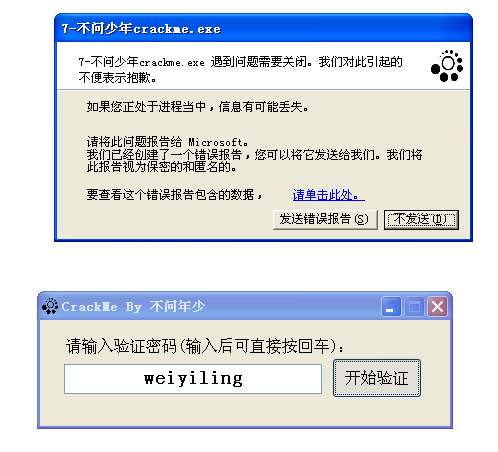
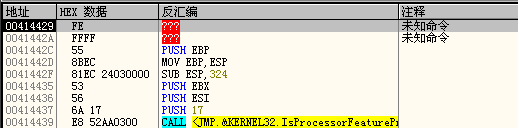
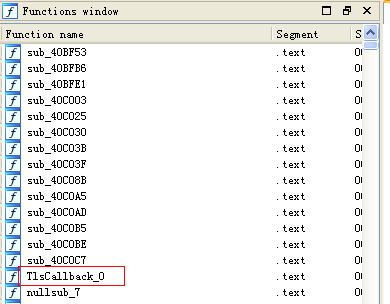
程序运行起来输入测试key，点击验证没反应，再点一次程序崩溃，这其实和题目的设计有关，后面会揭晓原因。



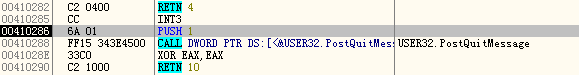
直接拖到od里运行，发现入口点被修改了：



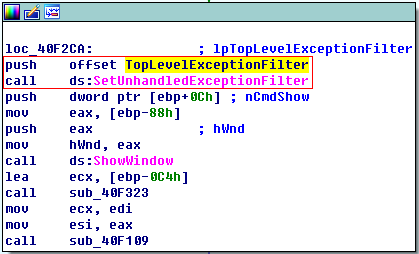
拖IDA看一下，一眼发现有TLS，先于入口点执行了：



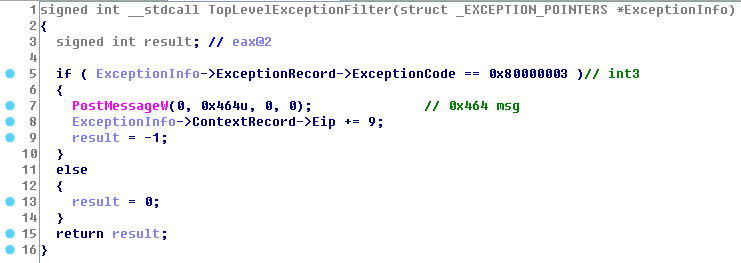
所以省事就先运行程序，然后OD附加进程调试，去下断获取输入的地方，发现没开始验证呢就不断的采集输入的字符串，估计和想实现直接回车验证有关，这样也稍微增加了下断验证点的难度。后来直接不下断点点击验证了，但是调试器却自己中断了：



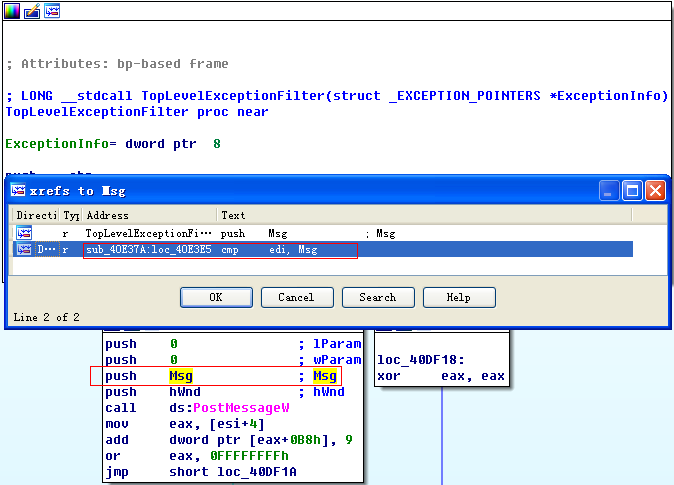
可以看到代码里直接写了个INT3断点，应该是为了触发异常进入异常处理函数，先看一下代码里的异常处理设置，找到了一处可疑的：



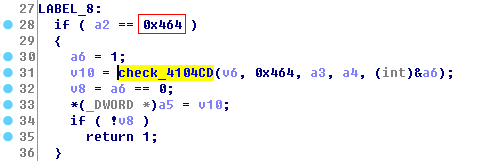
进入异常处理函数，正好设定了处理int3异常，然后就发送0x464号消息：



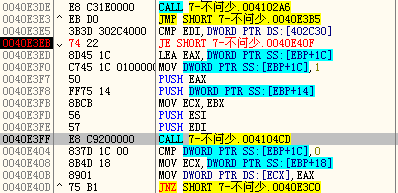
仔细一看，该消息号保存在一个全局变量里，交叉引用可以看到另一处调用：



该处调用处于一些消息处理的函数里，处理该消息时发现了验证的主函数：



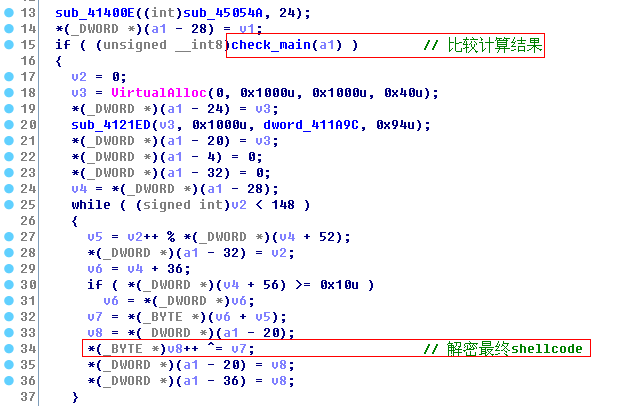
后面要调试验证过程可以直接在这个地方修改流程，可以重复的进入验证函数跟踪：



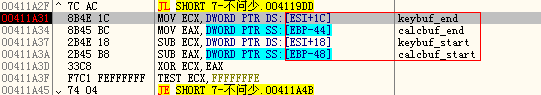
进入分析验证流程，可以看到作者还特意把计算输入key的函数加密了，需要的时候解密调用，最后计算进入**check**函数验证：



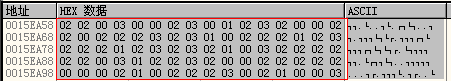
Check函数的流程是比较上面计算的结果和预先计算的结果是否一致，如果通过比较，就解密一个shellcode去执行：



跟踪一下比较函数吧，毕竟比较关键，发现在下图位置计算两个比较对象的长度：



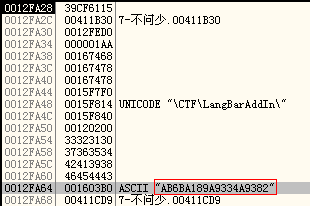
Esi+0x18指向的即期望计算后的对象地址，发现是一个数组：



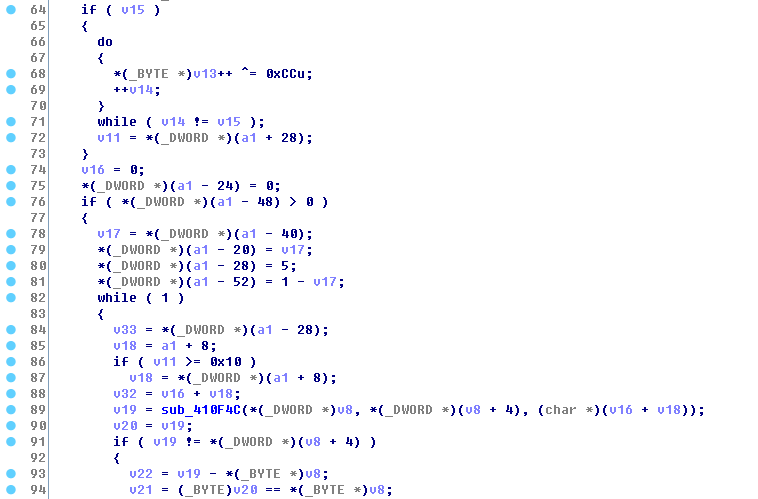
而ebp-0x48指向的为使用测试输入串（“weiyiling”）计算后的数组：



在跟踪过程中还发现了一个16进制字符串：



仔细分析一下能够确定，输入串一个字符对应上面16进制字符串的两个字符，且对应计算后数组的4个字节数据，从而可以猜测最终输入的计算结果应该和期望数组一致（算法在**sub\_41612a**），并且间接得到最终输入key的长度：0x50 / 4 = 0x14，也就是20个字符。这样一来，关键就在计算输入串转换为数组的那个被加密的函数calc了，点进去一看，妈呀真长，下面截取部分：



有点不想去跟踪这个算法了，于是萌生了编写OD脚本自动爆破每一位的想法，正好测试过程发现，输入串每一位若输入固定字符，输出的对应数组元素的4个字节数据是一样的（其实我只看了16进制的对应两个字符是一样的）。这样一来就很好爆破了，枚举每一位即可，下面贴出调试的OD枚举脚本：

var patch

var gettext

var aftercalc

var inputbuf

var calcbuf

var keybuf

var inputoffset

var calcoffset

var calcdowrd

var keydowrd

var testchar

mov patch,40e3eb

mov gettext,4104fc

mov aftercalc,411a31

mov inputoffset, 0

mov calcoffset, 0

mov testchar, 10

sub [patch],1//patch

go patch

test:

go gettext

mov inputbuf,[esp+8]

sto

add inputbuf, inputoffset

//mov [inputbuf],testchar

fill inputbuf, 1, testchar

go aftercalc

mov calcbuf,[ebp-48]

add calcbuf, calcoffset

mov keybuf,[esi+18]

add keybuf, calcoffset

mov calcdowrd, [calcbuf]

mov keydowrd, [keybuf]

cmp keydowrd, calcdowrd

je testok

add testchar, 1

cmp testchar, 9f

ja loopinput

jmp test

testok:

log "find"

log inputoffset

log testchar

add testchar, 1

jmp test

loopinput:

mov testchar, 10

add inputoffset, 1

add calcoffset, 4

cmp inputoffset, 14

ja end

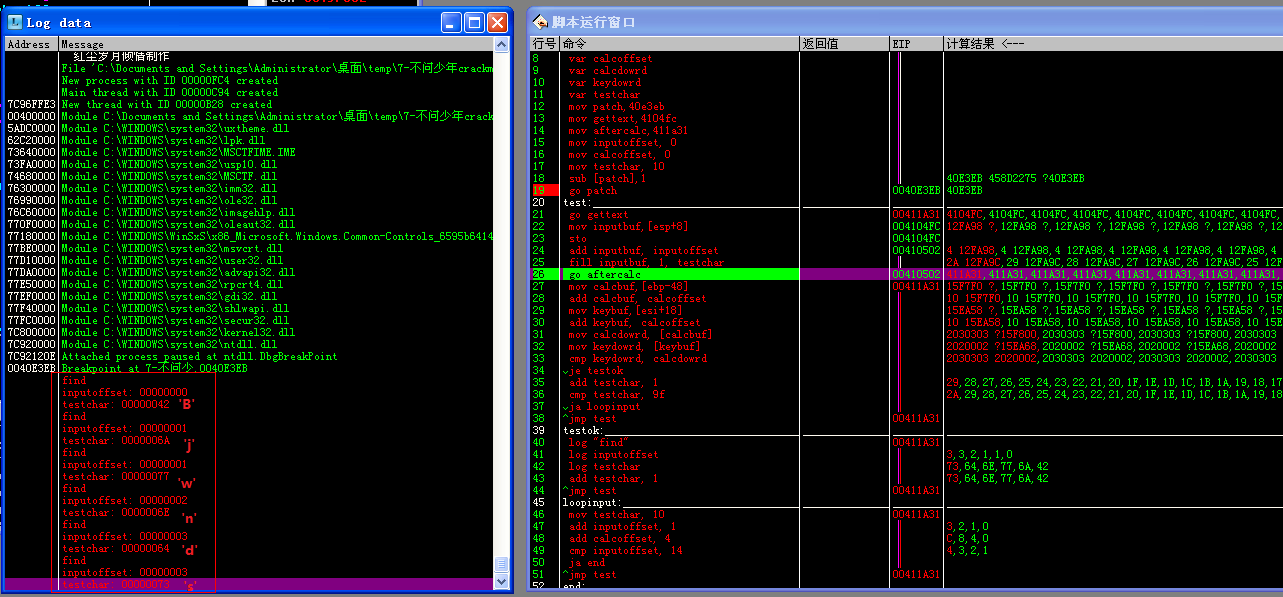
jmp test

end:

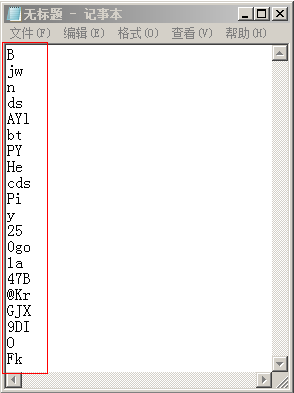
log "end"

//bc patch

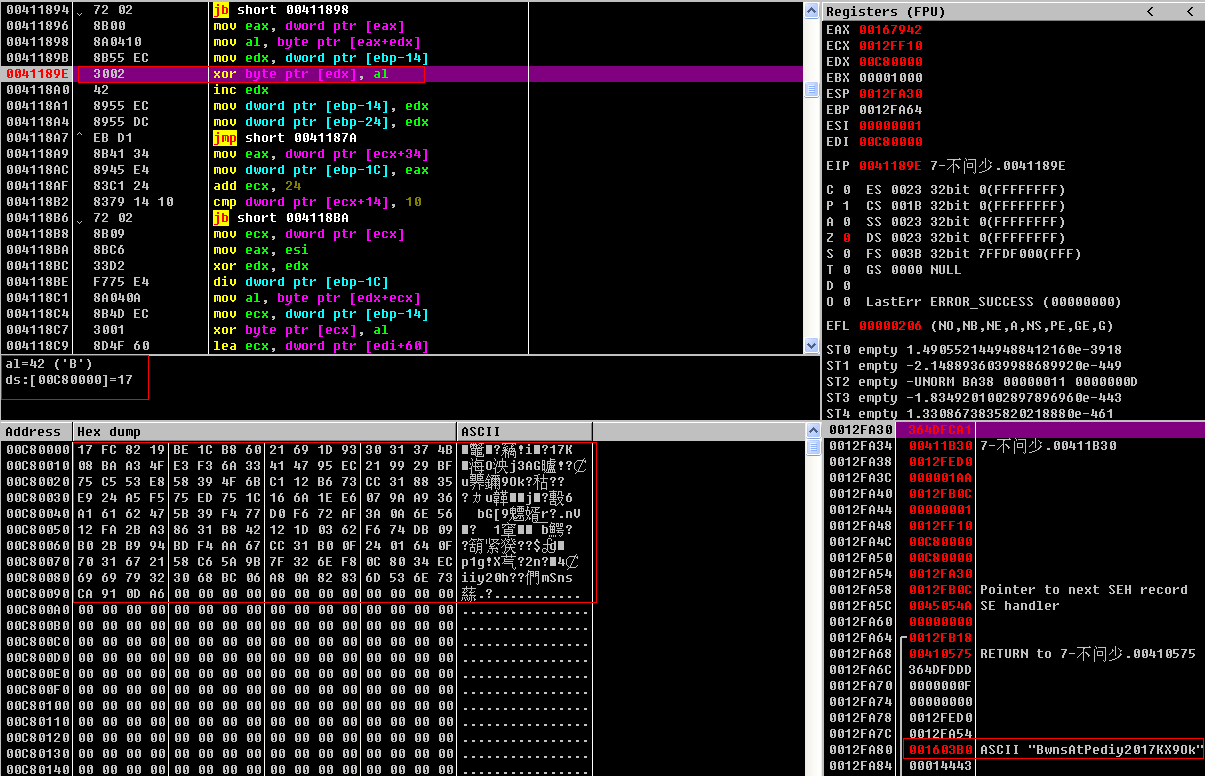
脚本的算法就不解释了，调试时载入OD跟一下很容易理解，最终能够把每一位可能的字符打印到OD的日志窗口里：



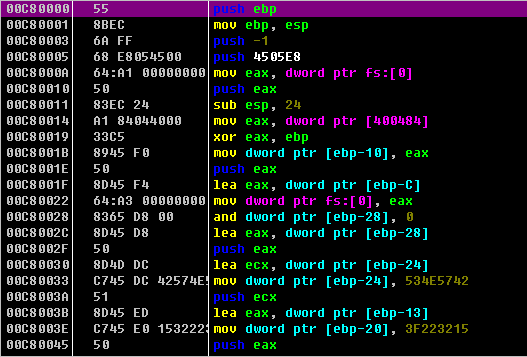
打印出来后发现有多解，下面每一行为第n行的字符集：



这样看的话那个比较只是第一层校验，后面的shellcode解密是第二层校验，但是能通过第一层校验的字符串有那么多（上万个），怎么破呢？看运气吧，比如如果你能从以上字符集看到连续的一个字符串“Pediy2017”，那你就成功了一大半，剩下的还得靠解密shellcode的过程。通过第一层校验后，会先分配一段内存，将加密的shellcode拷贝进去，然后使用输入串进行简单的异或解密：



成功解密后shellcode的样子，所以，这里别无选择，只能看你对shellcode代码的熟悉程度进行测试了：



只有正确猜出输入key才能解密如上正确的shellcode，弹出成功提示，否则就会像开始那样，点击两次验证按钮就程序崩溃，因为没有校验解密出的shellcode校验和就直接执行了。。。

