2、Protecting TOP SEH

MicroSoft has gravely adjusted the code of SetUnhandledExceptionFilter function. SetUnhandledExceptionFilter is an exported function in kernel32.dll. it sets a filter of the exception handling callback function. The callback function didn't replace the exception handling program of default system, only disposed something in its advance. Finally, the result was sent into the exception handling program of default system. The course is as equal as the exception has filtratid once.

The mode of calling SetUnhandledExceptionFilter function:

LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter);

The sole parameter of the function is an address of the callback function which need to set. The return value is the address of last setting. The function is not hung before the original callback function again, and that using the new callback function replace the original callback function. If the address parameter is specified NULL, then the system throw off the filter, and the exception is sent to the exception handling program of default system. Winxp sp2 has gravely changed the function. Before replaced the original callback function, she encrypt the new address of the callback function first. After then, she replace the callback function. She decrypt the address before the address of the original callback function is returned. The function is relatively simple.

```
.text:7C810386 SetUnhandledExceptionFilter proc near
.text:7C810386 lpTopLevelExceptionFilter = dword ptr  8
.text:7C810386
.text:7C810386                 mov     edi, edi
.text:7C810388                 push     ebp
.text:7C810389                 mov     ebp, esp
     ; here is encrypting the address (lpTopLevelExceptionFilter)
.text:7C81038B                 push     [ebp+ lpTopLevelExceptionFilter]
.text:7C81038E                 call      RtlEncodePointer
     ; and then, exchanged among the encrypted address and the address of the
     ; original callback function, i.e. the encrypted address is written into
     ; a Global variable. Simultaneous, she returned the address of the original
     ; callback function in the Global variable.
.text:7C810393                 push     eax                 ; Value
.text:7C810394                 push     offset Target       ; Target
.text:7C810399                 call     InterlockedExchange
     ; decrypt before the address of the original callback function is returned
     ; since it is encrypted.
.text:7C81039E                 push     eax
.text:7C81039F                 call     RtlDecodePointer
.text:7C8103A4                 pop      ebp
.text:7C8103A5                 retn     4
.text:7C8103A5 SetUnhandledExceptionFilter endp ; sp = -8
.text:7C8103A5
     ; the address of callback function was directly written onto a Global
     ; point ago, didn't handle anything. it is obvious that we can't use again
     ; the Stack Overflow with rewriting the function point ago. Discovered by
     ; the analysis, Winxp sp2 encrypted similarly all of Global point with
     ; the method。The follow is how encrypt the address.

     ; Both of RtlEncodePointer and RtlDecodePointer is the function of ntdll.dll
     ; exported. A point is encrypted by RtlEncodePointer. A point is decrypted
     ; by RtlDecodePointer. Actually, the whole course of encrypting and
     ; decrypting is simple. The encrypting executed XOR between the point and
     ; a random value. The decrypting executed XOR between the point and the
     ; random value.

     ; encrypting:    point = point ^ rand
     ; decrypting:    point = point ^ rand

     ; rand is a random value who is relating to the process. It is gotten by
     ; calling the ZwQueryInformationProcess function. The random value of every
     ; process is not same.

     ; Here is the code of this two functions.

     ; The code of RtlEncodePointer function is as following:
.text:7C933917 RtlEncodePointer proc near
.text:7C933917 var_4           = dword ptr -4
.text:7C933917 arg_4           = dword ptr  8
```

```
.text:7C933917
.text:7C933917                 mov     edi, edi
.text:7C933919                 push    ebp
.text:7C93391A                 mov     ebp, esp
      ; A random value who is relating to the process is gotten by calling the
      ; ZwQueryInformationProcess function.
.text:7C93391C                 push    ecx
.text:7C93391D                 push    0
.text:7C93391F                 push    4
      ; An address of temp variable in stack is gotten here. The random value
      ; is gotten finally will store in the temp variable.
.text:7C933921                 lea     eax, [ebp+var_4]
.text:7C933924                 push    eax
.text:7C933925                 push    24h  ; the number of children function is 0x24.
.text:7C933927                 push    0FFFFFFFFh
.text:7C933929                 call    ZwQueryInformationProcess
      ; The random value is done XOR with the point for encrypting. The course
      ; of decrypting is same as the course of encrypting.
.text:7C93392E                 mov     eax, [ebp+var_4]
.text:7C933931                 xor     eax, [ebp+arg_4]
.text:7C933934                 leave
.text:7C933935                 retn    4
.text:7C933935 RtlEncodePointer endp ; sp =  4

      ; The function of RtlDecodePointer is more simple, only jump to
      ; RtlEncodePointer since the course of decrypting is entirely same as the
      ; course of encrypting.

.text:7C93393D RtlDecodePointer proc near
      ; The four following lines is not any effect.
.text:7C93393D                 mov     edi, edi
.text:7C93393F                 push    ebp
.text:7C933940                 mov     ebp, esp
.text:7C933942                 pop     ebp
      ; The following line jump to RtlEncodePointer, is equal to call the
      ; function directly.
      ; RtlEncodePointer
.text:7C933943                 jmp     short RtlEncodePointer
.text:7C933943 RtlDecodePointer endp

      ; ZwQueryInformationProcess call finally a system calling, jump to ring0,
      ; The last calling function in ring0 is NtQueryInformationProcess. The
      ; childen number of calling the function finally is 0x24. The children
      ; number is fetched directly and store into a random value of the process,
      ; and copy it to a temp variable in user stack. If the random value is 0,
      ; then will creat the random value again according to the system time.
      ; Generally, The random value is 0 while the process is just created at
      ; beginning. So it must be made again. Because the random value is
      ; relating to the time which the process created, therefore the random
      ; value can't be guessed. The function is exported in the ntoskrnl.exe.
      ; The code which related to the function is as follow:

PAGE:004970CC loc_4970CC:
      ; The following code got a sole random value of relating to process.
      ; The children function number is 0x24.
PAGE:004970CC                  cmp     edi, edx        ; case 0x24
PAGE:004970CE                  jnz     loc_497349
PAGE:004970D4                  cmp     dword ptr [ebp+8], 0FFFFFFFFh
PAGE:004970D8                  jnz     loc_4977B8
      ; The following code got the address which will store the random value.
PAGE:004970DE                  mov     eax, large fs:124h
PAGE:004970E4                  mov     eax, [eax+44h]
PAGE:004970E7                  mov     [ebp-34h], eax
PAGE:004970EA
PAGE:004970EA loc_4970EA:
PAGE:004970EA                  mov     edi, [ebp-34h]
PAGE:004970ED                  add     edi, 258h
      ; Stored in edi address is a random value which is relating to the process.
      ; Here, the random value is gotten.
PAGE:004970F3                  mov     eax, [edi]
PAGE:004970F5                  test    eax, eax
PAGE:004970F7                  jz      loc_4B2379
      ; If the Random value is 0, then the Random value will be gotten again.
      ; The follow is the course of getting the random value:

      ; 1、The system time is gotten first.
```

```
153         ; 2、The system time is been continuously done XOR with a value in the
154         ;    system kernel for creating the random value.
155
156   PAGE:004B2379
157   PAGE:004B2379 loc_4B2379:
158         ; getting the system time
159   PAGE:004B2379                 lea     eax, [ebp-3Ch]
160   PAGE:004B237C                 push    eax
161   PAGE:004B237D                 call    KeQuerySystemTime
162   PAGE:004B2382                 db      3Eh
163         ; getting a Global variable in the system kernel, The Global variable is
164         ; also a random value.
165   PAGE:004B2382                 mov     eax, ds:0FFDFF020h
166   PAGE:004B2388                 mov     ecx, [eax+518h]
167   PAGE:004B238E                 xor     ecx, [eax+4B8h]
168         ; The random value is done XOR with the system time.
169   PAGE:004B2394                 xor     ecx, [ebp-38h]
170   PAGE:004B2397                 xor     ecx, [ebp-3Ch]
171         ; The result will be stored in the Global variable which is relating to
172         ; the process. The address is stored in edi.
173   PAGE:004B239A                 mov     [ebp-0CCh], ecx
174   PAGE:004B23A0                 mov     [ebp-0D4h], edi
175   PAGE:004B23A6                 mov     eax, 0
176   PAGE:004B23AB                 mov     ecx, [ebp-0D4h]
177   PAGE:004B23B1                 mov     edx, [ebp-0CCh]
178   PAGE:004B23B7                 cmpxchg [ecx], edx
179   PAGE:004B23BA                 push    4
180   PAGE:004B23BC                 pop     edx
181         ; Jump to loc_4970EA, create the random value. If the random value is 0,
182         ; then it will be created again.
183   PAGE:004B23BD                 jmp     loc_4970EA
184         ; After the random value is created, it is copied to a temp variable in
185         ; user stack. The address of the temp variable is stored in esi.
186
187         ; By this time, The random value which is relating to the time of creating
188         ; process has been gotten.
189   PAGE:004970FD                 mov     dword ptr [ebp-4], 15h
190   PAGE:00497104                 mov     [esi], eax
191   PAGE:00497106                 test    ebx, ebx
192   PAGE:00497108                 jnz     loc_497AA5
193   PAGE:0049710E                 jmp     loc_4955F5
194
195         ; Here, we have already understood the whole course how get all of the
196         ; random value. The random value is relating to the time of process created.
197         ; So, we can't guess the random value. But, the random value is born when
198         ; the process is created. until the process is die, the random value is
199         ; not changed. Hence, if we can get the random value, we can use it before
200         ; the process is die. For example, we do XOR between the random value and
201         ; the address which we will jump to. We write it into the address of
202         ; Top Overflow Handle Exception, then use it as same as used ago.
203
204         ; The method can't be used about the remote overflow. But if we can rewrite
205         ; the Import table of the program or the code segment, it is a best status.
206         ; The Import table of the system DLL can't be modified. But the Import table
207         ; of a program can be modified. So it is possible to use it. If the point
208         ; of some functions is existed in the code segment, then we can rewrite it
209         ; for using. If the case is existed, it is possible to create a general usage.
210
```