

```

1 // EncryptData_Initial.cpp
2 __asm {
3     // Copy the parameters before we set up our stack frame.
4     mov esi,data
5     mov edx,dataLength
6     dec edx
7     // Set up the main stack frame.
8     push ebp
9     mov ebp,esp
10    /*
11     * Put every variable we will need into the stack.
12     * Easy Reference Table:
13     *     -28     hop_count
14     *     -24     starting_index
15     *     -20     Key[]
16     *     -16     pass Hash
17     *     -12     gNumRounds
18     *     -8      data[]
19     *     -4      dataLength
20     *     ebp     old ebp
21     *
22     * Pushing 2 empty values at the end to reserve stack space
23     * for hop count and starting index
24     *
25     */
26    push edx                // dataLength
27    push esi                // data[]
28    mov edx,gNumRounds
29    dec edx
30    push edx                // gNumRounds
31    mov esi,gptrPasswordHash
32    push esi                // passwordHash[]
33    mov esi,gptrKey
34    push esi                // Key[]
35    xor eax,eax
36    push eax                // reserve space on stack for starting index
37    push eax                // reserve space on stack for hop count
38
39    /*
40     * Set up for the main looping structure
41     * Using eax to get results from function calls
42     * ebx will be used if needed to debug
43     * Using ecx as the counter variable
44     */
45    xor eax,eax             // result storer
46    xor ebx,ebx             // debug info (not used)
47    xor ecx,ecx             // round counter = 0
48
49    /*
50     * LOOP_ROUNDS
51     * 1. Calls function to calculate the starting index,
52     * then stores the result in the reserved stack space.
53     * 2. Calls function to calculate hop count,
54     * then stores the result in the reserved stack space.
55     * 3. Saves the counter variable.
56     * 4. Data Loop Structure
57     */
58    lbl_LOOP_ROUNDS:
59        call calculateStartingIndex // 1
60        mov [ebp-24],eax            // 1
61        call calculateHopCount     // 2
62        mov [ebp-28],eax            // 2
63
64        push ecx                    // 3
65        xor ecx,ecx

```

```

66  /*
67  * LOOP_DATA
68  * 1. Calls function to xor the data byte with key file byte
69  * 2. Calls function to increment the index by the hop count
70  * 3. Calls step C
71  * 4. Calls Step D
72  * 5. Calls Step E
73  * 6. Calls Step B
74  * 7. Calls Step A
75  */
76  lbl_LOOP_DATA:
77      // DATA ENCRYPTION STEPS
78      call xorByte           // 1
79      call incrementIndex   // 2
80      call stepC            // 3
81      call stepD            // 4
82      call stepE            // 5
83      call stepB            // 6
84      call stepA            // 7
85
86      // LOOP_DATA control Logic
87      cmp ecx,[ebp-4]        // check if reached end of data
88      je lbl_EXIT_LOOP_DATA // end; exit data loop
89      inc ecx
90      jmp lbl_LOOP_DATA      // not end; keep looping
91
92  lbl_EXIT_LOOP_DATA:
93      pop ecx
94      // END data Loop
95
96      // LOOP_ROUND control Logic
97      cmp ecx,[ebp-12]       // check if current round = num rounds
98      je lbl_EXIT_LOOP_ROUNDS // yes; exit round loop
99      inc ecx
100     jmp lbl_LOOP_ROUNDS    // no; keep looping
101
102  // FUNCTIONS
103  /*
104  * calculateStartingIndex
105  * 1. Copy the password hash stored in the stack to esi
106  * 2. Copy the [0+round*4]th byte of password hash to al
107  * 3. Arithmetic shift left eax 8 times; equivalent to multiplying by 256
108  * 4. Copy the [1+round*4]th byte of password hash to bl
109  * 5. Add eax and ebx
110  * 6. Return
111  *
112  * Note:
113  * The result is in eax so the caller can obtain the value.
114  */
115  calculateStartingIndex:
116      mov esi,[ebp-16]        // 1
117      xor eax,eax
118      mov al,byte ptr[esi+ecx*4] // 2
119      sal eax,8              // 3
120      inc esi
121      mov bl, byte ptr[esi + ecx * 4] // 4
122      add eax,ebx            // 5
123      ret
124      // END calculateStartingIndex
125
126
127
128
129
130

```

```

131  /*
132  * calculateHopCount
133  *     1. Copy the password hash from the stack to esi
134  *     2. Copy the [2+round*4]th byte of the hash to al
135  *     3. Arithmetic shift left eax 8 times; (eax * 256)
136  *     4. Copy the [3+round*4]th byte of the hash to bl
137  *     5. Sum eax and ebx
138  *     6. Check if eax is 0
139  *         6a. If eax is 0, set it to 65536 (0xFFFF)
140  *
141  *     Note:
142  *         The result is in eax so caller can get value
143  */
144  calculateHopCount:
145      mov esi,[ebp-16]          // 1
146      add esi,2
147      xor eax,eax
148      mov al,byte ptr[esi+ecx*4] // 2
149      sal eax,8                // 3
150      inc esi
151      mov bl,byte ptr[esi+ecx*4] // 4
152      add eax,ebx              // 5
153      cmp eax,0                // 6 (check)
154      je lbl_FIX_HOP
155      ret
156
157      lbl_FIX_HOP:              // 6a
158      mov eax,0xFFFF
159      ret
160      // END calculateHopCount
161
162  xorByte:
163      push ebx
164      xor ebx,ebx
165      mov ebx,[ebp-24]          // get index from stack frame
166      mov esi,[ebp-8]           // get *data from stack frame
167      mov edi,[ebp-20]          // get *gKey from stack frame
168      mov al,byte ptr[edi+ebx]   // al = gKey[index]
169      xor al,byte ptr[esi+ecx]   // al = al ^ data[x]
170      mov byte ptr[esi+ecx],al   // update data buffer byte
171      pop ebx
172      ret
173
174  incrementIndex:
175      mov eax,[ebp-24]          // eax = index
176      add eax,[ebp-28]          // index += hopcount
177      // fix index if necessary
178      cmp eax,65537
179      jae lbl_FIX_INDEX         // fix
180      mov [ebp-24],eax          // set new index
181      ret                       // dont fix
182
183      lbl_FIX_INDEX:
184      sub eax,65537
185      mov [ebp-24],eax          // set new index
186      ret
187
188
189
190
191
192
193
194
195

```

```

196 stepA:
197     push ebx
198     xor ebx,ebx
199     mov esi,[ebp-8]          // get *data
200     mov al,byte ptr[esi+ecx] // get data[x]
201     xor ebx,ebx
202     mov bh,0xAA             // 0xAA has all even bits 1 and odd 0,
203                             // bitwise and with this value will get all even bits
204     and bh,al               // bh is now all the even bits of our byte
205     mov bl,0x55             // 0x55 has all even bits 0 and odd 1,
206                             // bitwise and with this value will get all odd bits
207     and bl,al               // bl is not all the odd bits of our byte
208     shr bh,1                // shift all even bits right 1 time
209     shl bl,1                // shift all odd bits left 1 time
210     or bl,bh                // combine them back together
211     mov al,bl               // al is now out byte with even and odd bits swapped
212     mov byte ptr[esi+ecx],al // update data buffer byte
213     pop ebx
214     ret
215
216 stepB:
217     push ebx
218     xor ebx,ebx
219     mov esi,[ebp-8]          // get *data
220     mov al, byte ptr[esi+ecx] // get data[x]
221     xor al,00111100b         // this inverts the middle 4 bits
222     mov byte ptr[esi+ecx],al // update data buffer byte
223     pop ebx
224     ret
225
226 stepC:
227     mov esi,[ebp-8]          // get *data
228     mov al,byte ptr[esi+ecx] // get data[x]
229     ror al,4                 // rotate 4 to right
230     mov byte ptr[esi+ecx],al // update data buffer byte
231     ret
232
233 stepD:
234     mov esi,[ebp-8]          // get *data
235     xor eax,eax
236     push ebx
237     xor ebx,ebx
238     mov al,byte ptr[esi+ecx] // get data[x]
239     lea edi, gEncodeTable    // put the address of the first byte of gEncodeTable into esi
240     mov bl, byte ptr[edi+eax] // copy the value at the index al from gEncodeTable
241     mov byte ptr[esi + ecx],bl // update data buffer byte
242     pop ebx
243     ret
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260

```

```

261 stepE:
262     push ebx                // save old ebx value
263     push edx                // save old ecx value
264     xor eax,eax
265     mov esi,[ebp-8]         // get *data
266     mov al,byte ptr[esi+ecx] // get data[x]
267     xor ebx,ebx            // set ebx to 0, will be our counter
268     xor edx,edx            // set ecx to 0, will be the new reversed value
269     jmp lbl_ELOOP          // start looping
270
271     lbl_ELOOP:
272         shl al,1            // shift the right most bit into the carry
273         rcr dl,1            // rotate the carry into dl
274         cmp ebx,7           // compare counter to 7
275         je lbl_EEND         // if counter is 7 end the loop
276         inc ebx             // else increment count
277         jmp lbl_ELOOP       // and keep looping
278
279     lbl_EEND:
280         mov byte ptr[esi + ecx],dl // update data buffer byte
281         pop edx                // restore ecx
282         pop ebx                // restore ebx
283         ret                   // return
284
285 //                          END FUNCTIONS
286
287     lbl_EXIT_LOOP_ROUNDS:
288         // Restore the previous stack
289         add esp,28
290         pop ebp
291     }

```