

【jarvisoj刷题之旅】逆向题目DDCTF - Android Normal的writeup

原创

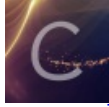
iqiqiya 于 2018-09-27 15:34:16 发布 777 收藏 1

分类专栏: [我的逆向之路](#) [我的CTF之路](#) [-----jarvisojCTF](#) [我的CTF进阶之路](#) 文章标签: [【jarvisoj刷题之旅】逆向题目DDCTF - Andr](#) [逆向题目DDCTF - Android Normal的wri](#) [DDCTF - Android Normal的writeup](#) [Android Normal](#)

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/xiangshangbashaonian/article/details/82868095>

版权



[我的逆向之路](#) 同时被 3 个专栏收录

108 篇文章 10 订阅

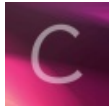
订阅专栏



[我的CTF之路](#)

92 篇文章 5 订阅

订阅专栏



[-----jarvisojCTF](#)

2 篇文章 0 订阅

订阅专栏

Android Normal

下载后输入解压密码进行解压 得到Readme.txt与DDCTF-Normal.apk

```
Readme.txt
1 赛题背景: 本挑战结合了Android, Java,
  C/C++, 加密算法等知识点, 考察了挑战者的binary逆向技术和加密算法能力。
2
3 赛题描述: 本题是一个app, 请试分析app中隐藏的key, 逆向加密算法并得到对应的秘钥
  。可以在app中尝试输入key, 如果正确会显示"correct", 如果错误会显示"Wrong"。
4 提 示: 阅读assembly code, 理解xor的加密逻辑和参数, 解出答案。
5 评分标准: key正确则可进入下一题。
https://blog.csdn.net/xiangshangbashaonian
```

将apk载入模拟器运行 (顺便吐槽下 蓝叠咋不能竖屏。。。)

Guess Key

.....

TEST

Wrong

<https://blog.csdn.net/xiangshangbashaonian>

输入123456789 出现Wrong

载入jeb

反编译成java代码分析

```

Certificate Assembly Decompiled Java Strings Constants Notes
static {
    System.loadLibrary("hello-libs");
}

public MainActivity() {
    super();
}

public void onClickTest(View v) {
    if(this.mFlagEntryView.getText().toString().equals(this.stringFromJNI())) {
        this.mFlagResultView.setText("Correct");
    }
    else {
        this.mFlagResultView.setText("Wrong");
    }
}

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    this.setContentView(2130968602);
    this.mFlagEntryView = this.findViewById(2131427413);
    this.mFlagResultView = this.findViewById(2131427415);
}

public native String stringFromJNI() {
}

```

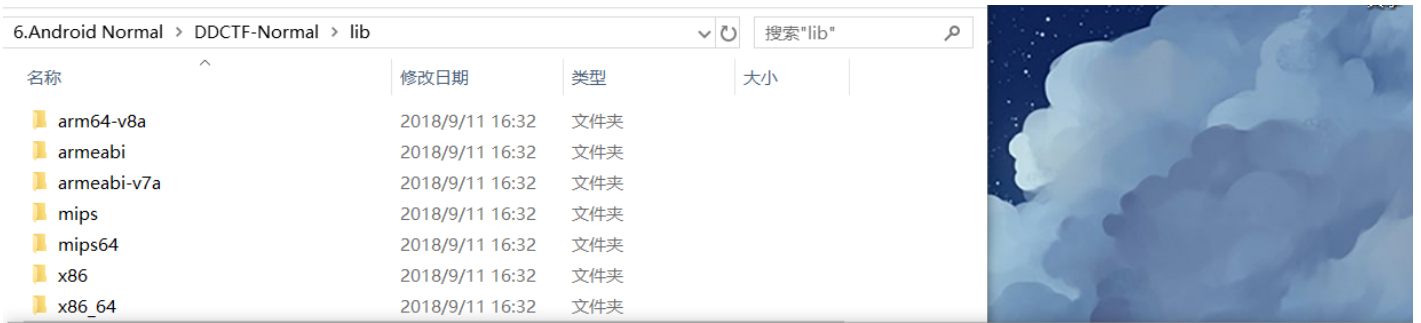
com.didictf.helloli
BuildConfig
MainActivity
R

<https://blog.csdn.net/xiangshangbashaonian>

程序流程很简单 就是一个简单的字符串比较 只不过要对比的字符串放进了native层

这个stringFromJNI()函数就是要在Java代码中调用的Native函数

那么就要IDA载入分析so文件



新建文本文档.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

其中，mips、armeabi、armeabi-v7a和x86都表示CPU的类型。一般的手机或平板都是用arm的cpu。

armeabi 是针对普通的或旧的arm v5 cpu，32位

armeabi-v7a 是针对有浮点运算或高级扩展功能的arm v7 cpu，32位

arm64-v8a 针对64位的

mips 是一种采取精简指令集（RISC）的处理器架构，32位

mips64 64位

x86 IA-32位指令集

x86_64 64位

<https://blog.csdn.net/xiangshangbashaonian>

因为前面有调用hello-libs.so

所以想找和这个名字一样的 结果发现每个文件夹里都有

这时候就蒙了

看了大佬的writeup

<https://blog.csdn.net/wannafly1995/article/details/80954371>

大佬说通常采用armeabi-v7a的so文件

那就用它咯

可以发现

f	__cxa_atexit	.plt	000000
f	__cxa_finalize	.plt	000000
f	__aeabi_memcpy	.plt	000000
f	__stack_chk_fail	.plt	000000
f	GetTicks	.plt	000000
f	gpower	.plt	000000
f	__android_log_print	.plt	000000
f	__aeabi_wind_cpp_pr45(char *)	.plt	000000
f	sub_518	.text	000000
f	__aeabi_wind_cpp_pr45(char *)	.text	000000
f	Java_com_didictf_hellolib_MainActivity_stringFromJNI	.text	000000
f	__imp__cxa_finalize	extern	000020
f	__imp__cxa_atexit	extern	000020
f	__imp_GetTicks	extern	000020
f	__imp__aeabi_memcpy	extern	000020
f	__aeabi_unwind_cpp_pr0	extern	000020
f	__imp__android_log_print	extern	000020
f	__imp__stack_chk_fail	extern	000020
f	__imp_gpower	extern	000020

<https://blog.csdn.net/xiangshangbashaonian>

F5变成c代码

分析可知有用的函数就是他了__aeabi_wind_cpp_pr45()

```

IDA View-A  Pseudocode-B  Pseudocode-A  Hex View-1  Struc
1 int __fastcall Java_com_didictf_hellolib_MainActivity_stringFromJNI(int a1)
2 {
3     int v1; // r4
4     int v2; // r7
5     int v3; // r0
6     char v5[4]; // [sp+8h] [bp-D0h]
7
8     v1 = a1;
9     GetTicks();
10    v2 = 0;
11    do
12    {
13        v3 = gpower(v2++);
14        *(_DWORD *)v5 = v3;
15    }
16    while ( v2 != 32 );
17    GetTicks();
18    __android_log_print(4, (int)"hell-libs::", "calculation time: %llu");
19    __aeabi_wind_cpp_pr45(v5);
20    return (*(int (__fastcall *)*)(int, char *))*(_DWORD *)v1 + 668)(v1, v5);
21 }

```

<https://blog.csdn.net/xiangshangbashaonian>

双击再双击

```

IDA View-A x Pseudocode-B x Pseudocode-A x Hex View-1 x Structures x Enums x Imports x
1 int __fastcall __aeabi_wind_cpp_pr45(char *a1)
2 {
3     char *v1; // r4
4     int i; // r1
5     int j; // r3
6     int v4; // r6
7     int v5; // r5
8     int v6; // r2
9     char c[182]; // [sp+6h] [bp-CAh]
10    int v9; // [sp+BCh] [bp-14h]
11
12    v1 = a1;
13    i = 0;
14    do
15    {
16        c[i] = a[i] ^ b[i];
17        ++i;
18    }
19    while ( i != 182 );
20    j = 0;
21    do
22    {
23        v4 = c[(c[0] >> 1) + j++];
24    }
25    while ( v4 );
26    v5 = j - 1;
27    v6 = 0;
28    if ( j - 1 >= 1 )
29    {
30        _aeabi_memcpy(a1, &c[(c[0] >> 1), j - 1);
31        v6 = v5;
32    }
33    v1[v6] = 0;
34    return _stack_chk_guard - v9;
35 }
00000580 _Z21__aeabi_wind_cpp_pr45Pc:31 (580) https://blog.csdn.net/xiangshangbashaonian

```

// c[0] = a[0] ^ b[0] 所以c[0]=57 c[0] >> 1 =28
// 因为当v4等于0时才退出循环 所以就是要找值是0 且下标在28以后的c数组中的元素
// 那么我们将他们全部打印出来 就可以看到flag

双击a[i]与b[i]就能得到内容 dump即可

Py大法好:

```

a = [0xD8, 0xC2, 0x6B, 0x42, 0x82, 0x67, 0xC8, 0x4D, 0x7A, 0x95,
0xE8, 0x81, 0x48, 0xC1, 0x9E, 0x40, 0xE8, 0xFB, 0xCF, 0xE6,
0x4F, 0xBA, 0xE6, 0xAF, 0x78, 0x19, 0x6F, 0x9C, 0xE9, 0xF7,
0x7A, 0xDD, 0x42, 0xCE, 0x8C, 0x03, 0xB8, 0x66, 0xD3, 0xAB,
0x00, 0x7E, 0xDE, 0x3E, 0x53, 0xDE, 0x30, 0x91, 0x3D, 0xF7,
0xCD, 0x72, 0x14, 0x51, 0x82, 0xEE, 0x1B, 0x8D, 0xB4, 0x8C,
0xD0, 0x8A, 0xF6, 0x9A, 0x96, 0x71, 0x98, 0x62, 0x93, 0x4A,
0x30, 0x2F, 0x9C, 0xA8, 0x79, 0x16, 0xC1, 0xE0, 0xEC, 0xD7,
0xE5, 0xEC, 0x8A, 0x64, 0xB4, 0x46, 0xCF, 0xD9, 0xE5, 0x96,
0xF3, 0x94, 0x73, 0xA9, 0xFF, 0xEA, 0xCB, 0x15, 0x9C, 0x7C,
0xA1, 0xD8, 0x3E, 0xBB, 0x1D, 0x38, 0xCB, 0x55, 0xD0, 0x19,
0x25, 0xB2, 0x0B, 0x92, 0xE8, 0x88, 0xAE, 0x06, 0xA2, 0x9B,
0x93, 0x64, 0x5E, 0xFB, 0x09, 0x05, 0xF6, 0x2F, 0x1F, 0x35,
0xCC, 0xEF, 0x05, 0x6C, 0x19, 0x42, 0x38, 0xA5, 0x59, 0x2E,
0x80, 0x0A, 0x19, 0xFC, 0x33, 0x5B, 0xBB, 0xD6, 0xFB, 0x2B

```

```
0x00, 0x0A, 0x15, 0x1E, 0x25, 0x2B, 0x30, 0x36, 0x3C, 0x42, 0x49,
0xAC, 0xF7, 0x0E, 0xAD, 0xD8, 0x57, 0x40, 0x98, 0x71, 0x2C,
0x78, 0x68, 0x91, 0x82, 0x4F, 0x5B, 0xD6, 0x40, 0x8F, 0x03,
0xBD, 0x55, 0x0B, 0x47, 0x3D, 0xF4, 0x5A, 0x49, 0x5B, 0xF2,
0xA2, 0x9E]
```

```
b = [0xE1, 0xA1, 0x01, 0xE4, 0x82, 0x56, 0x9D, 0x70, 0xD9, 0xF5,
0x08, 0x10, 0x22, 0xA7, 0x2D, 0x2B, 0x41, 0xF0, 0xBD, 0xA4,
0x67, 0x3D, 0x9A, 0x20, 0xB9, 0xFB, 0x11, 0xD3, 0xAD, 0xB3,
0x39, 0x89, 0x04, 0xE3, 0xBF, 0x3A, 0x8F, 0x07, 0xEA, 0x9B,
0x61, 0x4D, 0xEC, 0x08, 0x64, 0xE8, 0x04, 0xA0, 0x0B, 0xC2,
0xF5, 0x10, 0x76, 0x32, 0xBB, 0xD9, 0x2E, 0xBE, 0x86, 0xBA,
0xE7, 0xBA, 0xC6, 0xFC, 0xA2, 0x13, 0xD8, 0x06, 0xFA, 0x2E,
0x59, 0x4C, 0xF4, 0xDD, 0x01, 0x7F, 0xAF, 0x87, 0xC2, 0xB4,
0x8A, 0x81, 0x8A, 0xF2, 0xB6, 0x60, 0x9A, 0x13, 0x52, 0xC0,
0x6D, 0x9E, 0x5A, 0x52, 0xB5, 0x8F, 0x47, 0x5E, 0xE6, 0x41,
0xAD, 0xF5, 0xBB, 0xA9, 0x7A, 0x6C, 0xA1, 0x4C, 0x38, 0x60,
0xF2, 0x4B, 0x5C, 0xE8, 0x5B, 0xE5, 0xE3, 0xBA, 0x46, 0x70,
0x33, 0x04, 0xA7, 0x58, 0x19, 0x10, 0x49, 0x20, 0x1D, 0x51,
0x48, 0x9D, 0x78, 0xF9, 0xB4, 0x2E, 0x66, 0x58, 0x1B, 0xE8,
0xEE, 0x51, 0x09, 0x21, 0x80, 0xBC, 0xC8, 0x7B, 0xF5, 0x4E,
0x99, 0xFD, 0xFC, 0x9A, 0xFD, 0x65, 0x20, 0x13, 0x57, 0xD1,
0x83, 0x4D, 0xF6, 0x2C, 0xAF, 0x25, 0x3C, 0x12, 0xF0, 0x7C,
0x16, 0x66, 0x97, 0x7F, 0x6A, 0x02, 0xBC, 0x98, 0x52, 0xD7,
0xE3, 0x56]
```

```
c = []
```

```
j = 0
```

```
flag = ''
```

```
for i in range(0, len(a)):
```

```
    c.append(a[i] ^ b[i])
```

```
    if c[i] == 0:
```

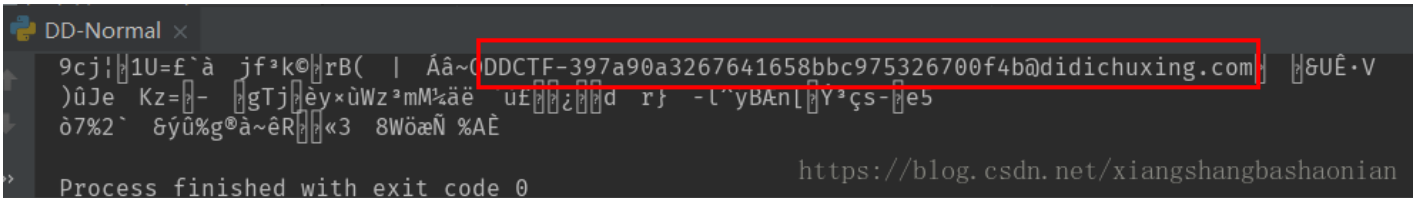
```
print(i)

for i in c:

    flag += chr(i)

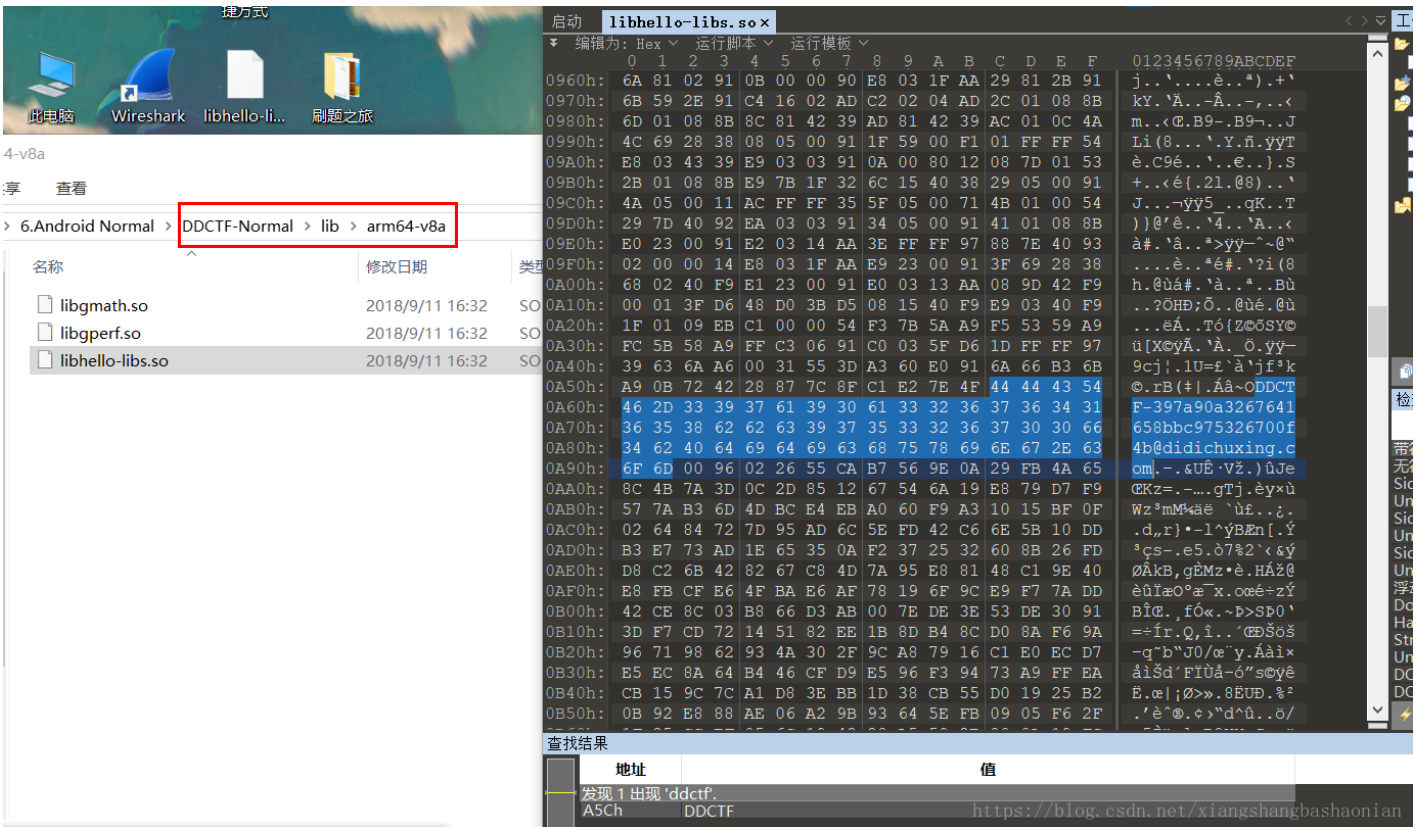
print(flag)
```

最后找到flag:



后来像再看看其他大佬思路:

十六进制编辑器打开lib\arm64-v8a\libhello-libs.so居然直接可以看到flag。。。



lib\x86\libhello-libs.so与

lib\x86_64\libhello-libs.so中都可以直接发现flag

可怕。。。

更多的是疑惑 如果有哪位大佬 看到 可不可以麻烦给小弟讲解一下

还有就是可以改smile代码进行动态调试