp
.text:080485CE                 mov     ebp, esp
.text:080485D0                 mov     eax, [ebp+pseudo_length]
.text:080485D3                 and     eax, 0FFh
.text:080485D8                 xor     al, 3Bh
.text:080485DA                 and     eax, 3Fh
.text:080485DF                 leave
.text:080485E0                 retn    4
.text:080485E0 func_1          endp
```

This short function boils down to::

```
int func_1(int pseudo_length)
{
    return (pseudo_length ^ 0x3B) & 0x3F;
}
```

Key's Second Character

Next follows disassembly that looks almost the same as the one from the previous section::

```
.text:080484ED                mov     edi, [esp]      ; edi = pseudo_length
.text:080484F0                push    edi             ; arg1 = pseudo_length
.text:080484F1                mov     edi, [esp+8]    ; edi = pseudo
.text:080484F5                push    edi             ; arg0 = pseudo
.text:080484F6                call    func_2
.text:080484FB                mov     ebx, offset hardcoded_str ; "A-CHRDw87lNS0E9B2TibgpnMV
.text:08048500                add     ebx, eax        ; ebx = &hardcoded_str[func_2_rv]
.text:08048502                mov     dl, [ebx]       ; dl = hardcoded_str[func_2_rv]
.text:08048504                mov     edi, [esp+8]    ; edi = clef
.text:08048508                inc     edi             ; edi = &clef[1]
.text:08048509                mov     dh, [edi]       ; dh = clef[1]
.text:0804850B                cmp     dl, dh          ; clef[1] == hardcoded_str[func_2_rv]
.text:0804850D                jnz     exit            ; if not equal then exit
```

The only differences are:

```
- a different subroutine gets called, I renamed it as ``func_2``. The first parameter to ``fu
- the instruction ``inc edi`` makes ``edi`` reference ``clef[1]``.
```

There's one pitfall: Because of the preceeding `push edi` in the second line, `[esp+8]` points to `pseudo`, and not `clef` as you might expect. Be careful when interpreting stack references based on `esp` rather than `ebp`.

This is the C code for the snippet::

```
int func_2_rv = func_2(pseudo, pseudo_length);
if( clef[1] != hardcoded_str[func_2_rv] )
    exit(1); // we failed
```

The subroutine `func_2` looks as follows::

```
.text:080485E3 func_2          proc near               ; CODE XREF: main+A6p
.text:080485E3
.text:080485E3 pseudo          = dword ptr  8
.text:080485E3 pseudo_length   = dword ptr  0Ch
.text:080485E3
.text:080485E3                 push    ebp
.text:080485E4                 mov     ebp, esp
.text:080485E6                 sub     esp, 8
.text:080485EC                 mov     ecx, [ebp+pseudo_length]
.text:080485EF                 mov     esi, [ebp+pseudo]
.text:080485F2                 add     esi, ecx
.text:080485F4                 xor     eax, eax
.text:080485F6                 jmp     loc_8048606
.text:080485FB ; ---------------------------------------------------------------------------
.text:080485FB
.text:080485FB loc_80485FB:                            ; CODE XREF: func_2+2Aj
.text:080485FB                 dec     esi
.text:080485FC                 mov     dl, [esi]
```

```
.text:080485FE                 and     edx, 0FFh
.text:08048604                 add     eax, edx
.text:08048606
.text:08048606 loc_8048606:                            ; CODE XREF: func_2+13j
.text:08048606                 dec     ecx
.text:08048607                 cmp     ecx, 0FFFFFFFFh
.text:0804860D                 jnz     short loc_80485FB
.text:0804860F                 xor     al, 4Fh
.text:08048611                 and     eax, 3Fh
.text:08048616                 leave
.text:08048617                 retn    8
.text:08048617 func_2          endp
```

Which represents the following loop::

```
int func_2(char* pseudo, int pseudo_length)
{
    int res = 0;
    for(int i = 0; i < pseudo_length; i++)
        res += pseudo[i];
    return (res ^ 0x4F) & 0x3F;
}
```

Key's Third Character

The next lines again look almost like the checks in for the first two characters of `clef`::

```
.text:08048513                 mov     edi, [esp]
.text:08048516                 push    edi
.text:08048517                 mov     edi, [esp+8]
.text:0804851B                 push    edi
.text:0804851C                 call    func_3
.text:08048521                 mov     ebx, offset hardcoded_str ; "A-CHRDw87lNS0E9B2TibgpnMV
.text:08048526                 add     ebx, eax
.text:08048528                 mov     dl, [ebx]
.text:0804852A                 mov     edi, [esp+8]
.text:0804852E                 add     edi, 2
.text:08048534                 mov     dh, [edi]
.text:08048536                 cmp     dl, dh          ; clef[2] == hardcoded_str[func_3_rv]
.text:08048538                 jnz     exit
```

The only difference being the new subroutine `func_3` and the reference to the third character of `clef`::

```
int func_3_rv = func_3(pseudo, pseudo_length);
if( clef[2] != hardcoded_str[func_3_rv] )
    exit(1); // we failed
```

This is `func_3`::

```
.text:0804861A func_3          proc near               ; CODE XREF: main+CCp
.text:0804861A
.text:0804861A pseudo          = dword ptr  8
.text:0804861A pseudo_length   = dword ptr  0Ch
.text:0804861A
.text:0804861A                 push    ebp
.text:0804861B                 mov     ebp, esp
.text:0804861D                 mov     eax, 1
.text:08048622                 mov     esi, [ebp+pseudo]
.text:08048625                 mov     ecx, [ebp+pseudo_length]
```

```
.text:08048628                 jmp     loc_804863C
.text:0804862D ; ---------------------------------------------------------------------------
.text:0804862D
.text:0804862D loc_804862D:                            ; CODE XREF: func_3+29j
.text:0804862D                 xor     ebx, ebx
.text:0804862F                 mov     bl, [esi]
.text:08048631                 and     bl, 0FFh
.text:08048634                 mul     ebx
.text:08048636                 and     eax, 0FFh
.text:0804863B                 inc     esi
.text:0804863C
.text:0804863C loc_804863C:                            ; CODE XREF: func_3+Ej
.text:0804863C                 dec     ecx
.text:0804863D                 cmp     ecx, 0FFFFFFFFh
.text:08048643                 jnz     short loc_804862D
.text:08048645                 xor     al, 55h
.text:08048647                 and     eax, 3Fh
.text:0804864C                 leave
.text:0804864D                 retn    8
.text:0804864D func_3          endp
```

Again we loop over all characters in `pseudo`, this time multiplying the ASCII codes rather than adding them up as in `func_2`::

```
int func_3(char* pseudo, int pseudo_length)
{
    int res = 1;
    for(int i = 0; i < pseudo_length; i++)
        res *= pseudo[i];
    return (res ^ 0x55) & 0x3F;
}
```

Key's Fourth Character

Next in our `main` routine follows more of the same::

```
.text:0804853E                 mov     edi, [esp]
.text:08048541                 push    edi
.text:08048542                 mov     edi, [esp+8]
.text:08048546                 push    edi
.text:08048547                 call    func_4
.text:0804854C                 mov     ebx, offset hardcoded_str ; "A-CHRDw87lNS0E9B2TibgpnMV
.text:08048551                 add     ebx, eax
.text:08048553                 mov     dl, [ebx]
.text:08048555                 mov     edi, [esp+8]
.text:08048559                 add     edi, 3
.text:0804855F                 mov     dh, [edi]
.text:08048561                 cmp     dl, dh          ; clef[3] == hardcoded_str[func_4_rv]
.text:08048563                 jnz     exit
```

or in C:

```
int func_4_rv = func_4(pseudo, pseudo_length);
if( clef[3] != hardcoded_str[func_4_rv] )
    exit(1); // we failed
```

The subroutine `func_4` is::

```
.text:08048650 func_4          proc near               ; CODE XREF: main+F7p
.text:08048650
.text:08048650 pseudo          = dword ptr  8
.text:08048650 pseudo_length   = dword ptr  0Ch
.text:08048650
.text:08048650                 push    ebp
.text:08048651                 mov     ebp, esp
.text:08048653                 sub     esp, 4
.text:08048659                 mov     esi, [ebp+pseudo]
.text:0804865C                 mov     al, [esi]
.text:0804865E                 mov     ecx, [ebp+pseudo_length]
.text:08048661                 jmp     loc_804866F
.text:08048666 ; ---------------------------------------------------------------------------
.text:08048666
.text:08048666 loc_8048666:                            ; CODE XREF: func_4+26j
.text:08048666                 inc     esi
.text:08048667                 mov     bl, [esi]
.text:08048669                 cmp     bl, al
.text:0804866B                 jbe     short loc_804866F
.text:0804866D                 mov     al, bl
.text:0804866F
.text:0804866F loc_804866F:                            ; CODE XREF: func_4+11j
.text:0804866F                                         ; func_4+1Bj
.text:0804866F                 dec     ecx
.text:08048670                 cmp     ecx, 0FFFFFFFFh
.text:08048676                 jnz     short loc_8048666
.text:08048678                 xor     al, 0Eh
.text:0804867A                 push    eax             ; seed
.text:0804867B                 call    _srand
.text:08048680                 call    _rand
.text:08048685                 and     eax, 3Fh
.text:0804868A                 leave
.text:0804868B                 retn    8
.text:0804868B func_4          endp
```

It contains two library calls that IDA identifies as `_srand` and `_rand`. The former initializes the random number generator, the latter generates a random integer between 0 and `RAND_MAX`. This is the `func_4` in C++::

```
#include <stdlib.h>  /* for srand and rand */
int func_4(char* pseudo, int pseudo_length)
{
    int res = pseudo[0];
    for(int i = 0; i < pseudo_length; i++)
        if(pseudo[i] > res)
            res = pseudo[i];
    srand(res ^0xE);
    return rand() & 0x3F;
}
```

Key's Fifth Character

The next block still offers nothing new::

```
.text:08048569                 mov     edi, [esp]
.text:0804856C                 push    edi
.text:0804856D                 mov     edi, [esp+8]
.text:08048571                 push    edi
.text:08048572                 call    func_5
.text:08048577                 mov     ebx, offset hardcoded_str ; "A-CHRDw87lNS0E9B2TibgpnMV
.text:0804857C                 add     ebx, eax
.text:0804857E                 mov     dl, [ebx]
```

```
.text:08048580                 mov     edi, [esp+8]
.text:08048584                 add     edi, 4
.text:0804858A                 mov     dh, [edi]
.text:0804858C                 cmp     dl, dh
.text:0804858E                 jnz     exit
```

It's yet another one of our checks::

```
int func_5_rv = func_5(pseudo, pseudo_length);
if( clef[4] != hardcoded_str[func_5_rv] )
    exit(1); // we failed
```

The routine `func_5` is::

```
.text:0804868E func_5          proc near               ; CODE XREF: main+122p
.text:0804868E
.text:0804868E pseudo          = dword ptr  8
.text:0804868E pseudo_length   = dword ptr  0Ch
.text:0804868E
.text:0804868E                 push    ebp
.text:0804868F                 mov     ebp, esp
.text:08048691                 xor     ebx, ebx
.text:08048693                 mov     esi, [ebp+pseudo]
.text:08048696                 mov     ecx, [ebp+pseudo_length]
.text:08048699                 dec     ecx
.text:0804869A                 jmp     loc_80486BC
.text:0804869F ; ---------------------------------------------------------------------------
.text:0804869F
.text:0804869F loc_804869F:                            ; CODE XREF: func_5+34j
.text:0804869F                 xor     edx, edx
.text:080486A1                 mov     dl, [esi]
.text:080486A3                 push    ecx
.text:080486A4                 push    ebx
.text:080486A5                 push    2
.text:080486AA                 push    edx
.text:080486AB                 call    sub_8048708
.text:080486B0                 pop     ebx
.text:080486B1                 pop     ecx
.text:080486B2                 add     ebx, eax
.text:080486B4                 and     ebx, 0FFh
.text:080486BA                 inc     esi
.text:080486BB                 dec     ecx
.text:080486BC
.text:080486BC loc_80486BC:                            ; CODE XREF: func_5+Cj
.text:080486BC                 cmp     ecx, 0FFFFFFFFh
.text:080486C2                 jnz     short loc_804869F
.text:080486C4                 mov     eax, ebx
.text:080486C6                 xor     al, 0EFh
.text:080486C8                 and     eax, 3Fh
.text:080486CD                 leave
.text:080486CE                 retn    8
.text:080486CE func_5          endp
```

With another call to a subroutine `sub_8048708`::

```
.text:08048708 sub_8048708     proc near               ; CODE XREF: func_5+1Dp
.text:08048708
.text:08048708 arg_0           = dword ptr  8
.text:08048708 arg_4           = dword ptr  0Ch
.text:08048708
.text:08048708                 push    ebp
.text:08048709                 mov     ebp, esp
.text:0804870B                 mov     ecx, [ebp+arg_4]
```

```
.text:0804870E                 mov     eax, 1
.text:08048713                 cmp     ecx, 0
.text:08048719                 jz      short locret_804872A
.text:0804871B                 mov     ebx, [ebp+arg_0]
.text:0804871E                 mul     ebx
.text:08048720                 jmp     loc_8048727
.text:08048725 ; ---------------------------------------------------------------------------
.text:08048725
.text:08048725 loc_8048725:                            ; CODE XREF: sub_8048708+20j
.text:08048725                 mul     ebx
.text:08048727
.text:08048727 loc_8048727:                            ; CODE XREF: sub_8048708+18j
.text:08048727                 dec     ecx
.text:08048728                 jnz     short loc_8048725
.text:0804872A
.text:0804872A locret_804872A:                         ; CODE XREF: sub_8048708+11j
.text:0804872A                 leave
.text:0804872B                 retn    8
.text:0804872B sub_8048708     endp
```

This second subroutine returns `arg_0` raised to the power of `arg_4`::

```
sub_8048708(arg_0, arg_4) = pow(arg_0, arg_4); // arg_0 ** arg_4
```

So `func_5` translates to the following C code::

```
int func_5(char* pseudo, int pseudo_length)
{
    int res = 0;
    for(int i = 0; i < pseudo_length; i++)
    {
        res += pseudo[i]*pseudo[i]; // sub_8048708(pseudo[i], 2)
    }
    return (res ^ 0xEF) & 0x3F;
}
```

Key's Sixth Character

Finally we get a snippet that looks slightly different::

```
.text:08048594                 mov     edi, [esp+4]    ; edi = pseudo
.text:08048598                 xor     edx, edx        ; edx = 0
.text:0804859A                 mov     dl, [edi]       ; dl = pseudo[0]
.text:0804859C                 push    edx             ; arg0 = pseudo[0]
.text:0804859D                 call    func_6
.text:080485A2                 mov     ebx, offset hardcoded_str ; "A-CHRDw87lNS0E9B2TibgpnMV
.text:080485A7                 add     ebx, eax
.text:080485A9                 mov     dl, [ebx]
.text:080485AB                 mov     edi, [esp+8]
.text:080485AF                 add     edi, 5
.text:080485B5                 mov     dh, [edi]
.text:080485B7                 cmp     dl, dh          ; clef[5] == hardcoded_str[func_6_rv]
.text:080485B9                 jnz     exit
```

This time the subroutine `func_6` takes the first character of `pseudo` as the only parameter::

```
int func_6_rv = func_6(pseudo[0]);
if( clef[5] != hardcoded_str[func_6_rv] )
    exit(1); // we failed
```

Here's `func_6`::

```
.text:080486D1 func_6          proc near               ; CODE XREF: main+14Dp
.text:080486D1
.text:080486D1 pseudo_0        = dword ptr  8
.text:080486D1
.text:080486D1                 push    ebp
.text:080486D2                 mov     ebp, esp
.text:080486D4                 xor     eax, eax
.text:080486D6                 mov     esi, [ebp+pseudo_0]
.text:080486D9                 cmp     esi, 0
.text:080486DF                 jz      short loc_80486FF
.text:080486E1                 call    _rand
.text:080486E6                 mov     ecx, [ebp+pseudo_0]
.text:080486E9                 jmp     loc_80486F5
.text:080486EE ; ---------------------------------------------------------------------------
.text:080486EE
.text:080486EE loc_80486EE:                            ; CODE XREF: func_6+25j
.text:080486EE                 push    ecx
.text:080486EF                 call    _rand
.text:080486F4                 pop     ecx
.text:080486F5
.text:080486F5 loc_80486F5:                            ; CODE XREF: func_6+18j
.text:080486F5                 dec     ecx
.text:080486F6                 jnz     short loc_80486EE
.text:080486F8                 and     eax, 0FFh
.text:080486FD                 xor     al, 0E5h
.text:080486FF
.text:080486FF loc_80486FF:                            ; CODE XREF: func_6+Ej
.text:080486FF                 and     eax, 3Fh
.text:08048704                 leave
.text:08048705                 retn    4
.text:08048705 func_6          endp
```

The snippet just takes `n` random numbers and returns the last one, where `n` is the ASCII code of our letter `pseudo[0]`::

```
int func_6(char pseudo0)
{
    int res = 0;
    for(int i = 0; i < pseudo0; i++)
        res = rand();
    return (res ^ 0xE5) & 0x3F;
}
```

The Goodboy Message

If we passed all 6 tests for the characters in `clef` we get to this snippet::

```
.text:080485BF                 push    offset aBravo   ; "Bravo !!\n"
.text:080485C4                 call    _printf
.text:080485C9                 xor     eax, eax
.text:080485CB                 leave
.text:080485CC                 retn
.text:080485CC main            endp ; sp-analysis failed
```

Which is prints the `Bravo !!\n` goodboy message before returning 0::

```
printf("Bravo !!\n")
return 0;
```

The Keygen

Putting together all six tests we obtain our keygenerator::

```
#include <stdio.h>
#include <cstring>
#include <stdlib.h>

int func_1(int pseudo_length)
{
    return (pseudo_length ^ 0x3B) & 0x3F;
}

int func_2(char* pseudo, int pseudo_length)
{
    int res = 0;
    for(int i = 0; i < pseudo_length; i++)
        res += pseudo[i];
    return (res ^ 0x4F) & 0x3F;
}

int func_3(char* pseudo, int pseudo_length)
{
    int res = 1;
    for(int i = 0; i < pseudo_length; i++)
        res *= pseudo[i];
    return (res ^ 0x55) & 0x3F;
}

int func_4(char* pseudo, int pseudo_length)
{
    int res = pseudo[0];
    for(int i = 0; i < pseudo_length; i++)
        if(pseudo[i] > res)
            res = pseudo[i];
    srand(res ^0xE);
    return rand() & 0x3F;
}

int func_5(char* pseudo, int pseudo_length)
{
    int res = 0;
    for(int i = 0; i < pseudo_length; i++)
    {
        res += pseudo[i]*pseudo[i];
    }
    return (res ^ 0xEF) & 0x3F;
}

int func_6(char pseudo0)
{
    int res = 0;
    for(int i = 0; i < pseudo0; i++)
        res = rand();
    return (res ^ 0xE5) & 0x3F;
}

int main(int argc, char *argv[])
{
    if( argc != 2 )
        printf("usage: keygen <pseudo>\n");
```

```
    char hardcoded[] = "A-CHRDw87lNS0E9B2TibgpnMVys5XzvtOGJcYLU+4mjW6fxqZeF3Qa1rPhdKIouk";
    char* pseudo = argv[1];
    char clef[7];
    int pseudo_length = strlen(pseudo);
    clef[0] = hardcoded[func_1(pseudo_length)];
    clef[1] = hardcoded[func_2(pseudo, pseudo_length)];
    clef[2] = hardcoded[func_3(pseudo, pseudo_length)];
    clef[3] = hardcoded[func_4(pseudo, pseudo_length)];
    clef[4] = hardcoded[func_5(pseudo, pseudo_length)];
    clef[5] = hardcoded[func_6(pseudo[0])];
    clef[6] = 0;
    printf("pseudo: %s\n", pseudo);
    printf("clef: %s\n", clef);
}
```