1. 观察程序行为

运行cmd后，在文件rev\_homework.exe所在路径运行程序，出现如图1的界面。

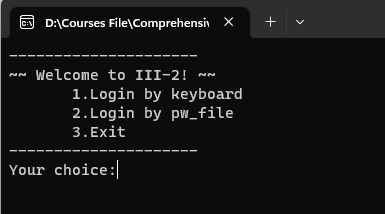


图 1 程序运行界面

根据提示，输入“3”，程序会退出；输入“1”，是需要从键盘输入登录信息﹔输入“2”，是需要从键盘输入登录信息。此处我们同样利用整数溢出漏洞来完成打印flag的操作，接下来对源程序进行反汇编分析。

1. 反汇编二进制代码并寻找溢出点

该程序的目标不仅要返回“Success!”的提示，还需要打印flag。现在借用IDA pro 7.6对该程序进行反汇编分析。

IDA成功定位到main函数，此处main函数只执行对main\_0函数的调用，查看main\_0，对反汇编代码进一步反编译。

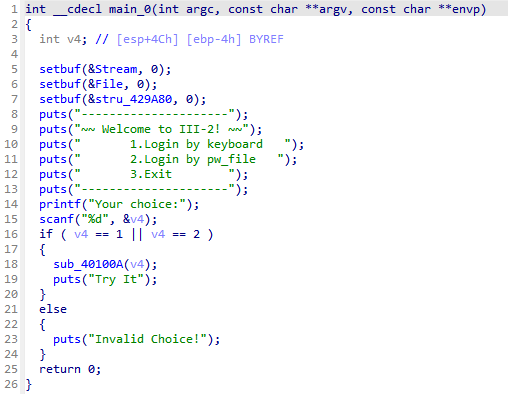


图 2 反编译的main函数

可以看到此处在v4等于1或者2的时候，需要执行sub\_40100A函数，也就是说在选择从键盘输入或者从文件读入时都需要执行该函数，则进一步跟踪至sub\_40100A。

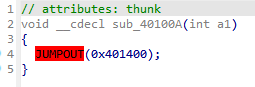


图 3 反编译的sub\_40100A函数

此处可以发现IDA显示了JUMPOUT(0x401400)，出现JUMPOUT的原因是IDA无法识别跳转指令的跳转地址，因此我们解除反编译，可以看到汇编代码如图4所示。



图 4 sub\_40100A函数的原始汇编代码

通过此处的汇编代码可以发现，实际上执行的是一个jmp指令，执行跳转指令跳转至loc\_401400。双击loc\_401400进行跟踪，可以跳转至该部分的完整汇编代码如图5所示。

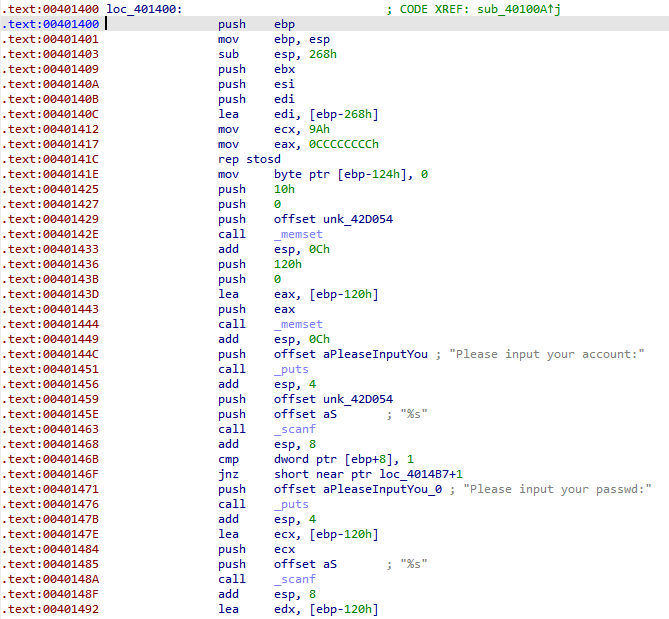


图 5 loc\_401400部分的汇编代码

此时可以将这部分代码块选中，按P执行MakeFunction操作，将该部分汇编代码作为一个函数进行识别，然后返回看反编译的sub\_40100A函数如图6所示。

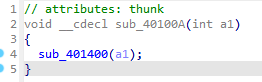


图 6 MakeFunction后的反编译的sub\_40100A函数

很显然可以看出此时IDA已经识别出这部分代码，代码执行任务即为执行我们刚make出来的函数sub\_401400。进一步跟踪，可以看到sub\_401400函数如图7所示。

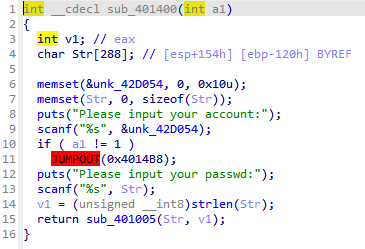


图 7 反编译的sub\_401400函数

此时可以看到仍然有JUMPOUT指令出现，则采取同样的方式进行跟踪。跟踪到目标地址0x4014B8可以看到汇编代码如图8所示。

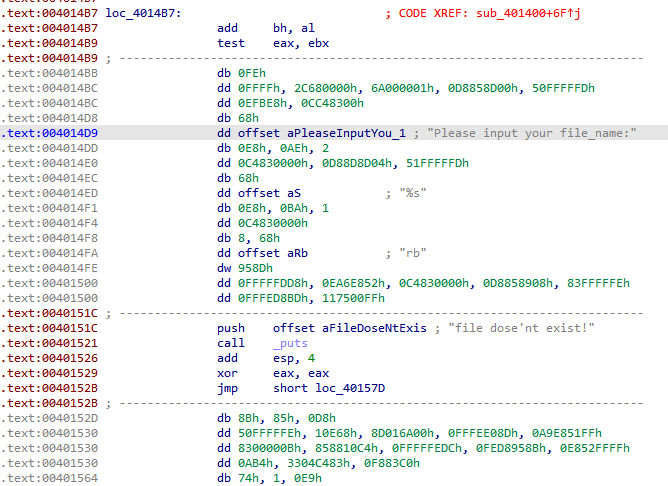


图 8 0x4014B8部分汇编代码

可以看到此处的代码大部分为dd，dw这样的指令，并且首个指令的初地址为0x4014B7，说明此处IDA识别汇编代码时出现了对齐问题，错误地将代码首地址视为0x4014B7得到了错误的汇编代码，所以此处可以按D使用MakeData指令将其转换为原始数据，转换后如图9所示。

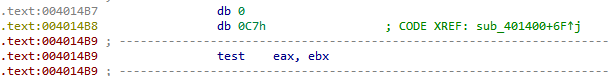


图 9 MakeData后的部分代码

此时可以再将0x4014B8后的代码块全部选中后按C使用MakeCode指令将其转换为汇编代码，转换后如图10所示。

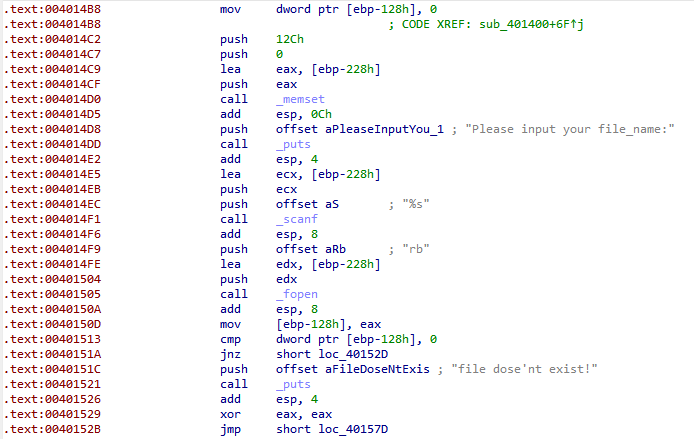


图 10 MakeCode后的0x4014B8部分汇编代码

可以看出此时的该部分代码被正确识别，使用MakeFunction功能后，再次回到反编译的sub\_401400函数部分，可以发现JUMPOUT指令被消除。

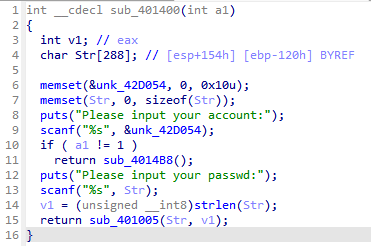


图 11 消除JUMPOUT指令后的sub\_401400函数

则可以看到在初始界面选择文件输入时，会执行sub\_4014B8函数，进一步跟踪sub\_4014B8函数，可以看到函数内又调用了sub\_401005函数，sub\_401005又进一步调用了sub\_401600函数。终于在此处可以发现整数溢出漏洞的目标函数strcpy(Destination, Source)。

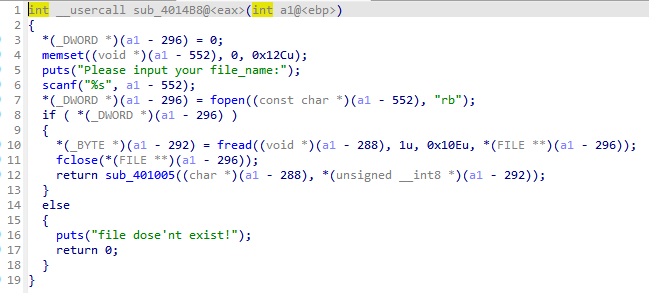


图 12 反编译的sub\_4014B8函数

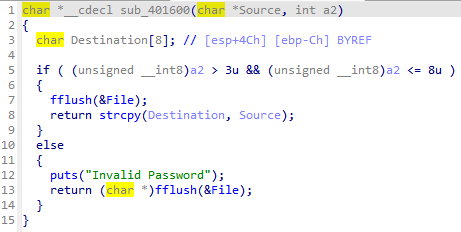


图 13 反编译的sub\_401600函数

可以看到此处为Destination分配了Ch的内存空间，并且在进行strcpy操作之前对a2的值进行了检测，应保证a2满足a2 > 3且a2 < 8，按理说并不会发生溢出。但是由于a2原本为int型，此处使用强制转换使其转换为unsigned \_int8类型，在进行这样的转换时会舍弃原本高字节的内容，也就是说如果Source的长度为255 + 3~8，则也会被识别为满足条件而进行strcpy操作，从而导致整数溢出漏洞的产生。所以我们需要构造一个255 + x（x在3到8之间）长度的字符串用于输入，则可以利用整数溢出漏洞。

1. 定位关键函数

由于程序目标是输出“Success!”字符串和“Flag”字段，所以使用Shift + F12进行字符串查找，可以很明显地看到字符串“Flag：”出现如图14。

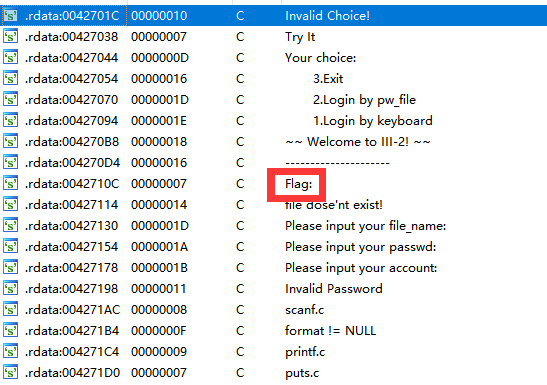


图 14 进行字符串查找

进行跟踪后找到对应数据段，点击xref可以找到引用该字符串的代码段，转到sub\_4012E0函数如图15所示。

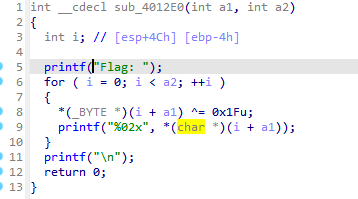


图 15 反编译后的sub\_4012E0函数

此处可以看到不仅打印了“Flag：”字符串，还通过循环打印了一段“%02x”格式的数据，猜测此处即为Flag对应值。但此时还没有发现“Success!”字符串，所以对该函数进行xref溯源，可以追踪到sub\_401019，进一步追踪到text段0x401293处，可以看到该部分代码块如图16所示。

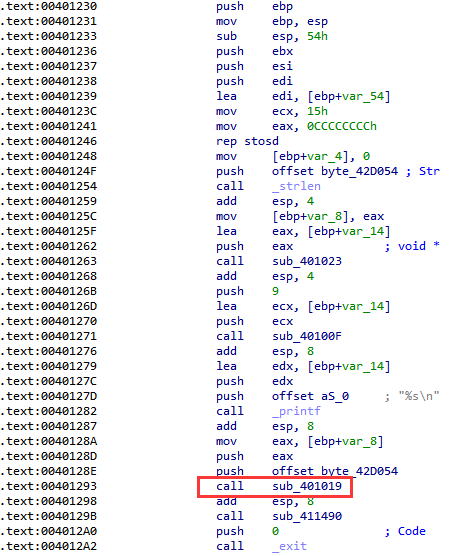


图 16 调用sub\_401019函数的汇编代码段

将这部分代码块使用MakeFunction进行处理，反编译该部分代码，及可以看到反编译后的函数sub\_401230如图17所示。

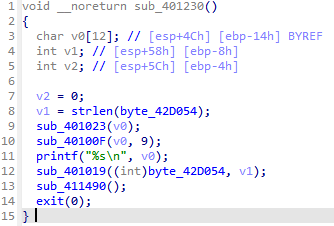


图 17 反编译的sub\_401230函数

可以看到该函数先后调用了sub\_401023，sub\_40100F，sub\_401009，sub\_411490，通过进一步跟踪可以发现sub\_401023调用了sub\_4011B0函数，sub\_4011B0中即包含打印“Success!”字符串的代码。

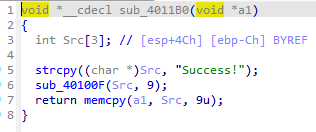


图 18 反编译的sub\_4011B0函数

从此处我们可以看出，sub\_401230函数包含了两个字符串的打印函数，也就是说将溢出地址设为0x00401230即可实现实验目标。

1. 动态跟踪溢出点附近栈的变化

通过动态跟踪可以分析在strcpy函数执行时栈的变化，可以尝试用一个长度为105h的文件作为输入，在之后向该文件插入目标溢出地址。

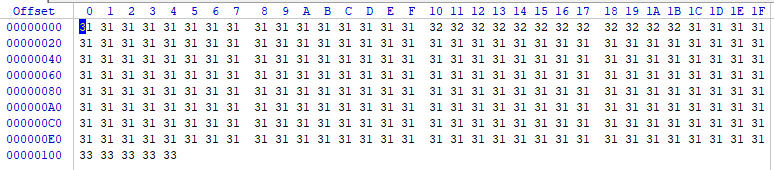


图 19 构造的测试用二进制文件

在执行strcpy函数的语句上打上断点，并且在debugger选项中选择Local Windows Debugger，然后开始进行动态调试。

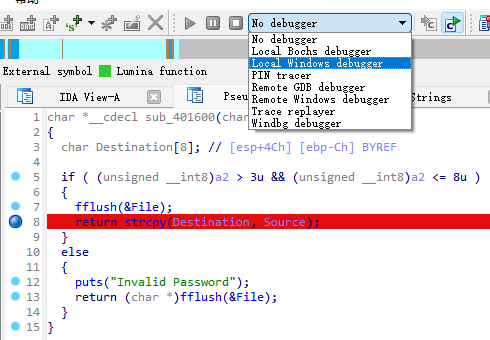


图 20 设置断点并选择Debugger

开始进行动态调试，在exe执行后选择对应的通过文件输入，并且输入对应的学号和文件名，回车后调试即会在断点处暂停。

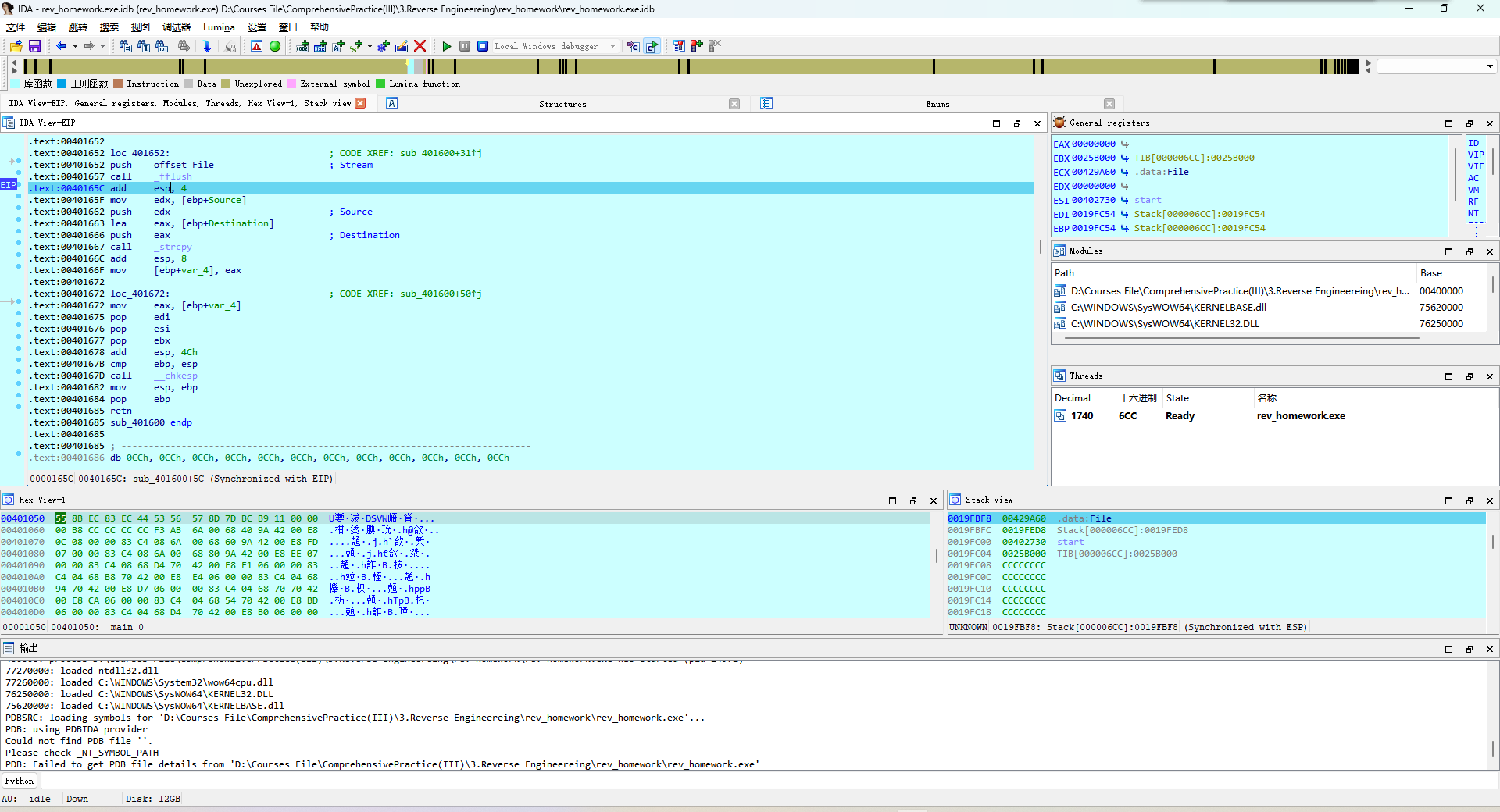


图 21 调试至断点处

通过按F8进行单步调试，单步执行到call \_strcpy时，查看右上方的寄存器状态栏，可以看到此时EBP的值为0019FC54。

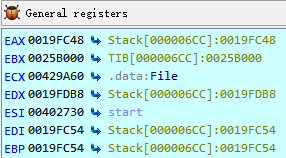


图 22 寄存器状态栏

同时，在栈中找到相应的地址位置，可以看到0019FC58保存了函数的返回地址，这部分内容即为我们的溢出目标。

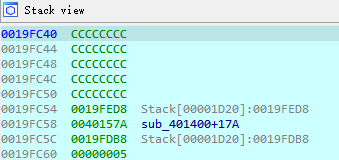


图 23 栈状态栏

继续通过F8进行单步调试执行，执行至retn语句时，可以观察到栈状态栏如图25所示。

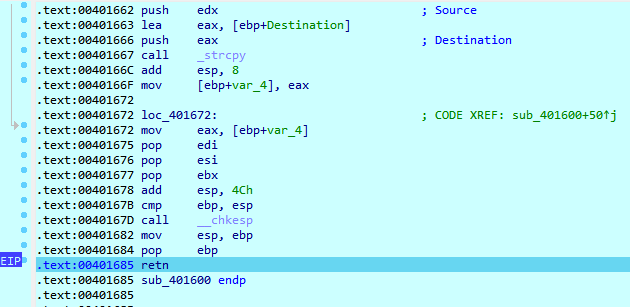


图 24 单步执行至retn语句

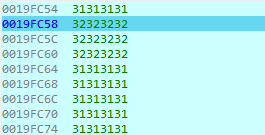


图 25 执行至retn语句时栈状态栏

可以看到此时地址为0019FC58的内存空间被写入了32323232，也就是溢出完成了。终止执行程序。

1. 构造特定输入

通过前面的分析可以发现，如果在构造的255 + x字节的数据中，前0x10 + 4字节中的后4个字节为我们的目标函数sub\_401230的偏移地址，就可以将图25中的返回地址修改为00401230，进而执行目标函数。

对应于构造的输入文件的位置即如图26所示，因此将这部分内容更改为00401230，即目标函数地址。注意此处需要保持低字节在前，高字节在后，所以按照字节顺序应为：30 12 40 00。构造二进制文件如图26所示。

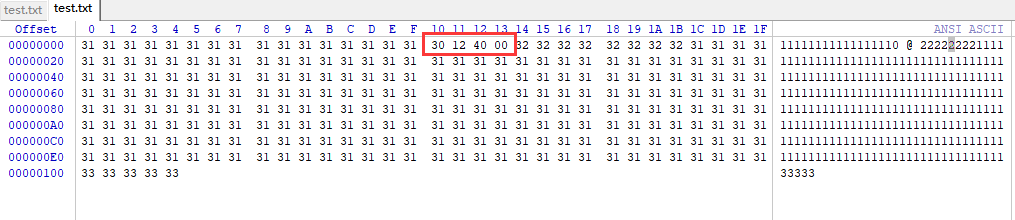


图 26 构造二进制文件输入

1. 测试

基于图26构造的文件，测试程序，成功显示了“Success!”和Flag字段。

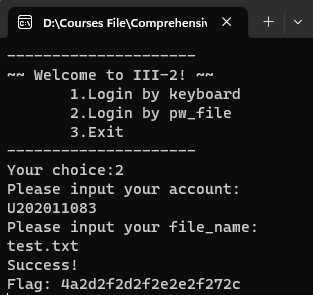


图 27 测试结果

信安2004班张楷奇

U202011083