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。

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

修改 patch 函数里对 memset 的处理

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

优化 patch 函数里对 memcpy 的打印

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

优化 patch 函数里对 memmove 的打印

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

优化 patch 函数里对 strlen 的打印

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

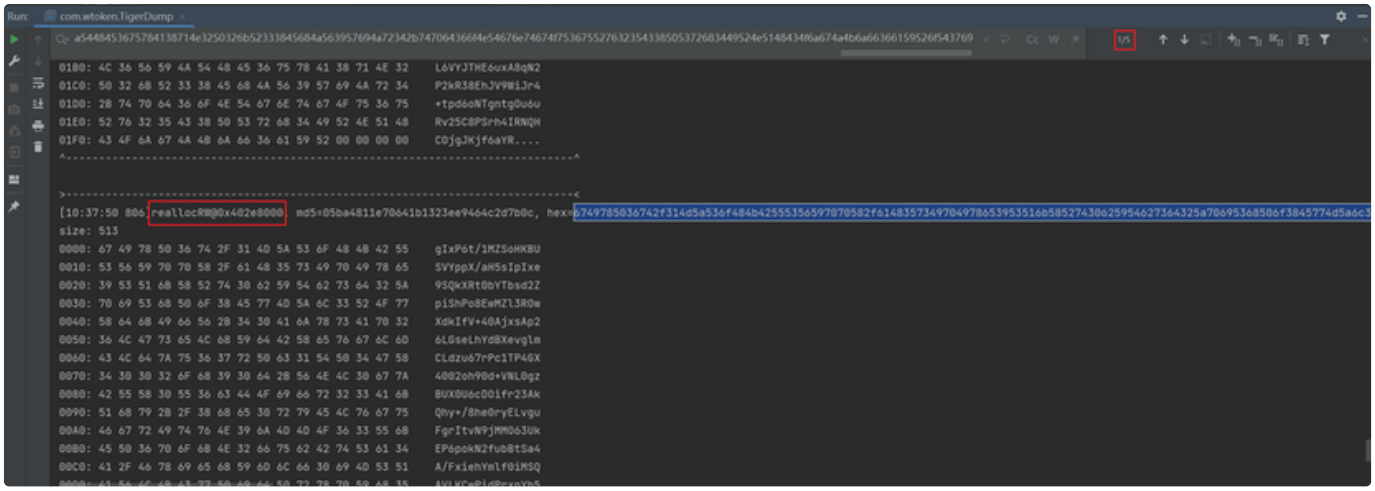
优化 patch 函数里对 realloc 的打印

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

优化 patch 函数里对 sprintf 的打印

```
emulator.attach().addBreakPoint(0xb99e73b8L, new BreakPointCallback() {
    @Override
    public boolean onHit(Emulator<?> emulator, long address) {
        RegisterContext registerContext = emulator.getContext();
        final UnidbgPointer str = registerContext.getPointerArg(0);
        emulator.attach().addBreakPoint(registerContext.getLR(), new BreakPointCallback() {
            @Override
            public boolean onHit(Emulator<?> emulator, long address) {
                System.out.println("sprintf result:"+str.getString(0));
                return true;
            }
        });
        emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC, module.findSymbolByName("sprintf").getAddress());
        return true;
    }
});
```

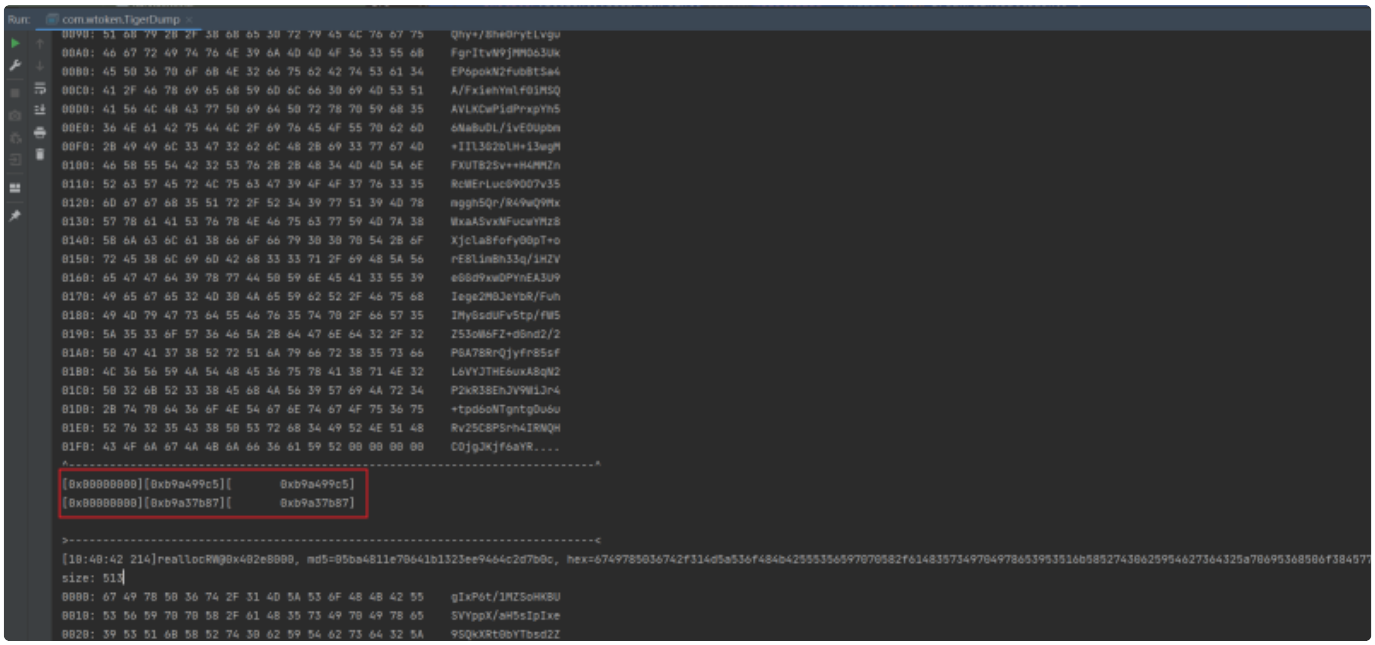
运行代码，上篇我们分析的那一长串疑似 base64 的内容，最早出现位置是下图日志处。



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。

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

运行



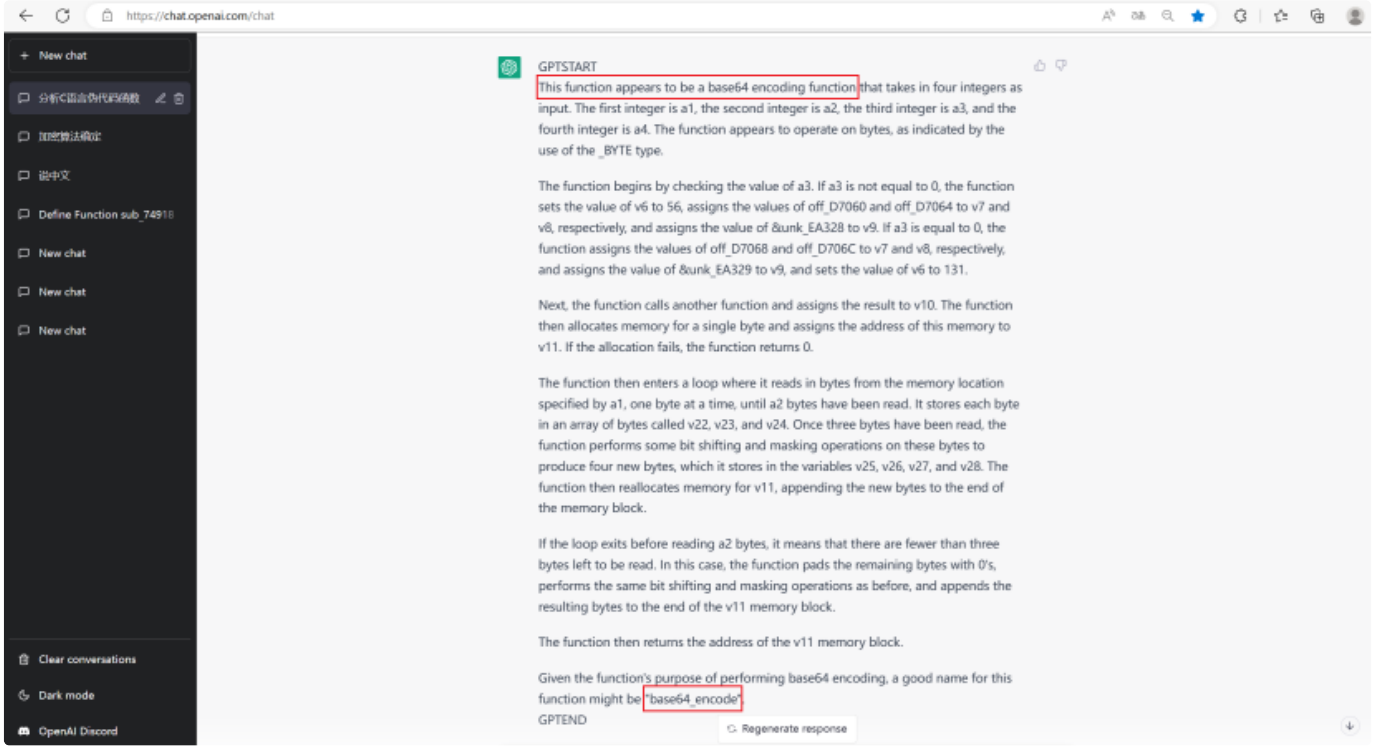
0xb9a499c5 - 0xb99d5000 = 0x749c5，IDA 里跳过去，来到 sub_74918。它就是上篇我们说的 Base64 实现函数。

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

```
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
_BYTE *__fastcall sub_74918(int a1, int a2, int a3, int a4)
{
  int v6; // r2
  void *v7; // r0
  void *v8; // r1
  void *v9; // r3
  int v10; // r10
  char *v11; // r0
  int v12; // r8
  unsigned int v13; // r5
  int i; // r1
  unsigned int j; // r2
  _BYTE *result; // r0
  int v17; // r4
  int v18; // r9
  int k; // r6
  char *v20; // r1
  unsigned __int8 v22; // [sp+9h] [bp-27h]
  unsigned __int8 v23; // [sp+Ah] [bp-26h]
  unsigned __int8 v24; // [sp+Bh] [bp-25h]
  char v25; // [sp+Ch] [bp-24h]
  char v26; // [sp+Dh] [bp-23h]
  char v27; // [sp+Eh] [bp-22h]
  char v28; // [sp+Fh] [bp-21h]

  if ( a3 )
  {
    v6 = 56;
    v7 = off_D7060;
    v8 = off_D7064;
    v9 = &unk_EA328;
  }
  else
  {
    v7 = off_D7068;
    v8 = off_D706C;
    v9 = &unk_EA329;
    v6 = 131;
  }
  v10 = ((int (__fastcall *)(void *, void *, int, void *, int))loc_74AC4)(v7, v8, v6, v9, a4);
  v11 = (char *)malloc(1u);
  if ( !v11 )
    return 0;
  v12 = 0;
LABEL_6:
  v13 = 0;
  while ( a2 != v13 )
  {
    *(&v22 + v13) = *(_BYTE *)(a1 + v13);
    if ( ++v13 == 3 )
    {
      v25 = v22 >> 2;
      v28 = v24 & 0x3F;
      v26 = (v23 >> 4) | (16 * (v22 & 3));
      v27 = (v24 >> 6) | (4 * (v23 & 0xF));
      v11 = (char *)realloc(v11, v12 + 4);
      a2 -= 3;
      a1 += 3;
      for ( i = 0; i != 4; ++i )
        v11[v12 + i] = *(_BYTE *)(v10 + (unsigned __int8)*(&v25 + i));
      v12 += 4;
      goto LABEL_6;
    }
  }
  if ( v13 )
  {
    for ( j = v13; j <= 2; ++j )
      *(&v22 + j) = 0;
    v25 = v22 >> 2;
    v28 = v24 & 0x3F;
    v26 = (v23 >> 4) | (16 * (v22 & 3));
    v17 = 0;
    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> ），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。

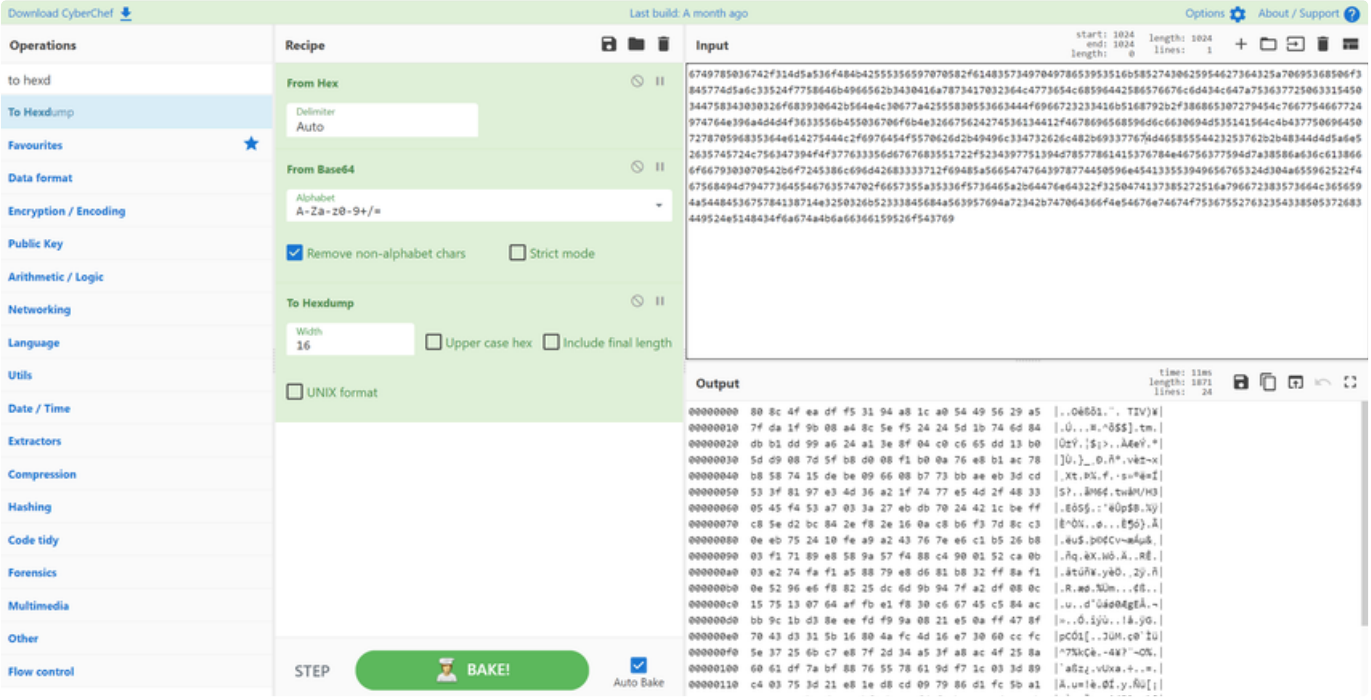
```
int __cdecl sub_10001000(int a1, int a2, char *Str)
{
  int result; // eax
  int v4; // esi
  int i; // edx

  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械危机。

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。



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。

```
            @Override
            public void hook(Backend backend, long address, int size, long value, Object user) {
                for (int i = 0; i < size; i++) {
                    addressUsed.add(address + i);
                }

            }

            @Override
            public void onAttach(UnHook unHook) {

            }

            @Override
            public void detach() {

            }
        }, 1, 0, null);

        emulator.getBackend().hook_add_new(new ReadHook() {
            @Override
            public void hook(Backend backend, long address, int size, Object user) {
                for(int i=0;i<size;i++){
                    addressUsed.add(address+i);
                }
            }

            @Override
            public void onAttach(UnHook unHook) {

            }

            @Override
            public void detach() {

            }
        }, 1, 0,  null);
    }

    public void hookBlock(){
        emulator.getBackend().hook_add_new(new BlockHook() {
            @Override
            public void hookBlock(Backend backend, long address, int size, Object user) {
                count ++;
                if(!find && (count % interval == 0)){
```

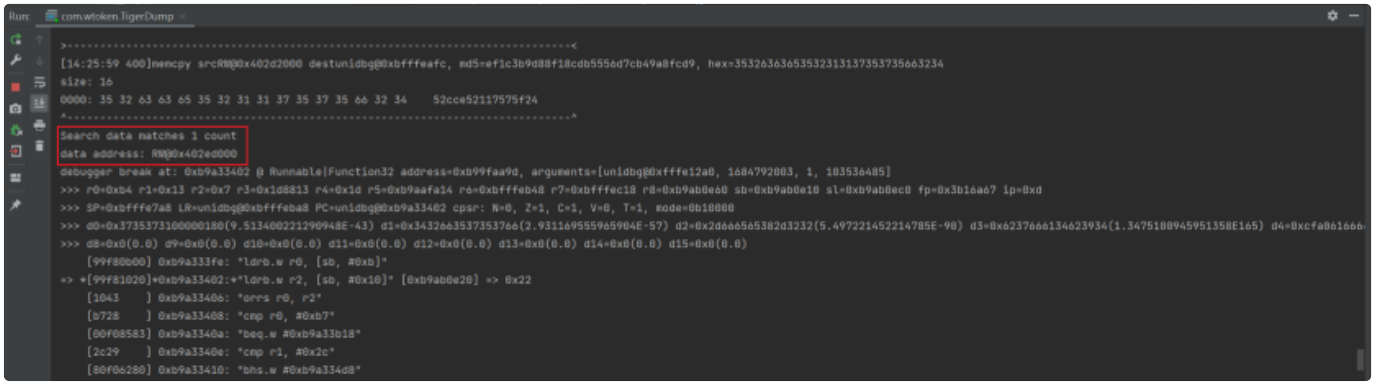
```java


    public void printAddress(){
        List<Pointer> list = new ArrayList<>();
        for (long[] range : activePlace) {
            Collection<Pointer> pointers = searchMemory(range[0],range[1], data);
            list.addAll(pointers);
        }
        System.out.println("Search data matches " + list.size() + " count");
        for (Pointer pointer : list) {
            System.out.println("data address: " + pointer);
        }
    }

    private Collection<Pointer> searchMemory(long start, long end, byte[] data) {
        Backend backend = emulator.getBackend();
        List<Pointer> pointers = new ArrayList<>();
        if(end - start >= data.length){
            for (long i = start, m = end - data.length; i <= m; i++) {
                byte[] oneByte = backend.mem_read(i, 1);
                if (data[0] != oneByte[0]) {
                    continue;
                }

                if (Arrays.equals(data, backend.mem_read(i, data.length))) {
                    pointers.add(UnidbgPointer.pointer(emulator, i));
                    i += (data.length - 1);
                }
            }
        }
        return pointers;
    }

    public boolean AcontainB(byte[] a, byte[] b){
        if(a.length < b.length){
            return false;
        }else {
            int p1 = 0;
            int p2 = 0;
            while (p1 < a.length){
                if(a[p1] == b[p2]){
                    p2++;
                    if(p2 == b.length){
                        return true;
                    }
```

```
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 }
```

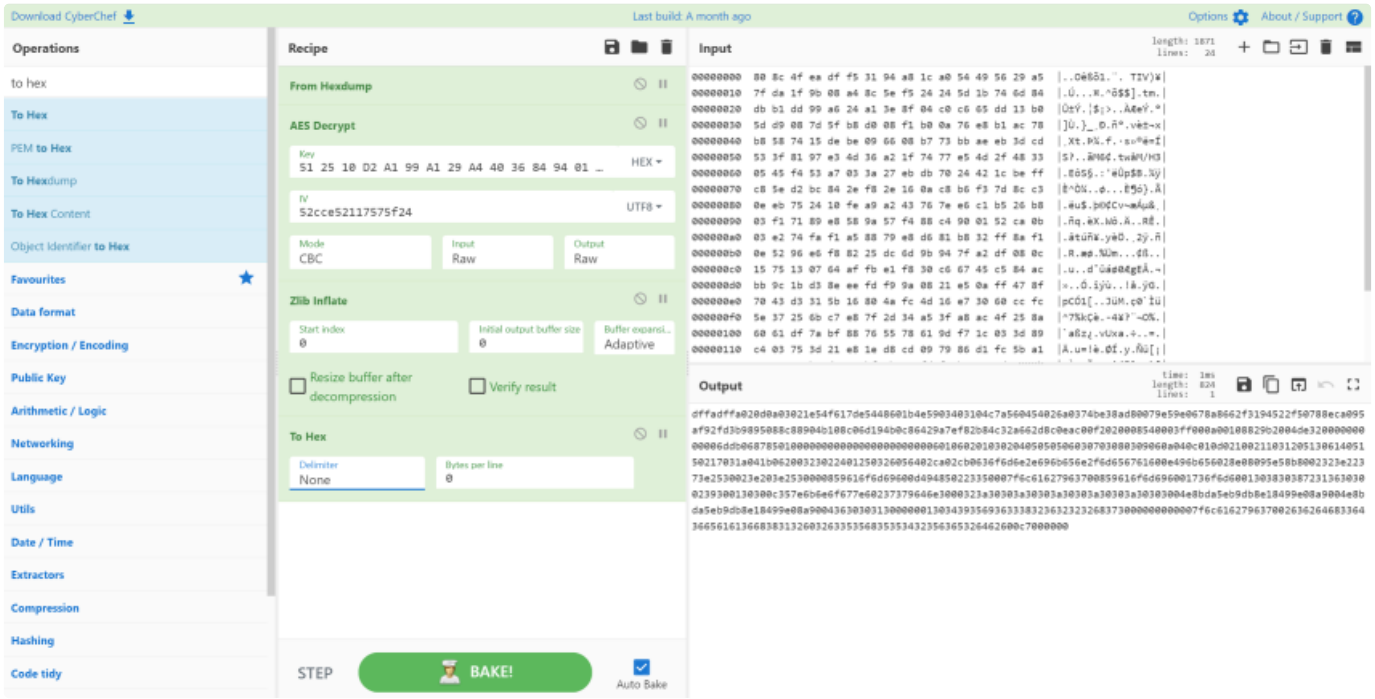
搜索那 0x180 字节的十六进制形式

```
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()AES_Decrypt(%7B'option':'Hex','string':'51%2025%2010%20D2%20A1%2099%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/1MS6oLqBUSVYppXwKMDAwMDAwMTAgIDdmIGRhIDFmIDliIDA4IGE0IDhjIDVlIGY1IDI0IDI0IDVkIDFiIDc0IDZkIDg0ICB8LtouLi6kLl71JCRdLnRtLnwKMDAwMDAwMjAgIGRiIGIxIGRkIDk5IGE2IDI0IGExIDNlIDhmIDA0IGMwIGM2IDY1IGRkIDEzIGIwICB827HdLqYkoT4uLsDGZd0usHwKMDAwMDAwMzAgIDVkIGQ5IDA4IDdkIDVmIGI4IGQwIDA4IGYxIGIwIDBhIDc2IGU4IGIxIGFjIDc4ICB8XdkufV%2B40C7xsC526LGseHwKMDAwMDAwNDAgIGI4IDU4IDc0IDE1IGRlIGJlIDA5IDY2IDA4IGI3IDczIGJiIGFlIGViIDNkIGNkICB8uFh0Lt6%2BLmYut3O7rus9zXwKMDAwMDAwNTAgIDUzIDNmIDgxIDk3IGUzIDRkIDM2IGEyIDFmIDc0IDc3IGU1IDRkIDJmIDQ4IDMzICB8Uz8uLuNNNqIudHflTS9IM3wKMDAwMDAwNjAgIDA1IDQ1IGY0IDUzIGE3IDAzIDNhIDI3IGViIGRiIDcwIDI0IDQyIDFjIGJlIGZmICB8LkX0U6cuOifr23AkQi6%2B/3wKMDAwMDAwNzAgIGM4IDVlIGQyIGJjIDg0IDJlIGY4IDJlIDE2IDBhIGM4IGI2IGYzIDdkIDhjIGMzICB8yF7SvC4u%2BC4uLsi2830uw3wKMDAwMDAwODAgIDBlIGViIDc1IDI0IDEwIGZlIGE5IGEyIDQzIDc2IDdlIGU2IGMxIGI1IDI2IGI4ICB8Lut1JC7%2BqaJDdn7mwbUmuHwKMDAwMDAwOTAgIDAzIGYxIDcxIDg5IGU4IDU4IDlhIDU3IGY0IDg4IGM0IDkwIDAxIDUyIGNhIDBiICB8LvFxLuhYLlf0LsQuLlLKLnwKMDAwMDAwYTAgIDAzIGUyIDc0IGZhIGYxIGE1IDg4IDc5IGU4IGQ2IDgxIGI4IDMyIGZmIDhhIGYxICB8LuJ0%2BvGlLnno1i64Mv8u8XwKMDAwMDAwYjAgIDBlIDUyIDk2IGU2IGY4IDgyIDI1IGRjIDZkIDliIDk0IDdmIGEyIGRmIDA4IDBjICB8Lllu5vguJdxtLi4uot8uLnwKMDAwMDAwYzAgIDE1IDc1IDEzIDA3IDY0IGFmIGZiIGUxIGY4IDMwIGM2IDY3IDQ1IGM1IDg0IGFjICB8LnUuLmSv%2B%2BH4MMZnRcUurHwKMDAwMDAwZDAgIGJiIDljIDFiIGQzIDhlIGVlIGZkIGY5IDlhIDA4IDIxIGU1IDBhIGZmIDQ3IDhmICB8uy4u0y7u/fkuLiHlLv9HLnwKMDAwMDAwZTAgIDcwIDQzIGQzIDMxIDViIDE2IDgwIDRhIGZjIDRkIDE2IGU3IDMwIDYwIGNjIGZjICB8cEPTMVsuLkr8TS7nMGDM/HwKMDAwMDAwZjAgIDVlIDM3IDI1IDZiIGM3IGU4IDdmIDJkIDM0IGE1IDNmIGE4IGFjIDRmIDI1IDhhICB8Xjcla8foLi00pT%2BorE8lLnwKMDAwMDAxMDAgIDYwIDYxIGRmIDdhIGJmIDg4IDc2IDU1IDc4IDYxIDlkIGY3IDFjIDAzIDNkIDg5ICB8YGHfer8udlV4YS73Li49LnwKMDAwMDAxMTAgIGM0IDAzIDc1IDNkIDlxIGU4IDFlIGQ4IGNkIDA5IDc5IDg2IGQxIGZjIDViIGExICB8xC51PSHoLtjNLnku0fxboXwKMDAwMDAxMjAgIDIwIGNjIDg2IGIxIGQ1IDA1IGJmIDliIDY5IGZkIGY1IGI5IDY3IDlkIGU4IDViICB8IMwusdUuvy5p/fW5Zy7oW3wKMDAwMDAxMzAgIGExIDU5IGY5IGQxIGE3IDc3IDZmIGY2IDNjIDYwIDNiIGYxIDFhIGQwIDhmIDI3ICB8oVn50ad3b/Y8YDvxLtAuJ3wKMDAwMDAxNDAgIGViIGYzIDliIDFmIDJmIGE1IDU4IDI1IDMxIGM0IGVhIGVjIDQwIGYyIGEzIDc2ICB86/MuLi%2BlWCUxxOrsQPKjdnwKMDAwMDAxNTAgIDNmIDY5IDExIGRmIGMxIDxIDI1IDVmIDU2IDg4IDlhIGY4IGZhIGRhIDVkIGVhICB8P2ku38EhJV9WLi74%2Btpd6nwKMDAwMDAxNjAgIDgzIDUzIDgyIDdiIDYwIDNhIGVlIGFlIDQ2IGZkIGI5IDBiIGMzIGQyIGFlIDFlICB8LlMue2A67q5G/bkuw9KuLnwKMDAwMDAxNzAgIDA4IDQ0IGQ0IDA3IDA4IGU4IGUwIDI0IGE4IGRmIGU5IGE2IDExIGExIDNlIGUyICB8LkTULi7o4CSo3%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。

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

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=0xbfffecf8 LR=unidbg@0xbffff598 PC=unidbg@0xb9a333ae cpsr: N=0, Z=1, C=1, V=0, T=1, mode=0b10000
>>> d0=0x4626533656234354(8.843862599486257E29) d1=0x9343031300000f6(-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, [r8, #0x3e]"
    [cef80000] 0xb9a333c0: "str.w r0, [lr]"
    [b8f85c00] 0xb9a333c4: "ldrh.w r0, [r8, #0x5c]"
    [5040    ] 0xb9a333c8: "eors 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]"
    [5043    ] 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。