

看雪.京东 2018CTF 第十二题 破解之道

原创

太...白 于 2021-02-15 10:10:11 发布 148 收藏 1

分类专栏: [# C/C++逆向 \(windows\)](#) 文章标签: [算法](#) [c语言](#) [字符串](#) [数据结构](#) [逆向工程](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: https://blog.csdn.net/Onlyone_1314/article/details/113813918

版权



[C/C++逆向 \(windows\)](#) 专栏收录该内容

18 篇文章 0 订阅

订阅专栏

目录

[CTF \(看雪.京东 2018CTF 第十二题 破解之道\)](#)

第一步:

第二步:

第三步:

第四步:

第五步:

第六步:

CTF (看雪.京东 2018CTF 第十二题 破解之道)

x6412.exe

先打开程序:

```
C:\WINDOWS\system32\cmd.exe
C:\Users\Legends\Desktop>x6412 123465
registration failed
C:\Users\Legends\Desktop>_
```

是一个输入字符串再判断对错的程序。

打开程序进行动态调试,一步一步走。首先会遇到一个对输入数据的长度进行判断的地方如下图所示:

第一步:

```
00007FF74D153453  BB 04000000  mov  edx,4
00007FF74D153458  49:83F8 1E    cmp  r8,1E
00007FF74D15345C  74 3D       je   x6412.7FF74D15349B  长度应该为0x1E
00007FF74D15345E  4C:89A5 70050000  mov  qword ptr ss:[rbp+570],r12
00007FF74D153465  4C:89A5 78050000  mov  qword ptr ss:[rbp+578],r12
```

这里会对输入的数据的长度与0x1E进行比较,如果不相等的话就会跳到错误的地方。

所以得到第一个条件就是输入的长度必须是30;

第二步:

然后继续向下执行,找到再次出现我们输入的字符串的地方发现如下汇编语言:

```
00007FF74D1534A4  0F84 1C050000  je   x6412.7FF74D1539C6
00007FF74D1534AA  4C:8D35 4F5E0300  lea  r14,qword ptr ds:[7FF74D189300]  r14:"KXCTF20181234597819
00007FF74D153481  48:BE 25232284E49CF2CB  mov  rsi,CBF29CE484222325
00007FF74D15348B  49:BC B301000000010000  mov  r12,100000001B3
00007FF74D1534C5  49:BD 94A801864CB463AF  mov  r13,AF63B44C8601A894
00007FF74D1534CF  90       nop
```

```
00007FF74D1534C5  49:BD 94A801864CB463AF  mov  r13,AF63B44C8601A894
00007FF74D1534CF  90       nop
00007FF74D1534D0  33C0     xor  eax,eax
00007FF74D1534D2  66:894424 28  mov  word ptr ss:[rsp+28],ax
00007FF74D1534D7  41:0FB606  movzx eax,byte ptr ds:[r14]  r14:"KXCTF2018123459781934
00007FF74D1534DB  884424 28  mov  byte ptr ss:[rsp+28],al
00007FF74D1534DF  48:8D5424 28  lea  rdx,qword ptr ss:[rsp+28]
00007FF74D1534E4  48:8BCE    mov  rcx,rsi
00007FF74D1534E7  84C0     test  al,al
00007FF74D1534E9  74 1E    je   x6412.7FF74D153509
00007FF74D1534EB  0F1F4400 00  nop  dword ptr ds:[rax+rax],eax
00007FF74D1534F0  48:0FBEC0  movsx rax,al
00007FF74D1534F4  48:33C1    xor  rax,rcx
00007FF74D1534F7  48:8BC8    mov  rcx,rax
00007FF74D1534FA  49:0FAFCC  imul rcx,r12
00007FF74D1534FE  48:8D52 01  lea  rdx,qword ptr ds:[rdx+1]
00007FF74D153502  0FB602    movzx eax,byte ptr ds:[rdx]
00007FF74D153505  84C0     test  al,al
00007FF74D153507  75 E7    jne  x6412.7FF74D1534F0
00007FF74D153509  41:8D40 01  lea  eax,qword ptr ds:[r8+1]  r8+1给eax
00007FF74D15350D  49:3BCD    cmp  rcx,r13  比较异或后的数据
00007FF74D153510  41:0F45C0  cmovne eax,r8d  如果不相等r8不变,相等则r8+1
00007FF74D153514  44:8BC0    mov  r8d,eax
00007FF74D153517  41:FFC7    inc  r15d
00007FF74D15351A  49:FFC6    inc  r14  r14:"KXCTF2018123459781934
00007FF74D15351D  49:63C7    movsxd rax,r15d
00007FF74D153520  49:3BC1    cmp  rax,r9
00007FF74D153523  72 AB    jb  x6412.7FF74D1534D0
00007FF74D153525  41:83F8 03  cmp  r8d,3  r8和3作比较
00007FF74D153529  BE 20000000  mov  esi,20  20:
```

这段代码的意思是从我们输入的所有数据中每一个符号都要进行运算。

运算过程是先与0xCBF29CE484222325进行异或,然后再乘以0x100000001B3。

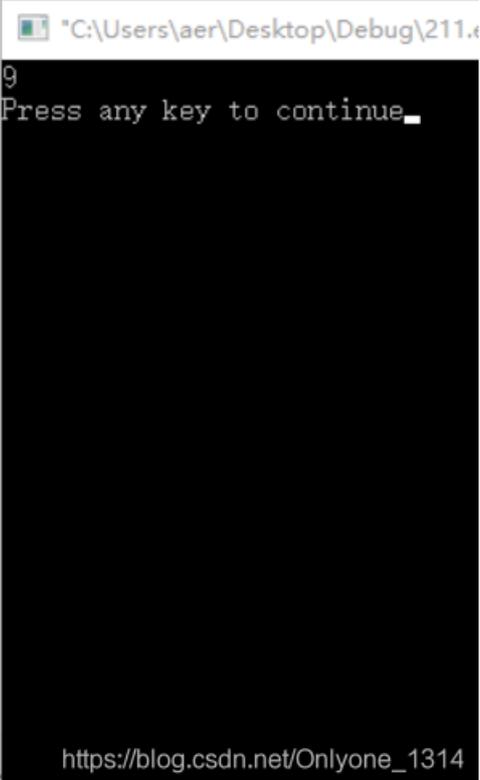
然后运算的结果会和0xAF63B44C8601A894进行比较,如果相等的话就会给r8+1,

最终r8的数据应该要大于等于3。

那么我们写一个程序来看一下哪个字符的运算后会和给的结果相同,最终运算结果是'9',

所以我们输入的字符串中应该至少有3个'9';

```
        printf("%c\n",d);
        break;
    }
}
while(eax!=final)
{
    /*
    _int64 b=0x100000001B3;
    _int64 d=0xAF63B44C8601A894;
    _int64 c=0xCBF29CE484222325;
    _int64 i=0;
    for(i=0;i<255;i++)
    {
        _int64 temp=c^i;
        temp=temp*b;
        if(temp==d)
        {
            printf("%c\n",i);
        }
    }
    return 0;
}
```



https://blog.csdn.net/Onlyone_1314

第三步:

运行到如下图所示的地方,又出现了我们输入的字符串,所以这里应该也是一处关键性的代码,通过分析我们发现这里取了输入的前9个字符进行和上图一样的运算,而且都对结果进行了检验,由于每一个字符的选择是从0-255,所以这里选择爆破的方法获得前九位字符.

地址	指令	注释
3F25	66:894424 20	mov word ptr ss:[rsp+20],ax
3F2A	0FB605 CF530300	movzx eax,byte ptr ds:[7FF692959300]
3F31	884424 20	mov byte ptr ss:[rsp+20],al
3F35	48:8D4C24 20	lea rcx,qword ptr ss:[rsp+20]
3F39	E8 81EBFFFF	call x6412.7FF692922AC0
3F3F	48:B9 EA3302864C0664AF	mov rcx,AF64064C860233EA
3F49	48:38C1	cmp rax,rcx
3F4C	74 15	je x6412.7FF692923F63
3F4E	48:B8 0000000001000000	mov rax,1000000000
3F58	41:B9 00383900	mov r9d,393800
3F5E	E9 C8010000	jmp x6412.7FF69292412E
3F63	0FB605 97530300	movzx eax,byte ptr ds:[7FF692959301]
3F6A	884424 20	mov byte ptr ss:[rsp+20],al
3F6E	48:8D4C24 20	lea rcx,qword ptr ss:[rsp+20]
3F73	E8 48EBFFFF	call x6412.7FF692922AC0
3F78	48:B9 674D02864C1564AF	mov rcx,AF64154C86024D67
3F82	48:38C1	cmp rax,rcx
3F85	74 15	je x6412.7FF692923F9C
3F87	48:B8 0000000001000000	mov rax,1000000000
3F91	41:B9 00383900	mov r9d,393800
3F97	E9 92010000	jmp x6412.7FF69292412E
3F9C	0FB605 5F530300	movzx eax,byte ptr ds:[7FF692959302]
3FA3	884424 20	mov byte ptr ss:[rsp+20],al
3FA7	48:8D4C24 20	lea rcx,qword ptr ss:[rsp+20]
3FAC	E8 0FEBFFFF	call x6412.7FF692922AC0
3FB1	48:B9 522602864CFE63AF	mov rcx,AF63FE4C86022652
3FBB	48:38C1	cmp rax,rcx
3FBE	74 15	je x6412.7FF692923FD5
3FC0	48:B8 0000000001000000	mov rax,1000000000
3FCA	41:B9 00383900	mov r9d,393800
3FDD	E9 59010000	jmp x6412.7FF69292412E

00007FF692959300: "KXCTF2018"

00007FF692959301: "XCTF20181"

00007FF692959302: "CTF201812"

这里相等

https://blog.csdn.net/Onlyone_1314

爆破代码及运行结果如下图所示:

```

#include<stdio.h>
int main()
{
    __int64 a[9]={0xAF64064C860233EA,
    0xAF64154C86024D67,
    0xAF63FE4C86022652,
    0xAF64094C86023903,
    0xAF63FB4C86022139,
    0xAF63AF4C8601A015,
    0xAF63AD4C86019CAF,
    0xAF63AC4C86019AFC,
    0xAF63B54C8601AA47};
    __int64 b=0x100000001B3;
    __int64 d=0;
    int flag=0;
    int i=0;
    __int64 c=0xCBF29CE484222325;
    __int64 final=0x4F8075587499C0FF;
    __int64 rax=0xA189FA2D2B8F61B6;
    for(i=0;i<9;i++)
    {
        for(d=0;d<255;d++)
        {
            __int64 temp=d^c;
            temp=temp*b;
            if(temp==a[i])
            {
                printf("%c",d);
                break;
            }
        }
    }
    return 0;
}

```

运行结果:



所以我们可以确定输入的前九个字符就是KXCTF2018

到目前为止我们可以确定输入的字符串的格式大概是KXCTF2018999XXXXXXXXXXXXXXXXXXXX

第四步:

又找到如图的关键代码处,此处的功能是将所有输入的字符进行计算,然后得出的结果必须和0x4F8075587499C0FF相同,由于这里共有30个字符串我们仅仅已知12个,还有18个字符未知,爆破是不可取的,所以这里应该可以留下来做一个检验的功能,这里先修改跳转,不进行分析:

```

00007FF6929244AD 89BD E8020000 mov dword ptr ss:[rbp+2E8],edi
00007FF6929244B3 48:B8 A694524A29A59452 mov rax,5294A5294A5294A6
00007FF6929244B8 48:8985 08090000 mov qword ptr ss:[rbp+908],rax
00007FF6929244C4 48:8D0D 354E0300 lea rcx,qword ptr ds:[7FF692959300]
00007FF6929244CB E8 F0E5FFFF call x6412.7FF692922AC0
00007FF6929244D0 48:B9 FFC099745875804F mov rcx,4F8075587499C0FF

```

第五步:

```

0F85 7C100000 jne x6412.7FF69292600E
33D2 xor edx,edx
44:8D47 04 lea r8d,qword ptr ds:[rdi+4]
48:8D8D F00A0000 lea rcx,qword ptr ss:[rbp+AF0]
E8 2C680000 call x6412.7FF6929287D0
48:8D15 8DF80100 lea rdx,qword ptr ds:[7FF692944838]
48:8D0D 4E430300 lea rcx,qword ptr ds:[7FF692959300]
E8 216E0000 call x6412.7FF692928DD8
48:8BD8 mov rbx,rax
0FB640 01 movzx eax,byte ptr ds:[rax+1]
8885 F00A0000 mov byte ptr ss:[rbp+AF0],al
0FB643 02 movzx eax,byte ptr ds:[rbx+2]
8885 F10A0000 mov byte ptr ss:[rbp+AF1],al
0FB643 03 movzx eax,byte ptr ds:[rbx+3]
8885 F20A0000 mov byte ptr ss:[rbp+AF2],al
0FB643 04 movzx eax,byte ptr ds:[rbx+4]
8885 F30A0000 mov byte ptr ss:[rbp+AF3],al
0FB643 05 movzx eax,byte ptr ds:[rbx+5]
8885 F40A0000 mov byte ptr ss:[rbp+AF4],al
41:B9 04010000 mov r9d,104
4C:8D05 43F80100 lea r8,qword ptr ds:[7FF69294483C]
41:8BD1 mov edx,r9d
48:8D8D F00A0000 lea rcx,qword ptr ss:[rbp+AF0]
E8 64940000 call x6412.7FF69292E46C
48:8D15 29F80100 lea rdx,qword ptr ds:[7FF692944838]
48:8D48 01 lea rcx,qword ptr ds:[rbx+1]

```

如图所示,这个地方应该是关键代码处,由于上面的跳转会跳过这一部分,所以我们修改跳转之后让程序进入这一部分运行。

```

00007FF692924FB7 E8 216E0000 call x6412.7FF692928DD8
00007FF692924FB8 48:8BD8 mov rdx,rax
00007FF692924FB9 0FB640 01 movzx eax,byte ptr ds:[rax+1]
00007FF692924FBA 8885 F00A0000 mov byte ptr ss:[rbp+AF0],al
00007FF692924FBC 0FB643 02 movzx eax,byte ptr ds:[rbx+2]
00007FF692924FCD 8885 F10A0000 mov byte ptr ss:[rbp+AF1],al
00007FF692924FCE 0FB643 03 movzx eax,byte ptr ds:[rbx+3]
00007FF692924FD2 8885 F20A0000 mov byte ptr ss:[rbp+AF2],al
00007FF692924FD8 0FB643 04 movzx eax,byte ptr ds:[rbx+4]
00007FF692924FDC 8885 F30A0000 mov byte ptr ss:[rbp+AF3],al
00007FF692924FE2 0FB643 05 movzx eax,byte ptr ds:[rbx+5]
00007FF692924FE6 8885 F40A0000 mov byte ptr ss:[rbp+AF4],al
00007FF692924FEC 41:B9 04010000 mov r9d,104
00007FF692924FF2 4C:8D05 43F80100 lea r8,qword ptr ds:[7FF69294483C]
00007FF692924FF9 41:8BD1 mov edx,r9d
00007FF692925003 48:8D8D F00A0000 lea rcx,qword ptr ss:[rbp+AF0]
00007FF692925008 E8 64940000 call x6412.7FF69292E46C
00007FF69292500F 48:8D15 29F80100 lea rdx,qword ptr ds:[7FF692944838]
00007FF692925015 48:8D48 01 lea rcx,qword ptr ds:[rbx+1]

```

当运行过BDD8这个函数之后发现返回的字符串是在第一个'9'之后的字符串,之后又取了截取后的字符串的前五位作为参数,同时又传进了".DLL"作为参数,执行了E46C函数,我们继续运行发现执行了E46C函数之后返回值里有7819c.DLL这个就是我们截取的5个字符和.DLL进行拼接后的结果

```

R10 0000000000000000
R11 0000008D40BCF690 "78193.DLL"
R12 0000000000000007
R13 0000000000000EC51
R14 0000000000000005

```

所以这个E46C的作用应该是拼接。

```

00007FF69292501B 48: 8D15 16F80100 lea rdx,qword ptr ds:[7FF692944838] rdx:"90123"
00007FF692925022 48: 8D48 01 lea rcx,qword ptr ds:[rax+1] rax+1:"0123"
00007FF692925026 E8 AD6D0000 call x6412.7FF692928D08
00007FF69292502B C600 00 mov byte ptr ds:[rax],0 rax:"90123"
00007FF69292502E 33D2 xor edx,edx
00007FF692925030 44: 8D47 04 lea r8d,qword ptr ds:[rdi+4]
00007FF692925034 48: 8D8D 00C00000 lea rcx,qword ptr ss:[rbp+C00]
00007FF692925038 E8 90670000 call x6412.7FF6929287D0
00007FF692925040 mov r9d,104
00007FF692925046 4C: 8D43 01 lea r8,qword ptr ds:[rbx+1] r8:"90123", rbx+1:
00007FF69292504A 41: 88D1 mov edx,r9d
00007FF69292504D 48: 8D8D 00C00000 lea rcx,qword ptr ss:[rbp+C00]
00007FF692925054 E8 03950000 call x6412.7FF69292E55C
00007FF692925059 45: 88C7 mov r8d,r15d

```

这里将第三个'9'赋值为0,然后取出了第二个'9'到没赋值前第三个'9'之间的字符串。

所以这一段的功能应该是将第一个'9'之后的五位取出来与.DLL进行拼接,生成一个动态链接库的名字,然后将第二个'9'和第三个'9'之间的字符取出来,暂时不清楚有什么用,但是大概分析出格式为:

KXCTF20189XXXXX9xxxxxxxxxxxx9

其中的XXXXX应该是一个DLL的名字。

猜测可能是NTDLL

通过网上查找发现这里的计算是一种hash算法。

第六步:

```

007FF692925384 8B5F 20 mov ebx,dword ptr ds:[rdi+20]
007FF692925387 66:0F1F8400 00000000 nop word ptr ds:[rax+rax],ax
007FF692925390 45: 8BD0 mov r10d,r8d
007FF692925393 4A: 8D0493 lea rax,qword ptr ds:[rbx+r10*4]
007FF692925397 42: 8B0C08 mov ecx,dword ptr ds:[rax+r9]
007FF692925398 49:03C9 add rcx,r9
007FF69292539E BA C59D1C81 mov edx,811C9DC5
007FF6929253A3 0FB601 movzx eax,byte ptr ds:[rcx]
007FF6929253A6 84C0 test al,al
007FF6929253A8 74 28 je x6412.7FF6929253D2
007FF6929253AA 66:0F1F4400 00 nop word ptr ds:[rax+rax],ax
007FF6929253B0 0FBEC0 movsx eax,al
007FF6929253B3 33C2 xor eax,edx
007FF6929253B5 690D 93010001 imul edx,eax,1000193
007FF6929253B8 48: 8D49 01 lea rcx,qword ptr ds:[rcx+1]
007FF6929253BF 0FB601 movzx eax,byte ptr ds:[rcx]
007FF6929253C2 84C0 test al,al
007FF6929253C4 75 EA jmp x6412.7FF6929253B0
007FF6929253C6 81FA 0F07B253 cmp edx,53B2070f
007FF6929253C8 0F84 98000000 je x6412.7FF69292546D
007FF6929253D2 41:FFC9 jnc r8d
007FF6929253D5 45:3BC2 cmp r8d,r11d
007FF6929253D8 72 86 jb x6412.7FF692925390

```

函数名: rax+r9*1:"program cannot be run in rcx:"SHAInit"

检测函数名是否为空

进行hash值的计算

判断函数名的hash值是否等于0x53b2070f

分析这段代码大概功能是从当前加载的DLL库中进行查找所有函数,然后进行HASH值的计算,找到HASH值等于0x53b2070f的函数名,我们直接执行到他找到之后的代码,在寄存器里会发现这段代码找到了LoadLibraryEXA函数。

继续向下执行

```

007FF692925470 49:03C1 add rax,r9
007FF692925473 8B4F 1C mov ecx,dword ptr ds:[rdi+1C]
007FF692925476 42:0FB70450 movzx eax,word ptr ds:[rax+r10*2]
007FF692925478 49:03C9 add rcx,r9
007FF69292547E 8B1481 mov edx,dword ptr ds:[rcx+rax*4]
007FF692925481 48: 8D8D F0A00000 lea rcx,qword ptr ss:[rbp+AF0]
007FF692925488 49:03D1 add rdx,r9
007FF69292548B FFD2 call rdx
007FF69292548D 4C: 8BE8 mov r13,rcx
007FF692925490 48: 8905 713F0300 mov qword ptr ds:[7FF692959408],rax
007FF692925497 65:48: 8B0425 30000000 mov rax,qword ptr ds:[30]
007FF6929254A0 48: 8B48 60 mov rcx,qword ptr ds:[rax+60]
007FF6929254A4 48: 8B41 18 mov rax,qword ptr ds:[rcx+18]
007FF6929254A8 48: 8B70 10 mov rsi,qword ptr ds:[rax+10]
007FF6929254AC 45: 33F6 xor r14d,r14d
007FF6929254AF 90 nop
007FF6929254B0 4C: 8B4E 30 mov r9,qword ptr ds:[rsi+30]
007FF6929254B4 49: 6341 3C movsxd rax,dword ptr ds:[r9+3C]
007FF6929254B8 42: 8B8C08 88000000 mov edi,dword ptr ds:[rax+r9+88]
007FF6929254C0 49:03F9 add rd1,r9
007FF6929254C3 49: 3BF9 cmp rd1,r9

```

我们输入的DLL名称: rcx:"78193.DLL"

调用LoadLibraryA函数: rdx=<kernel32.LoadLibraryA> (00007FFA3254E4A0)

这段的作用就是加载我们输入的DLL。

然后返回DLL地址00007FFA349C0000;

然后继续向下执行发现了和上面查找函数一样的代码,只不过是查找的hash值不同而已,采取同样的方法,直接看他找到了什么函数

007FF6929254F6	84C0	test al,al	
007FF6929254F8	74 24	jne X6412.7FF69292551E	
007FF6929254FA	66:0F1F4400 00	nop word ptr ds:[rax+rax],ax	
007FF692925500	0FBEC0	movsx eax,al	
007FF692925503	33C2	xor eax,edx	
007FF692925505	6900 93010001	imul edx,eax,1000193	
007FF692925508	48:8049 01	lea rcx,qword ptr ds:[rcx+1]	rcx+1: "GetProcessAffinityMask"
007FF69292550F	0FB601	movzx eax,byte ptr ds:[rcx]	
007FF692925512	84C0	test al,al	
007FF692925514	75 EA	jne X6412.7FF692925500	
007FF692925516	81FA 2557F4F8	cmp edx,F8F45725	
007FF69292551C	74 0D	jne X6412.7FF69292552B	
007FF69292551E	41:FFC0	inc r8d	
007FF692925521	45:3BC3	cmp r8d,r11d	
007FF692925524	72 BA	jb X6412.7FF6929254E0	
007FF692925526	48:8B36	mov rsi,qword ptr ds:[rsi]	
007FF692925529	EB 85	jmp X6412.7FF6929254B0	
007FF69292552B	884F 1C	mov ecx,dword ptr ds:[rdi+1C]	
007FF69292552E	49:03C9	add rcx,r9	
007FF692925531	8847 24	mov eax,dword ptr ds:[rdi+24]	rdi+24: "@\t"
007FF692925534	49:03C1	add rax,r9	
007FF692925537	42:0F870450	movzx eax,word ptr ds:[rax+r10*2]	
007FF69292553C	44:8B0481	mov r8d,dword ptr ds:[rcx+rax*4]	
007FF692925540	4D:03C1	add r8,r9	
007FF692925543	48:8D95 000C0000	lea rdx,qword ptr ss:[rbp+C00]	传入了我们输入的第二个'9' 到第三个'9'之间的数据
007FF69292554A	49:8BCD	mov rcx,r13	
007FF69292554D	41:FFD0	call r8	调用该函数
007FF692925550	4C:8BC0	mov r8,rax	

发现他找到了GetProcessAffinitMask函数,但是这个函数好像没什么用。

007FF692925526	48:8B36	mov rsi,qword ptr ds:[rsi]	
007FF692925529	EB 85	jmp X6412.7FF6929254B0	
007FF69292552B	884F 1C	mov ecx,dword ptr ds:[rdi+1C]	
007FF69292552E	49:03C9	add rcx,r9	
007FF692925531	8847 24	mov eax,dword ptr ds:[rdi+24]	rdi+24: "@\t"
007FF692925534	49:03C1	add rax,r9	
007FF692925537	42:0F870450	movzx eax,word ptr ds:[rax+r10*2]	
007FF69292553C	44:8B0481	mov r8d,dword ptr ds:[rcx+rax*4]	
007FF692925540	4D:03C1	add r8,r9	
007FF692925543	48:8D95 000C0000	lea rdx,qword ptr ss:[rbp+C00]	传入了我们输入的第二个'9' 到第三个'9'之间的数据
007FF69292554A	49:8BCD	mov rcx,r13	
007FF69292554D	41:FFD0	call r8	调用该函数

我们发现在用CALL R8的时候其实就是在调用GetProcAddress函数,并且使用了我们输入的第二个'9'到第三个'9'之间的数据作为一个函数名。

并且在调用该函数的时候使用的第一个参数是我们输入的DLL的地址。

RAX	00000000000002B1	L' A'
RBX	000000000000908EC	
RCX	00007FFA349C0000	ntdll.00007FFA349C0000
RDX	0000007BB999F5D0	"345678"
RBP	0000007BB999E9D0	"3333"
RSP	0000007BB999E8D0	
RSI	00000271900A37B0	
RDI	00007FFA325BEF70	kernel32.00007FFA325BEF70

所以这一段我们分析出来他应该是从我们输入的DLL中去找到我们输入的函数的地址,如果找到的话会返回函数地址,如果没有找到就会返回NULL

00007FF692925540	4D:03C1	add r8,r9	
00007FF692925543	48:8D95 000C0000	lea rdx,qword ptr ss:[rbp+C00]	传入了我们输入的第二个'9' 到第三个'9'之间的数据
00007FF69292554A	49:8BCD	mov rcx,r13	
00007FF69292554D	41:FFD0	call r8	调用该函数
00007FF692925550	4C:8BC0	mov r8,rax	
00007FF692925553	48:85C0	test rax,rax	
00007FF692925556	41:BE 05000000	mov r14,c	
00007FF69292555C	74 28	jne X6412.7FF692925589	如果函数调用成功则不跳转
00007FF69292555E	48:C7C0 FFFFFFFF	mov rcx,FFFFFFFF	
00007FF692925565	48:800D 943D0300	lea rcx,qword ptr ds:[7FF692959300]	00007FF692959300: "KXCTF20189NTDL"
00007FF69292556C	0F1F40 00	nop dword ptr ds:[rax],eax	
00007FF692925570	48:FFC0	inc rax	
00007FF692925573	80C01 00	cmp byte ptr ds:[rcx+rax],0	
00007FF692925577	75 F7	jne X6412.7FF692925570	
00007FF692925579	48:8BD0	mov rdx,rax	
00007FF69292557C	33C9	xor ecx,ecx	
00007FF69292557E	41:FFD0	call r8	
00007FF692925581	48:63D0	movsxd rdx,eax	
00007FF692925584	E9 71090000	jmp X6412.7FF692925EFA	
00007FF692925589	41:8BCF	mov ecx,r15d	
00007FF69292558C	894D B8	mov dword ptr ss:[rbp-48],ecx	
00007FF69292558F	44:8965 BC	mov dword ptr ss:[rbp-44],r12d	

这里必须不跳转,所以GetProcAddress要调用成功.然后这里就会返回一个负数在接下来的一个判断中起作用,然后跳转到成功的地方

断点未设置 2925EF7	49:03D1	add rdx,r9	r9: "3333"
00007FF692925EFA	85D2	test edx,edx	
00007FF692925EFC	0F89 0C010000	jns x6412.7FF69292600E	这里不能跳转
00007FF692925F02	8B05 50210300	mov eax,dword ptr ds:[7FF692958058]	
00007FF692925F08	8D0480	lea eax,qword ptr ds:[rax+rax*4]	
00007FF692925F0B	8985 30010000	mov dword ptr ss:[rbp+130],eax	

所以综上所述我们输入的flag应该是

KXCTF20189[XXX]9{XXXX}9

条件1: 长度为30位

条件2: 必须有至少3个'9'

条件3: 前九位为 KXCTF2018

条件4: 整个字符串的hash值应该是0x4F8075587499C0FF

条件5: []之间的数据应该是一个长度为5的DLL名称

条件6: {}之间应该是一个长度为13的函数名称

还有可能有4个'9'或者5个'9'等等但是这样的话就输入了很多无用的数据,所以这里应该是只有3个'9'在字符串中的。

所以综上所述可以写出代码来爆破得出最终的flag;

```
def Hash(string_xx):
    temp = 0xcbf29ce484222325
    a = 0x100000001b3
    for i in string_xx:
        temp = temp ^ ord(i)
        temp = (temp * a) & 0xffffffffffff
    return temp

def Find():
    file = open("C:\Users\ aer\Desktop\06x64\api.txt")
    while 1:
        str = file.readlines(100000)
        if not str:
            break
        for line in str:
            Function = line.strip()
            str_in = "KXCTF20189NTDLL9" + Function + "9"
            Code = Hash(str_in)
            if Code == 0x4f8075587499c0ff:
                print(str_in)
                break
        file.close()
    Find()
```

运行结果:

```
C:\Users\ aer\AppData\Local\Programs\Python
KXCTF20189NTDLL9DbgUiContinue9
```

得到正确的flag: KXCTF20189NTDLL9DbgUiContinue9

将flag输入exe: x6412 KXCTF20189NTDLL9DbgUiContinue9

```
命令提示符
Microsoft Windows [版本 10.0.17134.648]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\aer>cd Desktop
C:\Users\aer\Desktop>cd 06x64
C:\Users\aer\Desktop\06x64>cd .
C:\Users\aer\Desktop>x6412 KXCTF20189NTDLL9DbgUiContinue9
registered successfully
```

正确