# Mobile

出题思路：反调试、反射调用、字符串加密、简单加解密算法、脚本编程、java逆向、c逆向

执行流程：用户输入flag，点击确定进行检查，输入正确提示成功，输入错误提示失败

***检查流程：***

存在三个加密函数对输入字节数组进行加密处理，检查函数通过反射调用不断使用这三个函数对输入的flag进行加密处理，并且反射调用中所有的字符串都经过加密处理。为了增加难度，三个加密函数调用的次数达到1000次。Java层经过1000次加密处理之后，将结果传入native层进一步进行加密处理，先进行一轮rc4加密，再进行一轮aes加密（前16字节aes加密，后16字节直接加下标），inline hook了libc的write函数，在最后调用write函数进入到被我们hook的函数中对输入的每个字节进行异或0x13操作，最后与答案进行比较返回结果

***难点：***

1. Java层反调试

通过调用isDebuggerConnected()判断

需要修改smali进行绕过

1. 字符串加密

隐藏反射调用的同时，使用签名进行解密，一定程度上增加修改smali的限制（需要修改更多突破限制）

1. 反－反编译

检查函数主体使用反射调用了1000次加密，smali代码量达到3w＋行，使得反编译工具都失效

1. native层反调试
   1. 检查调试状态

fork一个新进程，并且通过ptrace自身阻止调试器调试，该进程每隔10秒通过读取/proc/pid/status判断是否存在调试器调试

* 1. 检测断点

在进行rc4加密过程中，将一函数的前256字节作为key，如果调试器在这256字节中存在断点，由于断点（软断点，非硬断点）的影响，则会得到错误的key

* 1. 通过signal监测子进程状态，作为反调试手段的一种，一旦子进程状态发生变化，父进程直接退出

1. native层代码自篡改

某些函数段加密处理，运行是解密，运行结束后再加密

1. ELF节区信息删除
2. Ollvm混淆处理
3. Hook: inline hook了libc的write函数，在加密输入的过程中调用write函数进入到被我们hook的函数中对输入的每个字节进行异或0x13操作

***解题思路：***

自底向上逆向解密，通过分析so文件，可直接获取最终需要对比的字节码ans[]，如下依次通过异或、aes、rc4解密

for (i = 0; I < 32; i++)

ans[i] ^= 0x13

for (i = 0; i < 32; i++) { //前16字节aes解密，后16字节直接减去下标

if (i < 16)

dat[i] = ans[i];

else

ans[i] = (ans[i]-i)&0xff;

}

inv\_aes(dat, ans, key);

rc4(buf, 256, (char \*)ans, 32);

//buf为函数Java\_com\_ctf\_crackme1\_Check\_check前256字节

通过上述解密可得到［0x99, 0x85, 0x64, 0xa1, 0xe7, 0xa3, 0xe5, 0xc, 0x30, 0x1a, 0xee, 0x25, 0xf3, 0x19, 0xc9,

0xa9, 0xb0, 0x20, 0xf9, 0x4d, 0x22, 0xd7, 0xcc］

此时分析java层加密过程为：不断调用3个加密函数进行加密，调用总数为1000次。

三个加密函数如下：

**public** **void** b(**byte**[] str) **throws** Exception {

**for** (**int** i = 0; i < str.length; i++) {

str[i] += 0x3a;

**if** (i != 0)

str[i] += str[i-1];

**else**

str[i] += str[str.length-1];

}

}

**public** **void** d(**byte**[] str) **throws** Exception {

**for** (**int** i = 0; i < str.length; i++) {

str[i] ^= 0x96;

**if** (i != 0) {

str[i] ^= str[i-1];

}**else** {

str[i] ^= str[str.length-1];

}

}

}

**public** **void** e(**byte**[] str) **throws** Exception {

**for** (**int** i = 0; i < str.length; i++) {

str[i] = (**byte**) (((str[i] >> 4) & 0xf) + ((str[i] & 0xf) << 4));

}

}

对应的解密函数如下：

**public** **void** b2(**byte**[] str) **throws** Exception {

**for** (**int** i = str.length-1; i >= 0; i--) {

str[i] -= 0x3a;

**if** (i != 0)

str[i] -= str[i-1];

**else**

str[i] -= str[str.length-1];

}

}

**public** **void** d2(**byte**[] str) **throws** Exception {

**for** (**int** i = str.length-1; i >= 0; i--) {

str[i] ^= 0x96;

**if** (i != 0) {

str[i] ^= str[i-1];

}**else** {

str[i] ^= str[str.length-1];

}

}

}

**public** **void** e(**byte**[] str) **throws** Exception {

**for** (**int** i = 0; i < str.length; i++) {

str[i] = (**byte**) (((str[i] >> 4) & 0xf) + ((str[i] & 0xf) << 4));

}

}

通过脚本实现提取三个加密函数的调用序列，我们将其放到list中，

**for** (**int** i = list.size()-1; i >= 0; i--) {

**switch** (list.get(i)) {

**case** 3:

b2(ans);

**break**;

**case** 4:

d2(ans);

**break**;

**case** 5:

e(ans);

**break**;

**default**:

**break**;

}

}

System.***out***.print("ans: ");

**for** (**byte** one : ans) {

System.***out***.print(one + " ");

}

逆向调用解密函数即可得到flag：

CTF{Y0u\_Ar3\_Go0d\_a7\_1t}