A Simple Analysis Of CTF Challenge From A CTF Beginner

Author:ITh4cker

0x0 Analysis

First,Let’s run the crack.exe in CMD.exe with the test SN “12345”:

```
C:\Users\[redacted]\Desktop\CTF3r\第一周11月19-24\Re\题目1\CrackMe>CrackMe.exe  12345
registration failed
```

It shows the common string of registration failer,then I try to search the string in IDA,but no result,so it should be encrypted by author,let’s extract all strings with string.exe,and I found some suspicious strings:

```
umen
!fac
{{T5|Y
ucce
lly
y
rastontiai fqed
```

```
0000000140038050  93 90 8D 96 86 03 BF F7  4A 14 7B 7B 54 35 7C 59  舜·崞·..亏·J.{{T5|Y
0000000140038060  9B 9B DF 8C 75 63 63 65  8E 8C 99 8A 6C 6C 79 00  涼·邔·ucce盼·榎·lly·
0000000140038070  8E 9A 98 96 72 61 73 74  6F 6E 74 69 61 69 20 66  帑·檟·rastontiai·f
0000000140038080  71 65 64 00 00 00 00 00  34 94 03 40 01 00 00 00  qed.....4..@....
0000000140038090  FF FF FF FF 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000001400380A0  0A 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

successfully

it seems like the encrypted registration successfully string and registration failed string,look them in IDA pseudo-code:

```
  if ( v118 >= 0 )
  {
LABEL_118:
    v321 = -dword_140038070;
    v322 = (dword_140038074 >> 16) + (dword_140038074 << 16);
    v323 = (dword_140038078 >> 16) + (dword_140038078 << 16);
    v324 = (dword_14003807C >> 16) + (dword_14003807C << 16);
    v325 = dword_140038080 - 5;
    sub_140001A40(&v506, &v321, 19i64);
    v180 = &v506;
    if ( v508 >= 0x10 )
      v180 = v506;
    v181 = sub_140001340(&qword_140039930, v180, v507);// registration failed
    sub_1400018B0(v181);
    sub_140001FD0(&v506);
    v226 = -293063854;
    v178 = -1773405844;
    v179 = -827325493;
  }
  else
  {
    v326 = 5 * dword_140038058;
    v327 = 7 * dword_14003805C - 217;
    v328 = -dword_140038060;
    v224 = dword_140038064;
    v329 = dword_140038064 + (dword_140038064 & 0xFFFF0000);
    v330 = 1 - dword_140038068;
    v331 = dword_14003806C;
    sub_140001A40(&v503, &v326, 23i64);
    v176 = &v503;
    if ( v505 >= 0x10 )
      v176 = v503;
    v177 = sub_140001340(&qword_140039930, v176, v504);// registration successfully
    sub_1400018B0(v177);
    sub_140001FD0(&v503);
    v225 = 0xEE883352;
    v178 = -858993452;
    v179 = -613566752;
  }
```

We can see they are in the if-else structure statement,if the value of v118 >=0,it shows us the failer string,otherwise the successful string(so the function sub_1400061A0 is the SN_Check function) where does the v118 come from? look forward for the previous reference, we found lots of Junk Code(judge them by the the var's reference):

```
  v124 = -(0xCCCCCCCD
         * (((0x33333333 * v122 + 7) >> 16)
          + ((0x33333333 * v122 + 7 + (((0x33333333 * v122 + 7) >> 16) * ((0x33333333 * v122 + 7) >> 16))) << 16)) >> 16)
       + ((0xCCCD * ((0x33333333 * v122 + 7) >> 16)) << 16)
       + 7;
  v125 = -(v122
         + ((7 - v123) << 32)
         + 1
         - (((v124
            + ((v124 >> 8) + 7 + (v124 << 8))
            - (7
             - (0xCCCCCCCD
              * (((0x33333333 * v122 + 7) >> 16)
               + ((0x33333333 * v122 + 7 + (((0x33333333 * v122 + 7) >> 16) * ((0x33333333 * v122 + 7) >> 16))) << 16)) >> 16))) >> 16)
          + ((5
            - (v124
             + (v124 >> 8)
             + (v124 << 8)
             + (0xCCCCCCCD
              * (((0x33333333 * v122 + 7) >> 16)
               + ((0x33333333 * v122 + 7 + (((0x33333333 * v122 + 7) >> 16) * ((0x33333333 * v122 + 7) >> 16))) << 16)) >> 16))) << 16))
         - v122);
  v126 = v125
       + 1
       - (((v124
          + ((v124 >> 8) + 7 + (v124 << 8))
          - (7
           - (0xCCCCCCCD
            * (((0x33333333 * v122 + 7) >> 16)
             + ((0x33333333 * v122 + 7 + (((0x33333333 * v122 + 7) >> 16) * ((0x33333333 * v122 + 7) >> 16))) << 16)) >> 16))) >> 16)
        + ((5
```

Junk Code

Finally we came to the key point:

```
  v107 = ((v94 + *(&v94[*(v94 + 2 * v98 + v95[9])] + v95[7])))(&v701);
  qword_140039408 = v107;
  for ( j = *(*(*(__readgsqword(0x30u) + 96) + 24i64) + 16i64); ; j = *j )
  {
    v109 = j[6];
    v110 = (v109 + *(v109 + v109[15] + 136));
    if ( v110 != v109 )
    {
      v111 = 0;
      v112 = v110[6];
      if ( v112 )
        break;
    }
LABEL_76:
    ;
  }
  while ( 1 )
  {
    v113 = v109 + *(&v109[v111] + v110[8]);
    v114 = 0x811C9DC5;
    v115 = *v113;
    if ( *v113 )
    {
      do
      {
        v114 = 16777619 * (v114 ^ v115);
        v115 = *++v113;
      }
      while ( *v113 );
      if ( v114 == 0xF8F45725 )
        break;
    }
    if ( ++v111 >= v112 )
      goto LABEL_76;
  }
  v116 = ((v109 + *(&v109[*(v109 + 2 * v111 + v110[9])] + v110[7])))(v107, v706);
  if ( v116 )
  {
    v117 = -1i64;
    do
      ++v117;
    while ( byte_140039300[v117] );
    v118 = v116(0i64, v117);
  }
```

We can see that v118 equals to the return value of v116() function,

```
loc_14000552B:                          ; CODE XREF: sub_140002E00+271C↑j
                mov     ecx, [rdi+1Ch]
                add     rcx, r9
                mov     eax, [rdi+24h]
                add     rax, r9
                movzx   eax, word ptr [rax+r10*2]
                mov     r8d, [rcx+rax*4]
                add     r8, r9
                lea     rdx, [rbp+0D40h+var_140]
                mov     rcx, r13
                call    r8
                mov     r8, rax
                test    rax, rax
                mov     r14d, 5
                jz      short loc_140005589
                mov     rax, 0FFFFFFFFFFFFFFFFh
                lea     rcx, byte_140039300
                nop     dword ptr [rax+00h]

loc_140005570:                          ; CODE XREF: sub_140002E00+2777↓j
                inc     rax
                cmp     byte ptr [rcx+rax], 0
                jnz     short loc_140005570
                mov     rdx, rax
                xor     ecx, ecx
                call    r8              ; v116
                movsxd  rdx, eax
```

From the disassemble code above,we can see that it’s the function call from export table,so we decided to debug it for clear(we set an bp at statement call r8),but we found the control flow can’t come here,it exited instead,then we found the following condition must be satisfied for runing to our bp by cross-reference analysis:

```
if ( sub_140002AC0(byte_140039300) == 0x4F8075587499C0FFi64 )
{
  LODWORD(v46) = 31;
  LODWORD(v47) = 0;
}
```

```
signed __int64 __fastcall sub_140002AC0(char *a1)
{
  char v1; // dl
  signed __int64 result; // rax
  signed __int64 v3; // rax

  v1 = *a1;
  for ( result = 0xCBF29CE484222325i64; *a1; result = 0x100000001B3i64 * v3 )
  {
    ++a1;
    v3 = result ^ v1;
    v1 = *a1;
  }
  return result;
}
```

By Google the “magic number” 0xCBF29CE484222325 in function sub_140002AC0(),It’s the hash calculation algorithm FNV-1(64bit):

0xCBF29CE484222325

All Maps Videos Images Shopping More Settings Tools

About 501 results (0.33 seconds)

Fowler–Noll–Vo hash function - Wikipedia
https://en.wikipedia.org/wiki/Fowler–Noll–Vo_hash_function
Fowler–Noll–Vo is a non-cryptographic hash function created by Glenn Fowler, Landon Curt ... The FNV_offset_basis is the 64-bit FNV offset basis value: 14695981039346656037 (in hex, **0xcbf29ce484222325**). The FNV_prime is the 64-bit ...
The hash · FNV-1 hash · FNV prime · FNV hash parameters
You've visited this page 3 times. Last visit: 11/18/18

FNV1a c++11 constexpr compile time hash functions, 32 and 64 bit ...
https://gist.github.com/underscorediscovery/81308642d0325fd386237cfa3b44785c
uint64_t hash = **0xcbf29ce484222325**;. uint64_t prime = 0x100000001b3;. for(int i = 0; i < len; ++i) {. uint8_t value = data[i];. hash = hash ^ value;. hash *= prime;. }.

picoquic/fnv1a.h at master · private-octopus/picoquic · GitHub
https://github.com/private-octopus/picoquic/blob/master/picoquic/fnv1a.h
14695981039346656037 (in hex, **0xcbf29ce484222325**). The prime is. 1099511628211 (in hex, 0x100000001b3; or as an expression 2^40 + 2^8 + 0xb3).

FNV-1 hash [edit]

The FNV-1 hash algorithm is as follows:[8][9]

```
hash = FNV_offset_basis
for each byte_of_data to be hashed
    hash = hash × FNV_prime
    hash = hash XOR byte_of_data
return hash
```

In the above pseudocode, all variables are unsigned integers. All variables, except for *byte_of_data*, have the same number of bits as the FNV hash. The variable, *byte_of_data*, is an 8 bit unsigned integer.

As an example, consider the 64-bit FNV-1 hash:

- All variables, except for *byte_of_data*, are 64-bit unsigned integers.
- The variable, *byte_of_data*, is an 8-bit unsigned integer.
- The ***FNV_offset_basis*** is the 64-bit **FNV offset basis** value: 14695981039346656037 (in hex 0xcbf29ce484222325).
- The ***FNV_prime*** is the 64-bit **FNV prime** value: 1099511628211 (in hex 0x100000001b3)
- The multiply returns the lower 64-bits of the product.
- The XOR is an 8-bit operation that modifies only the lower 8-bits of the hash value.
- The *hash* value returned is a 64-bit unsigned integer.

And the string var byte_140039300 is confirmed to be the inputed SN string by debugging:

```
.text:00007FF6CA4A69EE                                  ; ---------------------------------------------------
.text:00007FF6CA4A69EE
.text:00007FF6CA4A69EE                                  loc_7FF6CA4A69EE:                       ; CODE XREF: sub_7
.text:00007FF6CA4A69EE 41 B9 04 01 00 00                mov     r9d, 104h                       ; MaxCount
.text:00007FF6CA4A69F4 4C 8B 42 08                      mov     r8, [rdx+8]                     ; Src
.text:00007FF6CA4A69F8 41 8B D1                         mov     edx, r9d                        ; SizeInBytes
.text:00007FF6CA4A69FB 48 8D 0D FE 28 03 00             lea     rcx, byte_7FF6CA4D9300          ; Dst
.text:00007FF6CA4A6A02 E8 55 7B 00 00                   call    strncpy_s
RIP .text:00007FF6CA4A6A07 48 BA 00 00 00 00 06 00 00+mov     rdx, 600000000h                 ; SizeInBytes
.text:00007FF6CA4A6A11 48 8D 4D 77                      lea     rcx, [rbp+77h]                  ; Dst
.text:00007FF6CA4A6A15 E8 E6 C3 FF FF                   call    CheckSN
.text:00007FF6CA4A6A1A 33 C0                            xor     eax, eax
.text:00007FF6CA4A6A1C
.text:00007FF6CA4A6A1C                                  loc_7FF6CA4A6A1C:                       ; CODE XREF: sub_7
.text:00007FF6CA4A6A1C 4C 8D 9C 24 B0 00 00 00          lea     r11, [rsp+0C8h+var_18]
.text:00007FF6CA4A6A24 49 8B 5B 28                      mov     rbx, [r11+28h]
.text:00007FF6CA4A6A28 49 8B 73 38                      mov     rsi, [r11+38h]
.text:00007FF6CA4A6A2C 49 8B E3                         mov     rsp, r11

UNKNOWN 00007FF6CA4A69FB: sub_7FF6CA4A61A0+85B (Synchronized with RIP)

Hex View-1
00007FF6CA4D9300  69 74 68 34 63 6B 65 72  00 00 00 00 00 00 00 00  ith4cker.......
00007FF6CA4D9310  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF6CA4D9320  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF6CA4D9330  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF6CA4D9340  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF6CA4D9350  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Then the code execution flow goes into sub_140002E00(),it is confirmed to be the SN check function,all the check logics are in it,as I said before,there are also lots of junkcode in it for disturbing out analysis,it doesn't matter,just follow the inputed SN string and related vars,you will see the real codes,there are serveal check logics in the function,the first is the length of SN string must equal to 30 and there are at least 3 '9' in the SN string:

```
do
  ++v6;
while ( SN_Input[v6] );
LODWORD(v307) = v6;
HIDWORD(v307) = 5 - HIDWORD(v6);
v529 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v530 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v531 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v532 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v304 = v6 & 0xFFF;
v305 = v6 & 0xFFF;
v234 = v6 & 0xFFF;
v235 = v6 & 0xFFF;
v533 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v534 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v535 = (v6 + (HIDWORD(v6) << 32)) >> 12;
v236 = v6 & 0xFFF;
v237 = v6 & 0xFFF;
v238 = v6 & 0xFFF;
v7 = 4i64;
if ( (v6 & 0xFFF) + (v529 << 12) == 30 )      // length(SN) == 30
{
```

```
if ( (v6 & 0xFFF) + (v529 << 12) == 30 )      // length(SN) == 30
{
  v8 = 0;
  v9 = 0;
  if ( v6 )
  {
    v10 = SN_Input;
    do
    {
      v184 = 0;
      v11 = *v10;
      LOBYTE(v184) = v11;
      v12 = &v184;
      for ( i = 0xCBF29CE484222325i64; *v12; v11 = *v12 )
      {
        i = 0x100000001B3i64 * (i ^ v11);
        v12 = (v12 + 1);
      }
      count = v8 + 1;
      if ( i != 0xAF63B44C8601A894i64 )       // FNV-1(9) = 0xAF63B44C8601A894
        count = v8;
      v8 = count;
      ++v9;
      ++v10;
    }
    while ( v9 < v6 );
    v1 = 32i64;
    if ( count >= 3 )
    {
```

The second check is that the first 9 chars are “KXCTF2018”:

```
LOBYTE(v183) = SN_Input[0];
if ( FNV_1_hash(&v183) == 0xAF64064C860233EAi64 )// 'K'
{
  LOBYTE(v183) = SN_Input_1_;
  if ( FNV_1_hash(&v183) == 0xAF64154C86024D67i64 )// 'X'
  {
    LOBYTE(v183) = SN_Input_2_;
    if ( FNV_1_hash(&v183) == 0xAF63FE4C86022652i64 )// 'C'
    {
      LOBYTE(v183) = SN_Input_3_;
      if ( FNV_1_hash(&v183) == 0xAF64094C86023903i64 )// 'T'
      {
        LOBYTE(v183) = SN_Input_4_;
        if ( FNV_1_hash(&v183) == -5808522788293435079i64 )// 'F'
        {
          LOBYTE(v183) = SN_Input_5_;
          if ( FNV_1_hash(&v183) == -5808606351177179115i64 )// '2'
          {
            LOBYTE(v183) = SN_Input_6_;
            if ( FNV_1_hash(&v183) == -5808608550200435537i64 )// '0'
            {
              LOBYTE(v183) = SN_Input_7_;
              if ( FNV_1_hash(&v183) == -5808609649712063748i64 )// '1'
              {
                LOBYTE(v183) = SN_Input_8_;
                if ( FNV_1_hash(&v183) == -5808599754107409849i64 )// '8'
                {
                  v42 = 0i64;
                  v37 = 0;
                  v43 = 0x50700i64;
```

The third is the FNV-1 hash of the whole SN string is **0x4F8075587499C0FF**(As we discussed earlier):

```
if ( FNV_1_hash(SN_Input) == 0x4F8075587499C0FFi64 )
{
  LODWORD(v46) = 31;
  LODWORD(v47) = 0;
}
else
{
```

After going through these checks,it came to the following string handling code block:

```
memset(&v701, 0, 260ui64);
v80 = strstr(SN_Input, "9");                    // locate the first '9'
v81 = v80;
v701 = v80[1];
v702 = v80[2];
v703 = v80[3];
v704 = v80[4];
v705 = v80[5];
strncat_s(&v701, 260ui64, ".DLL", 260ui64);    // XXXXX.DLL
v82 = strstr(v81 + 1, "9");                     // locate the second '9'
v83 = v82;
*strstr(v82 + 1, "9") = 0;                      // locate the third '9'
```

We can know that the layout of the SN string is like following:

KXCTF20189XXXXX9YYYYYYYYYYYY9

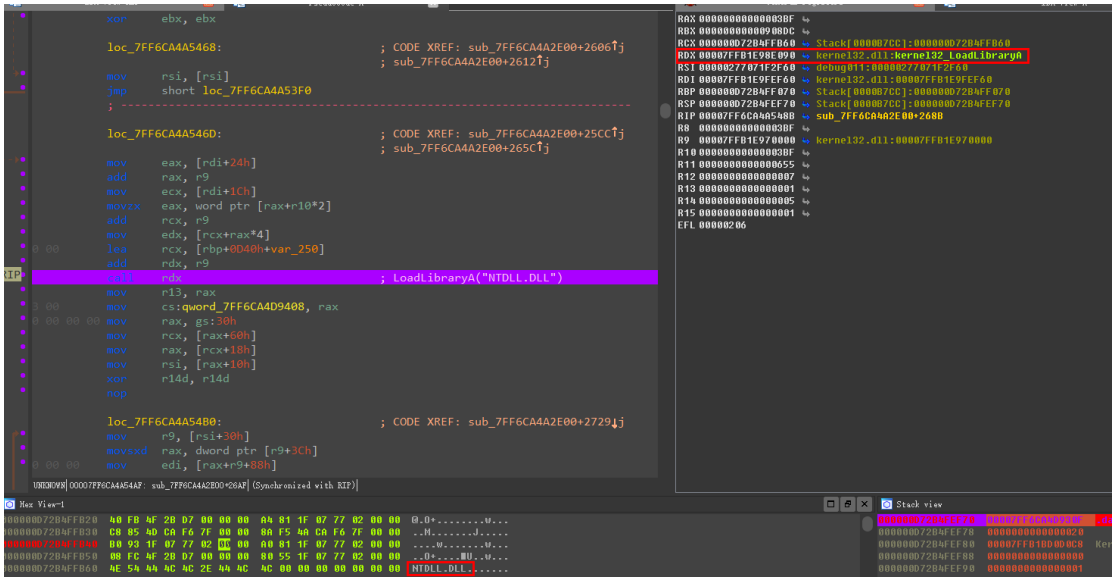
XXXXX represent the name of some a dll system directory,totally 5 bytes.

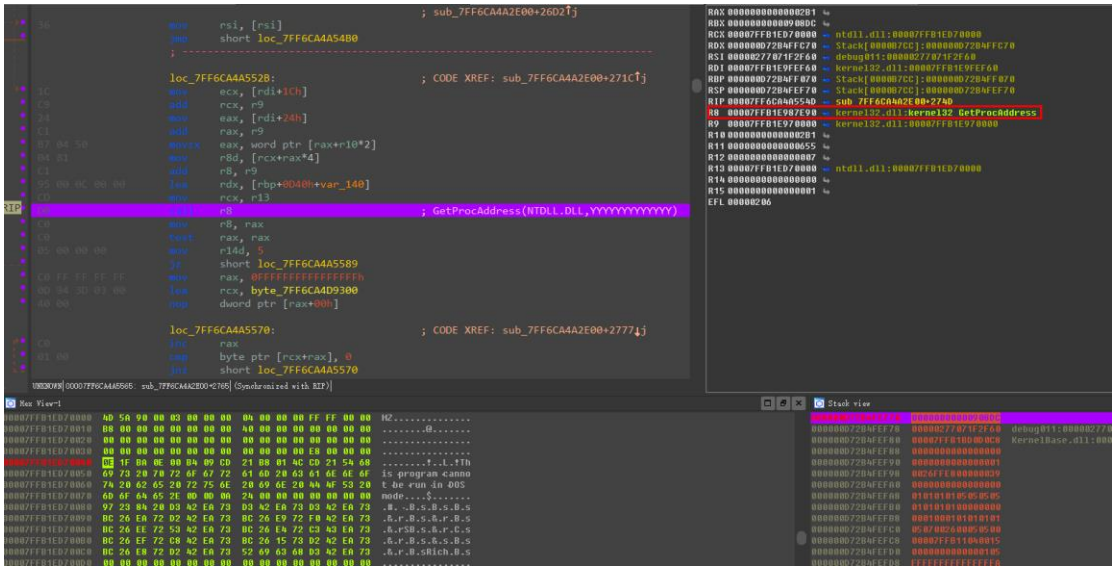
YYYYYYYYYYYY represent the name of some a api in XXXXX.dll,totally 13 bytes.

After knowing the layout of the SN string,now you can debug it in IDA for a further analysis,you will find that it will retrive the address of API LoadLibrary and GetProcAddress by PEB,and load the XXXXX.dll using LoadLibrary,after which call API YYYYYYYYYYYY,it using 32 bit FNV_hash to retrive the info it wanted.

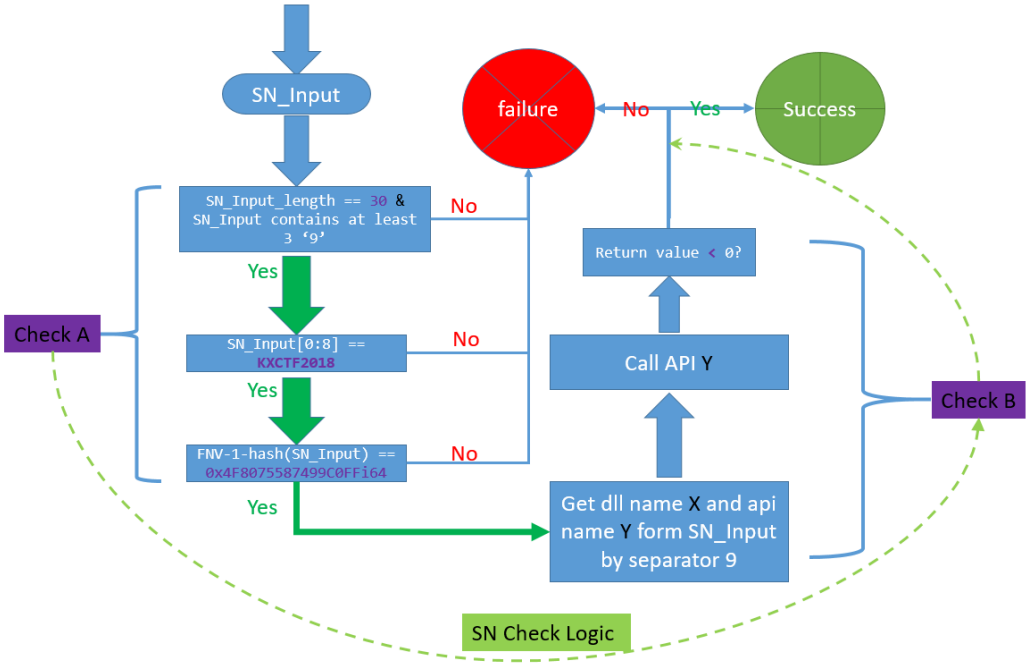
```
  v93 = *(*(*(__readgsqword(0x30u) + 96) + 24i64) + 16i64);// get the module address by PEB
  if ( (HIDWORD(v92) - abs(v92)) + (v92 << 32) )
  {
    while ( 1 )
    {
      v94 = v93[6];
      v95 = (v94 + *(v94 + v94[15] + 136));
      if ( v95 != v94 )
      {
        v96 = 0;
        v97 = v95[6];
        if ( v97 )
          break;
      }
LABEL_61:
      v93 = *v93;
    }
    while ( 1 )
    {
      v98 = v96;
      v99 = v94 + *(&v94[v96] + v95[8]);
      v100 = 0x811C9DC5;
      v101 = *v99;
      if ( *v99 )
      {
        do
        {
          v100 = 0x1000193 * (v100 ^ v101);
          v101 = *++v99;
        }
        while ( *v99 );
        if ( v100 == 0x53B2070F )                // LoadLibraryA
          break;
      }
      if ( ++v96 >= v97 )
        goto LABEL_61;
    }
```

```
  hModule_XXXXX = ((v94 + *(&v94[*(v94 + 2 * v98 + v95[9])] + v95[7])))(&XXXXX);// LoadLibraryA("XXXXX.dll")
  qword_140039408 = hModule_XXXXX;
  for ( j = *(*(*(__readgsqword(0x30u) + 96) + 24i64) + 16i64); ; j = *j )
  {
    v109 = j[6];
    v110 = (v109 + *(v109 + v109[15] + 136));
    if ( v110 != v109 )
    {
      v111 = 0;
      v112 = v110[6];
      if ( v112 )
        break;
    }
LABEL_76:
    ;
  }
  while ( 1 )
  {
    v113 = v109 + *(&v109[v111] + v110[8]);
    v114 = 0x811C9DC5;
    v115 = *v113;
    if ( *v113 )
    {
      do
      {
        v114 = 16777619 * (v114 ^ v115);
        v115 = *++v113;
      }
      while ( *v113 );
      if ( v114 == 0xF8F45725 )                  // GetProcAddress
        break;
    }
    if ( ++v111 >= v112 )
      goto LABEL_76;
  }
  v116 = ((v109 + *(&v109[*(v109 + 2 * v111 + v110[9])] + v110[7])))(hModule_XXXXX, YYYYYYYYYYYY);// GetProcAddress(hModule_XXXXX,YYYYYYYYYYYY)
  if ( v116 )
  {
    v117 = -1i64;
    do
      ++v117;
    while ( SN_Input[v117] );
    v118 = v116(0i64, v117);
```

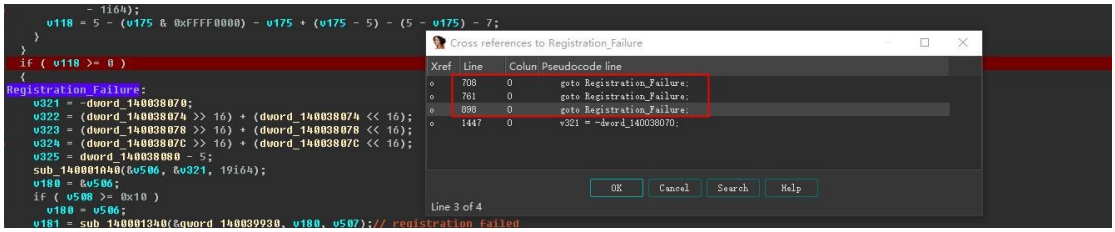


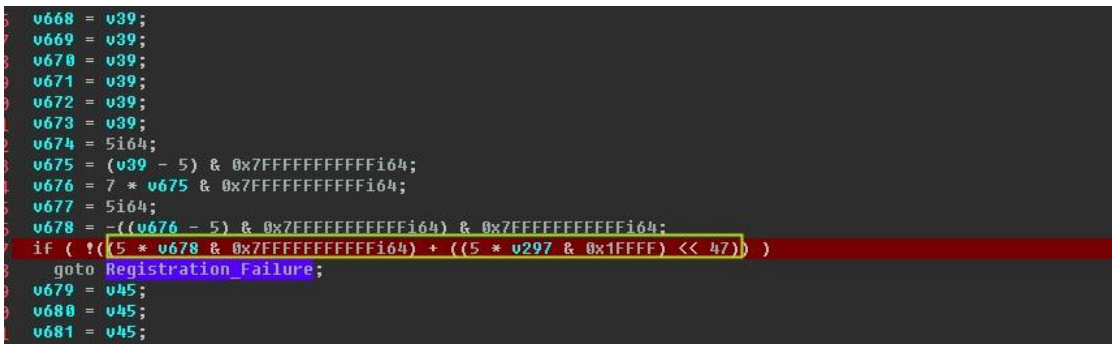


So the whole check logic seems clear now as following flow chart shows:

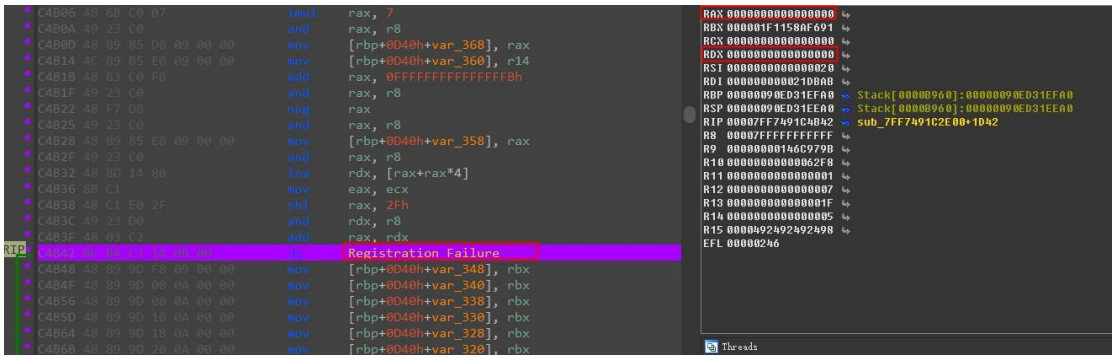


The above flow chart is the key check logic,I divide the check logics into 2 groups Check A and B ,why? as there are 3 additional if-else judgment statements pointing to the registration failure code block:





In dynamic debugging,the rip can run here,but I don't know what condition it need?when run here,the RAX and RDX wil be zero:



Only when we input the right flag,the RAX and RDX will not be zero,and the jump will be false,so it seems that all check condition excep the last condition(`return value < 0`) will jump here when the condition isn't satisfied,but I don't have a clear and accurate description for there are too many garbage instructions...

then we need to brute force the dll name(5 bytes)and api name(13 bytes) for the final flag using python script,the final crack output will look like as following:

```
Found SN_Input[0]:K
Found SN_Input[1]:X
Found SN_Input[2]:C
Found SN_Input[3]:T
Found SN_Input[4]:F
Found SN_Input[5]:2
Found SN_Input[6]:0
Found SN_Input[7]:1
Found SN_Input[8]:8
input[0:8]: KXCTF2018
Found SN_Input[9]:9
Found SN_Input[29]:9
Found API(e463da3c): GetModuleHandleA
Found API(f8f45725): GetProcAddress
Found API(53b2070f): LoadLibraryA
Congratulations to ITh4cker,Found flag: KXCTF20189NTDLL9DbgUiContinue9
```

0x1 Reference

0.https://ctf.pediy.com/game-fight-66.htm

1.https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function#FNV-1_hash

2.http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param