

dump 法从入门到熟练 (五)

一、引言

本篇介绍其他几种分析的思路。

二、思路一

根据经验，在处理数据流的过程里，很多样本都会使用 memcpy、malloc 这些函数操纵内存块。我们可以 Hook 这些函数，从侧面观察程序的数据流。

观察我们的 patch 函数，样本执行流里所有的外部库访问都在这里，我决定对这些函数做参数的打印。

在这个处理过程里，我发现之前的代码有个小 bug。样本的 _aeabi_memset 函数调用到外部库 memset 时，参数 2/3 要倒一下。

```
1 // attributes: thunk
2 void __fastcall _aeabi_memset(void *a1, size_t a2, int a3)
3 {
4     memset(a1, a3, a2);
5 }
```

修改 patch 函数里对 memset 的处理

```
1 emulator.attach().addBreakPoint(0xb99e7478L, new BreakPointCallback() {
2     @Override
3     public boolean onHit(Emulator<?> emulator, long address) {
4         RegisterContext registerContext = emulator.getContext();
5         int num = registerContext.getIntArg(1);
6         int length = registerContext.getIntArg(2);
7         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R1, length);
8         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R2, num);
9         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("memset").getAddress());
10        return true;
11    }
12 });
```

优化 patch 函数里对 memcpy 的打印

```
1 emulator.attach().addBreakPoint(0xb99e7250L, new BreakPointCallback() {
2     @Override
3     public boolean onHit(Emulator<?> emulator, long address) {
4         RegisterContext registerContext = emulator.getContext();
5         int length = registerContext.getIntArg(2);
6         UnidbgPointer str1 = registerContext.getPointerArg(0);
7         UnidbgPointer str2 = registerContext.getPointerArg(1);
8         Inspector.inspect(str2.getByteArray(0, length), "memcpy src"+str2 + " dest"+str1);
9         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("memcpy").getAddress());
10        return true;
11    }
12 });
```

优化 patch 函数里对 memmove 的打印

```
1 emulator.attach().addBreakPoint(0xb99e728cL, new BreakPointCallback() {
2     @Override
3     public boolean onHit(Emulator<?> emulator, long address) {
4         RegisterContext registerContext = emulator.getContext();
5         int length = registerContext.getIntArg(2);
6         UnidbgPointer str1 = registerContext.getPointerArg(0);
7         UnidbgPointer str2 = registerContext.getPointerArg(1);
8         Inspector.inspect(str2.getByteArray(0, length), "memmove src"+str2 + " dest"+str1);
9         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("memmove").getAddress());
10        return true;
11    }
12 });
```

优化 patch 函数里对 strlen 的打印

```
1 emulator.attach().addBreakPoint(0xb99e7178L, new BreakPointCallback() {
2     @Override
3     public boolean onHit(Emulator<?> emulator, long address) {
4         RegisterContext registerContext = emulator.getContext();
5         UnidbgPointer str = registerContext.getPointerArg(0);
6         Inspector.inspect(str.getString(0).getBytes(), "strlen "+str);
7         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("strlen").getAddress());
8         return true;
9     }
10 });
```

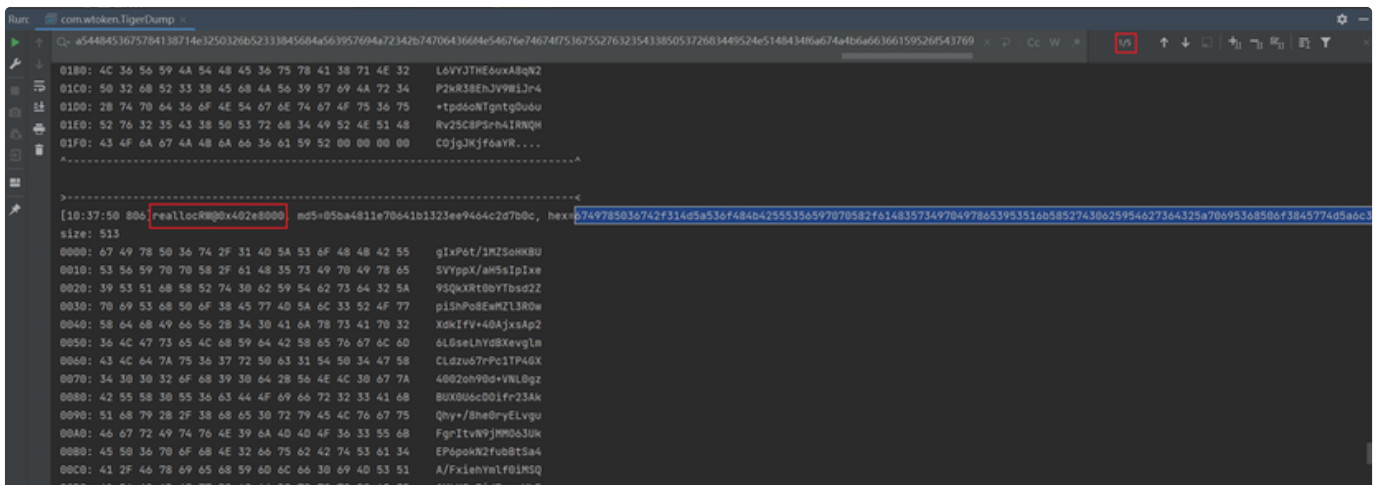
优化 patch 函数里对 realloc 的打印

```
1 emulator.attach().addBreakPoint(moduleBase + 0x1259c, new BreakPointCallback() {
2     @Override
3     public boolean onHit(Emulator<?> emulator, long address) {
4         RegisterContext registerContext = emulator.getContext();
5         int length = registerContext.getIntArg(1);
6         UnidbgPointer ptr = registerContext.getPointerArg(0);
7         Inspector.inspect(ptr.getByteArray(0, length), "realloc"+ptr);
8         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("realloc").getAddress());
9         return true;
10    }
11 });
```

优化 patch 函数里对 sprintf 的打印

```
1 emulator.attach().addBreakPoint(0xb99e73b8L, new BreakPointCallback() {
2     @Override
3     public boolean onHit(Emulator<?> emulator, long address) {
4         RegisterContext registerContext = emulator.getContext();
5         final UnidbgPointer str = registerContext.getPointerArg(0);
6         emulator.attach().addBreakPoint(registerContext.getLR(), new BreakPointCallback() {
7             @Override
8             public boolean onHit(Emulator<?> emulator, long address) {
9                 System.out.println("sprintf result:"+str.getString(0));
10                return true;
11            }
12        });
13        emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("sprintf").getAddress());
14        return true;
15    }
16 });
```

运行代码，上篇我们分析的那一长串疑似 base64 的内容，最早出现位置是下图日志处。



在它上方的 realloc 调用日志里，值也是它，只不过比它短一些，似乎是逐步生成出来的，我们可以在 realloc 上打印堆栈，看它的处理位置。

```

1  emulator.attach().addBreakPoint(moduleBase + 0x1259c, new BreakPointCallback() {
2      @Override
3      public boolean onHit(Emulator<?> emulator, long address) {
4          RegisterContext registerContext = emulator.getContext();
5          int length = registerContext.getIntArg(1);
6          UnidbgPointer ptr = registerContext.getPointerArg(0);
7          Inspector.inspect(ptr.getByteArray(0, length), "realloc"+ptr);
8          // print backtrace
9          emulator.getUnwinder().unwind();
10         emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("realloc").getAddress());
11         return true;
12     }
13 });

```

运行

The screenshot shows a debugger window with a memory dump and assembly view. A red box highlights two memory addresses: 0xb9a499c5 and 0xb9a37b87. Below the dump, assembly instructions are visible, including 'realloc' and 'reg_write'.

0xb9a499c5 - 0xb99d5000 = 0x749c5, IDA 里跳过去, 来到 sub_74918。它就是上篇我们说的 Base64 实现函数。

如果你看不出来, 可以去问 ChatGPT <<https://chat.openai.com/chat>>。

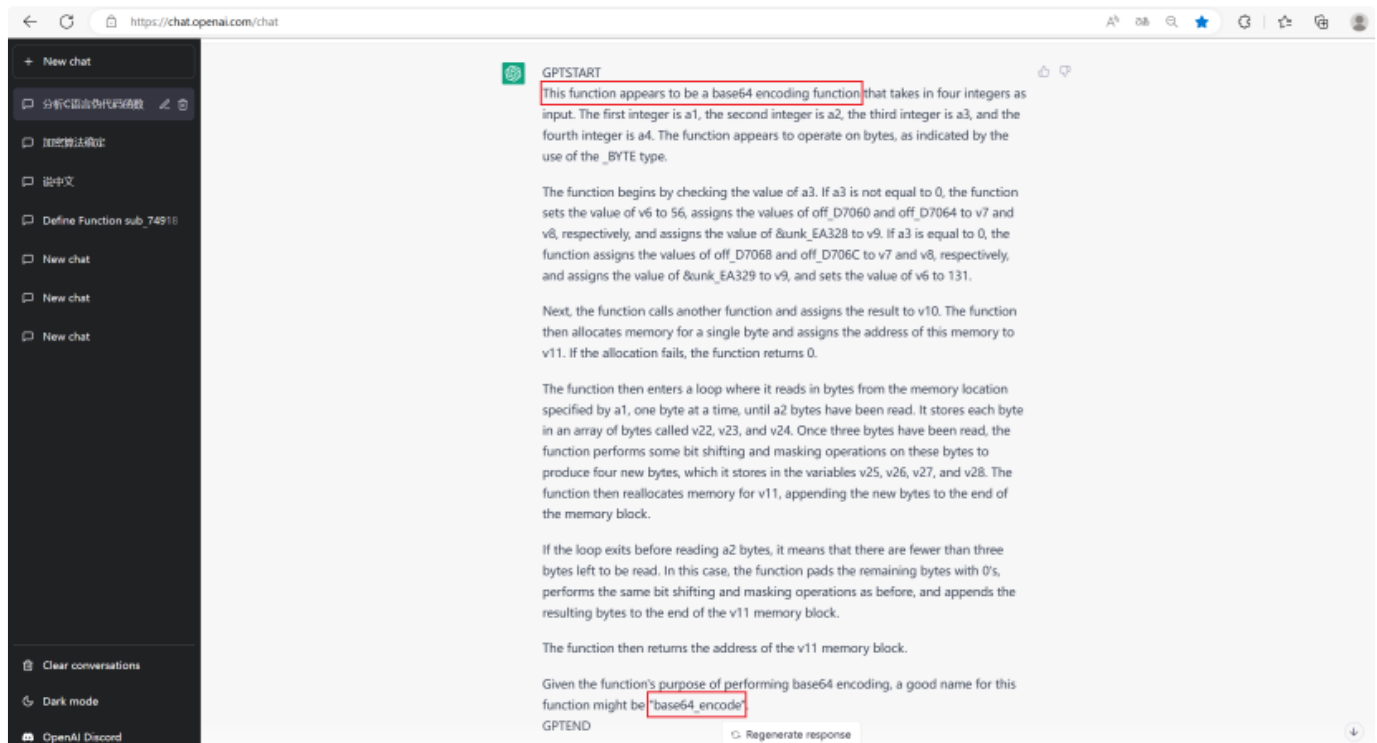
```
1 对下面的C语言伪代码函数进行分析 推测关于该函数的使用环境和预期目的详细的函数功能等信息 并为这个函数取一个新的名字 不要返回其他的内容 ^
2 _BYTE *__fastcall sub_74918(int a1, int a2, int a3, int a4)
3 {
4     int v6; // r2
5     void *v7; // r0
6     void *v8; // r1
7     void *v9; // r3
8     int v10; // r10
9     char *v11; // r0
10    int v12; // r8
11    unsigned int v13; // r5
12    int i; // r1
13    unsigned int j; // r2
14    _BYTE *result; // r0
15    int v17; // r4
16    int v18; // r9
17    int k; // r6
18    char *v20; // r1
19    unsigned __int8 v22; // [sp+9h] [bp-27h]
20    unsigned __int8 v23; // [sp+Ah] [bp-26h]
21    unsigned __int8 v24; // [sp+Bh] [bp-25h]
22    char v25; // [sp+Ch] [bp-24h]
23    char v26; // [sp+Dh] [bp-23h]
24    char v27; // [sp+Eh] [bp-22h]
25    char v28; // [sp+Fh] [bp-21h]
26
27    if ( a3 )
28    {
29        v6 = 56;
30        v7 = off_D7060;
31        v8 = off_D7064;
32        v9 = &unk_EA328;
33    }
34    else
35    {
36        v7 = off_D7068;
37        v8 = off_D706C;
38        v9 = &unk_EA329;
39        v6 = 131;
40    }
41    v10 = ((int (__fastcall *)(void *, void *, int, void *, int))loc_74AC4)(v7, v8, v6, v9, a4);
42    v11 = (char *)malloc(1u);
43    if ( !v11 )
44        return 0;
45    v12 = 0;
46    LABEL_6:
47    v13 = 0;
48    while ( a2 != v13 )
49    {
50        *(&v22 + v13) = *(_BYTE *)(a1 + v13);
51        if ( ++v13 == 3 )
52        {
53            v25 = v22 >> 2;
54            v28 = v24 & 0x3F;
55            v26 = (v23 >> 4) | (16 * (v22 & 3));
56            v27 = (v24 >> 6) | (4 * (v23 & 0xF));
57            v11 = (char *)realloc(v11, v12 + 4);
58            a2 -= 3;
59            a1 += 3;
60            for ( i = 0; i != 4; ++i )
61                v11[v12 + i] = *(_BYTE *)(v10 + (unsigned __int8)*(&v25 + i));
62            v12 += 4;
63            goto LABEL_6;
64        }
65    }
66    if ( v13 )
67    {
68        for ( j = v13; j <= 2; ++j )
69            *(&v22 + j) = 0;
70        v25 = v22 >> 2;
71        v28 = v24 & 0x3F;
72        v26 = (v23 >> 4) | (16 * (v22 & 3));
73        v17 = 0;
74        v27 = (v24 >> 6) | (4 * (v23 & 0xF));
```

```

75 while ( v13 + 1 != v17 )
76 {
77     v11 = (char *)realloc(v11, v12 + v17 + 1);
78     v11[v12 + v17] = *(_BYTE *) (v10 + (unsigned __int8)(&v25 + v17));
79     ++v17;
80 }
81 v18 = v12 + v17;
82 for ( k = 0; v13 + k <= 2; ++k )
83 {
84     v11 = (char *)realloc(v11, v18 + k + 1);
85     v20 = &v11[k];
86     v20[v12 + v17] = 61;
87 }
88 v12 = v18 + k;
89 }
90 result = realloc(v11, v12 + 1);
91 result[v12] = 0;
92 return result;

```

它认为这是一个 base64 编码函数，并认为 base64_encode 会是 sub_74918 好的标识名。



毫不夸张的说，chatgpt 比很多逆向分析人员对我们所处理的数据、汇编、代码都更熟悉。因此出现了不少 IDA 插件（比如 [Gepetto](https://github.com/JusticeRage/Gepetto) <<https://github.com/JusticeRage/Gepetto>>），这些插件的逻辑极其简单，就是像 chatgpt 提问再把回答注释在函数上。

```

int __cdecl sub_10001000(int a1, int a2, char *Str)
{
    int result; // eax
    int v4; // esi
    int i; // edx

    result = strlen(Str);
    v4 = 0;
    for ( i = 0; v4 < a2; ++v4 )
    {
        *(_BYTE *) (v4 + a1) ^= Str[i];
        if ( ++i >= result )
            i -= result;
    }
    return result;
}

```

➡

```

//
// This function is performing a bitwise XOR operation on the bytes of a character
// array starting at position "a1" and doing it for "a2" number of bytes. It is
// using the characters in Str to perform the XOR operations. A better name for
// this function could be 'xorCharacterArrayBytes()'.
int __cdecl sub_10001000(int startPos, int numBytes, char *stringToXorWith)
{
    int result; // eax
    int v4; // esi
    int i; // edx

    result = strlen(stringToXorWith);
    v4 = 0;
    for ( i = 0; v4 < numBytes; ++v4 )
    {
        *(_BYTE *) (v4 + startPos) ^= stringToXorWith[i];
        if ( ++i >= result )
            i -= result;
    }
    return result;
}

```

效果确实不错，读者可以体验一下这类插件，感受逆向工程领域的智能危机。

三、思路二

我们也可以直接大胆猜测，比如猜测这段数据来自于 BASE64。

Download CyberChef

Last build: A month ago

Options About / Support

Operations

to hexd

To Hexdump

Favourites

Data format

Encryption / Encoding

Public Key

Arithmetic / Logic

Networking

Language

Utils

Date / Time

Extractors

Compression

Hashing

Code tidy

Forensics

Multimedia

Other

Flow control

Recipe

From Hex

Delimiter

Auto

From Base64

Alphabet

A-Za-z0-9+/=

☒ Remove non-alphabet chars ☐ Strict mode

To Hexdump

Width

16

☐ Upper case hex ☐ Include final length

☐ UNIX format

STEP

BAKE!

Auto Bake

Input

start: 1024
end: 1024
length: 0
lines: 1

6749785036742f314d5a536f484b42555356597070582f6148357349704978653953516b58527430625954627364325a70695368506f3
845774d5a6c33524f7758646b4966562b3430416a7873417032364c4773654c68596442586576676c6d434c647a753637725063315450
344758343030326f683930642b564e4c30677a42555830553663444f6966723233416b5168792b2f386865307279454c7667754667724
974764e396a4d4d4f3633556b455036706f6b4e326675624274536134412f4678696568596d6c6630694d535141564c4b437750696450
727870596835364e614275444c2f6976454f5570626d2b49496c334732626c482b693377674d46585554423253762b2b48344d4d5a6e5
2635745724c756347394f4f377633566767683551722f5234397751394d78577861415376784e46756377594d7a38586a636c613866
6f6679303070542b6f7245386c696d42683333712f69485a566547476439774450596e45413553949656765324d304a655962522f4
67568494d794773645546763574782f6657355a3336f5736465a2b64476e64322f3250474137385272516a796672383573664c365659
4a5448453675784138714e3250326b52333845684a563957694a72342b747064366f4e454676474753675527632354338505372683
449524e5148434f6a674a4b6a66366159526f543769

Output

time: 18ms
length: 1871
lines: 24

00000000 00 8c 4f ea df f5 31 94 a8 1c a0 54 49 56 29 a5 |..00001.. TIV)W|
00000010 7f da 1f 9b 08 a4 8c 5e f5 24 24 5d 1b 74 6d 84 |.U...0000\$.tm|
00000020 db b1 dd 99 a6 24 a1 3e 8f 04 c0 c6 65 dd 13 b0 |00v.;\$>..AdeV.*|
00000030 5d d9 08 7d 5f b8 00 08 f1 b0 0a 76 e8 b1 ac 78 |JU.]_D.R".v&~x|
00000040 b8 58 74 15 de be 09 66 08 b7 73 bb ae eb 3d cd |.Xt.BK.f..s="a=I|
00000050 53 3f 81 97 e3 4d 36 a2 1f 74 77 e5 4d 2f 48 33 |S?..&M6d.tw&U/H3|
00000060 05 45 f4 53 a7 03 3a 27 eb db 70 24 42 1c be ff |.E00\$.:&0p\$B..y|
00000070 c8 5e d2 bc 84 2e f8 2e 16 0a c8 b6 f3 7d 8c c3 |E^0X..0...E50).A|
00000080 0e eb 75 24 10 fe a9 a2 43 76 7e e6 c1 b5 26 b8 |.eU.S.p0eCv-m&u&.|
00000090 03 f1 71 89 e8 58 9a 57 f4 88 c4 90 01 52 ca 0b |.hQ..eX..h0..A..RE.|
000000a0 03 e2 74 fa f1 a5 88 79 e8 d6 81 b8 32 ff 8a f1 |.AtuNk..y&0..2y.R|
000000b0 0e 52 96 e6 f8 82 25 dc 6d 9b 94 7f a2 df 08 0c |.R..eo..Uu...eR..|
000000c0 15 75 13 07 64 af fb e1 f8 30 c6 67 45 c5 84 ac |.u..d'0a00&gE&..|
000000d0 b8 9c 1b d3 8e ee fd f9 9a 08 21 e5 0a ff 47 8f |..0..iJu...l&..y0.|
000000e0 70 43 d3 31 5b 16 80 4a fc 4d 16 e7 30 60 cc fc |pC0\$[...JUM.c0'iu|
000000f0 5e 37 25 6b c7 e8 7f 2d 34 a5 3f a8 ac 4f 25 8a |^7X&C&..&X?~&0..|
00000100 60 61 df 7a bf 88 76 55 78 61 9d f7 1c 03 3d 89 |'a0z&..vuxa..+..|
00000110 c4 03 75 3d 21 e8 1e d8 cd 09 79 86 d1 fc 50 a1 |.A..u&e.0f..y..Nu[i|

当然，你也可以猜测这段数据来自 Base32 或其他编码，关键在于你要能验证自己的猜测。比如能在内存里搜到它 Base64 编码前的这 0x180 字节。一般来说，只有长度超过 32 字节，就不太可能在内存里恰好且随机的长成这样。

因为函数在执行过程里，会释放、清理或覆盖内存，因此在单一时机点搜索会有遗漏，需要在执行流里隔一点距离就搜索一次才靠谱，搜索的处理代码如下。



```
30     @Override
31     public void hook(Backend backend, long address, int size, long value, Object user) {
32         for (int i = 0; i < size; i++) {
33             addressUsed.add(address + i);
34         }
35     }
36 }
37
38 @Override
39 public void onAttach(UnHook unHook) {
40
41 }
42
43 @Override
44 public void detach() {
45
46 }
47 }, 1, 0, null);
48
49 emulator.getBackend().hook_add_new(new ReadHook() {
50     @Override
51     public void hook(Backend backend, long address, int size, Object user) {
52         for(int i=0;i<size;i++){
53             addressUsed.add(address+i);
54         }
55     }
56
57     @Override
58     public void onAttach(UnHook unHook) {
59
60 }
61
62     @Override
63     public void detach() {
64
65 }
66 }, 1, 0, null);
67 }
68
69 public void hookBlock(){
70     emulator.getBackend().hook_add_new(new BlockHook() {
71         @Override
72         public void hookBlock(Backend backend, long address, int size, Object user) {
73             count ++;
74             if(!find && (count % interval == 0)){
```

```
105
106
107     public void printAddress(){
108         List<Pointer> list = new ArrayList<>();
109         for (long[] range : activePlace) {
110             Collection<Pointer> pointers = searchMemory(range[0],range[1], data);
111             list.addAll(pointers);
112         }
113         System.out.println("Search data matches " + list.size() + " count");
114         for (Pointer pointer : list) {
115             System.out.println("data address: " + pointer);
116         }
117     }
118
119     private Collection<Pointer> searchMemory(long start, long end, byte[] data) {
120         Backend backend = emulator.getBackend();
121         List<Pointer> pointers = new ArrayList<>();
122         if(end - start >= data.length){
123             for (long i = start, m = end - data.length; i <= m; i++) {
124                 byte[] oneByte = backend.mem_read(i, 1);
125                 if (data[0] != oneByte[0]) {
126                     continue;
127                 }
128
129                 if (Arrays.equals(data, backend.mem_read(i, data.length))) {
130                     pointers.add(UnidbgPointer.pointer(emulator, i));
131                     i += (data.length - 1);
132                 }
133             }
134         }
135         return pointers;
136     }
137
138     public boolean AcontainB(byte[] a, byte[] b){
139         if(a.length < b.length){
140             return false;
141         }else {
142             int p1 = 0;
143             int p2 = 0;
144             while (p1 < a.length){
145                 if(a[p1] == b[p2]){
146                     p2++;
147                     if(p2 == b.length){
148                         return true;
149                     }
150                 }
151                 p1++;
152             }
153         }
154     }
```



```

180
181         while(i<nums.size()){
182             temp=nums.get(i);
183             if(i+1 != nums.size() && temp+1 != nums.get(i+1)){//特殊情况处理, 超出数组长度
184                 if(start == end){
185
186                 }else{
187                     activePlace.add(new long[]{nums.get(start), nums.get(end)});
188                 }
189                 start=i+1;
190                 end= start;
191                 ++i;
192                 if(start == nums.size() -1){//最后一个元素是起始元素直接添加
193                     break;
194                 }
195             }else{
196                 if(i+1 == nums.size() ){//后面没有元素了,
197                     if(start == end){
198
199                     }else{
200                         activePlace.add(new long[]{nums.get(start), nums.get(end)});
201                     }
202                     break;
203                 }
204                 ++i;
205                 end = i;
206             }
207         }
208     }
209 }
210
211 public static byte[] hexStringToByteArray(String s) {
212     int len = s.length();
213     byte[] data = new byte[len / 2];
214     for (int i = 0; i < len; i += 2) {
215         data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
216             + Character.digit(s.charAt(i+1), 16));
217     }
218     return data;
219 }
220
221
222 }

```

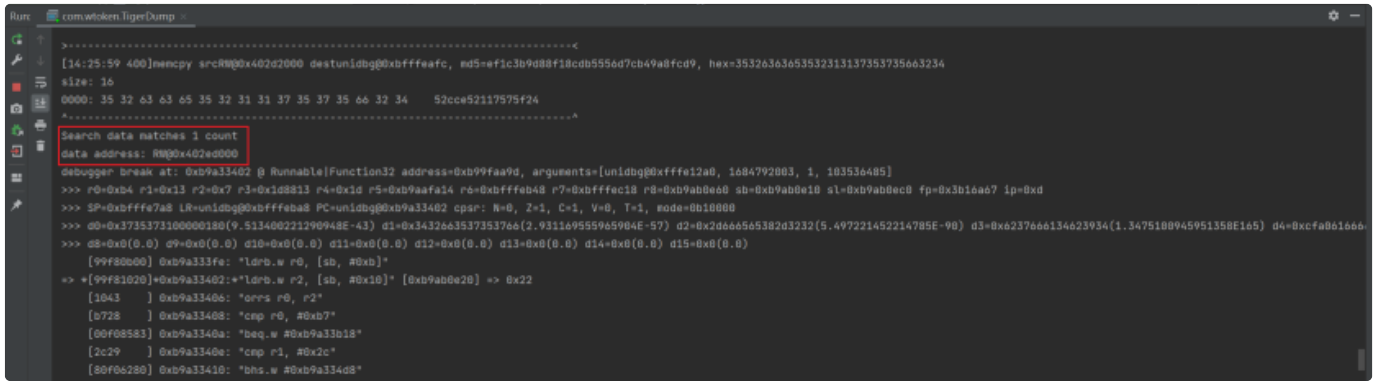
搜索那 0x180 字节的十六进制形式

```

1 private void callTarget() {
2     new DataSearch(emulator, "808c4feadff53194a81ca054495629a57fda1f9b08a48c5ef524245d1b746d84dbb1dd99a624a13e8f04c0c665dd1
3     List<Object> list = new ArrayList<>(10);
4     list.add(vm.getJNIEnv());
5     DvmObject<?> thiz = vm.resolveClass("com/aliyun/TigerTally/TigerTallyAPI").newObject(null);
6     list.add(vm.addLocalObject(thiz));
7     list.add(1);
8     ByteArray barr = new ByteArray(vm, "da965a94-97da-4730-b7d3-3d16e4061489".getBytes(StandardCharsets.UTF_8));
9     list.add(vm.addLocalObject(barr));
10    // 开始模拟执行
11    Number result = Module.emulateFunction(emulator, moduleBase + offset + 1, list.toArray());
12    String ret = vm.getObject(result.intValue()).getValue().toString();
13    System.out.println("result:"+ret);
14    Inspector.inspect(ret.getBytes(), "ret");
15 }

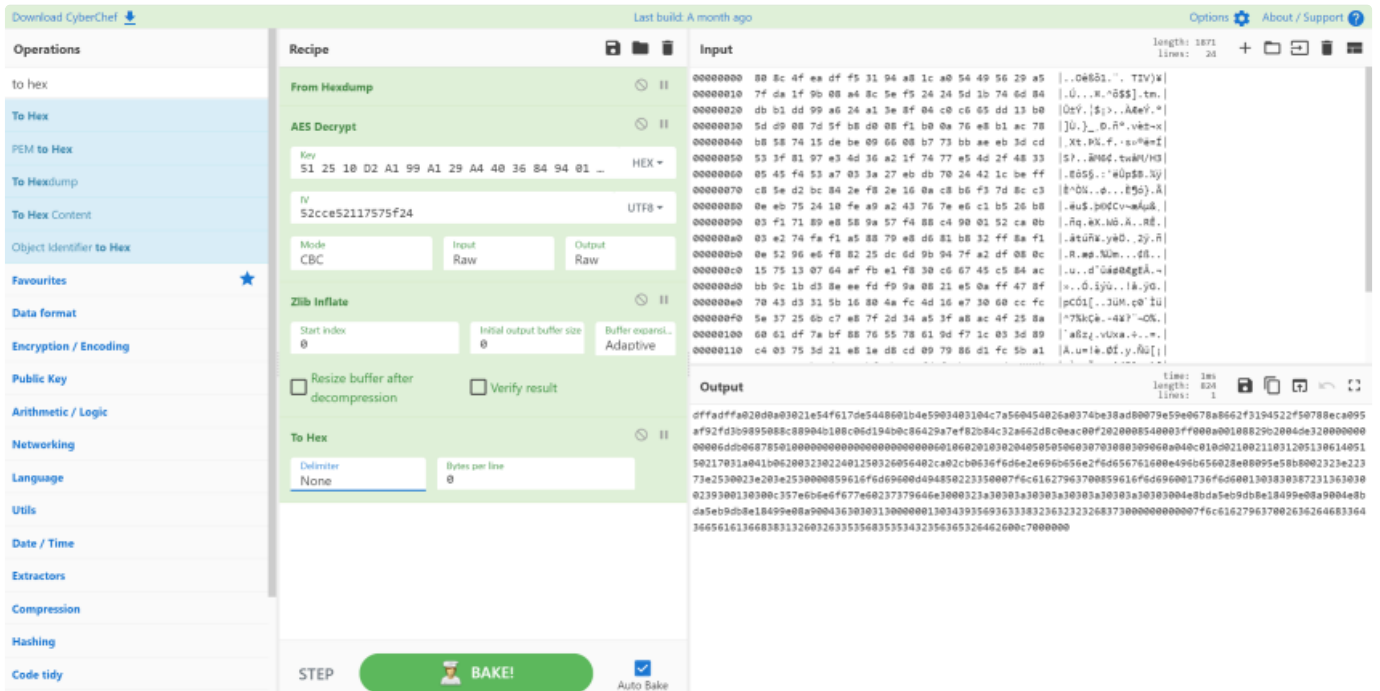
```

运行一两分钟后, 发现程序断下来了, 确实能在内存里找到, 位置是 0x402ed000, 这证明我们的想法没问题。



上篇最后，我们认为数据来自 zlib 压缩，通过 [cyberchef](#)

```
<https://gchq.github.io/CyberChef/#recipe=From_Hexdump(0AES_Decrypt(%7B'option':'Hex','string':'%1%2025%2010%202D%20A1%2029%20A1%2029%20A4%2040%2036%2084%2094%2001%2048%2096'%7D,%7B'option':'UTF8','string':'52cce52117575f24'%7D,'CBC','Raw','Raw',%7B'option':'Hex','string':'%7D,%7B'option':'Hex','string':'%7D)Zlib_Inflate(0,0,'Adaptive',false,false)To_Hex('None',0)&input=MDAwMDAwMDAgIDgwIDhjIDRmIGVhIGRmIGY1IDMxIDk0IGE4IDFjIGEWIDU0IDQ5IDU2IDi5IGE1ICB8Li5P6t/1MS6oLqBUSVYppXwKMDAwMDAwMTAgIDdmlGRhIDFmIDliIDA4IGE0IDhjIDVlIGY1IDi0IDi0IDVkiDFilDc0IDZkIDg0ICB8LtuLi6kLi71JCRdLnRtLnwKMDAwMDAwMjAgIGRiIGlxIGRkiDk5lGE2IDi0IGExIDNlIDhmIDAOIGMwiGM2IDY1IGRkiDEziGIwCB827HdLqYkoT4uLsDGZd0usHwKMDAwMDAwMzAgIDVkiGQ5IDA4IDdkiDVmIGl4IGQwIDA4IGYxIGlwIDBhIDc2IGU4IGlxIGFjIDc4ICB8XdkufV%2B40C7xsC526LGseHwKMDAwMDAwNDAgIGI4IDU4IDc0IDE1IGRIIGJIIDA5IDY2IDA4IGI3IDczIGJiIGFIIGVilIDNkiGNkICB8uFh0Lt6%2BLmYut3O7rus9zXwKMDAwMDAwNTAgIDUziDNmIDgxIDk3IGUziDRkiDM2IGEyiDFmIDc0IDc3IGU1IDRkiIDjmlDQ4IDMziCB8Uz8uLuNNNqludHflTS9IM3wKMDAwMDAwNjAgIDA1IDQ1IGY0IDUziGE3IDaziDNhIDi3IGVlIGRiiDcwiDI0lDQylIDFjIGJlIGZmICB8LkX0U6cuOifr23AkQi6%2B/3wKMDAwMDAwNzAgIGM4IDVlIGQyIGJjIDg0IDJlIGY4IDJlIDE2IDBhIGM4IGI2IGYziDdkIDhjIGMziCB8yF7SvC4u%2BC4uLsi2830uw3wKMDAwMDAwODAgIDBIIGVilIDc1IDi0IDewIGZlIGe5IGEyiDQzIDc2IDdIlGU2IGMxiGI1IDi2IGI4ICB8LuT1JC7%2BqaJDdn7mwbUmuHwKMDAwMDAwOTAgIDaziGYxiDcxIDg5IGU4IDU4IDhIDU3IGY0IDg4IGM0IDkwiDAXIDUyiGNhIDBiICB8LvFxLuhYlIf0LsQuLiLLKLnwKMDAwMDAwYTAgiDaziGUyiDc0IGZhiGYxiGE1IDg4IDc5IGU4IGQ2IDgxIGI4IDMyIGZmiDhhiGYxiCB8LuJ0%2BvGILnno1i64Mv8u8XwKMDAwMDAwYjAgIDBIIDUyiDk2IGU2IGY4IDgyIDi1IGRjiDZkiDliIDk0IDdmiGEyiGRmiDA4IDBjICB8Llu5vguJdxLi4uot8uLnwKMDAwMDAwYzAgIDE1IDc1IDEziIDA3IDY0IGFmiGZiiGUxiGY4IDMwiGM2IDY3IDQ1IGM1IDg0IGFjICB8LnuUuLmSv%2B%2BH4MMZnRcUurHwKMDAwMDAwZDAgiGJiIDjliIDFiiGQziDhliIGVliGZkiGY5IDhIDA4IDxiGU1IDBhiGZmiDQ3IDhmiCB8uy4u0y7u/fkuLiHlLv9HlnwKMDAwMDAwZTAgiDcwIDQziGQziDMxiDViiDE2IDgwiDRhiGZjiDRkiDE2IGU3IDMwiIDYwlgNjIGZjiCB8cEPTMVsuLkr8TS7nMGDM/HwKMDAwMDAwZjAgIDVliIDM3IDi1IDZiiGM3IGU4IDdmiDjkiDM0IGE1IDNmiGE4IGFjIDRmiDI1IDhhiCB8Xjcla8foLi00pT%2BorE8LlnwKMDAwMDAxMDAgIDYwiDYxiGRmiIDdhiGJmIDg4IDc2IDU1IDc4IDYxiIDkiGY3IDFjiDaziDNkiDg5ICB8YGHfer8udIV4YS73Li49LnwKMDAwMDAxMTAgIGM0IDaziDc1IDNkiIDxiGU4IDFliGQ4IGNkiDA5IDc5IDg2IGQxiGZjiDViiGExICB8xC51PSHoLtiJNLnu0fxboXwKMDAwMDAxMjAgIDlwiGNjiDg2IGlxIGQ1IDA1IGJmiDliIDY5IGZkiGY1IGI5IDY3IDikiGU4IDViiCB8IMwusDUuy5p/fW5Zy7oW3wKMDAwMDAxMzAgIGExIDU5IGY5IGQxiGE3IDc3IDZmiGY2IDNjiDYwiIDNiGYxiIDFhiGQwiDhmiDI3ICB8oVn50ad3b/Y8YDvxLtAuJ3wKMDAwMDAxNDAgIGViiGYziDliiDFmIDjmiGE1IDU4IDi1IDMxiGM0IGVhiGVjiDQwiGYyiGEziDc2ICB86/MuLi%2BliWCuxxOrsQPKjdnwKMDAwMDAxNTAgIDNmiIDY5IDExIGRmiIGMxiDxiDI1IDVmiIDU2IDg4IDhliGY4IGZhiGRhiDVkiGVhiCB8P2ku38EHJV9WLi74%2Btpd6nwKMDAwMDAxNjAgIDgziDUziDgyiDdiIDYwiIDNhiGVliIGFiIDQ2IGZkiGI5IDBiIGMziGQyiIGFiIDFiICB8LiMue2A67q5G/bkuw9KuLnwKMDAwMDAxNzAgIDA4IDQ0IGQ0IDA3IDA4IGU4IGUwiIDi0IGE4IGRmiIGU5IGE2IDExIGExIDNiIGUyiCB8LkTULi7o4CS03%2BmmLqE%2B4nw> 获取到压缩前是 dffaxxxx 一长串。
```



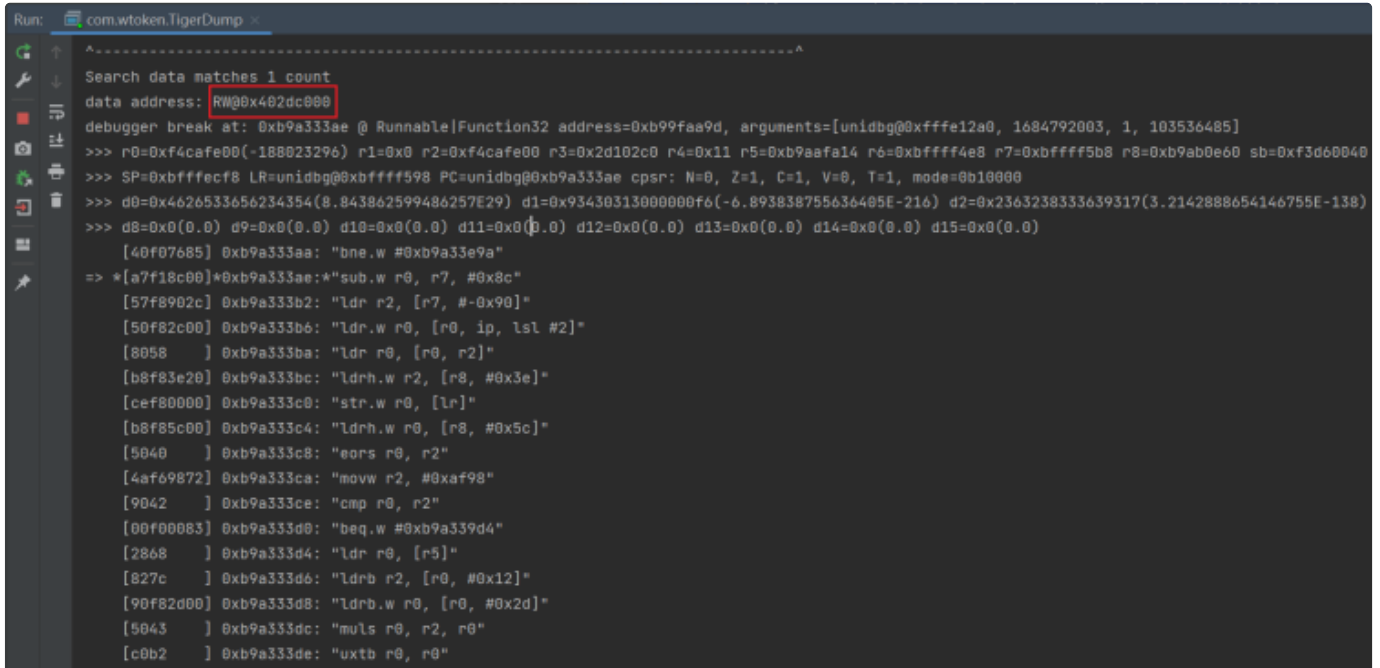
同样可以把这一堆放 dataSearch 里，有的话就说明猜想正确。

```

1 private void callTarget() {
2     new DataSearch(emulator, "dffadffa020d0a03021e54f617de5448601b4e5903403104c7a560454026a0374be38ad80079e59e0678a8662f319
3     List<Object> list = new ArrayList<>(10);
4     list.add(vm.getJNIEnv());
5     DvmObject<> thiz = vm.resolveClass("com/aliyun/TigerTally/TigerTallyAPI").newObject(null);
6     list.add(vm.addLocalObject(thiz));
7     list.add(1);
8     ByteArray barr = new ByteArray(vm, "da965a94-97da-4730-b7d3-3d16e4061489".getBytes(StandardCharsets.UTF_8));
9     list.add(vm.addLocalObject(barr));
10    // 开始模拟执行
11    Number result = Module.emulateFunction(emulator, moduleBase + offset + 1, list.toArray());
12    String ret = vm.getObject(result.intValue()).getValue().toString();
13    System.out.println("result:"+ret);
14    Inspector.inspect(ret.getBytes(), "ret");
15 }

```

运行后很快断下来，内存验证了我们的猜想，而且它的位置在 0x402dc000。



Run: com.wtoken.TigerDump

Search data matches 1 count

data address: RW@0x402dc000

debugger break at: 0xb9a333ae @ Runnable[Function32 address=0xb99faa9d, arguments=[unidbg@0xfffe12a0, 1684792003, 1, 103536485]

>>> r0=0xf4cafe00(-188023296) r1=0x0 r2=0xf4cafe00 r3=0x2d102c0 r4=0x11 r5=0xb9aafa14 r6=0xbffff4e8 r7=0xbffff5b8 r8=0xb9ab0e60 sb=0xf3d60040

>>> SP=0xbffffecf8 LR=unidbg@0xbffff598 PC=unidbg@0xb9a333ae cpsr: N=0, Z=1, C=1, V=0, T=1, mode=0b10000

>>> d0=0x4626533656234354(0.843862599486257E29) d1=0x93430313000000f6(-6.893838755636405E-216) d2=0x2363238333639317(3.2142888654146755E-138)

>>> d8=0x0(0.0) d9=0x0(0.0) d10=0x0(0.0) d11=0x0(0.0) d12=0x0(0.0) d13=0x0(0.0) d14=0x0(0.0) d15=0x0(0.0)

[40f07685] 0xb9a333aa: "bne.w #0xb9a33e9a"

=> *a7f18c00*0xb9a333ae:*"sub.w r0, r7, #0x8c"

[57f8902c] 0xb9a333b2: "ldr r2, [r7, #-0x90]"

[50f82c00] 0xb9a333b6: "ldr.w r0, [r0, ip, lsl #2]"

[8058] 0xb9a333ba: "ldr r0, [r0, r2]"

[b8f83e20] 0xb9a333bc: "ldrh.w r2, [r0, #0x3e]"

[cef80000] 0xb9a333c0: "str.w r0, [lr]"

[b8f85c00] 0xb9a333c4: "ldrh.w r0, [r0, #0x5c]"

[9040] 0xb9a333c8: "eor r0, r2"

[4af69872] 0xb9a333ca: "movw r2, #0xaf98"

[9042] 0xb9a333ce: "cmp r0, r2"

[00f00083] 0xb9a333d0: "beq.w #0xb9a339d4"

[2868] 0xb9a333d4: "ldr r0, [r5]"

[827c] 0xb9a333d6: "ldrb r2, [r0, #0x12]"

[90f82d00] 0xb9a333d8: "ldrb.w r0, [r0, #0x2d]"

[9043] 0xb9a333dc: "muls r0, r2, r0"

[c0b2] 0xb9a333de: "uxtb r0, r0"

四、尾声

下篇我们进一步分析这个数据的来源。