

期中作业

期中作业

1 PC端作业

- 1.1 protector分析与反调试绕过
- 1.2 动态调试寻找原始入口点
- 1.3 静态分析总结

2 移动端作业

- 2.1 packer分析与脱壳尝试
- 2.2 动态调试与绕过反调试
- 2.3 静态分析总结

- 分析环境

OS	Arch
Microsoft Windows 10	Intel64
Android 6.0 (Linux kernel v3.10.0+)	armv7l

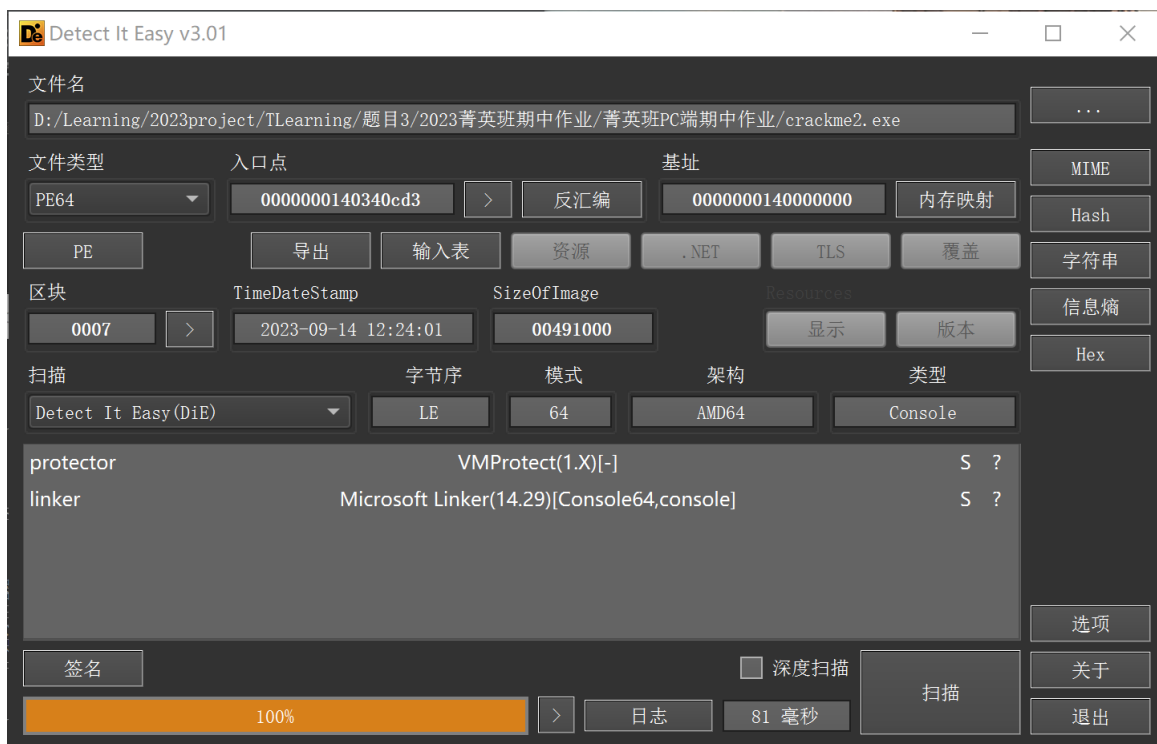
- 使用工具

Tools	Version
Detect It Easy	v3.01
x64dbg	Jun 15 2022
ScyllaHide Plugin	v1.4.750
IDA Pro	v7.5 SP3
Android Studio Giraffe	2022.3.1 Patch 2
Android Debug Bridge	v1.0.41
UPX	v4.1.0 & v3.95
OllyDumpEx	v1.84

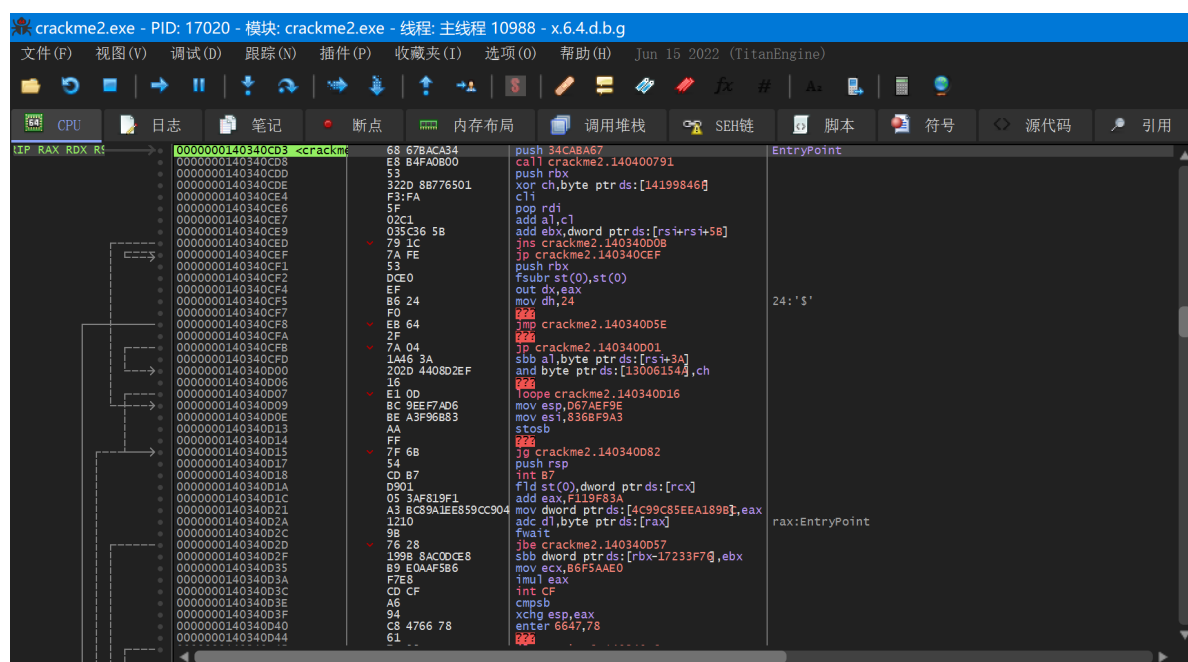
1 PC端作业

1.1 protector分析与反调试绕过

将 crackme2.exe 拖入 DIE 工具，确定文件类型 PE64、有壳 VMProtect、入口点 0x140340cd3 (肯定不是OEP)



尝试动态调试，使用 x64dbg 调试，Alt+F9 运行到入口点 0x140340cd3，“先push再call”属于典型 vmp特征，这里说明程序入口点被加壳，需要后续动态分析定位原始入口点 OEP。



单击 F9 继续运行，弹出提示 检测到调试器，说明程序存在反调试机制，不能直接进行动态调试寻找 OEP。

00007FFCB83CD058	✓ F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFCB83CD060	75 03	jne ntdll.7FFCB83CD065	
00007FFCB83CD062	0F05	syscall	
00007FFCB83CD064	C3	ret	
00007FFCB83CD065	CD 2E	int 2E	
00007FFCB83CD067	C3	ret	
00007FFCB83CD068	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
00007FFCB83CD070	4C:8B01	mov r10,rcx	NtCallbckReturn
00007FFCB83CD073	B8 05000000	mov eax,5	
00007FFCB83CD078	✓ F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFCB83CD080	75 03	jne ntdll.7FFCB83CD085	
00007FFCB83CD082	0F05	syscall	
00007FFCB83CD084	C3	ret	
00007FFCB83CD085	CD 2E	int 2E	
00007FFCB83CD087	C3	ret	
00007FFCB83CD088	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
00007FFCB83CD090	4C:8B01	mov r10,rcx	NtReadFile
00007FFCB83CD093	B8 06000000	mov eax,6	
00007FFCB83CD098	✓ F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFCB83CD0A0	75 03	jne ntdll.7FFCB83CD0A5	
00007FFCB83CD0A2	0F05	syscall	
00007FFCB83CD0A4	C3	ret	
00007FFCB83CD0A5	CD 2E	int 2E	
00007FFCB83CD0A7	C3	ret	
00007FFCB83CD0A8	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
00007FFCB83CD0B0	4C:8B01	mov r10,rcx	ZwDeviceIoControlFile
00007FFCB83CD0B3	B8 07000000	mov eax,7	
00007FFCB83CD0B8	✓ F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFCB83CD0C0	75 03	jne ntdll.7FFCB83CD0C5	
00007FFCB83CD0C2	0F05	syscall	
00007FFCB83CD0C4	C3	ret	
00007FFCB83CD0C5	CD 2E	int 2E	
00007FFCB83CD0C7	C3	ret	
00007FFCB83CD0C8	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
00007FFCB83CD0D0	4C:8B01	mov r10,rcx	NtWriteFile
00007FFCB83CD0D3	B8 08000000	mov eax,8	
00007FFCB83CD0D8	✓ F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
00007FFCB83CD0E0	75 03	jne ntdll.7FFCB83CD0E5	
00007FFCB83CD0E2	0F05	syscall	
00007FFCB83CD0E4	C3	ret	
00007FFCB83CD0E5	CD 2E	int 2E	
00007FFCB83CD0E7	C3	ret	
00007FFCB83CD0E8	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	

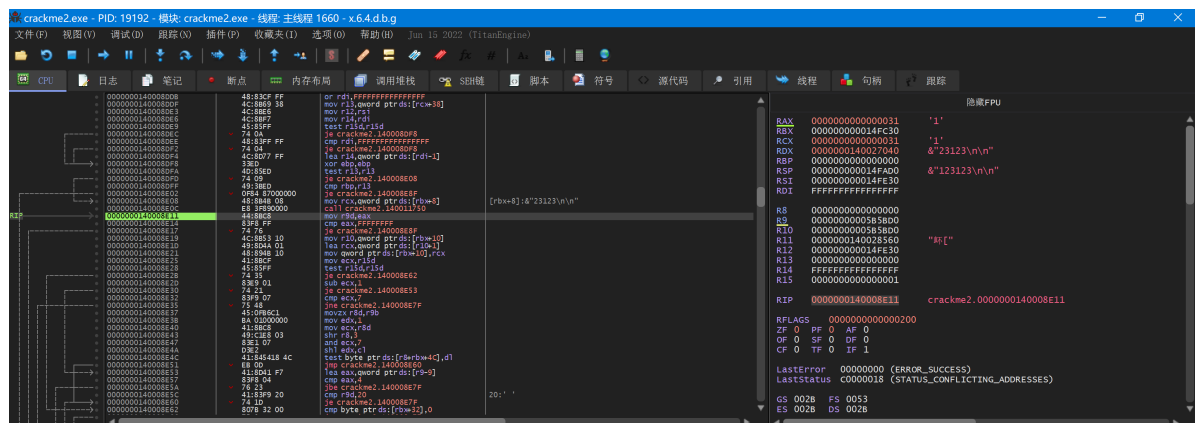
```
> kernelbase.0x00007FFCBBA45783 call NtReadFile->crackme2.0x140019635 call ReadFile
-> (此处省略近10次call, 因为找到ReadFile其实就基本找到了入口点), 我们对找到的整个调用链上的call语句下断点。
```

```

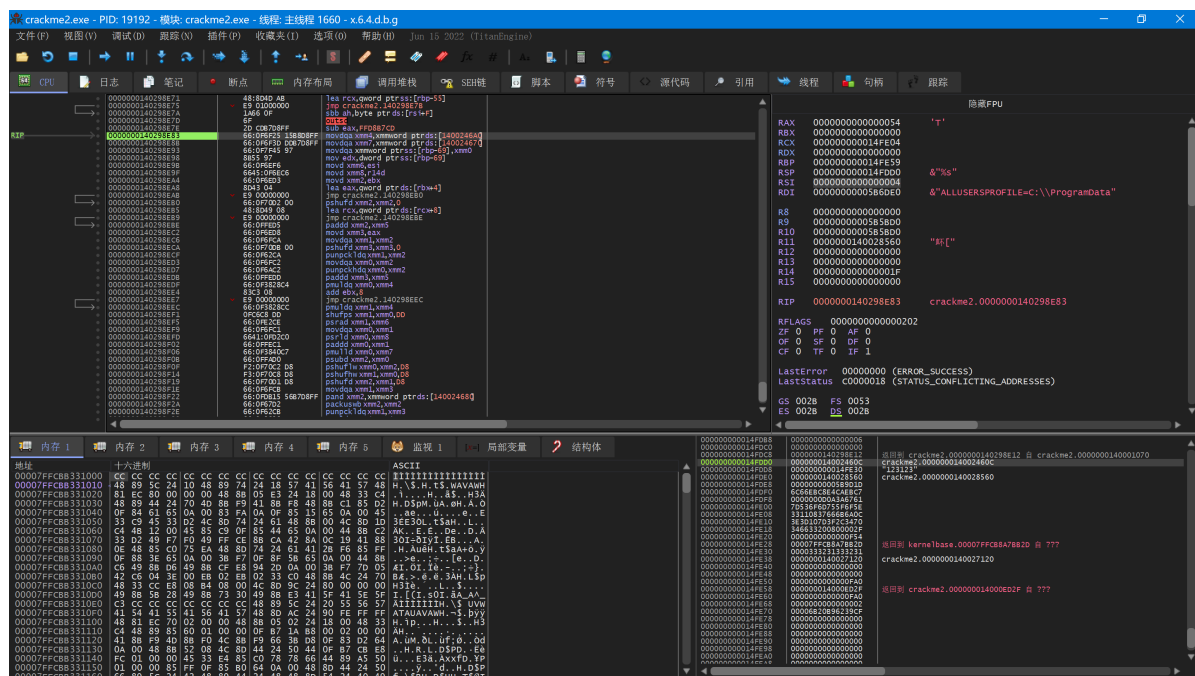
0000000000014F940 00000000000000081
0000000000014F948 00007FFCB835B44D 返回到 ntdll.00007FFCB835B44D 自 ntdll.00007FFCB835D160
0000000000014F950 00000000000460000
0000000000014F958 00007FFCB8A45783 返回到 kernelbase.00007FFCB8A45783 自 ???
0000000000014F960 0000000000001000
0000000000014F968 0000000000000000
0000000000014F970 00000000000469480
0000000000014F978 0000000000014F984
0000000000014F980 0000000000014F980
0000000000014F988 00000000000479BF0
0000000000014F990 00000000100001000
0000000000014F998 0000000000000000
0000000000014F9A0 0000000000000000
0000000000014F9A8 0000000000000000
0000000000014F9B0 0000000000000000
0000000000014F9B8 0000000000000000
0000000000014F9C0 0000000000000000
0000000000014F9C8 0000000000000000
0000000000014FD0 00000000140027040 crackme2.0000000140027040
0000000000014FD8 0000000014001963B 返回到 crackme2.000000014001963B 自 ???
0000000000014FE0 0000000000002688
0000000000014FE8 0000000000000000
0000000000014FF0 0000000000000000
0000000000014FF8 0000000000014FA98
0000000000014FA00 0000000000000000
0000000000014FA08 0000000000000000
0000000000014FA10 00000000000001F7
0000000000014FA18 0000000000000054
0000000000014FA20 0000000000000000
0000000000014FA28 0000000000000001
0000000000014FA30 0000000000000000

```

由于本程序被混淆处理得十分破碎，故只能在动态分析中猜测关键程序段的作用。输入字符串 123123 在函数 0x140011750 附近逐个取出 132123 的字符进行判断，这里猜测是执行类似 strlen() 的函数，其中 \n 不会算作字符串长度。（当然也可能是执行了类似 strcpy() 的函数）



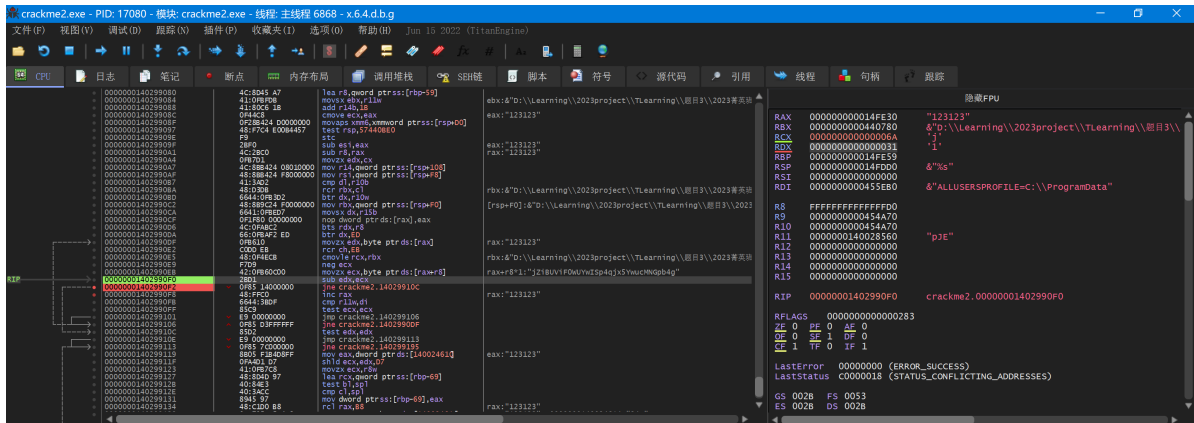
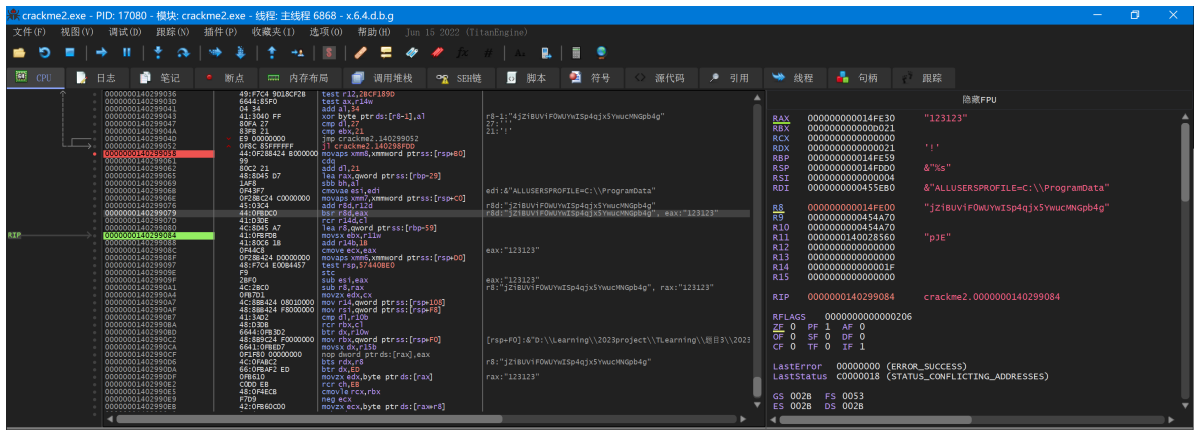
单步执行上述断点的函数后，RIP 指向了一段有着非常多 xmm 寄存器操作的地方（这很难不让人想起第一次作业判断分支前面的大量 xmm 寄存器），且继续往下翻不远处有一个 sleep() 调用，猜测这是为了在显示 flag 是否正确后延迟程序终止——故 flag 判断一定在他们两者之间！



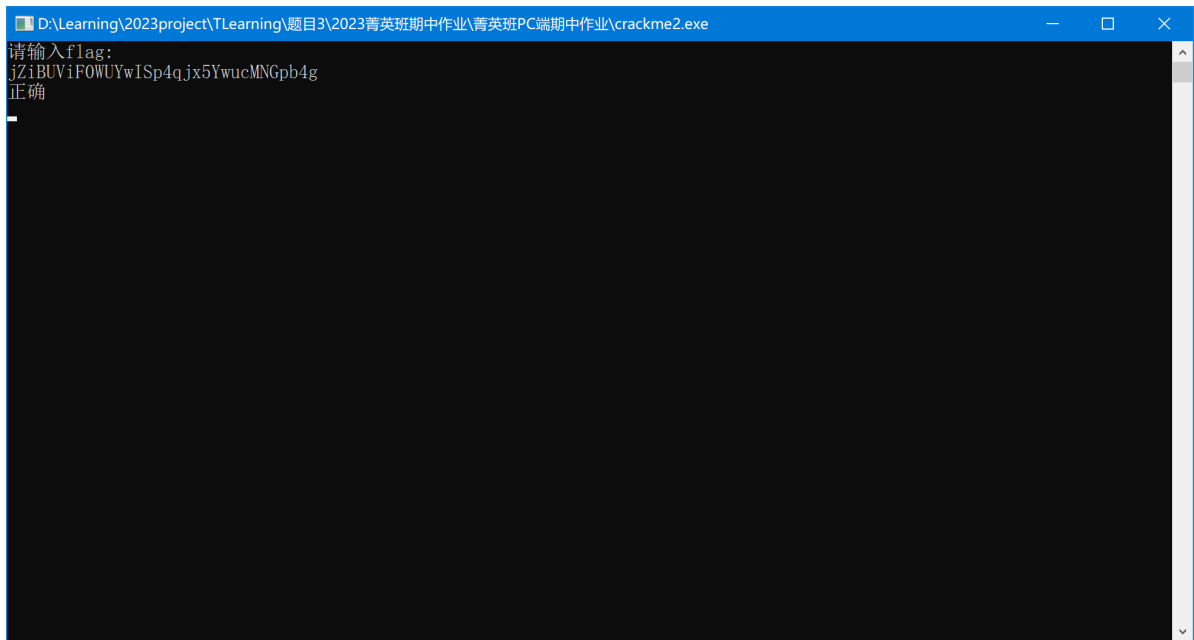
程序中有很多 cmp ebx, 20、cmp ebx, 21 的判断回跳，为加快调试效率，对于这类实际上回回回跳转循环执行的，且从内存中也不能明显看出其逻辑的循环，直接对后面设断点并 F9 即可。

在跳过了大量 xmm 寄存器处理后，发现 [rbp-59] = 0x14FE00 处有一处很长的字符串 jZiBUViF0WUYwiSp4qjx5YwucMNGpb4g，且在继续单步运行后，出现非常明显的字符串比较判断分支：

```
1 sub edx, ecx
2 jne notequal
3 inc rax
4 cmp r11w, di
5 test ecx, ecx
6 ...
```

可以初步判断此字符串为flag，在 crackme2.exe 中经过验证确实为flag



1.3 静态分析总结

用IDA反编译x64dbg中找到的疑似字符串比较代码，虽然已经面目全非，但还是基本能看出 RCX 存放输入字符串的某一位， v39 存放flag字符串相应位的地址， v48 存放输入字符串的首地址的指针(形如 &"123123")， v50 存放flag字符串的首地址，故很容易看出这里是直接取flag进行比较的。

```

203 do
204 {
205     __asm { rcr      ch, 0EBh }
206     LODWORD(_RCX) = (unsigned __int8)v39[(char *)v48 - v50];
207     v44 = (unsigned __int8)*v39 - (_DWORD)_RCX;
208     if ( v44 )
209         break;
210     ++v39;
211 }
212 while ( (_DWORD)_RCX );
213 if ( !v44 )
214 {
215     _RAX = 2164181217i64;
216     v47.m128i_i32[0] = -1211564587;
217     __asm { rcl      rax, 0B8h }
218     v47.m128i_i16[2] = 3338;
219     v47.m128i_i8[6] = 0;
220     sub_140001020((int)&v47);
221 }
222 return MEMORY[0x26330](5000i64);
223 }

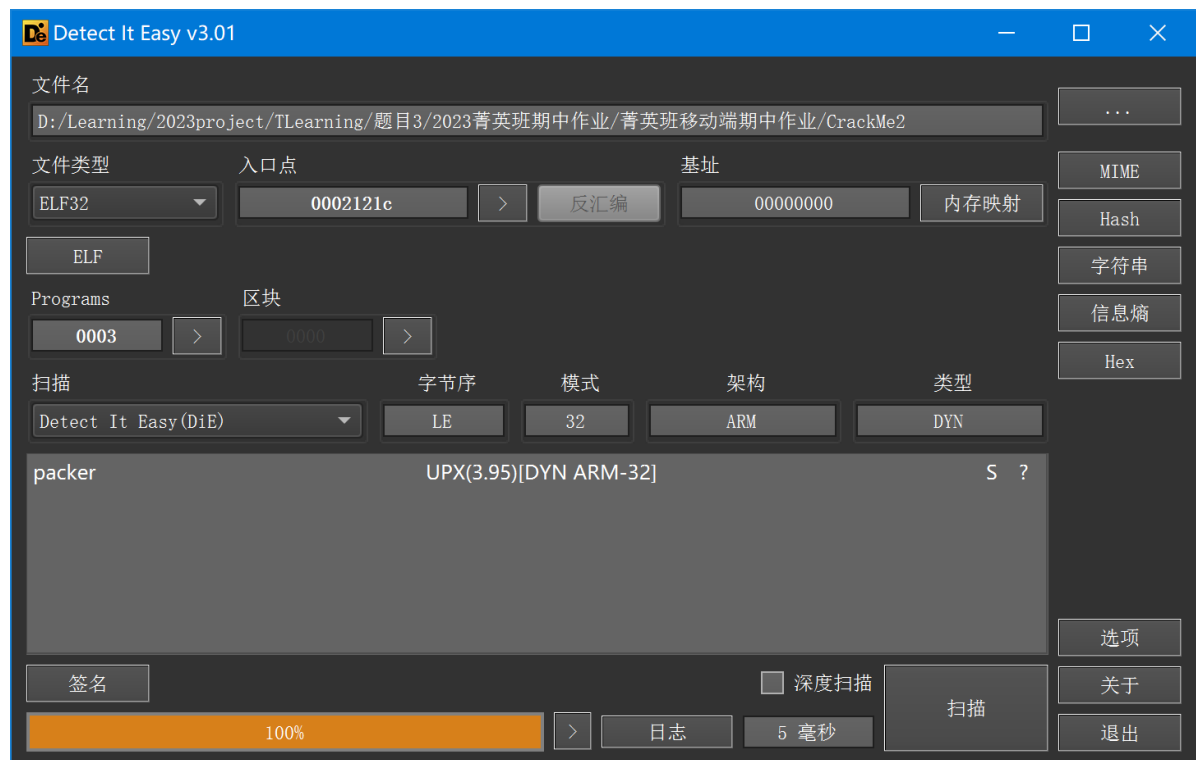
```

综上，本题最主要的难点在于绕过反调试，以及如何通过手动debug在复杂混乱的程序中找到关键的代码段，这里我通过 `ReadFile`、`%s`、`&"123123"` 等许多可以在x64dbg监视窗口上直接阅读到的信息，找到了程序真实入口，并获取到了flag。

2 移动端作业

2.1 packer分析与脱壳尝试

将CrackMe2拖入DIE工具，有upx3.95压缩加壳，arm32指令集架构，入口点0x0002121c



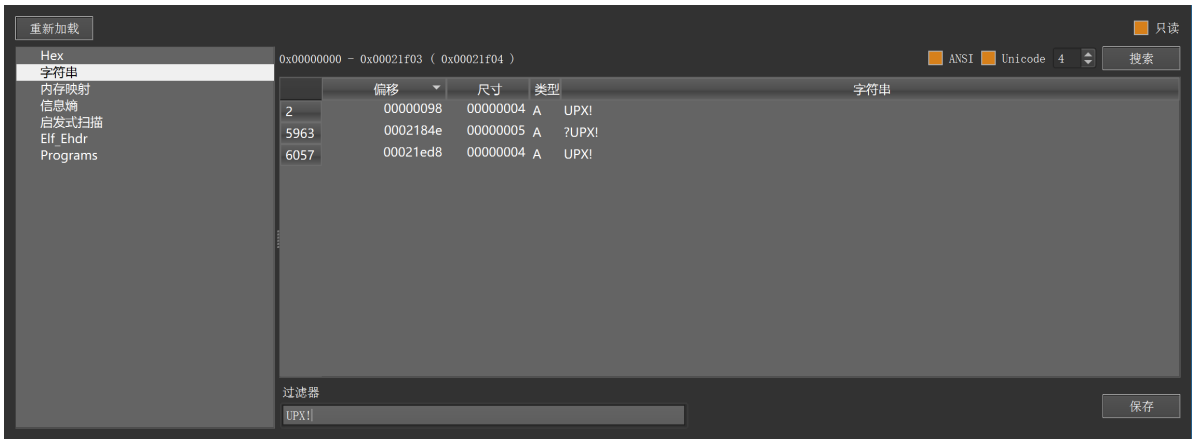
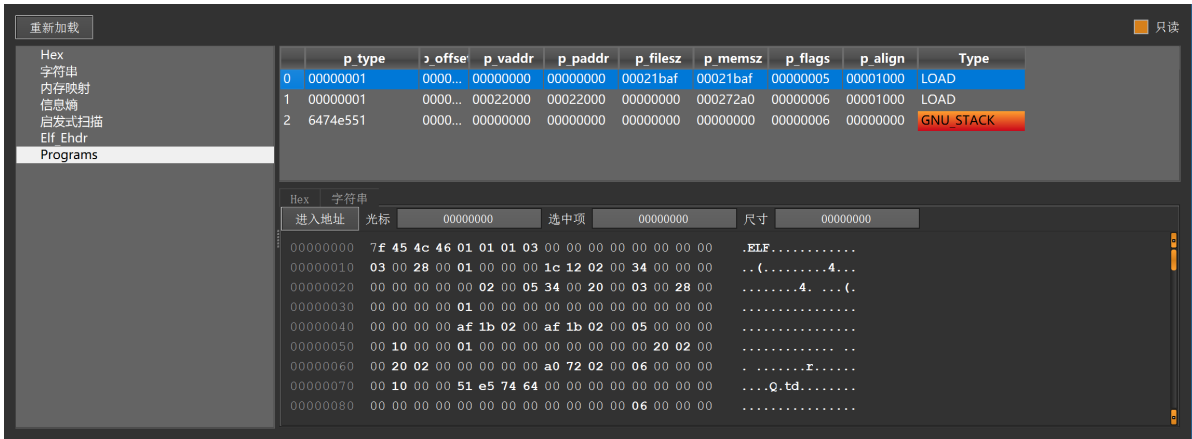
直接工具脱壳尝试，`upx -d CrackMe2`，居然提示 `not packed by UPX?` 说明做了反脱壳保护。“知己知彼，百战不殆”，[UPX防脱壳机脱壳、去除特征码、添加花指令小探](#)有许多UPX防脱壳的策略，包括①修改区段名；②改标识；③去掉UPX特征码；④添加花指令.....等方法。

```
D:\Learning\CTF\upx\upx-4.1.0-win64>upx -d D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业\CrackMe2
Ultimate Packer for executables
Copyright (C) 1996 - 2023
UPX 4.1.0 Markus Oberhumer, Laszlo Molnar & John Reiser Aug 8th 2023

File size      Ratio      Format      Name
-----
upx: D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业\CrackMe2: NotPackedException: not packed by UPX
Unpacked 0 files.

D:\Learning\CTF\upx\upx-4.1.0-win64>
```

通过 **DIE** 分析，并没有找到很明显的可以“反反脱壳”的地方，因为这个程序总共就3个程序段，连一个区也没有看到，且如果是 **UPX!** 标识被修改，我更是不可能遍历出被修改的地址。



2.2 动态调试与绕过反调试

既然不能直接工具脱壳，我们采取Plan B：动态调试进入OEP后，upx已经解压缩，利用 **ESP** 定理从 **libc**库返回到程序，dump内存

将CrackMe2 **adb push** 到 **Android Studio VAD** 设备中，同时将 **IDA Pro** 的 **android_server** 传入虚拟机，用 **adb forward tcp:23946 tcp:23946** 开启流量转发，IDA便可以附加到VAD的进程上进行调试。

但如下图所示，在 **android_server** 端口监听开启时，**./CrackMe2** 一运行就会提示 **undebug**，这是典型的检测端口反调试，但可以通过先运行 **./CrackMe2** 再开启端口监听解决。（如下左图最后一次运行CrackMe2时成功打印 **Input Your Answer**）

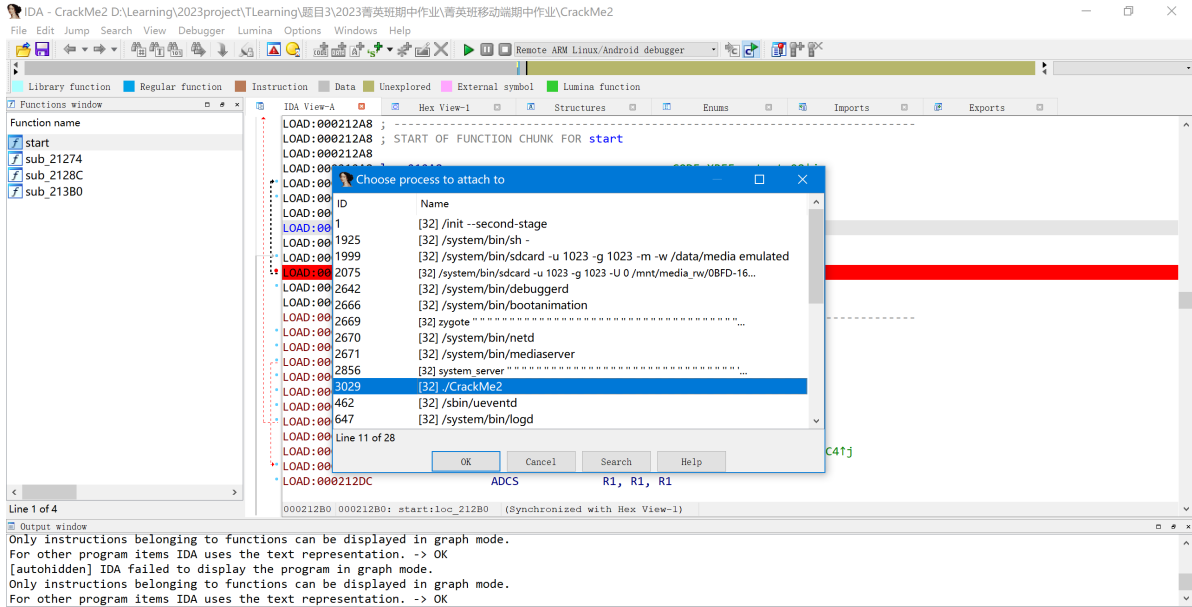

```
C:\Windows\System32\cmd.exe - adb shell
127|root@generic:/data/local/tmp # ./CrackMe2
undebug
C
C
Z
[1] + Stopped (signal) ./CrackMe2
root@generic:/data/local/tmp #
[1] + Aborted ./CrackMe2
root@generic:/data/local/tmp # ./CrackMe2
undebug
Aborted
134|root@generic:/data/local/tmp # ./CrackMe2
Input Your Answer:
123
Killed
137|root@generic:/data/local/tmp # ./CrackMe2
undebug
[1] + Stopped (signal) ./CrackMe2
root@generic:/data/local/tmp # ./CrackMe2
undebug
[2] + Stopped (signal) ./CrackMe2
[1] + Aborted ./CrackMe2
root@generic:/data/local/tmp # ./CrackMe2
undebug
[1] + Stopped (signal) ./CrackMe2
[2] - Aborted ./CrackMe2
root@generic:/data/local/tmp # ./CrackMe2
Input Your Answer:

C:\Windows\System32\cmd.exe - adb shell
(c) Microsoft Corporation。保留所有权利。
D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业>adb shell
C
D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业>
D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业>adb shell
root@generic:/data/local/tmp # ls
CrackMe2
android_server
root@generic:/data/local/tmp # ./android_server
IDA Android 32-bit remote debug server(ST) v7.5.26. Hex-Rays (c) 2004-2020
Listening on 0.0.0.0:23946...
1970-01-01 00:02:23 [1] Accepting connection from 127.0.0.1...
^Cgot signal #2, terminating
1|root@generic:/data/local/tmp #
130|root@generic:/data/local/tmp # ./android_server
IDA Android 32-bit remote debug server(ST) v7.5.26. Hex-Rays (c) 2004-2020
Listening on 0.0.0.0:23946...
2023-10-24 05:24:53 [1] Accepting connection from 127.0.0.1...
Looking for GNU DWARF file at "/usr/lib/debug/.build-id/ab/1f096f534b8f9c8f090d1c196dacef.debug"... no.
Looking for GNU DWARF file at "/system/lib/libc.so"... found!
2023-10-24 05:26:25 [1] Closing connection from 127.0.0.1...
^Cgot signal #2, terminating
1|root@generic:/data/local/tmp #
130|root@generic:/data/local/tmp # ./android_server
IDA Android 32-bit remote debug server(ST) v7.5.26. Hex-Rays (c) 2004-2020
Listening on 0.0.0.0:23946...
```

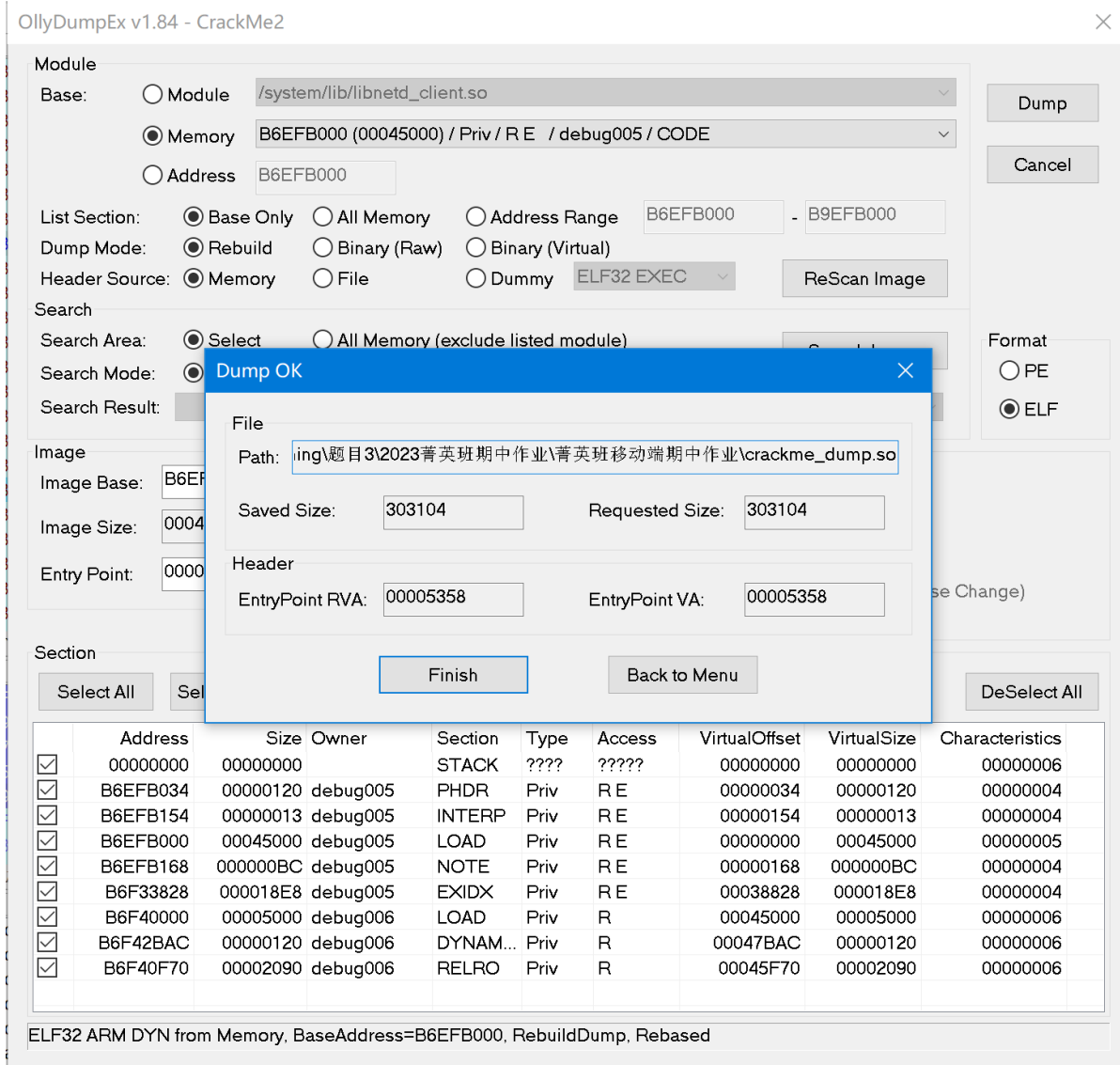
但是后续又发现第二个反调试机制，即在输入数据返回原程序后，会先进行一次 debugger 判断，并强制退出进程。（如下图在输入了 123123 后还是触发 undebug）

```
root@generic:/data/local/tmp # ./CrackMe2
undebug
[1] + Stopped (signal) ./CrackMe2
[2] - Aborted ./CrackMe2
root@generic:/data/local/tmp # ./CrackMe2
Input Your Answer:
123123
undebug
Aborted
[1] + Aborted ./CrackMe2
134|root@generic:/data/local/tmp #
```

解决方法是用IDA Pro附加进程后，通过栈中的地址回调，在返回到debugger判断函数时就立即 dump 内存，然后就不用管这个有壳的程序了，直接静态分析我们dump下来的程序。

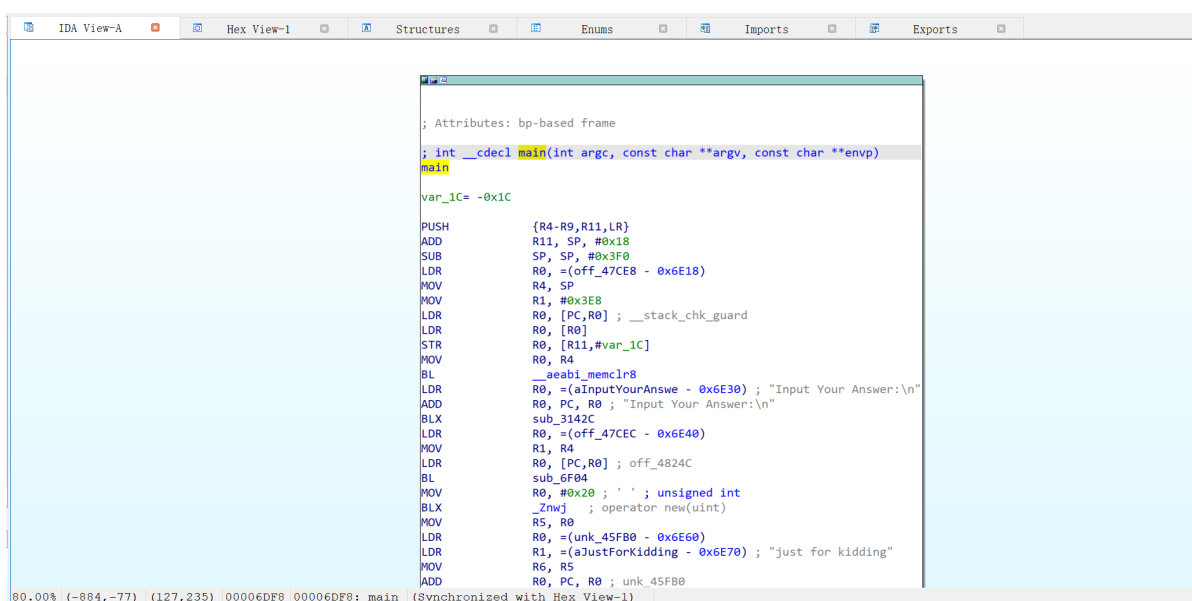


输入123123后，观察栈的回调地址，可以逐步进行断点：



2.3 静态分析总结

IDA Pro打开 crackme_dump.so , 很容易找到 main 位于 0x6DF8



常量声明区域，有很多可以解释上面反调试操作的字符串，检测栈溢出攻击的

__stack_chk_guard、检测端口监听的 cat /proc/net/tcp | grep :5D8A 和检测调试器 /proc/%d/status。其中有很可疑的 Base64 格式的编码：

DMD2vxKYDLvezuriqND2DhP3BJfdtuwrxepq== 但是用python脚本尝试，并不能直接decode()，说明

该编码已被部分或全部替换或者标准Base64索引表被替换。

```
IDA View-A | Pseudocode-B | Pseudocode-A | Hex View-1 | Structures | Enums | Imports
* LOAD:00047FFC off_47FFC DCD __imp_raise ; DATA XREF: raise+8↑r
* LOAD:00048000 aCatProcNetTcpG DCB "cat /proc/net/tcp |grep :5D8A",0
LOAD:00048000 ; DATA XREF: sub_53F4+AC↑o
LOAD:00048000 ; sub_53F4+B8↑o ...
* LOAD:0004801E aR DCB "r",0 ; DATA XREF: sub_53F4+C0↑o
LOAD:0004801E ; sub_53F4+C8↑o ...
* LOAD:00048020 aProcDStatus DCB "/proc/%d/status",0 ; DATA XREF: sub_59D4+28↑o
LOAD:00048020 ; sub_59D4+34↑o ...
* LOAD:00048030 aTracerpid DCB "TracerPid",0 ; DATA XREF: sub_59D4+60↑o
LOAD:00048030 ; sub_59D4+68↑o ...
* LOAD:0004803A aUndebug DCB "undebug",0xA,0 ; DATA XREF: sub_5AD0+20↑o
LOAD:0004803A ; sub_5AD0+24↑o ...
* LOAD:00048043 ALIGN 0x10
* LOAD:00048050 aDmd2vxkydlvezu DCB "DMD2vxKYDLvezuriqND2DhP3BJfduWwrx9pq==",0
LOAD:00048050 ; DATA XREF: LOAD:00005BC8↑o
LOAD:00048050 ; LOAD:00005BD8↑o ...
* LOAD:00048079 aWrongAnswer DCB "Wrong Answer",0xA,0
LOAD:00048079 ; DATA XREF: LOAD:00005B8C↑o
LOAD:00048079 ; LOAD:00005B9C↑o ...
* LOAD:00048087 ALIGN 0x10
* LOAD:00048090 aInputYourAnswer DCB "Input Your Answer:",0xA,0
LOAD:00048090 ; DATA XREF: main+2C↑o
LOAD:00048090 ; main+30↑o ...
* LOAD:000480A4 ALIGN 0x10
* LOAD:000480B0 aJustForKidding DCB "just for kidding",0
LOAD:000480B0 ; DATA XREF: main+58↑o
LOAD:000480B0 ; main+70↑o ...
* LOAD:000480C1 ALIGN 0x10
```

Base64编码规则

将字符串转换成二进制序列，每6个二进制位为一组，每6位组成一个新的字节，高位补00

编码表顺序：

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

0123456789+/-

在全局常量表上，出现了类似编码表的字符串，查看它们的调用位置，可能能找到编码的地址。

```
* LOAD:00048120 aAbcdefghijklmn DCB "abcdefghijklmnopqrstuvwxyz",0
LOAD:00048120 ; DATA XREF: sub_4E60+14↑o
LOAD:00048120 ; sub_4E60+38↑o ...
* LOAD:0004813B ALIGN 0x10
* LOAD:00048140 aAbcdefghijklmn_0 DCB "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0
LOAD:00048140 ; DATA XREF: sub_4E60+C8↑o
LOAD:00048140 ; sub_4E60+DC↑o ...
* LOAD:0004815B ALIGN 0x10
* LOAD:00048160 a0123456789 DCB "0123456789+/-",0 ; DATA XREF: sub_4E60+16C↑o
```

调用这3个字符串的函数是 `sub_4E60`、`sub_8998`，其中 `sub_8998` 可能是UPX用于初始化加密字符串的。`sub_4E60` 可能是跟base64编码有关，但是对于编码表的顺序值得注意，因为小写字母是始终排在前面的！

LOAD:00004EE8	STM	R0, {R1,R4,R5}
LOAD:00004EEC		
LOAD:00004EEC		loc_4EEC
LOAD:00004EEC	LDR	R0, =(aAbcdefghijklmn - 0x4EFC) ; "abcdefghijklmnopqrstuvwxy"
LOAD:00004EF0	MOV	R2, R4
LOAD:00004EF4	ADD	R1, PC, R0 ; "abcdefghijklmnopqrstuvwxy"
LOAD:00004EF8	MOV	R0, R5
LOAD:00004EFC	BL	__aeabi_memcpy
LOAD:00004F00		
LOAD:00004F00		loc_4F00
LOAD:00004F00		
LOAD:00004F04	LDR	R1, =(off_47CFC - 0x4F18) ; ABCD在abcd之后
LOAD:00004F08	MOV	R6, #0
LOAD:00004F08	LDR	R0, =(sub_7B28 - 0x4F1C)
LOAD:00004F0C	LDR	R3, =(off_481F8 - 0x4F20)
LOAD:00004F10	LDR	R2, [PC,R1] ; unk_481F0 ; lpdso_handle
LOAD:00004F14	ADD	R0, PC, R0 ; sub_7B28 ; lpfunc
LOAD:00004F18	ADD	R1, PC, R3 ; off_481F8 ; obj
LOAD:00004F1C	STRB	R6, [R5,R4]
LOAD:00004F20	BL	__cxa_atexit
LOAD:00004F24	LDR	R0, =(off_48208 - 0x4F34)
LOAD:00004F28	LDR	R1, =(aAbcdefghijklmn_0 - 0x4F44) ; "ABCDEFGHIIJKLMNOPQRSTUVWXYZ"
LOAD:00004F2C	ADD	R0, PC, R0 ; off_48208
LOAD:00004F30	STR	R6, [R0] ; dword_20
LOAD:00004F34	STR	R6, [R0,#(dword_4820C - 0x48208)]
LOAD:00004F38	STR	R6, [R0,#(dword_48210 - 0x48208)]
LOAD:00004F3C	ADD	R0, PC, R1 ; "ABCDEFGHIIJKLMNOPQRSTUVWXYZ"
LOAD:00004F40	BL	strlen
LOAD:00004F44	MOV	R4, R0

大胆尝试，编写一个自定义base64编码索引的脚本，尝试加解密文本。（其实这个编码索引的改变只是改变了字母的大小写而已）

```

1 import base64
2 # 标准索引
3 standard_charset =
4 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
5 # 自定义索引
6 my_charset =
7 "abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/"
8 # 构建转换
9 trans = str.maketrans(standard_charset, my_charset)
10 # 要解码的数据
11 encoded_data = "DMD2vxKYDLvezuriqND2DhP3BJfdtuWwrx9pq=="
12 for i in range(2):
13     # 先进行编码转换
14     encoded_data = encoded_data.translate(trans)
15     print(encoded_data)
16     # 再进行正常base64编码
17     data = base64.b64decode(encoded_data.encode()).decode()
18     print(data)
19     encoded_data = data

```

第一次进行编码转换和base64解密，发现还是base64 `vgvUy2vUDeDHBwvtzwn1CML0Eq==` 但是经过检验，发现还是不能用标准base64解密，故再用第二次编码转换和base64解密，最后得到 `TencentGameSecurity`，看起来很像flag的样子。

```

(base) PS D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业> python .\mybase64.py
dmd2VXkyd1VEZURIQnd2dHp3bjFDTUwWrx9PQ==
vgvUy2vUDeDHBwvtzwn1CML0Eq==

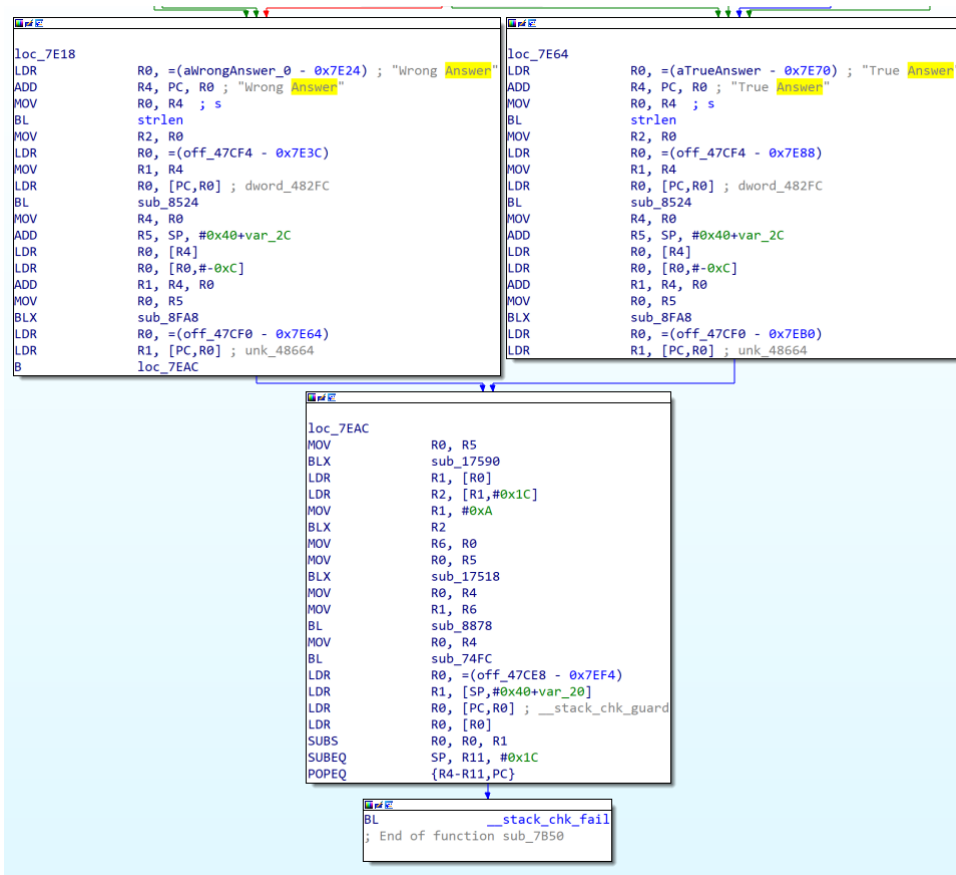
```

```

1 (base) PS D:\Learning\2023project\TLearning\题目3\2023菁英班期中作业\菁英班移动端期中作业> python .\mybase64.py
2 dmd2VXkyd1VEZURIQnd2dHp3bjFDTUwWrx9PQ==
3 vgvUy2vUDeDHBwvtzwn1CML0Eq==
4 VGVuY2VudEdhbWVtZW50cm10eQ==
5 TencentGameSecurity

```

直接找 True Answer 和 Wrong Answer 的调用记录，发现一个判断分支，说明分支上方就是字符串比较的过程。



上面的判断逻辑中，a1是一个有3个元素的指针数组，结合base64的编码特征，3字节数据会被编为4字节编码，a1可能是输入字符串的base64编码。

在下面的循环判断中 $v32 < v26 < *a4$ 肯定与编码的长度有关， $v31 = a4 + 1$ ，猜测a4指针的后面就是flag编码，而 $v29 = a1[2]$ ，猜测这里的a1[2]就是输入字符串的base64编码。

```

148  if ( v27 == v25 )
149  {
150      v29 = (unsigned __int8 *)a1[2];
151      v30 = v28 == 0;
152      v31 = a4 + 1;
153      if ( v30 )
154          v29 = (unsigned __int8 *)a1 + 1;
155      if ( (v26 & 1) == 0 )
156      {
157          if ( v27 )
158          {
159              v32 = -(v26 >> 1);
160              while ( *v31 == *v29 )
161              {
162                  ++v32;
163                  ++v29;
164                  ++v31;
165                  if ( !v32 )
166                      goto LABEL_43;
167              }
168              goto LABEL_42;
169          }
170      LABEL_43:
171          v36 = strlen(aTrueAnswer);

```

综上，猜测程序对输入的字符串进行了**两次自定义base64编码索引的编码**，然后与 `DMD2vxKYDLvezuriqND2Dhp3BJfdtuWwrxepq==` 比较，所以将该编码进行两次解码，可以得到需要输入的flag `TencentGameSecurity`

