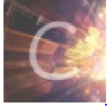
看雪CTF2017第六题 Ericky-apk writeup(安卓so逆向)

原创

anhkgg 于 2017-06-14 09:12:32 发布 4527 收藏 2
分类专栏： 原创 文章标签： android 反编译 apk ctf 看雪
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/angelxf/article/details/73198399
版权

原创 专栏收录该内容
47 篇文章 2 订阅
订阅专栏

概述

题目入口：http://ctf.pediy.com/game-fight-36.htm

本题是安卓cm，目测肯定需要调试so。

准备工具：

1. ApkIde改之理（其他类似的也行，能够反编译apk，得到jar,so等）
2. IDA（用于调试so），需要6.x以上，忘了是x几，我用的6.6
3. adb(ApkIde改之理就有)

反编译

将6-Ericky kanxue.apk拖进ApkIDE改之理，等待编译（没有加壳），ok。

在右侧树结构栏中，找到smali->android->com->miss->rfchen，列表中就是java层的主要函数。

点击MainActivity.smali，然后点击工具栏中jd-gui.exe，抓到java源码查看。

```
public class MainActivity extends Activity
{
  private EditText ´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊˊˊˊ´ˊˊˊ´ˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊ´´´ = null;
  private Button ´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊˊ´ˊ´ˊˊ´ˊˊ´ˊˊˊˊ´ˊ´ˊ´ˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊ´ˊˊˊ´ˊˊˊ´ˊ´ˊ´ˊˊ´ˊ´ˊ´ = null;

  protected void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130968603);
    this.´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊˊ´ˊ´ˊˊ´ˊˊ´ˊˊˊˊ´ˊ´ˊ´ˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊ´ˊˊˊ´ˊˊˊ´ˊ´ˊ´ˊˊ´ˊ´ˊ´ = ((Button)findViewById(2131427415));
    this.´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊˊˊˊ´ˊˊˊ´ˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊ´´´ = ((EditText)findViewById(2131427416));
    this.´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊˊ´ˊ´ˊˊ´ˊˊ´ˊˊˊˊ´ˊ´ˊ´ˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊ´ˊˊˊ´ˊˊˊ´ˊ´ˊ´ˊˊ´ˊ´ˊ´.setOnClickListener(new View.OnClickListener()
    {
      public void onClick(View paramView)
      {
        MainActivity.this.´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊˊˊˊ´ˊˊˊ´ˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊ´´´();
      }
    });
  }

  public void ´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊˊˊˊ´ˊˊˊ´ˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊ´´´()
  {
    String str = this.´ˊ´ˊˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊˊˊˊ´ˊˊˊ´ˊˊ´ˊˊ´ˊ´ˊˊ´ˊˊˊ´ˊ´ˊˊˊ´ˊˊ´ˊˊˊˊ´ˊˊ´ˊ´ˊˊ´ˊ´´´.getText().toString().trim();
    StringBuilder localStringBuilder = new StringBuilder();
    localStringBuilder.append(str);
    if (utils.check(localStringBuilder.toString().trim()))
    {
      Toast.makeText(this, MainActivity.1.utils.dbcb("⍰⍰⍰毀あ嵩0 ,("⍰ ).show();
      return;
    }
    Toast.makeText(this, MainActivity.1.utils.dbcb("⍰⍰⍰毀ぞ崌⍰� 劄�晌鬆屑諨"), 0).show();
  }
}
```

这混淆的函数名我也是醉了，但这都不重要。输入key之后，然后点击按钮，进入OnClick，调用了上面代码中第二个函数（什么？我怎么知道的，因为它们哪个…点号…的函数名相同！！）。

然后调用了utils.check来验证，成功提示！这里成功和错误提示的字符串做过变换，通过utils.dbcb解密，不细看了，不重要！

进入utils.java，看到加载了so，调用的是这个so的导出函数，看反编译目录lib/armeabi-v7a（只提供了arm的so，要有个x86的好了），知道这个so是librf-chen.so。

```
//典型的NDK调用，查查就知道了！
package com.miss.rfchen;

public class utils
{
  static
  {
    System.loadLibrary(MainActivity.1.utils.dbcb("ۛ▯雒▯�蹬"));
  }

  public static native boolean check(String paramString);
}
```

那么重点来了，要分析librf-chen.so的check函数，才能搞定此题。

准备调试

早上提前学习了一下so调试方法，找到了看雪安卓大神的教程，就是参考中的IDA动态调试技术，然后用上了，很好用！

跟着走

下面开始照着做。

1. 连上手机（或者模拟器），使用adb devices看看成功连上没有
2. adb push …/dbgsrv/android_server /sdcard/sv，教程是直接放入/data/data，一般权限不够
3. 然后进入shell，adb shell，输入su，获得root权限，然后cp /sdcard/sv /data/data/sv
4. 修改sv权限，chmod 777 /data/data/sv
5. 运行sv，/data/data/sv，默认监听到23946端口，Listening on port #23946。这步有个细节，不能直接adb shell /data/data/sv，这样权限不够，无法读取到进程信息，需要adb shell; su; /data/data/sv
6. 再开一个cmd，然后运行adb forward tcp:23946 tcp:23946
7. 运行一个idaq.exe，然后在菜单debugger->attach->remote Armlinux/android debugger，输入localhost, 23946,ok
8. 弹出进程框，按下Alt+T，输入chen，搜索到1808 [32] com.miss.rfchen，ok
9. F9运行

```
\ApkIDEz>    .\adb.exe shell
shell@your phone:/ $ su
su
root@your phone:/ # /data/data/sv
/data/data/sv
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2014
```

在界面中输入key，然后点击按钮，此时librf-chen.so才加载，然后ctrl+s，alt+t，输入librf找到librf-chen.so的基地址信息(记为base)，记下来。

用另一个ida打开librf-chen.so，找到check导出函数的偏移地址00002814，计算base+00002814，然后g在IDA调试器中输入该地址，加上断点。

```
check(_JNIEnv *,_jclass *,_jstring *) 00002814
```

IDA基本调试快捷键和OD一样:

```
F9: 运行
F8: 步过
F7: 步入
```

F9，跑起来，然后再次点击按钮，就断下来，进入了check。

下面就是跟和调试的过程了，看数据，看流程，分析算法！

arm汇编基础

得提前有个准备，看看arm指令，了解基本的指令，函数调用方式，下面列几个，更多的就看参考中的文章了

```
MOVS 同x86的mov
LDR 加载内存数据到寄存器
STR 寄存器数据存入内存
B/BL 跳转/函数调用
TST/CMP 比较
ADD/SUB 加/减
```

然后最主要的，函数调用的参数传递。arm默认使用的fastcall，通过r0,r1,r2,r3传递参数，超过4个参数，使用堆栈传递，r0也保存返回值。

关键点跟踪

在check断下之后，先是一段数据初始化，先滤过，然后blt sub_2874，进入关键函数

然后看到通过MOVS，STR将一些字符放入了内存。

```
.text:0000288A 000 01 60                                         STR             R1, [R0]
.text:0000288C 000 4A 20                                         MOVS            R0, #'J'
.text:0000288E 000 79 21                                         MOVS            R1, #'y'
.text:00002890 000 AD F8 22 00                                   STRH.W          R0, [SP,#arg_22]
.text:00002894 000 AD F8 24 10                                   STRH.W          R1, [SP,#arg_24]
.text:00002898 000 75 21                                         MOVS            R1, #'u'
.text:0000289A 000 AD F8 26 10                                   STRH.W          R1, [SP,#arg_26]
.text:0000289E 000 33 21                                         MOVS            R1, #'3'
```

接着就看让我恐惧的一幕，b loc_2898开始各种跳转，指令操作，然后刚跳完又是一个b xxx，接着各种跳转，毫无疑问，这是一段花指令了。

```
.text:0000289E                 B               loc_2898
```

花指令结构

经过多次跟踪，恶心到快吐的时候，终于看出话指令的基本结构了：

```
.text:00002BE8                  PUSH.W          {R4-R10,LR}
.text:00002BEC                  POP.W           {R4-R10,LR}
.text:00002BF0                  B               sub_2C1A                    ; 开始
 PUSH.W          {R4-R10,LR}
.text:00002BEC BD E8 F0 47                                    POP.W           {R4-R10,LR}
.text:00002BF0 13 E0                                          B               sub_2C1A
 ---------------------------------------------------------------------------
.text:00002BF2 BD E8 F0 47                                    POP.W           {R4-R10,LR}
.text:00002BF6 05 E0                                          B               sub_2C04
---------------------------------------------------------------------------
.text:00002BF8 00 F1 01 00                                    ADD.W           R0, R0, #1
.text:00002BFC 0A E0                                          B               loc_2C14
---------------------------------------------------------------------------
.text:00002BFE 1B 46                                          MOV             R3, R3
.text:00002C00 0E E0                                          B               loc_2C20
=======================================
.text:00002C02 10 E0                                          B               sub_2C26 ; 跳到快执行的位置
=======================================
.text:00002C04 B1 B5                                          PUSH            {R0,R4,R5,R7,LR}
.text:00002C06 01 E0                                          B               loc_2C0C
---------------------------------------------------------------------------
.text:00002C08 12 46                                          MOV             R2, R2
.text:00002C0A 01 E0                                          B               loc_2C10
.text:00002C0C 82 B0                                          SUB             SP, SP, #8
.text:00002C0E FB E7                                          B               loc_2C08
---------------------------------------------------------------------------
.text:00002C10 02 B0                                          ADD             SP, SP, #8
.text:00002C12 F1 E7                                          B               loc_2BF8
---------------------------------------------------------------------------
.text:00002C14 A0 F1 01 00                                    SUB.W           R0, R0, #1
.text:00002C18 F1 E7                                          B               loc_2BFE
=======================================
.text:00002C1A 2D E9 F0 47                                    PUSH.W          {R4-R10,LR}
.text:00002C1E E8 E7                                          B               loc_2BF2
---------------------------------------------------------------------------
.text:00002C20 BD E8 B1 40                                    POP.W           {R0,R4,R5,R7,LR}
.text:00002C24 ED E7                                          B               sub_2C02
=======================================
.text:00002C26 2D E9 F0 47                                    PUSH.W          {R4-R10,LR}
.text:00002C2A BD E8 F0 47                                    POP.W           {R4-R10,LR}
.text:00002C2E FF E7                                          B               sub_2C30 ; 进入有效代码，一般是接着的
地址

.text:00002C30                  PUSH            {R0,R4,R5,R7,LR} ; 开始一般会有一段对称没啥作用的话指令
.text:00002C32                  SUB             SP, SP, #8
.text:00002C34                  MOV             R2, R2
.text:00002C36                  ADD             SP, SP, #8
.text:00002C38                  ADD.W           R0, R0, #1
.text:00002C3C                  SUB.W           R0, R0, #1
.text:00002C40                  MOV             R3, R3
.text:00002C42                  POP.W           {R0,R4,R5,R7,LR}
.text:00002C46                  ADD.W           R1, R1, #1
.text:00002C4A                  SUB.W           R1, R1, #1
.text:00002C4E                  STRH.W          R0, [SP,#arg_30]
.text:00002C52                  MOVS            R0, #0x44
.text:00002C54                  PUSH.W          {R4-R10,LR}
.text:00002C58                  POP.W           {R4-R10,LR}
.text:00002C5C                  B               sub_2C86
```

特征：

1. 每跳转一个分支，基本都要一段花（记为A段），就是从上面代码中注释开始的问题
2. 进行几个跳转后，到了结束位置，跳入有效代码
3. 有效代码开头一般也有加一段花（记为B段）
4. 在A段话指令中，指令地址是向下增长的，也就是A开始往下拉一段，就能找到结束位置
5. B端一般无跳转，但是对称代码有多又少

所以根据特征，去除话指令也挺方便，我使用的IDA的patch功能手工去花的，脚本牛可以写个脚本。

所有花指令填充的00 bf（NOP），然后就可以F5了。

关键点跟踪2

然后接着调试跟踪。

接着上面，后续会接着向该段内存填充字符（非直接填充，还有个段算法，根据初始话的0x20的值来做的），我没有仔细跟踪算法了，通过对些内存关键点下断，然后跳出循环位置下断，下面0000357A就是循环位置，如此多次之后，循环结束。

```
.text:00003576 000 B4 F1 FF 3F                                      CMP.W           R4, #0xFFFFFFFF
.text:0000357A 000 3F F7 74 AD                                      BGT.W           loc_3066
```

查看该内存数据：

```
 5F019020 4A 00 79 00 75 00 33 00  43 00 4A 00 6C 00 56 00  J.y.u.3.C.J.l.V.
 5F019030 44 00 53 00 47 00 51 00  21 00 0A 00 00 00 00 00  D.S.G.Q.!.......
```

接着跳过一段花之后，调用了bl sub_19FC，跟入，发现结果和刚才那段基本一直，也是将字符写入内存，并且内存就是刚才那段，只是每次都有一个1偏移。

```
.text:0000364A 000 FE F7 D7 F9                                      BL              sub_19FC
...
librf_chen.so:5EFFB52E                  ORR.W           R3, LR, R2,LSL#1
librf_chen.so:5EFFB532                  LDRB.W          R0, [R8,R5,LSL#1]
librf_chen.so:5EFFB536                  ADDS            R2, #1
librf_chen.so:5EFFB538                  STRB.W          R0, [R12,R3] ；也是前面的位置，但是加了个1偏移
```

同样，结束之后，查看内存，通过后面分析，知道这段字符就是key加密变换之后要对比的字符串。

```
5ED12020  4A 50 79 6A 75 70 33 65  43 79 4A 6A 6C 6B 56 36  JPyjup3eCyJjlkV6
5ED12030  44 6D 53 6D 47 48 51 3D  21 21 0A 0A 00 00 00 00  DmSmGHQ=!!......
```

子过程返回之后，接着b进入另一段。调了这么久，我们输入的key去哪里了？下面来了！

```
text:00003680 000 D9 F8 00 00                                      LDR.W           R0, [R9] 之前传入的参_JNIEnv
.text:00003684 000 41 46                                            MOV             R1, R8 之前传入的参数，_jclass
.text:00003686 000 00 22                                            MOVS            R2, #0
.text:00003688 000 00 24                                            MOVS            R4, #0
.text:0000368A 000 D0 F8 A4 32                                      LDR.W           R3, [R0,#0x2A4] libdvm.so:_Z20d
vmDecodeIndirectRefP6ThreadP8_jobject+F55
.text:0000368E 000 48 46                                            MOV             R0, R9 this指针
.text:00003690 000 98 47                                            BLX             R3 libdvm.so:_Z20dvmDecodeIndir
ectRefP6ThreadP8_jobject+F55，返回输入的key的内存
```

先来看看check接口：

```
check(_JNIEnv *,_jclass *,_jstring *) 00002814
```

check参数在刚进入就被保存了，现在在00003680位置取出来，返回了我们输入的key到R0中（看注释）。

```
5DC4BEC0  31 32 33 34 35 36 00 40  10 00 00 00 4B 00 00 00  123456.@....K...
```

然后，又调用了一个子过程来处理key，我这里先没有跟入，直解F8，看了返回值

```
.text:00003792 000 16 F0 09 FB                                    BL              sub_19DA8
.text:00003796 000 01 46                                          MOV             R1, R0  ; key
.text:00003798 000 DF F8 A4 04                                    LDR.W           R0, =(unk_20020 - 0x38D2)
```

```
65 4B 2F 30 36 38 71 52  00 00 00 00 C0 BE C4 5D  eK/068qR
```

基本确认是加密函数，然后又把该结果和JPyjup3eCyJjlkV6DmSmGHQ=!!进行对比。

```
.text:000038CE 000 78 44                                          ADD             R0, PC ; 保存了JPyjup3eCyJjlkV6
DmSmGHQ=!!
.text:000038D0
.text:000038D0                                    AGAIN_18                                ; CODE XREF: sub_2874+1
0D
.text:000038D0 000 0A 5D                                          LDRB            R2, [R1,R4]; R1保存了eK/068qR 取
出一个字符
.text:000038D2 000 03 5D                                          LDRB            R3, [R0,R4]; 取出一个字符
.text:000038D4 000 93 42                                          CMP             R3, R2
.text:000038D6 000 40 F0 6B 80                                    BNE.W           loc_39B0 ; jmp 3A1A
.text:000038DA 000 01 34                                          ADDS            R4, #1

.text:00003942 000 18 2C                                          CMP             R4, #0x18
.text:00003944 000 C4 D1                                          BNE             AGAIN_18

.text:000039AC 000 01 20                                          MOVS            R0, #1
.text:000039AE 000 3B E1                                          B               loc_3C28

.text:00003A86 000 00 28                                          CMP             R0, #0
.text:00003A88 000 00 F0 67 80                                    BEQ.W           TAG_FAILED
.text:00003C26 000 00 20                                          MOVS            R0, #0
```

取出一个字符进行比较，不同则跳转，相同R4加1，继续比价直到超过0x18（也就是加密结果长度0x18），都相同了R0=1

看看不同时跳转的代码，sub_27C8是一个类似鱼strstr的代码，我本以为加密之后结果可以部分匹配也行，结果我错了，作者坑人，因为这个sub_27C8就算返回1，也就是部分匹配成功了，也会进入00003C26，R0=0。

```
.text:00003A1A 000 78 44                                          ADD             R0, PC  ; result
.text:00003A1C 000 FE F7 D4 FE                                    BL              sub_27C8 ; 在result中找key，找到
匹配的一段，返回匹配位置，否则返回0
```

所以加密结果必须是0x18，和JPyjup3eCyJjlkV6DmSmGHQ=!!完全匹配(0x18字节)

算法

现在重新跟入加密子过程sub_19DA8，看看是怎么个算法。

```
.text:00019DA8                                  sub_19DA8                     ; CODE XREF: sub_2874+F1E
.text:00019DA8
.text:00019DA8                                  var_10          = -0x10
.text:00019DA8
.text:00019DA8 000 2D E9 F0 43                                  PUSH.W          {R4-R9,LR}
.text:00019DAC 01C 03 AF                                        ADD             R7, SP, #0xC
.text:00019DAE 01C AD F5 81 6D                                  SUB.W           SP, SP, #0x408
.text:00019DB2 424 81 B0                                        SUB             SP, SP, #4
.text:00019DB4 428 81 46                                        MOV             R9, R0
.text:00019DB6 428 DF F8 5C 05                                  LDR.W           R0, =(__stack_chk_guard_ptr - 0
x19DBE)
.text:00019DBA 428 78 44                                        ADD             R0, PC ; __stack_chk_guard_ptr
.text:00019DBC 428 00 68                                        LDR             R0, [R0] ; __stack_chk_guard
.text:00019DBE 428 00 68                                        LDR             R0, [R0]
.text:00019DC0 428 47 F8 10 0C                                  STR.W           R0, [R7,#var_10]
.text:00019DC4 428 00 F0 AA FA                                  BL              sub_1A31C ;
.text:00019DC4                                                                           ; 返回199319124851!
.text:00019DC8 428 80 46                                        MOV             R8, R0
.text:00019DCA 428 48 46                                        MOV             R0, R9
```

先通过sub_1A31C子函数返回了一串字符199319124851!，算法和生成JPyjup3eCyJjlkV6DmSmGHQ=!!字符类似，不再细说。

```
.text:00019F80 428 20 46                                        MOV             R0, R4  ; size
.text:00019F82 428 E7 F7 14 EC                                  BLX             malloc //分配内存来保存第一次加密
结果
.text:00019F86 428 21 46                                        MOV             R1, R4
.text:00019F88 428 05 46                                        MOV             R5, R0
text:00019FF0 428 E7 F7 E2 EB                                  BLX             __aeabi_memclr；清零
.text:00019FF4 428 6C 46                                        MOV             R4, SP
.text:00019FF6 428 08 21                                        MOVS            R1, #8  ; a2
.text:00019FF8 428 20 46                                        MOV             R0, R4  ; result
.text:00019FFA 428 42 46                                        MOV             R2, R8  ; str

.text:0001A0C8 428 EB F7 8C FA                                  BL              sub_55E4 ; str = "199310124851!
"
.text:0001A0C8                                                                           ; a2 长度+2
.text:0001A0CC 428 20 46                                        MOV             R0, R4  ; p
.text:0001A0CE 428 31 46                                        MOV             R1, R6  ; key_len
.text:0001A0D0 428 4A 46                                        MOV             R2, R9  ; key
.text:0001A0D2 428 2B 46                                        MOV             R3, R5  ; pKeyResult
```

然后分配了一段内存，用于保存第一次加密的key结果。
调用sub_55E4，将199310124851!通过变换放入一个8字节+0x100*4的数组(初始化为0-0x100)空间,挺绕的，由于这个函数跟key没有多大关系，所以咩必要细究是怎么做的，可以直接将计算后内存dump出来用后面的逆运算（其实我没用上）。

```
.text:0001A13A 428 EA F7 A0 FA                                  BL              sub_467E;第一次加密变换
.text:0001A13E 428 28 46                                        MOV             R0, R5
```

然后sub_467E进行第一次加密变换，将key和前面的8字节+0x100*4的数组组队的xor，细节直接看代码(完整的我会放idb)：

```
v4 = p->unk_0;
  v5 = p->unk_4;
  if ( key_len >> 3 )                         // 8 >> 3 = 1
  {
    v6 = -(key_len >> 3);                     // -2
    v7 = pKeyResult + 8 * (key_len >> 3);     // 2*8
    key1 = key;
    do
    {
      ++v6;
      v9 = (unsigned __int8)(v4 + 1);         // 1
      v10 = p->index[v9];                     // p->Index[1]
      v11 = v5 + v10;                         // 0+p->Index[1]
      v12 = p->index[v11];
      p->index[v9] = v12;
      p->index[v11] = v10;
      *(_BYTE *)pKeyResult = p->index[(unsigned __int8)(v10 + v12)] ^ *(_BYTE *)key1;
      v13 = (unsigned __int8)(v4 + 2);        // 2
      v14 = p->index[v13];                    // p->Index[2]
    ...
```

这里我没有暂时没有渗入理解，直接进入第二次加密运算。

```
.text:0001A222 428 01 44                                    ADD             R1, R0 ; 长度
.text:0001A224 428 28 46                                    MOV             R0, R5 ; 第一次加密结果
.text:0001A226 428 EB F7 69 FC                              BL              sub_5AFC ;第二次加密
.text:0001A22A 428 3B 49                                    LDR             R1, =(__stack_chk_guard_ptr - 0
x1A300)
```

进入sub_5AFC，将key每3个字节一组，进行`<<8`拼接，也就是`a1<<16+a2<<8+a3`，举个例子`0xaa,0xbb,0xcc=>0xaabbcc`

然后拼接结果v15再左移，
如果是3个字符拼接的，这里v16是3，`v19=v15 << 8 * (3 - v16)`也就左移0，也就是不左移;
如果是两个字符或者一个字符拼接的，这里就需要左移8或者16位，说白了就是需要构成0x112233的结构。

然后v19进行4次移位，取aAbcdefghijklmn字符放入结果内存中。其实就是v19按6位进行分割（分别右移0x12,0xc,0x6,0x0，&03f），分割的值作为index，去aAbcdefghijklmn中对应字符，保存。
如果`v16<3`，也就是此次拼接没有3个字符，这里`index=0x40`，也就是增加额外的"="用于结果。

```
if ( _R10 > 0 )                               // len>0
  {
    i = 0;
    p1 = p;
    do
    {
      if ( i >= _R10 )
      {
        v16 = 0;
        v15 = 0;
      }
      else
      {
        ii = 0;
        v15 = 0;
        do
        {
          v15 = *(_BYTE *)(key + i + ii) | (v15 << 8);//
                                                // v15 = key[i] | 0<<8
                                                // v15 = key[i+1] | v15<<8
      // v15 = key[i+1] | v15<<8
          v16 = ii + 1;
          if ( ii + 1 > 2 )                     // 0, 1
            break;
          v17 = i + ii++;
        }
        while ( v17 + 1 < _R10 );
        i += v16;                               // v16 = 1, 2, 3
                                                // i += v16, 下次计算使用的i
      }
      j = 0;
      v19 = v15 << 8 * (3 - v16);
      v20 = 0x12;
      do
      {
        if ( v16 < j )
          index = 0x40;
        else
          index = (v19 >> v20) & 0x3F;
        v20 -= 6;
        *((_BYTE *)p1 + j++) = aAbcdefghijklmn[index];
      }
      while ( j != 4 );                         // 每4字节
      p1 = (char *)p1 + 4;
    }
    while ( i < _R10 );
  }
```

逆向算法

算法大致明白了，结果又是JPyjup3eCyJjlkV6DmSmGHQ=（取了0x18字节）。那么将第二次加密进行求逆。
先找JPyjup3eCyJjlkV6DmSmGHQ=每字节
在'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/='中的index。

```
k = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/='
r = 'JPyjup3eCyJjlkV6DmSmGHQ=' #//!!'
idd = []
def get_index_in_k(c):
    for i in range(0, len(k)):
        c1 = k[i:i+1]
        if c1 == c:
            return i
    return -1

def cc():
    j = 0
    for i in range(0, len(r)):
        c1 = r[i: i+1]
        index = get_index_in_k(c1)
        idd.append(index) #保存序号
        print '%d: %c %d %x' % (i+1, c1, index, index )
```

结果是：

```
1: J 9 9
2: P 15 f
3: y 50 32
4: j 35 23
5: u 46 2e
6: p 41 29
7: 3 55 37
8: e 30 1e
9: C 2 2
10: y 50 32
11: J 9 9
12: j 35 23
13: l 37 25
14: k 36 24
15: V 21 15
16: 6 58 3a
17: D 3 3
18: m 38 26
19: S 18 12
20: m 38 26
21: G 6 6
22: H 7 7
23: Q 16 10
24: = 64 40
```

然后每4个index一组，来自于v19的4次右移，那么反过来4个一组，左移相加就是v19

```
for i in range(0, len(idd), 4):
        a1 = idd[i] << 0x12
        a2 = idd[i+1] << 0xc
        a3 = idd[i+2] << 0x6
        a4 = 0
        if idd[i+3] == 0x40:
            a4 = 0
        else:
            a4 = idd[i+3] << 0
        a = a1+ a2+a3+a4
        rrr.append(a)
        print '%d: %x' % (i, a)
```

得到结果：

```
0: 24fca3
4: ba9dde
8: b2263
12: 96457a
16: e64a6
20: 1874
```

然后我们又知道v19其实是v15拼接的，所以拆开就得到v15（第一次加密结果），可以看到key长度应该是17。

```
24 fc a3 ba 9d de 0b 22 63 96 45 7a 0e 64 a6 18 74
```

然后接着求第一次加密的逆运算，看代码，好多啊，怎么办，难道要求逆，好难！
好吧，不装了，其实不难，我们看前面说的第一次加密其实就是分组xor！
xor好啊，xor好啊...我们知道xor两次会将结果还原，想到了什么？！
是的，既然我们拿到第一次加密结果，那让他再和哪个8字节+0x100*4的数组再xor一次不久可以了，但是要重写这个加密代码貌似也挺麻烦的，怎么办？！

这里我是这么做的，在调试中，第一次加密前，将key的值（本来是输入）修改为上面得到的第一次加密结果，然后开始第一次加密运算，这样不就完美的完成了一次求逆吗，哈哈！

具体操作，对1A13A下断，输入key（必须是17位，否则修改内存时可能会挂），确认，断下来，此时r2就是key

```
5E127B20  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  1234567890123456
5E127B30  37 00 6D 5F 1B 00 00 00  00 00 00 00 00 00 00 00  7.m_............
```

然后在hex窗口，f2修改内存，输入上面的24 fc...，然后f2确认修改。

```
5E127B20  24 FC A3 BA 9D DE 0B 22  63 96 45 7A 0E 64 A6 18  $.
5E127B30  74 A9 12 5E 0F 00 1F 00  FF FF 1F 00 0F 00 00 t..^..
```

然后f8。看看结果：

```
5E127B38  6D 61 64 65 62 79 65 72  69 63 6B 79 39 34 35 32  madebyericky9452
5E127B48  38 00 73 00 11 10 00 00  62 00 69 00 6C 00 69 00  8.s.....b.i.l.i.
```

答案就是：madebyericky94528

转载请注明出处：https://anhkgg.github.io/kxctf2017-writeup6

参考：

1. 安卓APP动态调试技术–以IDA为例
2. http://luleimi.blog.163.com/blog/static/175219645201210922139272/
3. http://blog.csdn.net/zhangmiaoping23/article/details/43445797
4. http://www.cnblogs.com/liujiahi/archive/2011/03/22/2196401.html
5. http://cncc.bingj.com/cache.aspx?q=arm++IT+EQ&d=4981012666125942&mkt=zh-CN&setlang=zh-CN&w=YEX3ioizXLDZGmlpVDBGFh_dhhHpfnYj