Crackmes.de – Matteo KeygenMe by Matteo

Johannes

April 28, 2015

The crackme Matteo KeygenMe by Matteo has been published February 24, 2015. It is rated at *4 - Needs special knowledge*. The crackme is written in Assembler and runs on Windows.

The crackme has two major parts. The first part is all about trying to stop you from getting to the relevant code by throwing a handful of anti-debugging and anti-disassembly techniques at you. The second part then validates the key file that you provide.

I used IDA Pro and WinDBG to solve the crackme. No anti-debugging scripts have been used — all tricks have been manually defused.

Part 1: Anti-Debugging Measures

Trick 1: TLS Callbacks

The crackme uses TLS callbacks. Those are invoked *before* the main entry point:

```
.data:0040400C TlsCallbacks_ptr dd offset TlsCallbacks
.data:00404010 TlsSizeOfZeroFill dd 0
.data:00404014 TlsCharacteristics dd 0
.data:00404018 TlsIndex        db    0
.data:00404019                 db    0
.data:0040401A                 db    0
.data:0040401B                 db    0
.data:0040401C TlsCallbacks    dd offset TlsCallback_0
.data:0040401C
.data:00404020 dword_404020    dd 0
.data:00404024 dword_404024    dd 0
```

Make sure you have set a breakpoint at the `TlsCallback_0` offset, or that the debugger is set to stop at TLS callbacks.

Trick 2: Dynamically Added TLS Callbacks

All the first TLS callback does is create another TLS callback:

```
00401EC7 TlsCallback_0 proc near
00401EC7 mov     dword_404020, offset loc_401E76
00401ED1 retn
00401ED1 TlsCallback_0 endp
```

Add a breakpoint at `loc_401E76` and run there.

Trick 3: Anti-Disassembly with Fake Jumps and Garbage Bytes

The second TLS callback routine at first probably looks like that:

```
00401E76 push    (offset loc_401E7F+1)
00401E7B stc
00401E7C jnb     short loc_401E7F
00401E7E retn
00401E7F ; ---------------------------------------------------------------------------
00401E7F
00401E7F loc_401E7F:
00401E7F
00401E7F jmp     dword ptr [esi-74h]
00401E82 ; ---------------------------------------------------------------------------
00401E82 shl     byte ptr [esi-72h], 1
00401E85 shl     byte ptr [esi-64h], 1
```

The above code uses a fake conditional jump (the jump is always taken), a PUSH/POP-jump (the RETN instruction actually jumps to loc_401E7F+1), and garbage bytes. This causes the wrong disassembly above. Once you step into the code with debugger, IDA should show the correct disassembly. You can also manually fix the disassembly:

```
00401E76 push    offset loc_401E80
00401E7B stc
00401E7C jnb     short near ptr byte_401E7F
00401E7E retn
00401E7E ; ---------------------------------------------------------------------------
00401E7F byte_401E7F db 0FFh
00401E80 ; ---------------------------------------------------------------------------
00401E80
00401E80 loc_401E80:
00401E80 mov     ax, ss
00401E83 mov     ss, ax
```

Trick 4: Trap Flag

Next come theses instructions:

```
00401E80 mov     ax, ss
00401E83 mov     ss, ax
00401E86 pushfw
00401E88 test    byte ptr [esp+1], 1
00401E8D jnz     short loc_401E9C
00401E8F popfw
```

The second MOV-instruction writes SS, this causes the processor to lock all interrupts until *after* the following instruction. The PUSHFW instruction then pushes the flags on the stack, and `[esp+1]` accesses the *trap flag*. Since interrupts, including INT 1, are locked until after PUSHFW, the debugger has no chance to unset the flag.

Either don't single step over the pushfw instruction, or manually set the zero flag at the jump. The routine sets a third callback routine.

Trick 5: Push/Ret-Jumps

Because of the anti-disassembly techniques before, the third callback routine is not yet marked as code:

```
00401DBB db  68h ; h
00401DBC db 0C5h ; +
00401DBD db  1Dh
00401DBE db  40h ; @
00401DBF db    0
00401DC0 db 0F8h ; °
00401DC1 db  72h ; r
00401DC2 db    1
```

Hitting `C` in IDA produces the following disassembly:

```
00401DBB ; ---------------------------------------------------------------------------
00401DBB push    offset off_401DC5
00401DC0 clc
00401DC1 jb      short near ptr unk_401DC4
00401DC3 retn
00401DC3 ; ----------------------------
```

This snippet uses the PUSH/RET-jump. Pushing the offset and then returning is almost equivalent to a regular jump, but they might prevent the disassembler from properly detecting the function boundaries.

Trick 6: Self-Modifying Code

The following instructions follow:

```
00401DCF loc_401DCF:
00401DCF mov     decrypted, 1
00401DD6 mov     eax, 771881844                  ; key
00401DDB mov     ecx, offset start_of_ciphertext
00401DE0 mov     edx, offset end_of_ciphertext
00401DE5
00401DE5 decryption_loop:
00401DE5 xor     [ecx], al
00401DE7 ror     eax, 1
00401DE9 inc     ecx
00401DEA cmp     ecx, edx
00401DEC jnz     short decryption_loop
00401DEC ; ---------------------------------------------------------------------------
00401DEE start_of_ciphertext db 23h
00401DEF db 0DEh ; ¦
00401DF0 db  56h ; V
00401DF1 db  7Bh ; {
00401DF2 db  87h ; ç
00401DF3 db 0DBh ; ¦
```

After the decryption the cipertext turns into meaningful code:

```
00401DEE start_of_ciphertext
00401DEE push    edi
00401DEF mov     edx, large fs:30h
...
```

The crackme uses similar snippets in many places. Make sure to not set memory breakpoint inside the modified section. Memory breakpoints affect the memory, and those changes will get encrypted or decrypted too, leading to corrupt code.

Trick 7: PEB->BeingDebugged

The code then checks the `BeingDebugged`-flag of the PEB (equivalent to calling `IsDebuggerPresent`, but more sneaky):

```
00401DEE push    edi
00401DEF mov     edx, large fs:30h
00401DF6 mov     al, [edx+2]
00401DF9 test    al, al
00401DFB jnz     short being_debugged
```

Manually patch the flag, or set the zero flag at the jump.

Trick 8: NtGlobalFlag

With `edx` still pointing to the PEB, the crackme also checks the `NtGlobalFlag` field. The field has three flags that indicate the presence of a debugger:

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

The proper way to check all three flags would therefore be with bitmask 0x70. The crackme is cruder and just checks if the entire `NtGlobalFlag` field is zero:

```
00401DFD cmp     dword ptr [edx+68h], 0
00401E01 jnz     short being_debugged
```

Again patch the field or manually set the zero flag.

Trick 9: Software Breakpoint Detection

The entry point of any executable is an obvious place to put software breakpoints (opcode 0xCC). The next lines search 0x26 bytes of the memory starting at the entry point for 0xCC:

```
00401E04 mov     edi, offset start
00401E09 mov     ecx, 26h
00401E0E mov     al, 0CCh
00401E10 repne scasb
00401E12 jz      short being_debugged
```

Remove software breakpoints before the check, or manually unset the zero flag.

Trick 10: SEH - Triggered with Single Step Exception

After three more self modifying loops, we end here:

```
00401D01 push    8E4C9A90h
00401D06 push    838042730
...
00401D13 mov     edx, large fs:18h
00401D1A mov     ecx, [edx]
00401D1C xchg    ebx, ecx
00401D1E pop     eax
00401D1F add     [esp], eax
00401D22 push    ebx
00401D23 mov     ebx, esp
00401D25 xchg    ebx, ecx
00401D27 mov     [edx], ecx
00401D29 pushfw
00401D2B or      byte ptr [esp+1], 1
00401D30 popfw
00401D32 mov     eax, offset off_404041
00401D37 mov     byte ptr [eax], 0D4h
```

The offset fs:18h points to the Structured Exception Handler (SEH) on the stack. The code adds another handler with address 8E4C9A90h + 31F3846Ah = 0x401efa (mod 2**32). The exception is triggered by setting the trap flag (pushfw, or, popfw). Set a breakpoint at the new SEH handler. Then make sure you pass the single step exception 0x80000004 to the application. Debuggers often consume this exception (it is intended for debuggers after all).

Trick 11: Exception's Context Structure - Safe Place

The exception handler at 0x401EFA first decrypts the following block:

```
00401F12 start_of_ciphertext_0:
00401F12 mov     eax, [esp+0Ch]
00401F16 mov     dword ptr [eax+0B8h], offset loc_401D3A
```

The third argument to the exception handler, passed in `[eax+0Ch]`, is the context structure of the exception:

```
struct _CONTEXT
    ....
    +B8     EIP;  # register
    ...
```

The field 0xB8 is set to 0x401D37, or the EIP we would return to after the exception handler. By overwriting the register to 0x401D3A, the crackme changes the flow to a different address. Take note of the address to later set a breakpoint there when returning from the SEH.

Trick 11: Exception's Context Structure - Hardware Registers

The crackme then also checks four other offsets into the context structure:

```
00401F23 test    [eax+4], edx
00401F26 jnz     being_debugged_0
00401F2C test    [eax+8], edx
00401F2F jnz     being_debugged_0
00401F35 test    [eax+0Ch], edx
00401F38 jnz     being_debugged_0
00401F3E test    [eax+10h], edx
00401F41 jnz     being_debugged_0
00401F47 push    esi
00401F48 xor     edx, edx
```

Those four values are the debug registers:

```
struct _CONTEXT
    ...
    +04     Dr0;
    +08     Dr1;
    +0C     Dr2;
    +10     Dr3;
    ...
```

They indicate if the hardware breakpoints are set. Disable all hardware breakpoints, or manually unset the zero flag at each jump.

Trick 12: Checksum

The following disassembly calculates a checksum of 0x332 bytes from the entry point of the crackme:

```
00401F4A mov     esi, offset start
00401F4F mov     ecx, 332h
00401F54 cld
00401F55
00401F55 loc_401F55:
00401F55 lodsd
00401F56 add     edx, eax
00401F58 rol     edx, 1
00401F5A dec     ecx
00401F5B jnz     short loc_401F55
00401F5D pop     esi
00401F5E cmp     edx, checksum
00401F64 jz      short checksum_is_good
00401F66 mov     eax, [esp+0Ch]
00401F6A mov     dword ptr [eax+0B8h], offset loc_401D32
00401F74 jmp     being_debugged_0
00401F79 ; ---------------------------------------------------------------------------
```

If the checksum does not match a hardcoded value, we are busted. The checksum method detects software breakpoints as well any kind of patches. Remove breakpoints and patches, or manually set the zero flag at 401f64.

Trick 13: Correct Exception Record -> C3

The first parameter of the exception handler points to an `EXCEPTION_RECORD`:

```
struct _EXCEPTION_RECORD
    +00:    ExceptionCode DWORD
    +04:    ExceptionFlags DWORD
    ...
```

The next instructions of the crackme read the exception code:

```
00401F79 checksum_is_good:
00401F79 mov     edx, [esp+4]
00401F7D mov     edx, [edx]
```

We know it was a single step exception that triggered the exception, see Trick 10. This exception has the code 0x80000004:

```
#define STATUS_SINGLE_STEP               ((DWORD) 0x80000004)
```

By shifting that right 5 bytes and adding 0x33h, we get the value 0xC3

```
00401F7F rol     edx, 5       ; ---> 80
00401F82 add     edx, 33h     ; ---> C3
```

The value 0xC3 is stored at 2 bytes into 0x404041

```
00401F85 mov     eax, offset byte_404041
00401F8A mov     [eax+2], dl
```

So the data becomes:

```
.data:00404041 dword_404041 dd 16C381B9h
```

The effect of the change will be relevant at the start of the entry point. Make sure that C3 is written, and manually correct the byte if your debugger changed the exception code.

Trick 14: Check Software Breakpoint at Popular API Calls

The crackme then checks if there is a software breakpoint at the jump table entry for *GetProcessAddress*:

```
00401F8D mov     edx, offset get_process_address
00401F92 cmp     byte ptr [edx], 0CCh
00401F95 jz      being_debugged_1
```

The *get_process_address* address disassembles to:

```
00402162 get_process_address:
00402162                                         ; 004020C5p ...
00402162 jmp     ds:GetProcAddress
```

The crackme also checks the actual offset of *GetProcAddress* by extracting the offset from the `jmp` instruction:

```
00401F9B mov     edx, [edx+2]
00401F9E mov     edx, [edx]
00401FA0 cmp     byte ptr [edx], 0CCh
00401FA3 jz      being_debugged_1
```

The crackme repeats the same two checks for *GetModuleHandleA* and *GetModuleHandleW*. Disable plugins that add breakpoints at these API, or just unset the zero flag.

Trick 15: Thread Hiding

The crackme dynamically loads the *NtSetInformationThread* function:

```
0040207A apis_clean:
0040207A push    offset aNtdll_dll               ; "NtDll.dll"
0040207F call    get_module_handle_w
00402084 test    eax, eax
00402086 jz      loc_40211A
0040208C push    offset aNtsetinformationth      ; "NtSetInformationThread"
00402091 push    eax
00402092 call    get_process_address
00402097 test    eax, eax
00402099 jz      short loc_40211A
```

The crackme also checks if there is a breakpoint at the beginning of the routine – hinting a potential anti-anti measure:

```
0040209B mov     edx, eax
0040209D cmp     byte ptr [edx], 0CCh
004020A0 jz      short being_debugged_1
```

Next, *NtSetInformationThread* is called with 0x11 as the second parameter:

```
004020A2 push    0
004020A4 push    0
004020A6 push    11h
004020A8 push    0FFFFFFFEh
004020AA call    eax
```

0x11 stands for *HideThreadFromDebugger*, which will causes the debugger to no longer receive any events. To skip the call in WinDbg you can use the following commands:

```
>r @eip = @eip + 2
>r @esp = @esp + 0x10
```

Trick 16: Patch Vectored Exception Handlers

The following disassembly follows. It patches the beginning of the API function *AddVectoredExceptionHandler* with the 5 bytes at *return_null*:

```
004020AC push    offset aKernel32_dll_0          ; "Kernel32.dll"
004020B1 call    get_module_handle_w
004020B6 push    ebx
004020B7 push    esi
004020B8 push    edi
004020B9 mov     ebx, eax
004020BB test    eax, eax
004020BD jz      short loc_402117
004020BF push    offset aWriteprocessmemory      ; "WriteProcessMemory"
004020C4 push    ebx
004020C5 call    get_process_address
004020CA test    eax, eax
```

```
004020CC jz      short loc_402117
004020CE mov     edi, eax
004020D0 cmp     byte ptr [edi], 0CCh
004020D3 jz      short being_debugged_1
004020D5 push    offset aAddvectoredexcepti      ; "AddVectoredExceptionHandler"
004020DA push    ebx
004020DB call    get_process_address
004020E0 test    eax, eax
004020E2 jz      short loc_402117
004020E4 push    0
004020E6 push    5
004020E8 push    offset return_null
004020ED push    eax
004020EE push    0FFFFFFFFh
004020F0 call    edi
```

The stub *return_null* is:

```
00402075            return_null:
00402075
00402075 33 C0      xor     eax, eax
00402077 C2 08 00   retn    8
```

So crackme replaces *AddVectoredExceptionHandler* with *return 0*. It does the same with *AddVectoredContinueHandler*. I had to ask the crackme author what the purpose of patching the VEH is. According to Matteo, they prevent VEH-based debugger from working, for example Cheat Engine.

After Trick 16, we leave the exception handler and return to the address that was set in Trick 11. First, the handler is removed:

```
00401D3A                            loc_401D3A:
00401D3A 64 8B 15 18 00 00 00       mov     edx, large fs:18h
00401D41 8F 02                      pop     dword ptr [edx]
00401D43 44                         inc     esp
00401D44 83 C4 08                   add     esp, 8
00401D47 83 EC 05                   sub     esp, 5
```

Next, we again calculate a checksum for the first bytes of the entry point:

```
00401D4A 33 D2                      xor     edx, edx
00401D4C 33 C0                      xor     eax, eax
00401D4E 56                         push    esi
00401D4F 57                         push    edi
00401D50 BE 00 10 40 00             mov     esi, offset start
00401D55 B9 26 00 00 00             mov     ecx, 26h
00401D5A FC                         cld
00401D5B
00401D5B                            loc_401D5B:
00401D5B AC                         lodsb
00401D5C 03 D0                      add     edx, eax
00401D5E D1 C2                      rol     edx, 1
00401D60 49                         dec     ecx
00401D61 75 F8                      jnz     short loc_401D5B
00401D63 81 FA 2B C6 16 5D          cmp     edx, offset checksum2
00401D69 74 17                      jz      short checksum_is_good2
```

This does the same as Trick 12. After that, we finally are done with the TlsCallbacks and enter the entry point of the crackme.

Trick 17: Timing Check

The first step of the entry point is to decrypt the content at 401026. The decryption key is stored at 0x404041, this is where Trick 13 made the modification based on the exception record:

```
00401000 public start
00401000 start:
00401000 mov     esi, offset loc_404041
00401005 mov     ecx, 0CA5h
0040100A mov     edx, offset loc_401026
0040100F cld
00401010
00401010 loc_401010:
00401010 lodsb
00401011 cmp     esi, offset loc_404047
00401017 jl      short loc_40101E
00401019 mov     esi, offset loc_404041
0040101E
0040101E loc_40101E:
0040101E dec     ecx
0040101F xor     [ecx+edx], al
00401022 test    ecx, ecx
00401024 jnz     short loc_401010
```

After some irrelevant code we get to these lines:

```
004010B4                add     esp, 4
004010B7                mov     time_in_secs, eax
004010BC                call    ds:GetTickCount
004010C2                xor     edx, edx
004010C4                mov     ecx, 3E8h
004010C9                div     ecx
004010CB                mov     tickcount_in_secs, eax
```

Here we store the current time in seconds (retrieved by an earlier call to `time`, not shown), and the tick count in seconds. These values become relevant further down in the crackme, here:

```
0040153D                cmp     tickcount_in_secs, 0
00401544                jl      short loc_40154F
00401546                cmp     tickcount_in_secs, 4
0040154D                jle     short good      ; At most 4 seconds passed
```

and here:

```
00401567                cmp     time_in_secs, 0
0040156E                jl      short bad
00401570                cmp     time_in_secs, 4
00401577                jg      short bad
```

These two blocks check if at most 4 seconds passed according to `time` or the tick count. If either one is true, we are fine. If on the other hand a debugger causes a greater delay than four seconds, then later the crackme jumps over setting a flag at offset *004015BC*:

```
004015B9 loc_4015B9:
004015B9                 jmp     short loc_4015BD
004015B9 ; ---------------------------------------------------------------------------
004015BB unk_4015BB      db 0EBh ; d
004015BC ; ---------------------------------------------------------------------------
004015BC
004015BC timing_ok:
004015BC                                         ; 004015A9j
004015BC                 inc     ecx
004015BD
004015BD loc_4015BD:
004015BD                                         ; loc_4015B9j
004015BD                 pop     edx
004015BE                 and     ecx, edx
```

These are all the anti-debugging checks that I found. Part 2 shows how the key validation works.

Part 2: Key Validation

The Keyfile

The registration information is stored in a file called `TheKey.k` in the same directory as the crackme:

```
004010E0                 push    0
004010E2                 push    80h
004010E7                 push    3
004010E9                 push    0
004010EB                 push    0
004010ED                 push    80000000h
004010F2                 push    offset aThekey_k ; "TheKey.k"
004010F7                 call    ds:CreateFile
004010FD                 xor     dword ptr ds:aThekey_k, offset unk_218F6F18 ; "TheKey.k"
00401107                 xor     dword ptr ds:aThekey_k+4, offset unk_218F6F18
00401111                 mov     fileHandle, eax
00401116                 test    eax, eax
00401118                 jz      fail
0040111E                 inc     eax
0040111F                 test    eax, eax
00401121                 jz      fail
00401127                 push    0
00401129                 push    offset ContentLength
0040112E                 push    40h
00401130                 push    offset keyContent
00401135                 push    fileHandle
0040113B                 call    ds:ReadFile
00401141                 push    fileHandle
00401147                 call    ds:CloseHandle
```

The keyfile needs to have 3 lines. The first two lines need to be terminated with `0xD` (carriage return). The last line must not have a line terminator. The following disassembly determines the length of the three lines, and store pointer to each line:

```
0040115A                mov     pKeyContent, offset keyContent
00401164                push    0Dh
00401166                push    15h
00401168                push    offset keyContent
0040116D                call    line_length
00401172                cmp     eax, 0FFFFFFFFh
00401175                jz      fail
0040117B                mov     line1_len, eax
00401180                mov     edx, offset keyContent
00401185                add     edx, eax
00401187                mov     byte ptr [edx], 0
0040118A                inc     edx
0040118B                inc     edx
0040118C                mov     dword ptr pLine2, edx
00401192                sub     ecx, eax
00401194                push    0Dh
00401196                push    15h
00401198                push    edx
00401199                call    line_length
0040119E                cmp     eax, 0FFFFFFFFh
004011A1                jz      fail
004011A7                mov     line2_len, eax
004011AC                mov     edx, offset keyContent
004011B1                add     edx, eax
004011B3                add     edx, line1_len
004011B9                add     edx, 2
004011BC                mov     byte ptr [edx], 0
004011BF                add     edx, 2
004011C2                mov     pLine3, edx
004011C8                push    15h
004011CA                push    edx
004011CB                call    line_length_f
004011D0                cmp     eax, 0FFFFFFFFh
004011D3                jz      fail
004011D9                mov     line3_len, eax
```

The content of the keyfile needs to be alphanumeric, i.e., only contain letters and digits:

```
004011EC                mov     esi, pKeyContent
004011F2
004011F2 loc_4011F2:
004011F2                cmp     ebx, 0
004011F5                jz      short loc_401205
004011F7                mov     al, [esi]
004011F9                push    eax
004011FA                call    is_alpha_numeric
004011FF                and     edi, eax
00401201                inc     esi
00401202                dec     ebx
00401203                jmp     short loc_4011F2
00401205 ; ---------------------------------------------------------------------------
00401205
00401205 loc_401205:
00401205                mov     ebx, line2_len
0040120B                mov     esi, dword ptr pLine2
```

```
00401211
00401211 loc_401211:
00401211                 cmp     ebx, 0
00401214                 jz      short loc_401224
00401216                 mov     al, [esi]
00401218                 push    eax
00401219                 call    is_alpha_numeric
0040121E                 and     edi, eax
00401220                 inc     esi
00401221                 dec     ebx
00401222                 jmp     short loc_401211
00401224 ; ---------------------------------------------------------------------------
00401224
00401224 loc_401224:
00401224                 mov     ebx, line3_len
0040122A                 mov     esi, pLine3
00401230
00401230 loc_401230:
00401230                 cmp     ebx, 0
00401233                 jz      short loc_401243
00401235                 mov     al, [esi]
00401237                 push    eax
00401238                 call    is_alpha_numeric
0040123D                 and     edi, eax
0040123F                 inc     esi
00401240                 dec     ebx
00401241                 jmp     short loc_401230
00401243 ; ---------------------------------------------------------------------------
00401243
00401243 loc_401243:
00401243                 mov     eax, edi
```

Finally, there is an obfuscated check to see if the first and second line of the key have the same length:

```
004012EF                 mov     ecx, line2_len
004012F5                 mov     unpredictable, eax
004012FA                 add     eax, 93E8h
004012FF                 sub     eax, unpredictable
00401305                 add     ecx, eax
00401307                 mov     edx, line1_len
0040130D                 sub     ecx, edx
0040130F                 xor     ecx, 75382
00401315                 cmp     ecx, 112030
0040131B                 jnz     fail
```

The above check boils down to:

```
(line2len - line1len + 0x93e8) XOR 75382 = 112030
(line2len - line1len + 0x93e8)           = 0x93e8
line1len                                 = line2len
```

Valid Second Line

The crackme applies a series of transformations to the first line:

```
00401267                mov     ecx, line1_len
0040126D                cmp     ecx, 3
00401270                jb      fail
00401276                xor     ecx, 5Ch
```

In pseudo-code:

```
line1_len = len(line1)
line1[0] ^= line1_len ^ 0x5c
```

Then we XOR characters from the end with characters at the start:

```
00401283                mov     ecx, line1_len
00401289                add     edx, ecx
0040128B                dec     edx
0040128C
0040128C loc_40128C:
0040128C                mov     cl, [edx]
0040128E                xor     [eax], cl
00401290                inc     eax
00401291                dec     edx
00401292                cmp     eax, edx
00401294                jl      short loc_40128C
```

In pseudo-code:

```
i = 1
j = line1_len-1
while i < j:
    line1[i] ^= line1[j]
    i += 1
    j -= 1
```

A similar routine follows:

```
00401353                mov     ecx, line1_len
00401359                dec     ecx
0040135A                shr     ecx, 1
0040135C                mov     eax, pLine1
00401361                mov     edx, eax
00401363                add     edx, ecx
00401365                inc     edx
00401366                mov     al, [eax]
00401368
00401368 loc_401368:
00401368                xor     [edx], al
0040136A                inc     al
0040136C                inc     edx
0040136D                cmp     byte ptr [edx], 0
00401370                jnz     short loc_401368
```

It does:

```
i = (line1_len-1)//2 + 1
c = line1[0]
for i in range(i, line1_len):
    line1[i] ^= c
    c += 1
```

Finally, the bytes in the line are made alphanumeric by calling:

```
00401378                 push    pLine1
0040137E                 call    make_alphanumeric
```

The routine *make_alphanumeric* is:

```
0040166A ; =============== S U B R O U T I N E =======================================
0040166A
0040166A ; Attributes: bp-based frame
0040166A
0040166A make_alphanumeric proc near
0040166A
0040166A data            = dword ptr  8
0040166A length          = dword ptr  0Ch
0040166A
0040166A                 push    ebp
0040166B                 mov     ebp, esp
0040166D                 push    edi
0040166E                 mov     edi, [ebp+data]
00401671
00401671 loc_401671:
00401671                 mov     cl, 25h
00401673                 cmp     [ebp+length], 0
00401677                 jz      short loc_4016A9
00401679
00401679 loc_401679:
00401679                 cmp     byte ptr [edi], '9'
0040167C                 jg      short loc_401685
0040167E                 cmp     byte ptr [edi], '0'
00401681                 jl      short loc_401685
00401683                 jmp     short loc_4016A3
00401685 ; ---------------------------------------------------------------------------
00401685
00401685 loc_401685:
00401685                                         ; make_alphanumeric+17j
00401685                 cmp     byte ptr [edi], 'Z'
00401688                 jg      short loc_401691
0040168A                 cmp     byte ptr [edi], 'A'
0040168D                 jl      short loc_401691
0040168F                 jmp     short loc_4016A3
00401691 ; ---------------------------------------------------------------------------
00401691
00401691 loc_401691:
00401691                                         ; make_alphanumeric+23j
00401691                 cmp     byte ptr [edi], 'z'
00401694                 jg      short loc_40169D
00401696                 cmp     byte ptr [edi], 'a'
00401699                 jl      short loc_40169D
```

```
0040169B                 jmp     short loc_4016A3
0040169D ; ---------------------------------------------------------------------------
0040169D
0040169D loc_40169D:
0040169D                                         ; make_alphanumeric+2Fj
0040169D                 add     [edi], cl
0040169F                 inc     cl
004016A1                 jmp     short loc_401679
004016A3 ; ---------------------------------------------------------------------------
004016A3
004016A3 loc_4016A3:
004016A3                                         ; make_alphanumeric+25j ...
004016A3                 inc     edi
004016A4                 dec     [ebp+length]
004016A7                 jmp     short loc_401671
004016A9 ; ---------------------------------------------------------------------------
004016A9
004016A9 loc_4016A9:
004016A9                 pop     edi
004016AA                 leave
004016AB                 retn    8
004016AB make_alphanumeric endp
004016AB
```

This routine decompiles to:

```
def make_alphanumeric(chars):
    for i in range(len(chars)):
        c = 37
        while not chr(chars[i]).isalnum():
            chars[i] += c
            c += 1
            chars[i] &= 0xFF
            c &= 0xFF
    return chars
```

The crackme then XORs the result with unpredictable data:

```
00401384                 mov     edi, pLine1
0040138A                 mov     edx, unpredictable
00401390                 mov     ecx, line1_len
00401396                 add     ecx, edi
00401398
00401398 loc_401398:
00401398                 xor     [edi], dl
0040139A                 inc     edi
0040139B                 rol     edx, 8
0040139E                 cmp     edi, ecx
004013A0                 jnz     short loc_401398
```

The crackme also XORs the second line with the same key. It then compares the transformed first and second line:

```
00401398 loc_401398:
```

```
00401398                xor     [edi], dl
0040139A                inc     edi
0040139B                rol     edx, 8
0040139E                cmp     edi, ecx
004013A0                jnz     short loc_401398
004013A2                push    esi
004013A3                mov     esi, pLine1
004013A9                mov     edi, dword ptr pLine2
004013AF                mov     ecx, line1_len
004013B5                cld
004013B6                repe cmpsb
004013B8                pop     esi
004013B9                pop     edi
004013BA                jnz     fail
```

The XOR encryption of the first and second line with the same key can be omitted, such that we have the following relationship between first and second line:

```
line1 = "phildunphy"
line1_len = len(line1)
line1_codes = [ord(c) for c in line1]
line1_codes[0] ^= line1_len ^ 0x5c
i = 1
j = line1_len-1

while i < j:
    line1_codes[i] ^= line1_codes[j]
    i += 1
    j -= 1

i = (line1_len-1)//2 + 1
c = line1_codes[0]
for i in range(i, line1_len):
    line1_codes[i] ^= c
    c += 1

x = make_alphanumeric(line1_codes)
line2 = ''.join([chr(xx) for xx in x])

# >>> second line is: K6LAUSIXAS
```

The Third Line

The third line is the trickiest. The crackme first generates two seeds based on the third and first line. The first seed is determined as follows:

```
004013C4                mov     edx, pLine3
004013CA                mov     eax, [edx]
004013CC                add     eax, [edx+4]
004013CF                mov     ecx, [edx+8]
004013D2                rol     eax, cl
004013D4                mov     ecx, [edx+0Ch]
004013D7                ror     eax, cl
004013D9                xor     eax, [edx+10h]
```

```
004013DC                 mov     edx, dword ptr line1_copy
004013E2                 add     edx, dword ptr line1_copy+4
004013E8                 mov     ecx, dword ptr line1_copy+8
004013EE                 rol     edx, cl
004013F0                 mov     ecx, dword ptr line1_copy+0Ch
004013F6                 ror     edx, cl
004013F8                 xor     edx, dword ptr line1_copy+10h
004013FE                 xor     edx, eax
00401400                 mov     seed1, edx
```

This boils down to the following pseudo-code:

```
def hash1(line):
    eax = get_int_from_string(line[:4])
    eax += get_int_from_string(line[4:8])
    ecx = get_int_from_string(line[8:12])
    eax = rol(eax, ecx & 0xFF)
    ecx = get_int_from_string(line[12:16])
    eax = ror(eax, ecx & 0xFF)
    ecx = get_int_from_string(line[16:20])
    eax ^= ecx
    return eax

eax = hash1(line3)
edx = hash1(line1)
eax ^= edx
seed1 = eax
```

With *get_int_from_string* converting a string into the little endian integer:

```
def get_str_from_int(val):
    s = ""
    for i in range(4):
        s += chr(val & 0xFF)
        val >>= 8

    return s
```

The second seed is calculated as follows:

```
00401406                 mov     eax, pLine3
0040140B                 add     eax, 9
0040140E                 mov     eax, [eax]
00401410                 rol     eax, 8
00401413                 xor     al, al
00401415                 ror     eax, 8
00401418                 mov     seed2, eax
```

Which is:

```
s = get_int_from_str(line3[9:13])
```

The seeds are then used to build a 16 bytes hash:

```
00401422                mov     eax, seed1
00401427                push    eax
00401428                call    ds:srand
0040142E                add     esp, 4
00401431                jmp     short loc_401442
00401433 ; ---------------------------------------------------------------------------
00401433
00401433 loc_401433:
00401433                mov     eax, seed2
00401438                push    eax
00401439                call    ds:srand
0040143F                add     esp, 4
00401442
00401442 loc_401442:
00401442                mov     esi, 4
00401447
00401447 loc_401447:
00401447                                          ; 00401473j
00401447                call    ds:rand
0040144D                push    4
0040144F                push    eax
00401450                call    modulus
00401455                mov     edi, eax
00401457                cmp     word ptr hash[ebx+edi*4], 0
00401460                jnz     short loc_401447
00401462                call    ds:rand
00401468                add     word ptr hash[ebx+edi*4], ax
00401470                dec     esi
00401471                test    esi, esi
00401473                jnz     short loc_401447
00401475                call    ds:rand
0040147B                xor     ecx, ecx
0040147D
0040147D loc_40147D:
0040147D                xor     word ptr hash[ecx*4], ax
00401485                inc     ecx
00401486                cmp     ecx, 4
00401489                jnz     short loc_40147D
0040148B                test    ebx, ebx
0040148D                jnz     short loc_40149C
0040148F                mov     ebx, 2
00401494                xor     seed2, eax
0040149A                jmp     short loc_401433
0040149C ; ---------------------------------------------------------------------------
0040149C
0040149C loc_40149C:
0040149C                call    ds:rand
004014A2                mov     ecx, eax
004014A4                rol     eax, 10h
004014A7                mov     ax, cx
004014AA                xor     ecx, ecx
004014AC
004014AC loc_4014AC:
004014AC                xor     dword ptr hash[ecx*4], eax
004014B3                inc     ecx
```

```
004014B4                cmp     ecx, 4
004014B7                jnz     short loc_4014AC
```

The following C-code shows how the hash is calculated for given two seeds:

```
#include <stdio.h>

inline int rand(int *seed) {
    *seed = *seed*0x343fd + 0x269EC3;
    return ((*seed) >> 0x10) & 0x7FFF;
}

long int main (long int argc, char *argv[]) {
    unsigned char hash[16];
    unsigned int i;
    unsigned int base = 0;
    unsigned int seed1 = 0x0110469A;
    unsigned int seed2 = 0x006C7972;
    unsigned int seed = seed1;
    unsigned int offset;
    unsigned int tmp;

    for(i = 0; i < 16; i++)
        hash[i] = 0;

    for(base = 0; base <= 2; base += 2) {
        for(i = 0; i < 4; i++)
        {
            do
            {
                offset = rand(&seed) % 4;
            } while ( hash[base + offset*4] != 0 || hash[base + offset*4 + 1] != 0 );
            tmp = rand(&seed);
            hash[base + offset*4 + 1] += (tmp >> 8);
            hash[base + offset*4] += (tmp & 0xFF);
        }

        tmp = rand(&seed);
        for(i = 0; i < 4; i++) {
            hash[i*4 + 1] ^= (tmp >> 8);
            hash[i*4] ^= tmp & 0xFF;
        }

        if(base == 0) {
            seed2 ^= tmp;
            seed = seed2;
        }
    }

    tmp = rand(&seed);
    tmp = (tmp<<16) + tmp;
    for(base = 0; base < 4; base++) {
        for(i = 0; i < 4; i++) {
            hash[base*4 + i] ^= (tmp & 0xFF);
            tmp = (tmp >> 8) | ((tmp & 0xFF) << 24);
```

```
        }
    }

    printf("hash is:\n");
    for(i = 0; i < 16; i++)
        printf("%x ", hash[i]);
    printf("\n");
}
```

The calculated hash is finally compared to a hardcoded value:

```
004014BB                 cmp     dword ptr hash, 3C0E7DEBh
004014C5                 jnz     short loc_4014EC
004014C7                 cmp     dword ptr hash+4, 3AD11611h
004014D1                 jnz     short loc_4014EC
004014D3                 cmp     dword ptr hash+8, 0B070195h
004014DD                 jnz     short loc_4014EC
004014DF                 cmp     dword ptr hash+0Ch, 36263E26h
004014E9                 jnz     short loc_4014EC
004014EB                 inc     ecx             ; set correct flag
```

Determining Valid Seeds

The first step to crack the algorithm is two find seeds that lead to the correct hash. I did this by first brute forcing the seeds for the second round, see program `brute_force2.c`. It should produce the following four seeds four the second round:

- 006f445a
- 406f445a
- 806f445a
- c06f445a

The starting seed of the second round is actually calulated like this:

```
seed2 ^= tmp;
seed = seed2;
```

where *tmp* is the last random number. All random numbers have at most 2 bytes. Also the *seed2* is at most 0xFFFFFF. Therefore, only the first of the four seeds can actually be produced by the code. The value of the last random number call is also output by the program, the number is 0x210e and we need to XOR the seed with this value to get the actual calculated value in *seed2*.

The seed for the first round *seed1* can be found similarly, I used the program *brute_force1.c*. I found four seeds:

- 01a01234
- 41a01234
- 81a01234
- c1a01234

The seed *81a01234* did not work with the crackme, I don't know why but all we need is one working seed anyways.

For the keygen we need a way to find lines that produce the desired hash. I used the following properties of the hashing to quickly find good lines:

1. The *seed1* is calculated with a hash routine, that uses bytes 16 to 19 of the second line in an XOR operation. We get the correct seed by simply adjusting these four bytes. Some of the adjustments will lead to strings that are not alpha numeric. I therefore simply loop over random strings until one corrects to an alphanumeric string.
2. The *seed2* is equivalent to the three bytes 9, 10, 11 of the second line. The *seed2* is known to be 0x6f445a XOR 0x210e = 6F6554, or “Teo” as a little endian ASCII string.

The keygen first generates a random string of 20 characters. It then sets bytes 9, 10, and 11 to “Tor”. After that, it patches the last four bytes to match the first seed. If the result is not alphanumeric, it simply tries again.

The Keygen

The following Python code expects a name as the first and only argument. The name must be alphanumeric (so no spaces allowed). It then calculates the second line and third line. The result is writen to *TheKey.k*, which you need to copy to the crackme directory. The Crackme should show a message box with *PERFECT* as the text.

```
import random
import sys


def keygen(name):

    def get_int_from_string(s):
        val = 0
        for x in s[::-1]:
            val <<= 8
            val += ord(x)
        return val


    def get_str_from_int(val):
        s = ""
        for i in range(4):
            s += chr(val & 0xFF)
            val >>= 8

        return s

    def rol(val, places):
        shift = places % 32;
        val = (val << shift)  + (val >> (32-shift))
        val &= 0xFFFFFFFF
        return val

    def ror(val, places):
        shift = places % 32;
        val = (val >> shift)  + (val << (32-shift))
        val &= 0xFFFFFFFF
        return val

    def make_alphanumeric(txt_codes):
        for i in range(len(txt_codes)):
            c = 37
            while not chr(txt_codes[i]).isalnum():
```

```
            txt_codes[i] += c
            c += 1
            txt_codes[i] &= 0xFF
            c &= 0xFF
    return txt_codes

def random_alphanumeric(l):
  return ''.join(random.sample(map(chr, range(48, 57) + range(65, 90) + range(97, 122)), l))

def hash1(line):
    eax = get_int_from_string(line[:4])
    eax += get_int_from_string(line[4:8])
    ecx = get_int_from_string(line[8:12])
    eax = rol(eax, ecx & 0xFF)
    ecx = get_int_from_string(line[12:16])
    eax = ror(eax, ecx & 0xFF)
    ecx = get_int_from_string(line[16:20])
    eax ^= ecx
    return eax

def find_line3_seed1(line1, line3):
    #valid_seed1 = [0x1a01234, 0x41a01234, 0x81a01234, 0xc1a01234]
    valid_seed1 = [0x1a01234, 0x41a01234, 0xc1a01234]
    eax = hash1(line3)
    edx = hash1(line1)
    eax ^= edx
    for s in valid_seed1:
        diff = eax ^ s
        ecx = get_int_from_string(line3[16:20])
        ecx ^= diff
        new_str = get_str_from_int(ecx)
        if new_str.isalnum():
            return line3[0:16] + new_str
    return None


def find_line3_seed2():
    line3 = random_alphanumeric(20)
    valid_seed2 = 0x6f445a ^ 0x210e
    s = get_str_from_int(valid_seed2)[:3]
    line3 = line3[0:9] + s + line3[12:]
    return line3


"""
    check if name is alphanumeric, crackme won't accept other names
"""
if not name.isalnum():
    print("The name must be alphanumeric")
    return

if not 3 <= len(name) <= 20:
    print("Name too long or too short")
    return
```

```
    """
        generate second line from name
    """
    line1 = name
    line1_len = len(line1)
    line1_codes = [ord(c) for c in line1]
    line1_codes[0] ^= line1_len ^ 0x5c

    i = 1
    j = line1_len-1
    while i < j:
        line1_codes[i] ^= line1_codes[j]
        i += 1
        j -= 1

    i = (line1_len-1)//2 + 1
    c = line1_codes[0]
    for i in range(i, line1_len):
        line1_codes[i] ^= c
        c += 1

    x = make_alphanumeric(line1_codes)
    line2 = ''.join([chr(xx) for xx in x])

    """
        semi brute force valid third line
    """
    while True:
        line3 = find_line3_seed2()
        line3 = find_line3_seed1(line1, line3)
        if line3:
            break

    with open("TheKey.k", "wb") as w:
        w.write("{}\r\n".format(line1))
        w.write("{}\r\n".format(line2))
        w.write("{}".format(line3))

keygen(sys.argv[1])
```

Example Keyfile

The following example for name “phildunphy” should give the *PERFECT* message:

```
phildunphy
K6LAUSIXAS
vS2LeI0ylTeocCnkbqDS
```