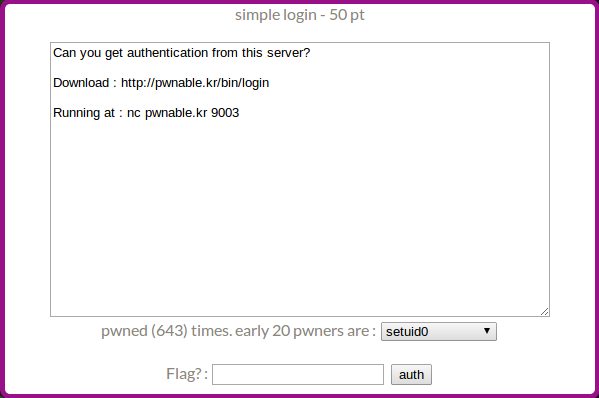
***解题思路***

# *题目要求*



# *题目解读*

应该还是溢出漏洞利用的题目,nc连上看下效果:

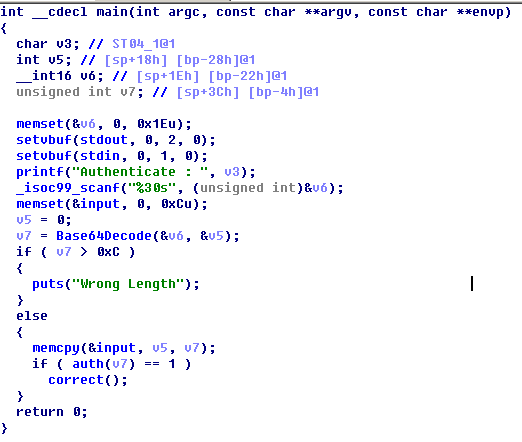


其中输入字符串,返回hash。

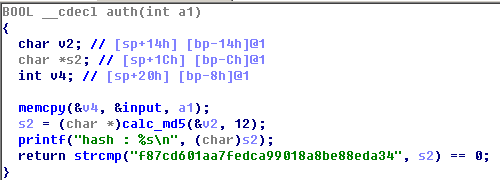
需要注意的是每次输入相同的数据，得出的hash值却不同。<可疑点>

接下来上IDA进行分析

# *解题过程*

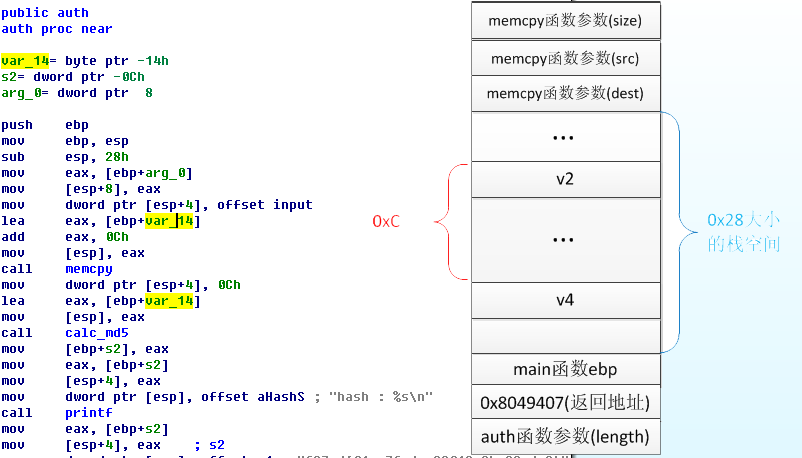
main函数的主要逻辑如下所示: 

从键盘上输入字符,并解base64编码,同时限制解码后的数据长度不能大于12。然后把解码后的数据拷贝到input数组当中，调用auth函数，如果认证结果为1，则通过，调用correct函数。很简单的逻辑，没有复杂的实现，下面仔细查看Base64Decode、auth、correct这三个函数，重点是auth函数。



从input数组中拷贝a1长度的数据到v4，而计算md5的函数参数为v2，并不是v4。这里是个可疑点。接下来打印计算出的哈希值，并与一个固定的哈希字符串进行比较，二者相等，则为认证通过。

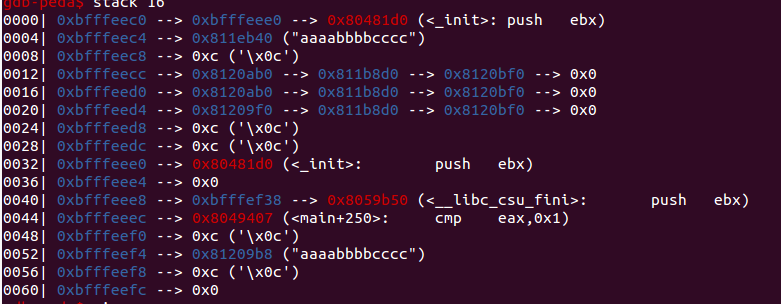
开始以为是暴力破解的问题，使用MD5在线工具来破解原文，没有得到有效结果，果断放弃这条路。后来在测试过程中发现相同的输入，每次打印的哈希值都不同，于是仔细分析哈希计算前后代码：



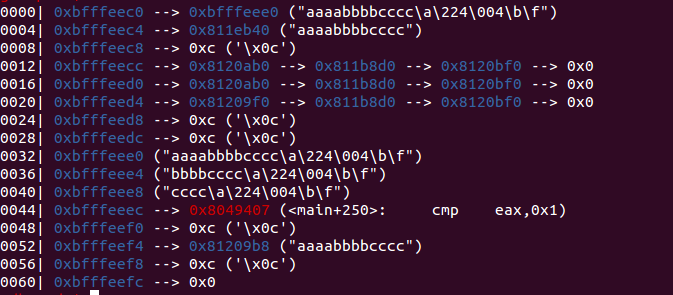
根据上述代码可以得出栈空间的分布。可以看出memcpy函数处有问题，dest的地址为ebp-14h+0ch=ebp-8h，即图中v4的位置，而在计算md5的时候，用的是ebp-14h的地址，并非我们输入并解base64之后的数据，这也说明了为什么每次输入相同的内容，而计算的md5值不同的问题。

而且从上图可以看出，当解base64之后的数据长度是12时，只能覆盖到main函数的ebp的值，不能覆盖auth函数的返回地址，还需要其他方法来解决。

上gdb-peda工具进行分析，首先构造数据b64e(“a”\*4+”b”\*4+”c”\*4)作为输入。在auth函数出下断点，执行memcpy之前的栈空间为：



执行完memcpy之后的栈空间为：



可以看到从input，也就是0x811eb40处拷贝了12个字节到0xbfffeee0处，其中cccc字符串覆盖了原来的0xbfffef38，也就是main函数的ebp的地址。接着往下执行，在auth函数执行完后，可以看到最后执行的两条指令为：



这里需要解释下这两条指令的作用：

leave指令的动作为：将栈指针esp执行帧指针ebp，（平衡栈空间），然后再将备份的原帧指针出栈到ebp，即：

***mov esp，ebp***

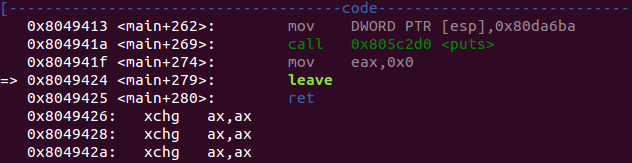
***pop ebp***

ret指令的动作为：将栈顶的返回地址弹出到EIP，然后按照EIP此时指示的指令地址继续执行程序。

对应到这里ebp的值为：0xbfffeee8，执行完leave指令之后ebp的值为0x63636363，此时的esp指向0xbfffeeec，然后执行ret指令，调到0x8049407处执行。执行ret指令的时候gdb会提示一个信息如下：



不用管它，继续单步执行。走到main函数的结尾处



同样是leave和ret操作。执行leave指令后，程序终止。为什么会终止呢？我们来看一下执行leave前ebp的值，这时ebp的值为0x63636363，执行leave指令时，把它的值赋给esp，然后出栈一个元素给ebp，而0x63636363这个地址无法访问，也就无法完成leave指令。

通过这个过程的分析，我们可以对用户的输入进行构造，使得ebp能够访问有效的地址，因为input在.bss段，且会拷贝用户的输入，所以可以把input的首地址来覆盖main函数的ebp，同时又因为correct函数里有system函数，且参数就是/bin/sh，因此构造如下输入：

**b64e(‘A’\*4 + system\_addr + input\_addr);**

这样在main函数返回时会把input的地址作为esp指针，然后执行ret指令时就会执行system\_addr处的指令，即调用了system(“/bin/sh”)函数。

即 ***b64e(‘a’\*4 + p32(0x08049284) + p32(0x0811eb40));***

最终结果为：control EBP, control ESP, control EIP, control the world~