

# dump 法从入门到熟练 (六)

## 一、引言

上篇我们发现，数据长 412 字节，位于从 0x402dc000 开始的内存区域里，本篇做继续分析。

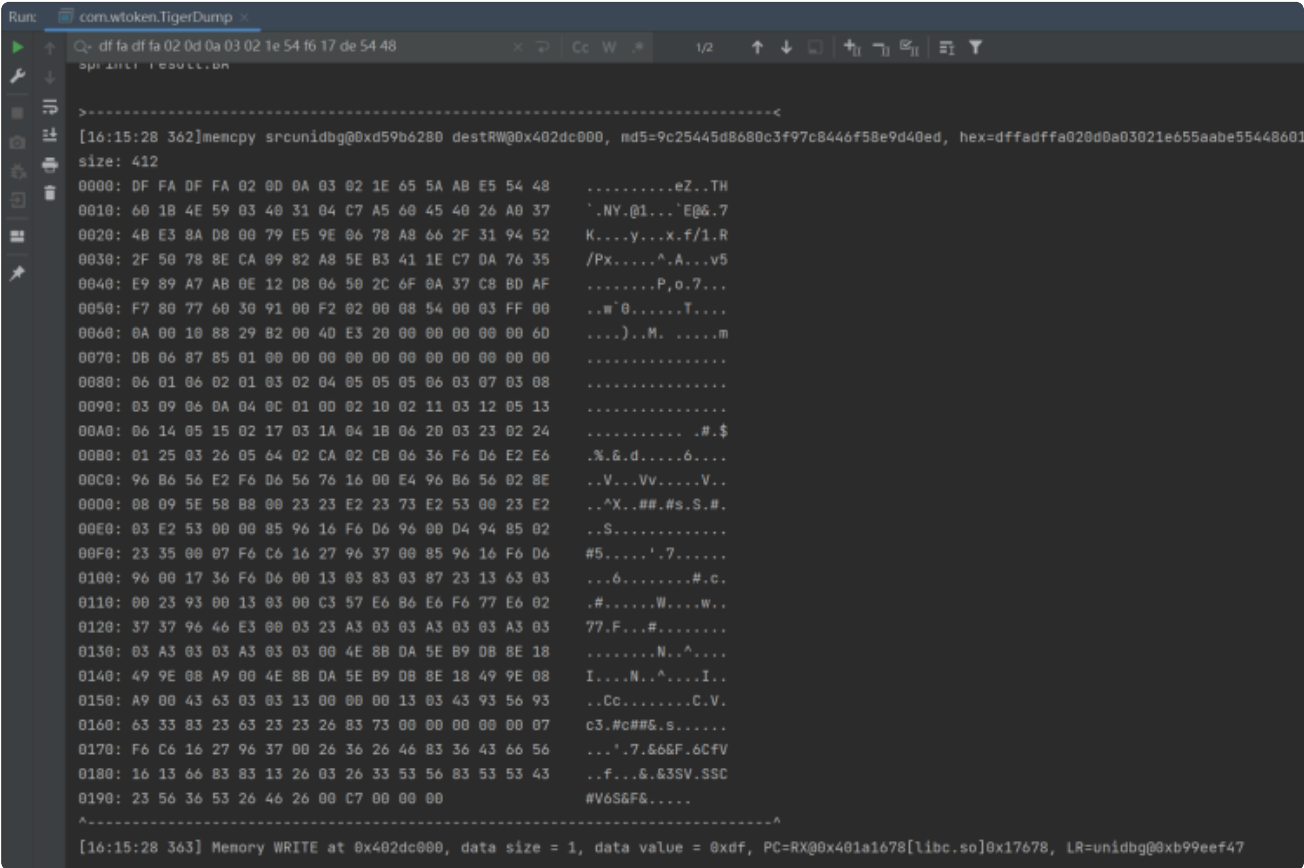
```
1  00000000 df fa df fa 02 0d 0a 03 02 1e 54 f6 17 de 54 48 |ßúßú.....Tö.ÞTH|
2  00000010 60 1b 4e 59 03 40 31 04 c7 a5 60 45 40 26 a0 37 |`.NY.@1.Ç¥`E@& 7|
3  00000020 4b e3 8a d8 00 79 e5 9e 06 78 a8 66 2f 31 94 52 |Kă.Ø.yå..x``f/1.R|
4  00000030 2f 50 78 8e ca 09 5a f9 2f d3 b9 89 50 88 c8 89 |/Px.Ê.Zù/Ó¹.P.È.|
5  00000040 04 b1 08 c0 6d 19 4b 0c 86 42 9a 7e f8 2b 84 c3 |.±.Àm.K..B.~ø+.Ã|
6  00000050 2a 66 2d 8c 0e ac 00 f2 02 00 08 54 00 03 ff 00 |*f-...-.ò...T..ÿ.|
7  00000060 0a 00 10 88 29 b2 00 4d e3 20 00 00 00 00 00 6d |.....)².Mă .....m|
8  00000070 db 06 87 85 01 00 00 00 00 00 00 00 00 00 00 |Û.....|
9  00000080 06 01 06 02 01 03 02 04 05 05 05 06 03 07 03 08 |.....|
10 00000090 03 09 06 0a 04 0c 01 0d 02 10 02 11 03 12 05 13 |.....|
11 000000a0 06 14 05 15 02 17 03 1a 04 1b 06 20 03 23 02 24 |..... .#.$|
12 000000b0 01 25 03 26 05 64 02 ca 02 cb 06 36 f6 d6 e2 e6 |.%.&.d.Ê.Ë.6öÖâæ|
13 000000c0 96 b6 56 e2 f6 d6 56 76 16 00 e4 96 b6 56 02 8e |.ŶVâöÖVv..ä.ŶV..|
14 000000d0 08 09 5e 58 b8 00 23 23 e2 23 73 e2 53 00 23 e2 |..^X,.##â#sâS.#â|
15 000000e0 03 e2 53 00 00 85 96 16 f6 d6 96 00 d4 94 85 02 |.âS.....öÖ..ô...|
16 000000f0 23 35 00 07 f6 c6 16 27 96 37 00 85 96 16 f6 d6 |#5..öÆ.'.7....öÖ|
17 00000100 96 00 17 36 f6 d6 00 13 03 83 03 87 23 13 63 03 |...6öÖ.....#.c.|
18 00000110 00 23 93 00 13 03 00 c3 57 e6 b6 e6 f6 77 e6 02 |.#.....ÃWæŶæöwæ.|
19 00000120 37 37 96 46 e3 00 03 23 a3 03 03 a3 03 03 a3 03 |77.Fä..#£..£..£.|
20 00000130 03 a3 03 03 a3 03 03 00 4e 8b da 5e b9 db 8e 18 |.£..£...N.Ú¹¹Û..|
21 00000140 49 9e 08 a9 00 4e 8b da 5e b9 db 8e 18 49 9e 08 |I..@.N.Ú¹¹Û..I..|
22 00000150 a9 00 43 63 03 03 13 00 00 00 13 03 43 93 56 93 |@.Cc.....C.V.|
23 00000160 63 33 83 23 63 23 23 26 83 73 00 00 00 00 07 |c3.#c##&.s.....|
24 00000170 f6 c6 16 27 96 37 00 26 36 26 46 83 36 43 66 56 |öÆ.'.7.&6F.6CfV|
25 00000180 16 13 66 83 83 13 26 03 26 33 53 56 83 53 53 43 |..f...&.&3SV.SSC|
26 00000190 23 56 36 53 26 46 26 00 c7 00 00 00 |#V6S&F&Ç...|
```

## 二、算法分析

这个数据看不出来什么门道，先对第一个字节做一下 traceWrite。

```
1  emulator.traceWrite(0x402dc000,0x402dc000);
```

运行



```
Run: com.wtoken.TigerDump
[16:15:28 362] memcpy src=unidbg@0xd59b6280 dest=RW@0x402dc000, md5=9c25445d8680c3f97c8446f58e9d40ed, hex=dfdfa020d0a03021e655aabe55448601
size: 412
0000: DF FA DF FA 02 0D 0A 03 02 1E 65 5A AB E5 54 48 .....eZ..TH
0010: 60 18 4E 59 03 40 31 04 C7 A5 60 45 40 26 A0 37 ..NY.@1...`E@G.7
0020: 4B E3 8A D8 00 79 E5 9E 06 78 A8 66 2F 31 94 52 K...y...x.f/1.R
0030: 2F 50 78 8E CA 09 82 A8 5E B3 41 1E C7 DA 76 35 /Px.....^..A...v5
0040: E9 89 A7 AB 0E 12 D8 06 50 2C 6F 0A 37 C8 BD AF .....P,o.7...
0050: F7 80 77 60 30 91 00 F2 02 00 08 54 00 03 FF 00 ..m`0.....T....
0060: 0A 00 10 88 29 82 00 4D E3 20 00 00 00 00 00 6D .....).M. ....m
0070: DB 06 87 85 01 00 00 00 00 00 00 00 00 00 00 .....
0080: 06 01 06 02 01 03 02 04 05 05 05 06 03 07 03 08 .....
0090: 03 09 06 0A 04 0C 01 00 02 10 02 11 03 12 05 13 .....
00A0: 06 14 05 15 02 17 03 1A 04 18 06 20 03 23 02 24 ..... .#.$
00B0: 01 25 03 26 05 64 02 CA 02 CB 06 36 F6 06 E2 E6 ..%.G.d....6...
00C0: 96 B6 56 E2 F6 06 56 76 16 00 E4 96 B6 56 02 8E ..V...Vv....V..
00D0: 08 09 5E 58 08 00 23 23 E2 23 73 E2 53 00 23 E2 ..^X...##.#s.#.#
00E0: 03 E2 53 00 00 85 96 16 F6 D6 96 00 04 94 85 02 ..S.....
00F0: 23 35 00 07 F6 C6 16 27 96 37 00 85 96 16 F6 D6 #5.....'.7.....
0100: 96 00 17 36 F6 D6 00 13 03 83 03 87 23 13 63 03 ...6.....#.C.
0110: 00 23 93 00 13 03 00 C3 57 E6 B6 E6 F6 77 E6 02 ..#.....W...w...
0120: 37 37 96 46 E3 00 03 23 A3 03 03 A3 03 03 A3 03 77.F...#.....
0130: 03 A3 03 03 A3 03 03 00 4E 8B DA 5E B9 DB 8E 18 .....N..^....
0140: 49 9E 08 A9 00 4E 8B DA 5E B9 DB 8E 18 49 9E 08 I...N..^....I..
0150: A9 00 43 63 03 03 13 00 00 00 13 03 43 93 56 93 ..Cc.....C.V.
0160: 63 33 83 23 63 23 23 26 83 73 00 00 00 00 07 c3.#c##G.s.....
0170: F6 C6 16 27 96 37 00 26 36 26 46 83 36 43 66 56 ...'.7.&6F.6CFV
0180: 16 13 66 83 83 13 26 03 26 33 53 56 83 53 43 ..f...G.&3SV.SSC
0190: 23 56 36 53 26 46 26 00 C7 00 00 00 #V6S&F&.....
[16:15:28 363] Memory WRITE at 0x402dc000, data size = 1, data value = 0xdf, PC=RX@0x401a1678[libc.so]0x17678, LR=unidbg@0xb99eef47
```

发现来此 memcpy，而且我们发现，memcpy 的这份数据，和我们的目标数据很像，以 DFFA 开头，但似乎又不完全相同，眼睛都看花了。

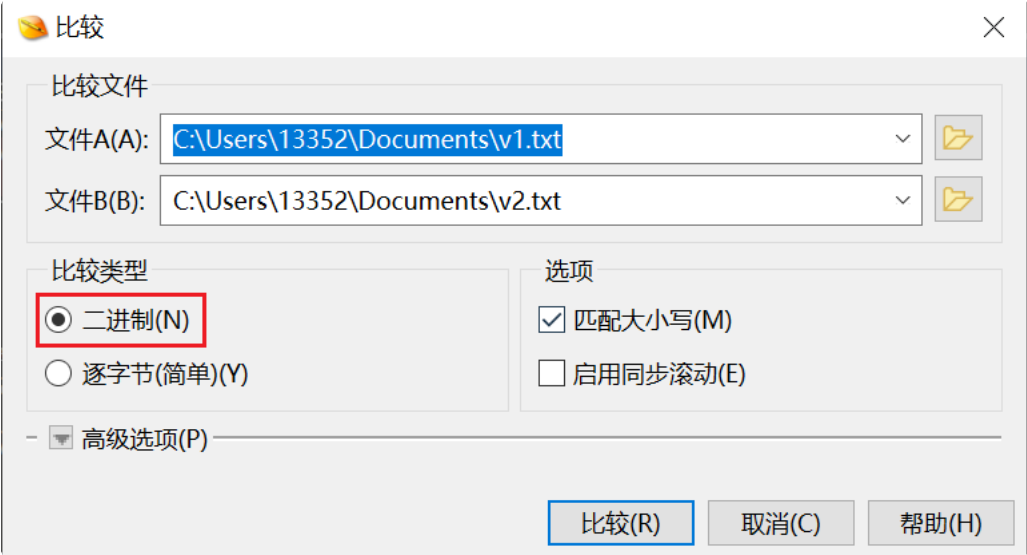
我们用 cyberchef 把两份数据都规整成 hexdump 格式，各保存到一个文件里。

 v1.txt (2 KB)  v2.txt (2 KB)

放到 010Editor 里做比较



选择二进制比较

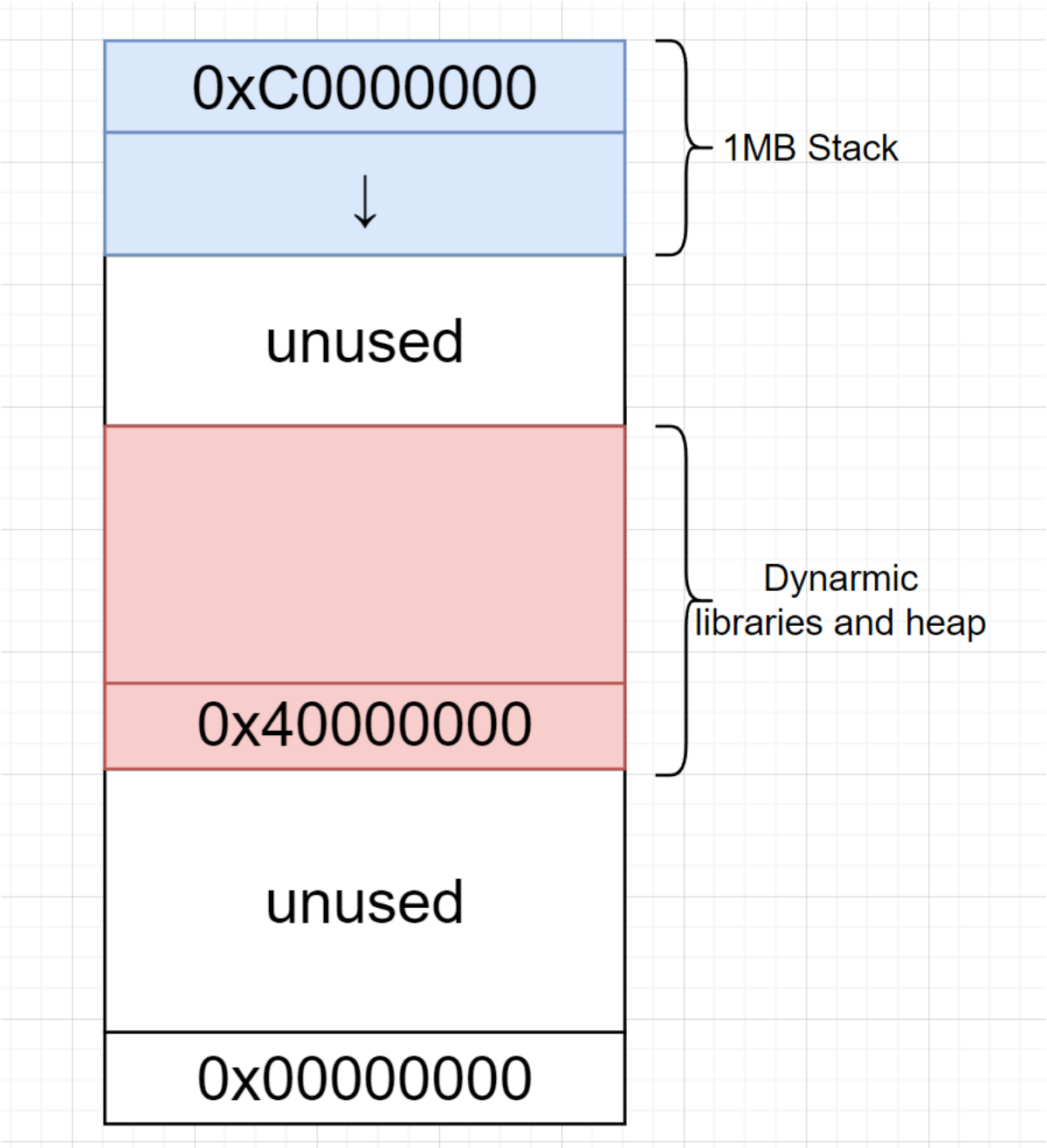


效果如下

起始页	v1.txt	
00000000	df fa df fa 02 0d 0a 03 02 1e 54 f6 17 de 54 48	ÄYÄ°ÄYÄ°.....TÄ¶.ÄžTH
00000010	60 1b 4e 59 03 40 31 04 c7 a5 60 45 40 26 a0 37	` .NY.@1.Ä±Ä¥`E@&Ä 7
00000020	4b e3 8a d8 00 79 e5 9e 06 78 a8 66 2f 31 94 52	KÄ£.Ä~.yÄ¥..xÄ" f/1.R
00000030	2f 50 78 8e ca 09 5a f9 2f d3 b9 89 50 88 c8 89	/Px.ÄŠ.ZÄ¹/Ä"Ä¹.P.Ä^.
00000040	04 b1 08 c0 6d 19 4b 0c 86 42 9a 7e f8 2b 84 c3	.Ä±.Ä€m.K..B.~Ä.+.Äf
00000050	2a 66 2d 8c 0e ac 00 f2 02 00 08 54 00 03 ff 00	*f-..Ä¬.Ä²...T..Ä¿.
00000060	0a 00 10 88 29 b2 00 4d e3 20 00 00 00 00 00 6d	....)Ä².MÄ£ .....m
00000070	db 06 87 85 01 00 00 00 00 00 00 00 00 00 00	Ä>.....
00000080	06 01 06 02 01 03 02 04 05 05 06 03 07 03 08	.....
00000090	03 09 06 0a 04 0c 01 0d 02 10 02 11 03 12 05 13	.....
000000a0	06 14 05 15 02 17 03 1a 04 1b 06 20 03 23 02 24	.....#.\$
000000b0	01 25 03 26 05 64 02 ca 02 cb 06 36 f6 d6 e2 e6	%.&.d.ÄŠ.Ä<.6Ä¶Ä-ÄcÄ
000000c0	96 b6 56 e2 f6 d6 56 76 16 00 e4 96 b6 56 02 8e	.Ä¶VÄcÄ¶Ä-Vv..Ä±.Ä¶V..
000000d0	08 09 5e 58 b8 00 23 23 e2 23 73 e2 53 00 23 e2	..^XÄ.##Äc#sÄcS.#Äc
000000e0	03 e2 53 00 00 85 96 16 f6 d6 96 00 d4 94 85 02	.ÄcS.....Ä¶Ä-..Ä"....
000000f0	23 35 00 07 f6 c6 16 27 96 37 00 85 96 16 f6 d6	#5..Ä¶Ä†.' .7....Ä¶Ä-
00000100	96 00 17 36 f6 d6 00 13 03 83 03 87 23 13 63 03	...6Ä¶Ä-.....#.c.
00000110	00 23 93 00 13 03 00 c3 57 e6 b6 e6 f6 77 e6 02	..#.....ÄfwÄ!Ä¶Ä!Ä¶wÄ!..
v2.txt		
00000000	df fa df fa 02 0d 0a 03 02 1e 65 5a ab e5 54 48	ÄYÄ°ÄYÄ°.....eZÄ«Ä¥TH
00000010	60 1b 4e 59 03 40 31 04 c7 a5 60 45 40 26 a0 37	` .NY.@1.Ä±Ä¥`E@&Ä 7
00000020	4b e3 8a d8 00 79 e5 9e 06 78 a8 66 2f 31 94 52	KÄ£.Ä~.yÄ¥..xÄ" f/1.R
00000030	2f 50 78 8e ca 09 82 a8 5e b3 41 1e c7 da 76 35	/Px.ÄŠ..Ä" ^Ä³A.Ä±Äšv5
00000040	e9 89 a7 ab 0e 12 d8 06 50 2c 6f 0a 37 c8 bd af	Ä©.ÄšÄ«..Ä~.P,o.7Ä^Ä½Ä
00000050	f7 80 77 60 30 91 00 f2 02 00 08 54 00 03 ff 00	Ä·.w`0..Ä²...T..Ä¿.
00000060	0a 00 10 88 29 b2 00 4d e3 20 00 00 00 00 00 6d	....)Ä².MÄ£ .....m
00000070	db 06 87 85 01 00 00 00 00 00 00 00 00 00 00	Ä>.....
00000080	06 01 06 02 01 03 02 04 05 05 06 03 07 03 08	.....
00000090	03 09 06 0a 04 0c 01 0d 02 10 02 11 03 12 05 13	.....
000000a0	06 14 05 15 02 17 03 1a 04 1b 06 20 03 23 02 24	.....#.\$
000000b0	01 25 03 26 05 64 02 ca 02 cb 06 36 f6 d6 e2 e6	%.&.d.ÄŠ.Ä<.6Ä¶Ä-ÄcÄ
000000c0	96 b6 56 e2 f6 d6 56 76 16 00 e4 96 b6 56 02 8e	.Ä¶VÄcÄ¶Ä-Vv..Ä±.Ä¶V..
000000d0	08 09 5e 58 b8 00 23 23 e2 23 73 e2 53 00 23 e2	..^XÄ.##Äc#sÄcS.#Äc
000000e0	03 e2 53 00 00 85 96 16 f6 d6 96 00 d4 94 85 02	.ÄcS.....Ä¶Ä-..Ä"....
000000f0	23 35 00 07 f6 c6 16 27 96 37 00 85 96 16 f6 d6	#5..Ä¶Ä†.' .7....Ä¶Ä-
00000100	96 00 17 36 f6 d6 00 13 03 83 03 87 23 13 63 03	...6Ä¶Ä-.....#.c.
00000110	00 23 93 00 13 03 00 c3 57 e6 b6 e6 f6 77 e6 02	..#.....ÄfwÄ!Ä¶Ä!Ä¶wÄ!..

整体一致，但局部又稍微有那么一些不同，按照先抓整体再管局部的原则，我们先看整体这个数据哪里来的，其地址是 0xd59b6280。

回忆一下 Unidbg 的内存布局



- 如果一个数的值在 0x40000000 - 0x50000000 区间里，那么它极大概率是一个地址，而且指向动态库或堆内存。
- 如果一个数的值是 0xBFF?????, 那么它大概率是一个指向栈空间的地址。

我们这里的地址是 0xd59b6280，所以它应该是我们 dump 出来的，而不是 Unidbg 内部产生的数据。根据地址范围判断，它应该在 d5900000\_d6000000.bin 里，到之前的 maps.log 里找一下，发现属于 [anon:libc\_malloc]，即由 malloc 开辟的内存。

```
1    d5900000-d6000000 rw-p 00000000 00:00 0 [anon:libc_malloc]
```

翻译一下就是这个数据产生在我们的调用点之前的某个时机，而且所属的内存块由 libc 的 malloc 函数开辟。

这其实展现了 dump 法的一个大缺陷，早于调用点的麻烦你确实不用管了，但与之相对应，早于调用点的这些逻辑你自然也分析不了。

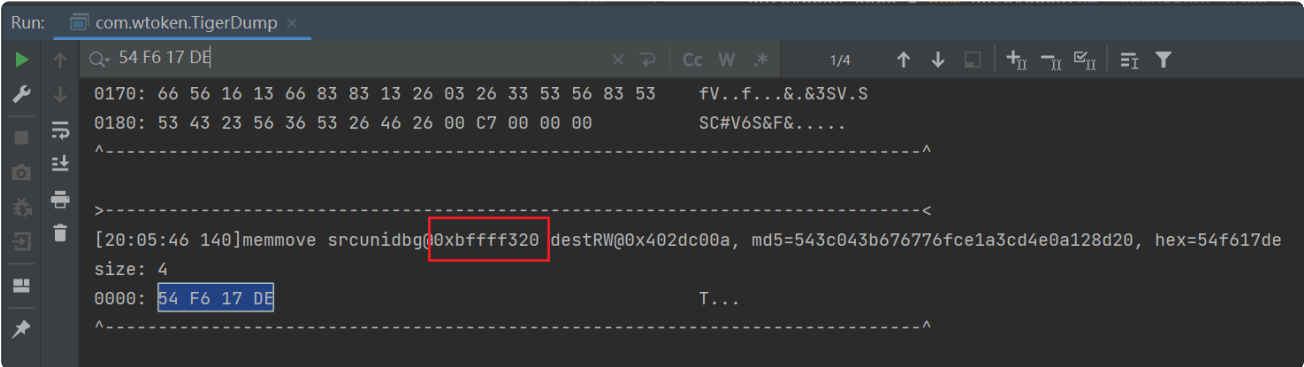
对于这个样本而言，这个数据块可能生成自 genericNt3 或 JNI\_OnLoad，需要我们通过其他方法分析。

整体分析完了，接下来看局部，似乎有两小部分不一致，分别是 4 + 32 字节共 36 字节。

先看四字节的这部分，值是 54 F6 17 DE。下面我们尝试用三种不同的方式对它的生成处做定位。一题多解是一件很有意义的事，因为并非所有人都和你有相同的思维方式，多几个思路可供参考能增加成功率。

## 2.1 思路一

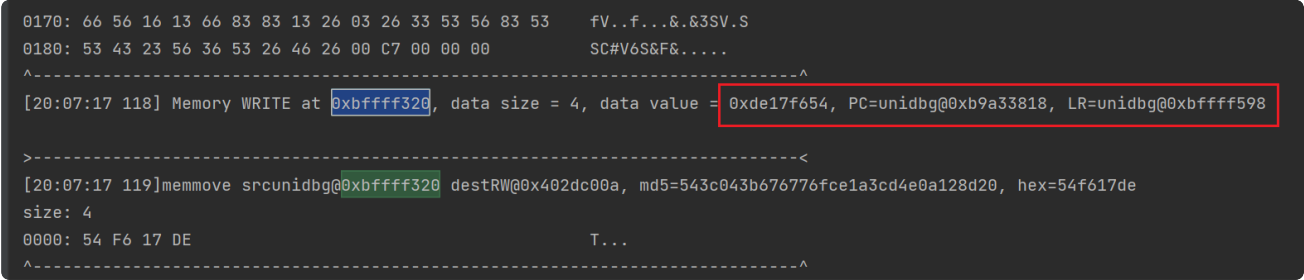
上篇我们 hook 打印了很多外部库函数的参数，在运行日志里搜一下 54 F6 17 DE，说不定它作为参数出现过。



发现竟然能找到，来源是 0xbffff320。下面对这个位置做内存追踪。

```
1 emulator.traceWrite(0xbffff320L,0xbffff320L);
```

运行如下



根据绝对地址计算出偏移是 0x5E818

☰ 程序员

B9A33818 - B99D5000 =  
**5 E818**

HEX 5 E818

IDA 跳过去看看

```

.text:0005E7F4
.text:0005E7F8 28 68          LDR      R0, [R5]
.text:0005E7FA 6F F0 6D 02    MOV      R2, #0xFFFFF92
.text:0005E7FE 00 79          LDRB     R0, [R0, #4]
.text:0005E800 50 43          MULS     R0, R2
.text:0005E802
.text:0005E802          loc_5E802          ; CODE XREF: sub_5DA10:loc_5E84E1j
.text:0005E804 C0 B2          UXTB     R0, R0
.text:0005E806 80 28          CMP      R0, #0x80
.text:0005E808 C5 D1          BNE      loc_5E794
.text:0005E80A
.text:0005E80A A7 F1 8C 00    SUB.W    R0, R7, #0x8C
.text:0005E80C 57 F8 90 3C    LDR.W    R3, [R7, #-0x90]
.text:0005E80E DE F8 00 20    LDR.W    R2, [LR]
.text:0005E810 50 F8 2C 00    LDR.W    R0, [R0, R12, #1*2]
.text:0005E812 C2 50          STR      R2, [R0, R3]
.text:0005E814 DA F8 98 00    LDR.W    R0, [R10, #0x98]
.text:0005E816 DA F8 D8 20    LDR.W    R2, [R10, #0xD8]
.text:0005E818 10 44          ADD      R0, R2
.text:0005E81A 49 F6 F2 52 C4 F6 85 62    MOV      R2, #0x4E859DF2
.text:0005E81C 90 42          CMP      R0, R2
.text:0005E81E 40 F0 BD 81    BNE.W    loc_5E8AC
.text:0005E820 FF F7 35 BD    B.W      loc_5E2A0
.text:0005E822
.text:0005E822          ;
.text:0005E822          loc_5E836          ; CODE XREF: sub_5DA10+AD41j
.text:0005E824 DA F8 1C 00    LDR.W    R0, [R10, #0x1C]
.text:0005E826 DA F8 CC 20    LDR.W    R2, [R10, #0xCC]
.text:0005E828 10 43          ORRS     R0, R2
.text:0005E82A 6F F4 C9 72    MOV      R2, #0xFFFFFE0D
.text:0005E82C 90 42          CMP      R0, R2
.text:0005E82E 00 F0 87 81    BEQ.W    loc_5E858
.text:0005E830 00 F0 68 BB    B.W      loc_5EF24
.text:0005E832
.text:0005E832          ;
.text:0005E832          loc_5E84E          ; CODE XREF: sub_5DA10+A101j
.text:0005E834 D8 DC          BGT      loc_5E802
.text:0005E836
.text:0005E836 01 20          MOV.S    R0, #1
.text:0005E838 DA EA A9 FA    PKHTBS.W R10, R10. R9.ASR#30
0005E838 0005E838: sub_5DA10+E08 (Synchronized with Hex View-1)

```

这个值来自 0x5E810，我们搞个 Hook 打印一下 LR 以及它指向的 32 位数据。

```

1  emulator.attach().addBreakPoint(0xb9a33810L, new BreakPointCallback() {
2
3      @Override
4      public boolean onHit(Emulator<?> emulator, long address) {
5          RegisterContext registerContext = emulator.getContext();
6          UnidbgPointer lr = registerContext.getLRPointer();
7          int value = lr.getInt(0);
8          System.out.println("hit 0xb9a33810L:" + lr);
9          System.out.println("value:"+Integer.toHexString(value));
10         return true;
11     }
12 });

```

运行代码

```

Run: com.wtoken.TigerDump x
Q: de17f654
hit 0xb9a33810L:unidbg@0xbffff580
value:4
hit 0xb9a33810L:unidbg@0xbffff580
value:a
hit 0xb9a33810L:unidbg@0xbffff59c
value:bffff2fc
hit 0xb9a33810L:unidbg@0xbffff598
value:de17f654
[20:18:02 690] Memory WRITE at 0xbffff320, data size = 4, data value = 0xde17f654, PC=unidbg@0xb9a33818, LR=unidbg@0xbffff598

```

对 0xbffff598 做写入的监控，找来源。

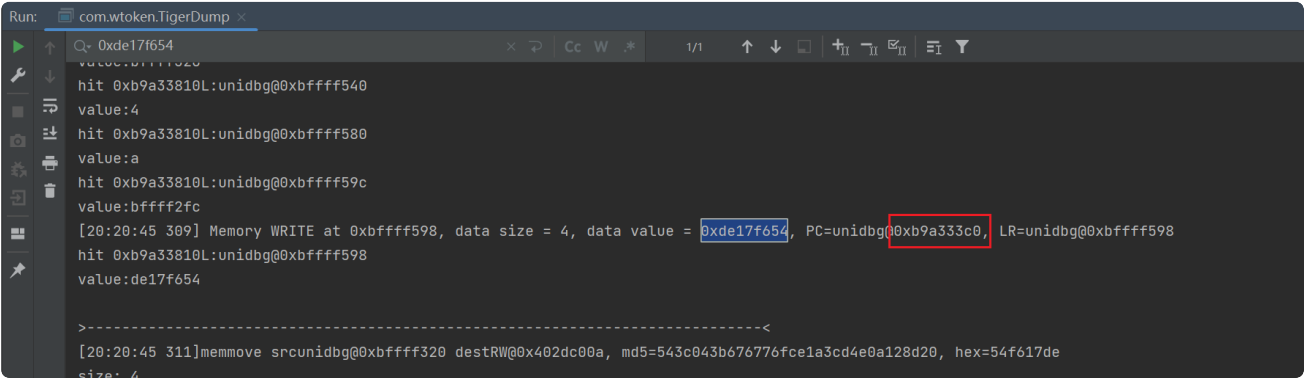
```

1  emulator.traceWrite(0xbffff598L, 0xbffff598L);

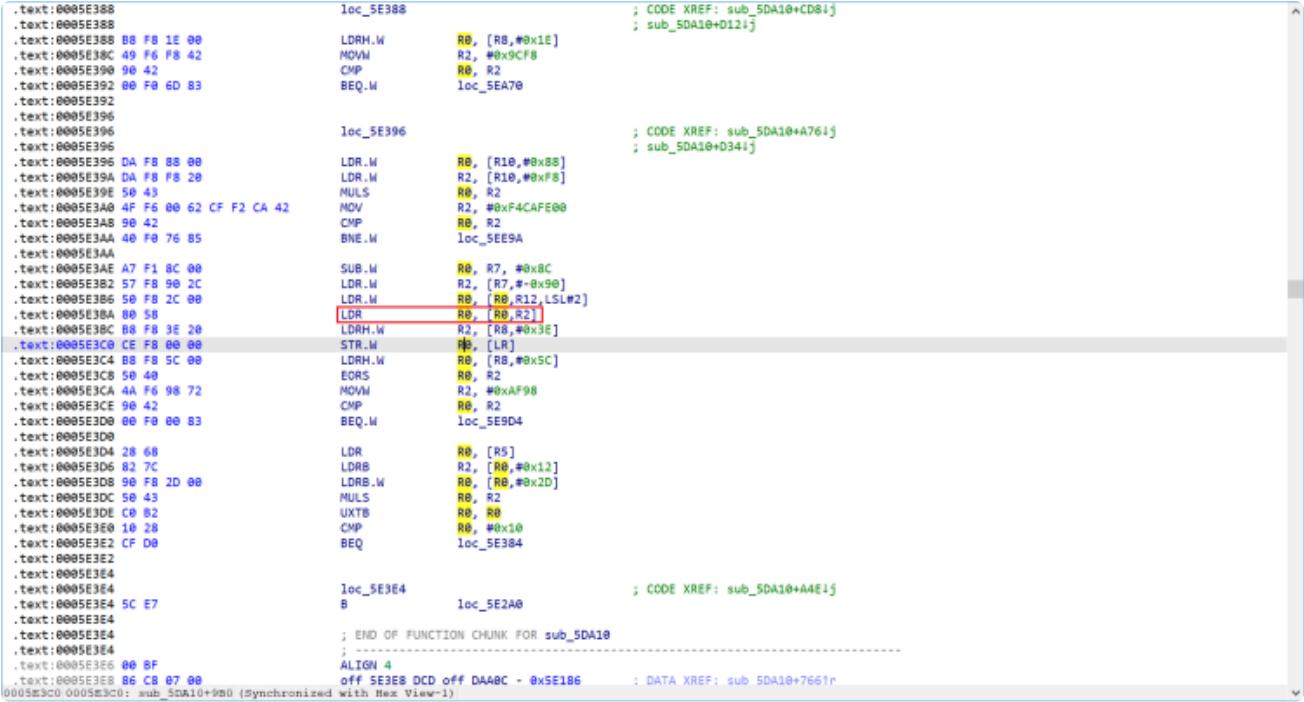
```

运行代码

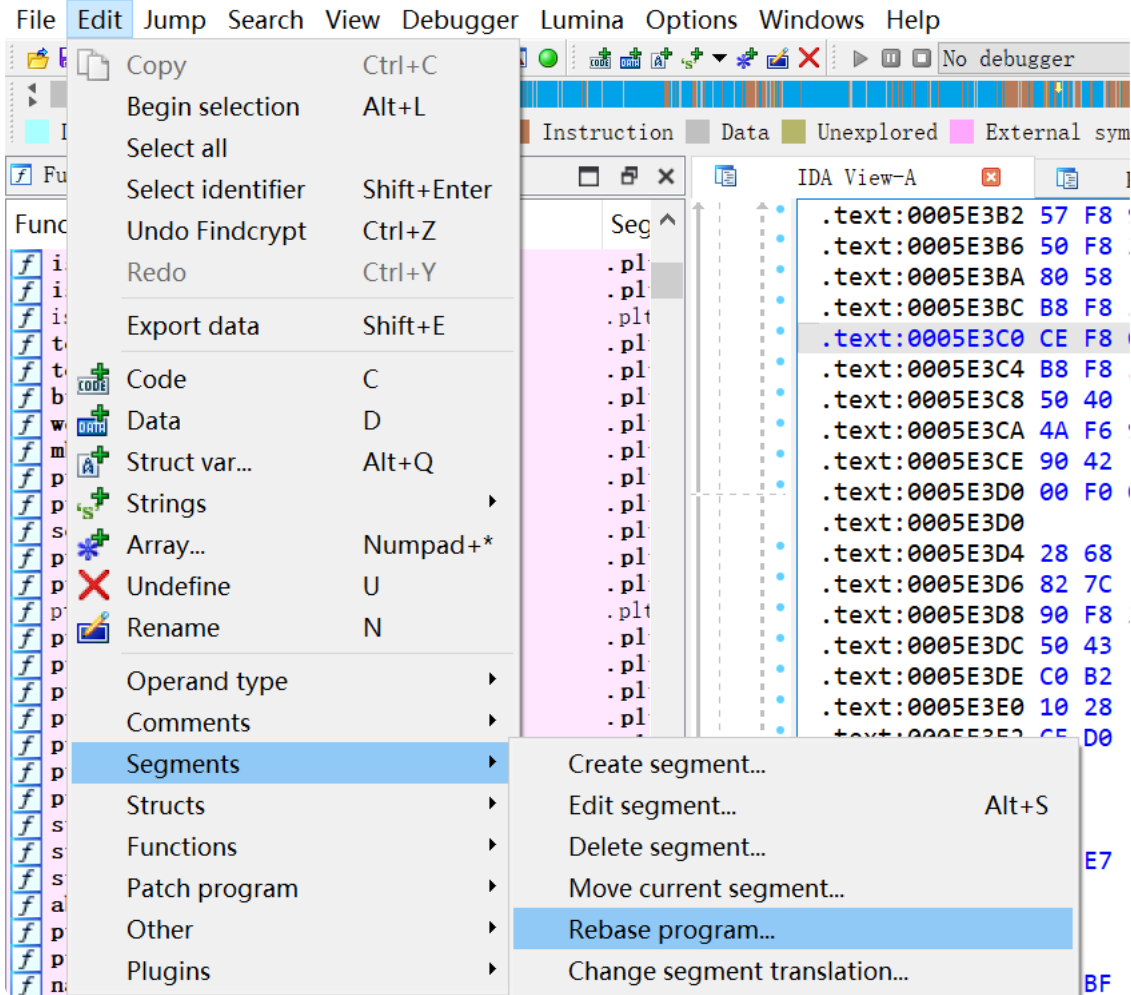




相对偏移是 0x5E3C0，IDA 里跳过去看看。



如果你讨厌计算偏移地址的步骤，可以把 IDA 的基地址设置为 0xb99d5000。



言归正传，这个值来自于 0x5E3BA，来源于是 R0 + R2 指向的 32 位数据，同理做 Hook 处理。

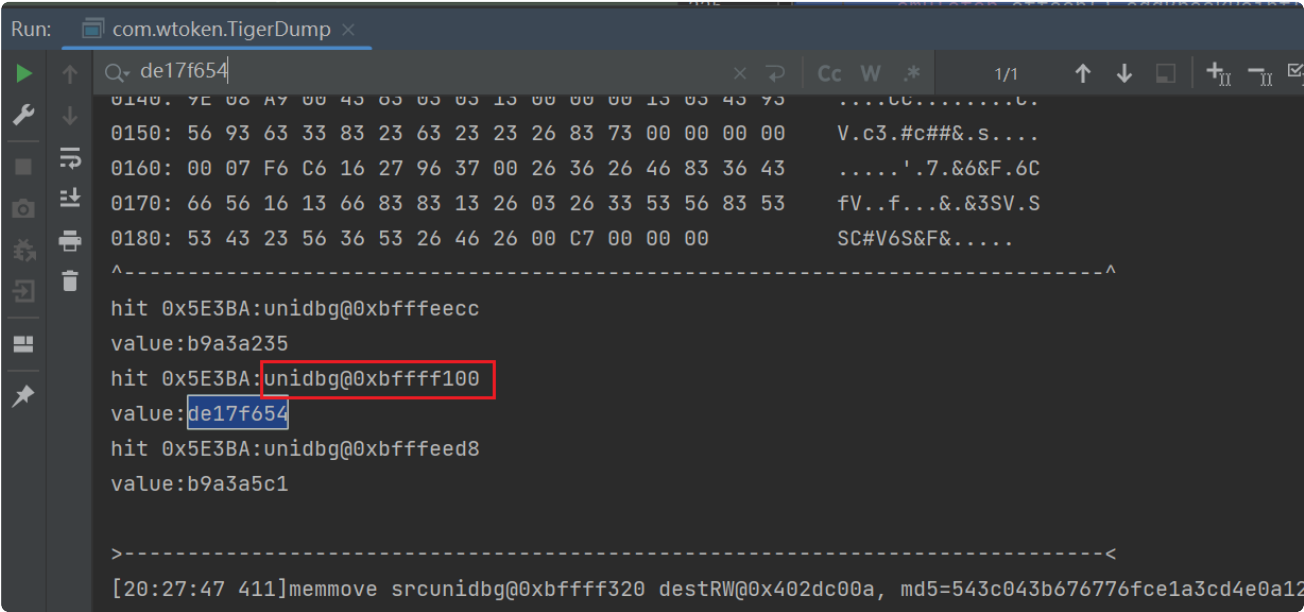
```

1  emulator.attach().addBreakPoint(moduleBase + 0x5E3BA, new BreakPointCallback()
2
3  @Override
4  public boolean onHit(Emulator<?> emulator, long address) {
5
6      RegisterContext registerContext = emulator.getContext();
7      int r0 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R0);
8      int r2 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R2);
9
10     UnidbgPointer ptr = UnidbgPointer.pointer(emulator, r0 + r2);
11     assert ptr != null;
12     int value = ptr.getInt(0);
13     System.out.println("hit 0x5E3BA:" + ptr);
14     System.out.println("value:" + Integer.toHexString(value));
15     return true;
16 }
17
18 });

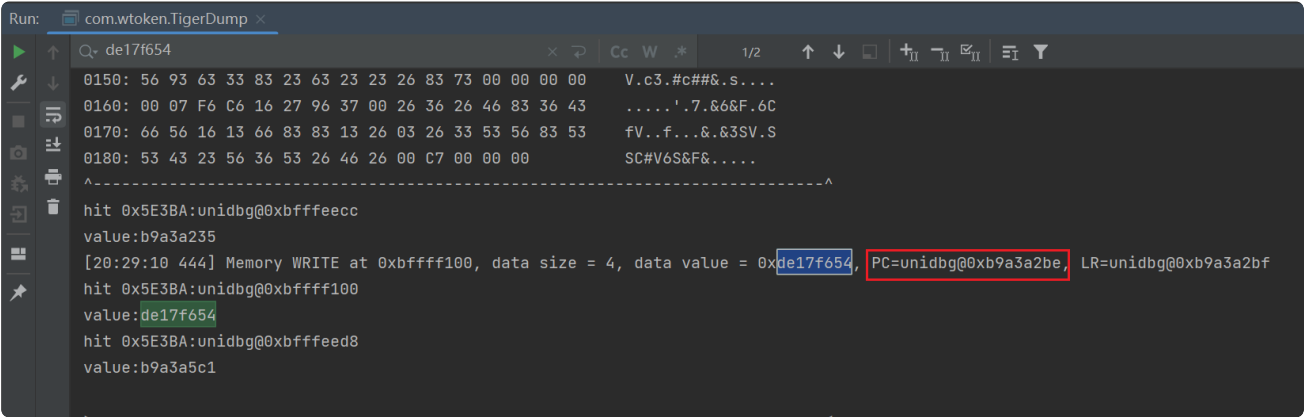
```

日志里搜索

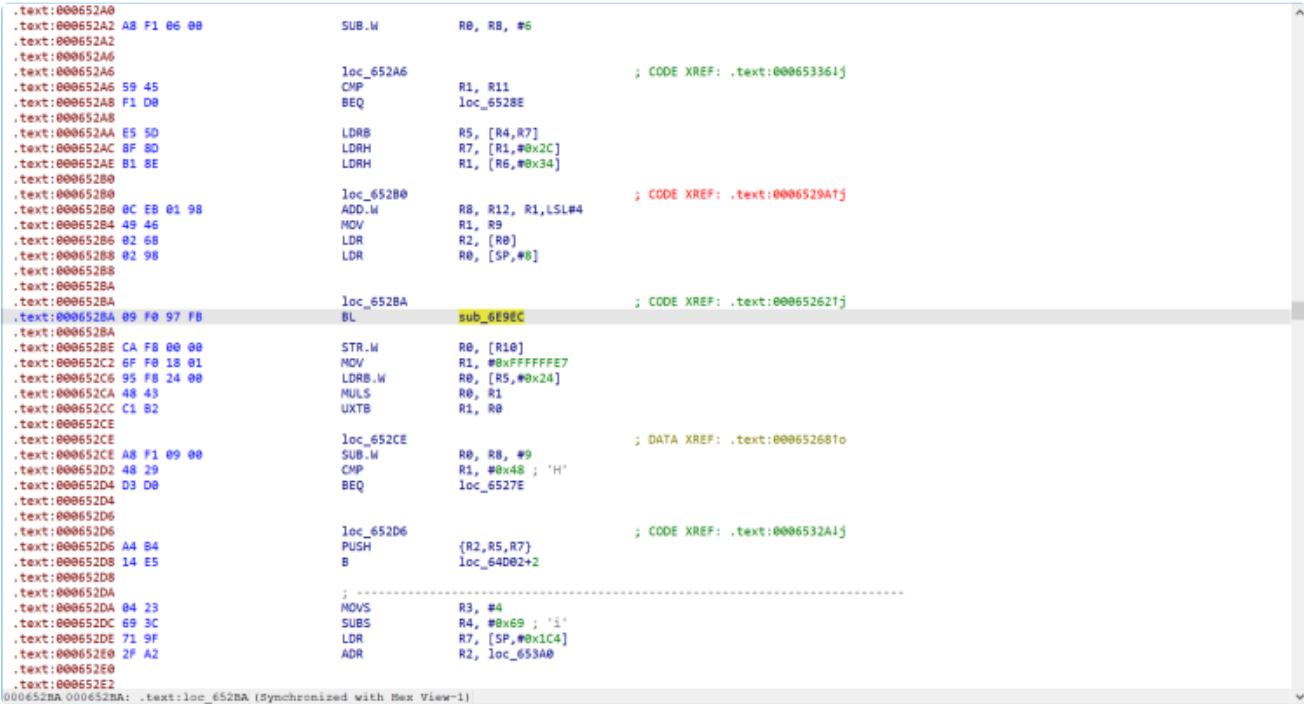




来源于 0xBFFFF100，继续追。



偏移地址 0x652BE



R0 是 sub\_6E9EC 的返回值，我们可以 Hook 做进一步的验证。Hook 代码如下

```

1  emulator.attach().addBreakPoint(moduleBase + 0x6E9EC, new BreakPointCallback() {
2      @Override
3      public boolean onHit(Emulator<?> emulator, long address) {
4          final RegisterContext registerContext = emulator.getContext();
5          emulator.attach().addBreakPoint(registerContext.getLR(), new BreakPointCallback() {
6              @Override
7              public boolean onHit(Emulator<?> emulator, long address) {
8                  int r0 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R0);
9                  System.out.println("call 0x6E9EC:" + Integer.toHexString(r0));
10                 return true;
11             }
12         });
13         return true;
14     }
15 });

```

运行发现确实是这个值的生成点，而且这个函数只运行一次。

```

Run: com.wtoken.TigerDump x
Q: call 0x6E9EC
0140: 9E 08 A9 00 43 63 03 03 13 00 00 00 13 03 43 93    ....Cc.....C.
0150: 56 93 63 33 83 23 63 23 23 26 83 73 00 00 00 00    V.c3.#c##&.s....
0160: 00 07 F6 C6 16 27 96 37 00 26 36 26 46 83 36 43    .....'.7.&6&F.6C
0170: 66 56 16 13 66 83 83 13 26 03 26 33 53 56 83 53    fV..f...&.&3SV.S
0180: 53 43 23 56 36 53 26 46 26 00 C7 00 00 00         SC#V6S&F&.....
^-----^
call 0x6E9EC:de17f654
>-----<
[20:50:33 540]memmove src:unidbg@0xbffff320 dest:RW@0x402dc00a, md5=543c043b676776fcea1a3cd4e0a128d20, hex=54f
size: 4

```

让我们仔细看看这个函数？

```
1 unsigned int __fastcall sub_6E9EC(unsigned __int8 *a1, unsigned int a2, int a3)
2 {
3     unsigned int v3; // r3
4     int v4; // r4
5
6     v3 = a3 ^ a2;
7     while ( a2 >= 4 )
8     {
9         v4 = *(_DWORD *)a1;
10        a1 += 4;
11        a2 -= 4;
12        v3 = (1540483477 * v3) ^ (1540483477 * ((1540483477 * v4) ^ ((unsigned int)
13    })
14    switch ( a2 )
15    {
16        case 1u:
17            goto LABEL_9;
18        case 2u:
19    LABEL_8:
20        v3 ^= a1[1] << 8;
21    LABEL_9:
22        v3 = 1540483477 * (*a1 ^ v3);
23        return (1540483477 * (v3 ^ (v3 >> 13))) ^ ((1540483477 * (v3 ^ (v3 >> 13)))
24    case 3u:
25        v3 ^= a1[2] << 16;
26        goto LABEL_8;
27    }
28    return (1540483477 * (v3 ^ (v3 >> 13))) ^ ((1540483477 * (v3 ^ (v3 >> 13)))
29 }
```

在过去，我们需要经过一些动静态分析，才能明白这个算法的语义，但现在我们有 chatGPT，来看看它怎么说，我的输入如下。

```
1 对下面的C语言伪代码函数进行分析 推测这是一个什么算法
2 unsigned int __fastcall sub_6E9EC(unsigned __int8 *a1, unsigned int a2, int a3)
3 {
4     unsigned int v3; // r3
5     int v4; // r4
6
7     v3 = a3 ^ a2;
8     while ( a2 >= 4 )
9     {
10         v4 = *(_DWORD *)a1;
11         a1 += 4;
12         a2 -= 4;
13         v3 = (1540483477 * v3) ^ (1540483477 * ((1540483477 * v4) ^ ((unsigned int)v3 >> 13)));
14     }
15     switch ( a2 )
16     {
17         case 1u:
18             goto LABEL_9;
19         case 2u:
20             LABEL_8:
21             v3 ^= a1[1] << 8;
22             LABEL_9:
23             v3 = 1540483477 * (*a1 ^ v3);
24             return (1540483477 * (v3 ^ (v3 >> 13))) ^ ((1540483477 * (v3 ^ (v3 >> 13))) ^ (1540483477 * v3));
25         case 3u:
26             v3 ^= a1[2] << 16;
27             goto LABEL_8;
28     }
29     return (1540483477 * (v3 ^ (v3 >> 13))) ^ ((1540483477 * (v3 ^ (v3 >> 13))) ^ (1540483477 * v3));
30 }
```

它的回答非常漂亮。



它认为这是一个散列函数（也即哈希函数），但到底是标准的哈希函数，还是样本自定义的哈希函数，它没有给出答案。chatGPT 进一步认为，参数 1 指向哈希的输入，参数 2 是它的长度，参数 3 是一个初始值。

将函数 Hook onEnter 时机的返回值改成 false，这意味着它会在经过时断下来，好让我们动态调试。

```

1  emulator.attach().addBreakPoint(moduleBase + 0x6E9EC, new BreakPointCallback(
2      @Override
3      public boolean onHit(Emulator<?> emulator, long address) {
4          final RegisterContext registerContext = emulator.getContext();
5          emulator.attach().addBreakPoint(registerContext.getLR(), new BreakPoint
6              @Override
7              public boolean onHit(Emulator<?> emulator, long address) {
8                  int r0 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R0);
9                  System.out.println("call 0x6E9EC:" + Integer.toHexString(r0));
10                 return true;
11             }
12         });
13         // 修改为 false
14         return false;
15     }
16 });

```

运行测试，按照 chatGPT 的说法，起始地址是 0x402dc1c0，长度是 0x18e，初始值是 0。

```

Run: com.wtoken.TigerDump x
0160: 00 07 f6 c6 16 27 96 37 00 26 36 26 46 83 36 43 .....'.7.&6&F.6C
0170: 66 56 16 13 66 83 83 13 26 03 26 33 53 56 83 53 fV..f...&.&3SV.S
0180: 53 43 23 56 36 53 26 46 26 00 c7 00 00 00 SC#V6S&F&.....
^-----^
debugger break at: 0xb9a439ec @ Runnable|Function32 address=0xb99faa9d, arguments=[unidbg@0xffffe12a0,
>>> r0=0x402dc1c0 r1=0x18e r2=0x0 r3=0xce r4=0x84123888 r5=0xb9ab0e10 r6=0x47 r7=0xb9ab0ec0 r8=0x19 s
>>> SP=0xbfffecc8 LR=unidbg@0xb9a3a2bf PC=unidbg@0xb9a439ec cpsr: N=1, Z=0, C=0, V=0, T=1, mode=0b100
>>> d0=0x4626533656234326(8.843862599486192E29) d1=0x9343031300000000(-6.893838755636088E-216) d2=0x2
>>> 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(
[0000] 0xb9a439ea: "movs r0, r0"
=> *[d0b5] *0xb9a439ec: "push {r4, r6, r7, lr}"
[02af] 0xb9a439ee: "add r7, sp, #8"
[82ea0103] 0xb9a439f0: "eor.w r3, r2, r1"
[104a] 0xb9a439f4: "ldr r2, [pc, #0x40]"
[0429] 0xb9a439f6: "cmp r1, #4"
[08d3] 0xb9a439f8: "blo #0xb9a43a0c"
[10c8] 0xb9a439fa: "ldm r0!, {r4}"
[5343] 0xb9a439fc: "muls r3, r2, r3"
[0439] 0xb9a439fe: "subs r1, #4"
[5443] 0xb9a43a00: "muls r4, r2, r4"
[84ea1464] 0xb9a43a02: "eor.w r4, r4, r4, lsr #24"
[5443] 0xb9a43a06: "muls r4, r2, r4"
[6340] 0xb9a43a08: "eors r3, r4"
[f4e7] 0xb9a43a0a: "b #0xb9a439f6"
[0129] 0xb9a43a0c: "cmp r1, #1"
[09d0] 0xb9a43a0e: "beq #0xb9a43a24"

```

打印看看



```

Run: com.wtoken.TigerDump x
debugger break at: 0xb9a439ec @ Runnable|Function32 address=0xb99faa9d, arguments=[unidbg@0xffff
>>> r0=0x402dc1c0 r1=0x18e r2=0x0 r3=0xce r4=0x84123888 r5=0xb9ab0e10 r6=0x47 r7=0xb9ab0ec0 r8=
>>> SP=0xbffffec8 LR=unidbg@0xb9a3a2bf PC=unidbg@0xb9a439ec cpsr: N=1, Z=0, C=0, V=0, T=1, mode
>>> d0=0x4626533656234326(8.843862599486192E29) d1=0x9343031300000000(-6.893838755636088E-216)
>>> 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) d1
[0000 ] 0xb9a439ea: "movs r0, r0"
=> *[d0b5 ]*0xb9a439ec:*"push {r4, r6, r7, lr}"
[02af ] 0xb9a439ee: "add r7, sp, #8"
[82ea0103] 0xb9a439f0: "eor.w r3, r2, r1"
[104a ] 0xb9a439f4: "ldr r2, [pc, #0x40]"
[0429 ] 0xb9a439f6: "cmp r1, #4"
[08d3 ] 0xb9a439f8: "blo #0xb9a43a0c"
[10c8 ] 0xb9a439fa: "ldm r0!, {r4}"
[5343 ] 0xb9a439fc: "muls r3, r2, r3"
[0439 ] 0xb9a439fe: "subs r1, #4"
[5443 ] 0xb9a43a00: "muls r4, r2, r4"
[84ea1464] 0xb9a43a02: "eor.w r4, r4, r4, lsr #24"
[5443 ] 0xb9a43a06: "muls r4, r2, r4"
[6340 ] 0xb9a43a08: "eors r3, r4"
[f4e7 ] 0xb9a43a0a: "b #0xb9a439f6"
[0129 ] 0xb9a43a0c: "cmp r1, #1"
[09d0 ] 0xb9a43a0e: "beq #0xb9a43a24"

0x402dc1c0 0x18e

>-----<
[21:02:12 689]RW@0x402dc1c0, md5=15de197b3e8e039ed6e063ed72cb66fd, hex=5448601b4e5903403104c7a5
size: 398
0000: 54 48 60 1B 4E 59 03 40 31 04 C7 A5 60 45 40 26 TH`.NY.@1...`E&
0010: A0 37 4B E3 8A D8 00 79 E5 9E 06 78 A8 66 2F 31 .7K...y...x.f/1
0020: 94 52 2F 50 78 8E CA 09 5A F9 2F D3 B9 89 50 88 .R/Px...Z./...P.
0030: C8 89 04 B1 08 C0 6D 19 4B 0C 86 42 9A 7E F8 2B .....m.K..B..+.
0040: 84 C3 2A 66 2D 8C 0E AC 00 F2 02 00 08 54 00 03 ..*f-.....T..
0050: FF 00 0A 00 10 88 29 B2 00 4D E3 20 00 00 00 00 .....).M. ....
0060: 00 6D DB 06 87 85 01 00 00 00 00 00 00 00 00 00 .m.....
0070: 00 00 06 01 06 02 01 03 02 04 05 05 06 03 07 .....
0080: 03 08 03 09 06 0A 04 0C 01 0D 02 10 02 11 03 12 .....
0090: 05 13 06 14 05 15 02 17 03 1A 04 1B 06 20 03 23 ..... .#

```

感觉它说的可能是对的，但也没啥证据。

我们可以注意到，函数通篇都在使用一个硬编码的值，在值上按 h 将它转成十六进制形式。

```

1 unsigned int __fastcall sub_6E9EC(unsigned __int8 *a1, unsigned int a2, int a3)
2 {
3     unsigned int v3; // r3
4     int v4; // r4
5
6     v3 = a3 ^ a2;
7     while ( a2 >= 4 )
8     {
9         v4 = *(_DWORD *)a1;
10        a1 += 4;
11        a2 -= 4;
12        v3 = (0x5BD1E995 * v3) ^ (0x5BD1E995 * ((0x5BD1E995 * v4) ^ ((unsigned int)(0x5BD1E995 * v4) >> 24)));
13    }
14    switch ( a2 )
15    {
16        case 1u:
17            goto LABEL_9;
18        case 2u:
19    LABEL_8:
20        v3 ^= a1[1] << 8;
21    LABEL_9:
22        v3 = 0x5BD1E995 * (*a1 ^ v3);
23        return (0x5BD1E995 * (v3 ^ (v3 >> 13))) ^ ((0x5BD1E995 * (v3 ^ (v3 >> 13))) >> 15);
24        case 3u:
25        v3 ^= a1[2] << 16;
26        goto LABEL_8;
27    }
28    return (0x5BD1E995 * (v3 ^ (v3 >> 13))) ^ ((0x5BD1E995 * (v3 ^ (v3 >> 13))) >> 15);
29 }

```

## Google 搜索这个数

hash algorithm 0x5BD1E995 - x +

google.com/search?q=hash+algorithm+0x5BD1E995&oq=hash+algorithm+0x5BD1E995&aqs=chrome..69i57j0i546j0i30i546j...

Launch bot: Youtu... FUNNYAPI-GIS: ... MapLocation-地... GitHub - bkidy/Di... GitHub - StanleyL... GitHub - yTang98... GitHub -

Google hash algorithm 0x5BD1E995

全部 图片 视频 地图 新闻 更多 工具

找到约 1,270 条结果 (用时 0.36 秒)

<https://github.com/murmur2/blob/master/MurmurHash2.c> at master · abandoned... - GitHub

Murmur2 Hashes in ansi C. Contribute to abandoned/murmur2 development by ... const uint32\_t m = 0x5bd1e995; ... Mix 4 bytes at a time into the hash \*/.

<https://www.cnblogs.com/findumars>

murmurhash2算法和DJB Hash算法是目前最流行的 ... - 博客园

2017年6月13日 — DJB HASH算法1 2 3 4 5 6 7 8 9 10 11 /\* the famous DJB Hash. ... the famous DJB Hash Function for strings \*/ ... k \*= 0x5bd1e995;.

<https://commons.apache.org/proper/jacoco>

MurmurHash2.java - Apache Commons

\* Generates a 64-bit hash from byte array with given length and a default seed value. \* This is a helper method that will produce the same result as: \* <pre>

<http://developer.aliyun.com/article>

【密码学】一文读懂MurMurHash2 - 阿里云开发者社区

2022年6月12日 — 32位哈希算法 ... int m = 0x5bd1e995; const int r = 24; // Initialize the hash to a ... 64-bit hash for 64-bit platforms // code from: ...

<https://developer.aliyun.com/article>

【密码学】一文读懂MurMurHash2 - 阿里云开发者社区

They're not really 'magic', they just happen to work well. const unsigned int m = 0x5bd1e995;

虽然我们可以直接把它当成自定义哈希函数处理，但如果可以的话，我很乐意挖掘它是否是某个标准算法，这是一个乐趣。比如这里，我们 Google 发现它可能是 murmurhash2 算法，点第一个链

接进去看看。

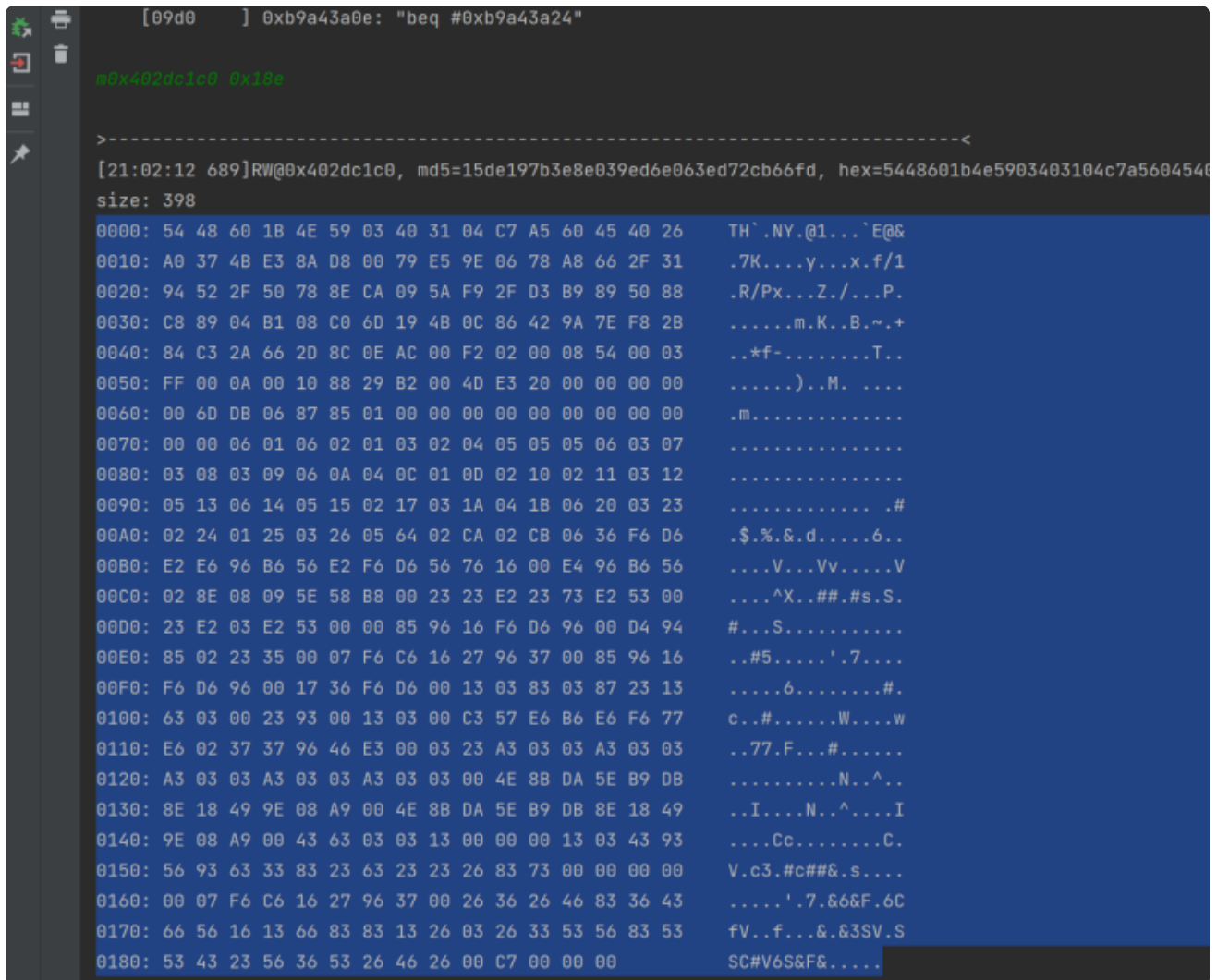
```
1  /* MurmurHash2, by Austin Appleby
2
3  // Note - This code makes a few assumptions about how your machine behaves -
4  // 1. We can read a 4-byte value from any address without crashing
5  // 2. sizeof(int) == 4
6
7  //
8  // And it has a few limitations -
9
10 //
11 // 1. It will not work incrementally.
12 // 2. It will not produce the same results on little-endian and big-endian
13 // machines. */
14
15
16
17 uint32_t MurmurHash2 ( const void * key, int len, uint32_t seed )
18 {
19     /* 'm' and 'r' are mixing constants generated offline.
20        They're not really 'magic', they just happen to work well. */
21
22     const uint32_t m = 0x5bd1e995;
23     const int r = 24;
24
25     /* Initialize the hash to a 'random' value */
26
27     uint32_t h = seed ^ len;
28
29     /* Mix 4 bytes at a time into the hash */
30
31     const unsigned char * data = (const unsigned char *)key;
32
33     while(len >= 4)
34     {
35         uint32_t k = *(uint32_t*)data;
36
37         k *= m;
38         k ^= k >> r;
39         k *= m;
40
41         h *= m;
42         h ^= k;
43
44         data += 4;
45         len -= 4;
46     }
```

这代码的相似率确实极高，找一个 Python 库验证一下，比如 [murmurhash2](https://pypi.org/project/murmurhash2/)

<<https://pypi.org/project/murmurhash2/>>，它的使用方式很简单，参数一是 `bytes` 类型的输入，参数二是种子，或者说初始值，像下面这样。

```
1 from murmurhash2 import murmurhash2
2
3
4 SEED = 0
   print(murmurhash2(b'key', SEED))
```

我把 Hook 到的结果放到 cyberchef 里转成 hexstring

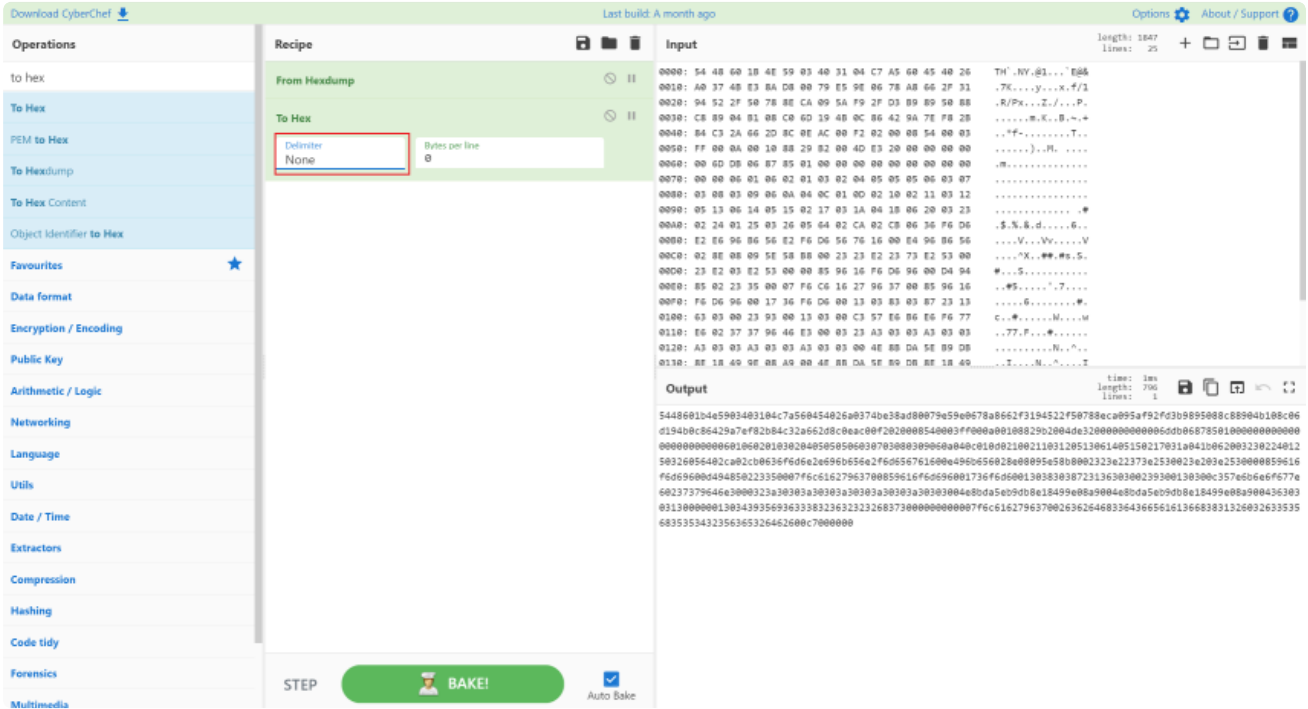


```
[09d0] 0xb9a43a0e: "beq #0xb9a43a24"

m0x4020c1c0 0x18e

>-----<
[21:02:12 689]RW@0x402dc1c0, md5=15de197b3e8e039ed6e063ed72cb66fd, hex=5448601b4e5903403104c7a5604540
size: 398
0000: 54 48 60 1B 4E 59 03 40 31 04 C7 A5 60 45 40 26 TH`.NY.@1...`E@&
0010: A0 37 4B E3 8A D8 00 79 E5 9E 06 78 A8 66 2F 31 .7K....y...x.f/1
0020: 94 52 2F 50 78 8E CA 09 5A F9 2F 03 B9 89 50 88 .R/Px...Z./...P.
0030: C8 89 04 B1 08 C0 6D 19 4B 0C 86 42 9A 7E F8 2B .....m.K..B..~.+
0040: 84 C3 2A 66 2D 8C 0E AC 00 F2 02 00 08 54 00 03 ..*f-.....T..
0050: FF 00 0A 00 10 88 29 B2 00 4D E3 20 00 00 00 00 .....).M. ....
0060: 00 6D DB 06 87 85 01 00 00 00 00 00 00 00 00 00 .m.....
0070: 00 00 06 01 06 02 01 03 02 04 05 05 05 06 03 07 .....
0080: 03 08 03 09 06 0A 04 0C 01 0D 02 10 02 11 03 12 .....
0090: 05 13 06 14 05 15 02 17 03 1A 04 1B 06 20 03 23 ..... .#
00A0: 02 24 01 25 03 26 05 64 02 CA 02 CB 06 36 F6 D6 .$.%.&.d.....6..
00B0: E2 E6 96 B6 56 E2 F6 D6 56 76 16 00 E4 96 B6 56 ....V...Vv....V
00C0: 02 8E 08 09 5E 58 B8 00 23 23 E2 23 73 E2 53 00 ....^X..##.#s.S.
00D0: 23 E2 03 E2 53 00 00 85 96 16 F6 D6 96 00 04 94 #...S.....
00E0: 85 02 23 35 00 07 F6 C6 16 27 96 37 00 85 96 16 ..#5.....'.7....
00F0: F6 D6 96 00 17 36 F6 D6 00 13 03 83 03 87 23 13 .....6.....#.
0100: 63 03 00 23 93 00 13 03 00 C3 57 E6 B6 E6 F6 77 c..#.....W....w
0110: E6 02 37 37 96 46 E3 00 03 23 A3 03 03 A3 03 03 ..77.F...#.....
0120: A3 03 03 A3 03 03 A3 03 03 00 4E 8B DA 5E B9 DB .....N..^..
0130: 8E 18 49 9E 08 A9 00 4E 8B DA 5E B9 DB 8E 18 49 ..I....N..^....I
0140: 9E 08 A9 00 43 63 03 03 13 00 00 00 13 03 43 93 ....Cc.....C.
0150: 56 93 63 33 83 23 63 23 23 26 83 73 00 00 00 00 V.c3.#c##&.s....
0160: 00 07 F6 C6 16 27 96 37 00 26 36 26 46 83 36 43 .....'.7.&6&F.6C
0170: 66 56 16 13 66 83 83 13 26 03 26 33 53 56 83 53 fV..f...&.&3SV.S
0180: 53 43 23 56 36 53 26 46 26 00 C7 00 00 00 SC#V6S&F&.....
```

转换过程



验证的 Python 代码如下

```
1  from murmurhash2 import murmurhash2
2
3  SEED = 0
4  input = bytes.fromhex("5448601b4e5903403104c7a560454026a0374be38ad80079e59e0678a8662f3194522f50788eca095af92fd3b9895088c88904b108c06
5  print("ret:"+hex(murmurhash2(input, SEED)))
```

运行结果，完全对的上。

```
1  ret:0xde17f654
```

因此我们分析出这四字节的来源了，它是名为 murmurhash2 的哈希算法的处理结果，输入源是对一个 0x18e 长度的字节数组，我们后续再看这个数组的出处，现在换下一个思路。

## 2.2 思路二

我们也可以直接在 traceCode 文本里搜索 0xde17f654，因为它很可能曾流经过什么寄存器，然后被我们的 trace 打印记录下来。



起始页 trace1.log x

```

[19:44:29 612][83ea0123] 0xb9a43a20: "eor.w r3, r3, r1, lsl #8" r3=0x9d035ac5 r1=0x0 => r3=0x9d035ac5
[19:44:29 612][0078] 0xb9a43a24: "ldrb r0, [r0]" r0=0x402dc34c => r0=0x0
[19:44:29 612][5840] 0xb9a43a26: "eors r0, r3" r0=0x0 r3=0x9d035ac5 => r0=0x9d035ac5
[19:44:29 612][00fb02f3] 0xb9a43a28: "mul r3, r0, r2" r0=0x9d035ac5 r2=0x5bd1e995 => r3=0x926621a9
[19:44:29 612][83ea5330] 0xb9a43a2c: "eor.w r0, r3, r3, lsr #13" r3=0x926621a9 => r0=0x9262b298
[19:44:29 612][5043] 0xb9a43a30: "muls r0, r2, r0" r0=0x9262b298 r2=0x5bd1e995 => r0=0x9262b298
[19:44:29 613][80ead030] 0xb9a43a32: "eor.w r0, r0, r0, lsr #15" r0=0x9262b298 => r0=0x9262b298
[19:44:29 613][d0bd] 0xb9a43a36: "pop {r4, r6, r7, pc}" sp=0xbffffecb8
[19:44:29 613][caf80000] 0xb9a43a2be: "str.w r0, [sl]" r0=0x9262b298 sl=0xbffff100
[19:44:29 613][6ff01801] 0xb9a43a2c2: "mvn r1, #0x18" => r1=0xffffffe7
[19:44:29 613][95f82400] 0xb9a43a2c6: "ldrb.w r0, [r5, #0x24]" r5=0xb9ab0e10 => r0=0x78
[19:44:29 613][4843] 0xb9a43a2ca: "muls r0, r1, r0" r0=0x78 r1=0xffffffe7 => r0=0xfffff448
[19:44:29 614][c1b2] 0xb9a43a2cc: "uxtb r1, r0" r0=0xfffff448 => r1=0x48
[19:44:29 614][a8f10900] 0xb9a43a2ce: "sub.w r0, r8, #9" r8=0x19 => r0=0x10
[19:44:29 614][4829] 0xb9a43a2d2: "cmp r1, #0x48" r1=0x48 => cpsr: N=0, Z=1, C=1, V=0
[19:44:29 614][d3d0] 0xb9a43a2d4: "beq #0xb9a3a27e"
[19:44:29 614][796c] 0xb9a43a27e: "ldr r1, [r7, #0x44]" r7=0xb9ab0ec0 => r1=0x7002cc38
[19:44:29 614][d7f89820] 0xb9a43a280: "ldr.w r2, [r7, #0x98]" r7=0xb9ab0ec0 => r2=0x68f9871a
[19:44:29 615][ab7e] 0xb9a43a284: "ldrb r3, [r5, #0x1a]" r5=0xb9ab0e10 => r3=0xce
[19:44:29 615][ee7f] 0xb9a43a286: "ldrb r6, [r5, #0x1f]" r5=0xb9ab0e10 => r6=0x47
[19:44:29 615][1143] 0xb9a43a288: "orrs r1, r2" r1=0x7002cc38 r2=0x68f9871a => r1=0x78fbcf3a
[19:44:29 615][06ea0302] 0xb9a43a28a: "and.w r2, r6, r3" r6=0x47 r3=0xce => r2=0x46
[19:44:29 615][18046] 0xb9a43a28e: "mov r8, r0" r0=0x10 => r8=0x10
[19:44:29 615][1828] 0xb9a43a290: "cmp r0, #0x18" r0=0x10 => cpsr: N=1, Z=0, C=0, V=0
[19:44:29 615][40f33380] 0xb9a43a292: "ble.w #0xb9a3a2fc"
[19:44:29 616][d7f89400] 0xb9a43a2fc: "ldr.w r0, [r7, #0x94]" r7=0xb9ab0ec0 => r0=0xad1e3ccb
[19:44:29 616][d7f8c410] 0xb9a43a300: "ldr.w r1, [r7, #0xc4]" r7=0xb9ab0ec0 => r1=0xc653389c
[19:44:29 616][0840] 0xb9a43a304: "ands r0, r1" r0=0xad1e3ccb r1=0xc653389c => r0=0x84123888
[19:44:29 616][a042] 0xb9a43a306: "cmp r0, r4" r0=0x84123888 r4=0x84123888 => cpsr: N=0, Z=1, C=1, V=0
[19:44:29 616][f2d1] 0xb9a43a308: "bne #0xb9a3a2f0"
[19:44:29 616][b8f1140f] 0xb9a43a30a: "cmp.w r8, #0x14" r8=0x10 => cpsr: N=1, Z=0, C=0, V=0
[19:44:29 617][b8f6ada] 0xb9a43a30e: "bge.w #0xb9a3a26c"
[19:44:29 617][b8f1110f] 0xb9a43a312: "cmp.w r8, #0x11" r8=0x10 => cpsr: N=1, Z=0, C=0, V=0
[19:44:29 617][11d1] 0xb9a43a316: "bne #0xb9a3a33c"
[19:44:29 617][4046] 0xb9a43a33c: "mov r0, r8" r8=0x10 => r0=0x10
[19:44:29 617][297a] 0xb9a43a33e: "ldrb r1, [r5, #8]" r5=0xb9ab0e10 => r1=0x33
[19:44:29 617][3629] 0xb9a43a340: "cmp r1, #0x36" r1=0x33 => cpsr: N=1, Z=0, C=0, V=0
[19:44:29 618][3ff4ceaf] 0xb9a43a342: "beq.w #0xb9a3a2e2"

```

查找结果

地址	值
已找到 6 个 '0x9262b298'.	
行 68010	[19:44:29 613][80ead030] 0xb9a43a32: "eor.w r0, r0, r0, lsr #15" r0=0x9262b298 => r0=0x9262b298
行 68012	[19:44:29 613][caf80000] 0xb9a43a2be: "str.w r0, [sl]" r0=0x9262b298 sl=0xbffff100
行 68879	[19:44:29 744][8058] 0xb9a333ba: "ldr r0, [r0, r2]" r0=0xbffffec7 r2=0x290 => r0=0x9262b298
行 68881	[19:44:29 744][cef80000] 0xb9a333c0: "str.w r0, [lr]" r0=0x9262b298 lr=0xbffff598
行 68977	[19:44:29 758][def80020] 0xb9a33810: "ldr.w r2, [lr]" lr=0xbffff598 => r2=0x9262b298
行 68979	[19:44:29 759][c250] 0xb9a33818: "str r2, [r0, r3]" r2=0x9262b298 r3=0x4b0

只有六个地方出现，最早的是 0xb9a43a32，偏移地址是 0x6EA32，直接就定位到目标函数了。

IDA View-A Pseudocode-A Findcrypt results Hex View-1 Structures

```

1 unsigned int __fastcall sub_6E9EC(unsigned __int8 *a1, unsigned int a2, int a3)
2 {
3     unsigned int v3; // r3
4     int v4; // r4
5
6     v3 = a3 ^ a2;
7     while ( a2 >= 4 )
8     {
9         v4 = *(_DWORD *)a1;
10        a1 += 4;
11        a2 -= 4;
12        v3 = (0x5BD1E995 * v3) ^ (0x5BD1E995 * ((0x5BD1E995 * v4) ^ ((unsigned int)(0x5BD1E995 * v4) >> 24)));
13    }
14    switch ( a2 )
15    {
16        case 1u:
17            goto LABEL_9;
18        case 2u:
19            goto LABEL_8;
20    LABEL_8:
21        v3 ^= a1[1] << 8;
22    LABEL_9:
23        v3 = 0x5BD1E995 * (*a1 ^ v3);
24        return (0x5BD1E995 * (v3 ^ (v3 >> 13))) ^ ((0x5BD1E995 * (v3 ^ (v3 >> 13))) >> 15);
25    case 3u:
26        v3 ^= a1[2] << 16;
27        goto LABEL_8;
28    }
29    return (0x5BD1E995 * (v3 ^ (v3 >> 13))) ^ ((0x5BD1E995 * (v3 ^ (v3 >> 13))) >> 15);
30 }

```

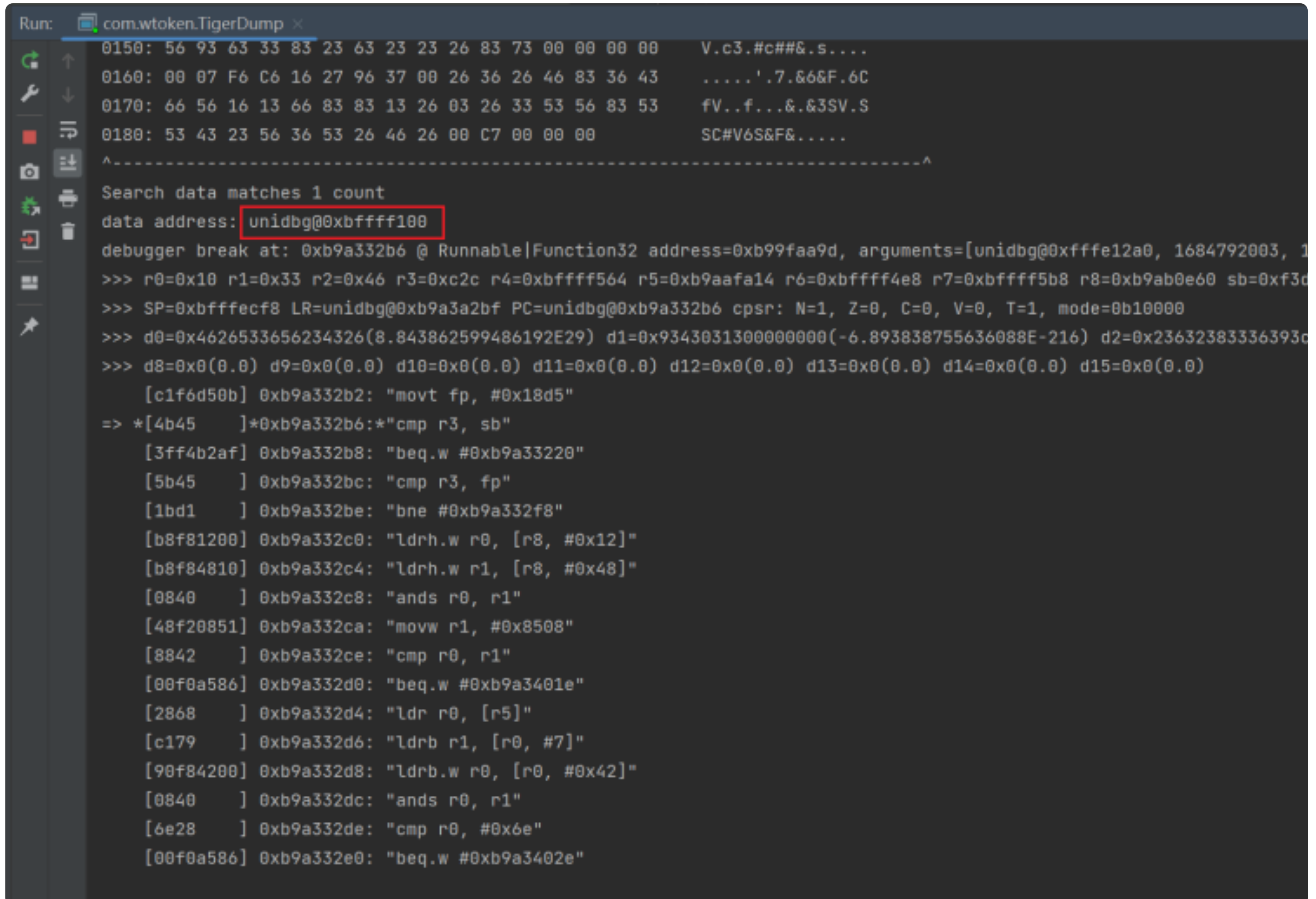
后面的流程同思路一。

## 2.3 思路三

我们用 dataSearch 搜索这四个字节所处的内存，这不是一个特别好的主意，因为 4 字节太短了，可能因为各种原因随机出现在内存里，但我们还是想碰碰运气。

```
1 new DataSearch(emulator, "54F617DE");
```

运行



```
Run: com.wtoker.TigerDump x
0150: 56 93 63 33 83 23 63 23 23 26 83 73 00 00 00 00 V.c3.#c##&.s....
0160: 00 07 F6 C6 16 27 96 37 00 26 36 26 46 83 36 43 .....'.7.66&F.6C
0170: 66 56 16 13 66 83 83 13 26 03 26 33 53 56 83 53 fV..f...&.63SV.S
0180: 53 43 23 56 36 53 26 46 26 00 C7 00 00 00 00 SC#V6S&F&.....
^-----^
Search data matches 1 count
data address: 0xbffff100
debugger break at: 0xb9a332b6 @ Runnable|Function32 address=0xb99faa9d, arguments=[unidbg@0xffffe12a0, 1684792003, 1
>>> r0=0x10 r1=0x33 r2=0x46 r3=0xc2c r4=0xbffff564 r5=0xb9aafa14 r6=0xbffff4e8 r7=0xbffff5b8 r8=0xb9ab0e60 sb=0xf3c
>>> SP=0xbfffcfc8 LR=unidbg@0xb9a3a2bf PC=unidbg@0xb9a332b6 cpsr: N=1, Z=0, C=0, V=0, T=1, mode=0b10000
>>> d0=0x4626533656234326(8.843862599486192E29) d1=0x9343031300000000(-6.893838755636088E-216) d2=0x23632383336393c
>>> 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)
[c1f6d50b] 0xb9a332b2: "movt fp, #0x18d5"
=> *[4b45 ] 0xb9a332b6: "cmp r3, sb"
[3ff4b2af] 0xb9a332b8: "beq.w #0xb9a33220"
[5b45 ] 0xb9a332bc: "cmp r3, fp"
[1bd1 ] 0xb9a332be: "bne #0xb9a332f8"
[b8f81200] 0xb9a332c0: "ldrh.w r0, [r8, #0x12]"
[b8f84810] 0xb9a332c4: "ldrh.w r1, [r8, #0x48]"
[0840 ] 0xb9a332c8: "ands r0, r1"
[48f20851] 0xb9a332ca: "movw r1, #0x8508"
[8842 ] 0xb9a332ce: "cmp r0, r1"
[00f0a586] 0xb9a332d0: "beq.w #0xb9a3401e"
[2868 ] 0xb9a332d4: "ldr r0, [r5]"
[c179 ] 0xb9a332d6: "ldrb r1, [r0, #7]"
[90f84200] 0xb9a332d8: "ldrb.w r0, [r0, #0x42]"
[0840 ] 0xb9a332dc: "ands r0, r1"
[6e28 ] 0xb9a332de: "cmp r0, #0x6e"
[00f0a586] 0xb9a332e0: "beq.w #0xb9a3402e"
```

发现这里运气不错，直接定位到了 0xbffff100，后续就是同思路一那样，追踪这个地址的来源，然后很快就会找到目标函数。

## 三、尾声

我们可以对这三个思路做简单的讨论，读者可能会觉得这里思路二最好，直接方便，但待分析的数据并不总是那么短，如果是较长的数据源，用思路三可以更快更方便的定位到生成点。在思路一和思路二之间，如果 trace 文本量数以千万行计，并且所追踪的数据比较“平凡”，在 trace 文本里出现成千上万次，那么思路一可能会更好用。

总结就是没有哪个思路是绝对的最优解，要根据场景和样本做具体分析。

下篇我们继续做样本分析。

