《软件安全》实验报告

姓名：蒋枘言　学号：2313546　班级：信息安全班

一、实验名称

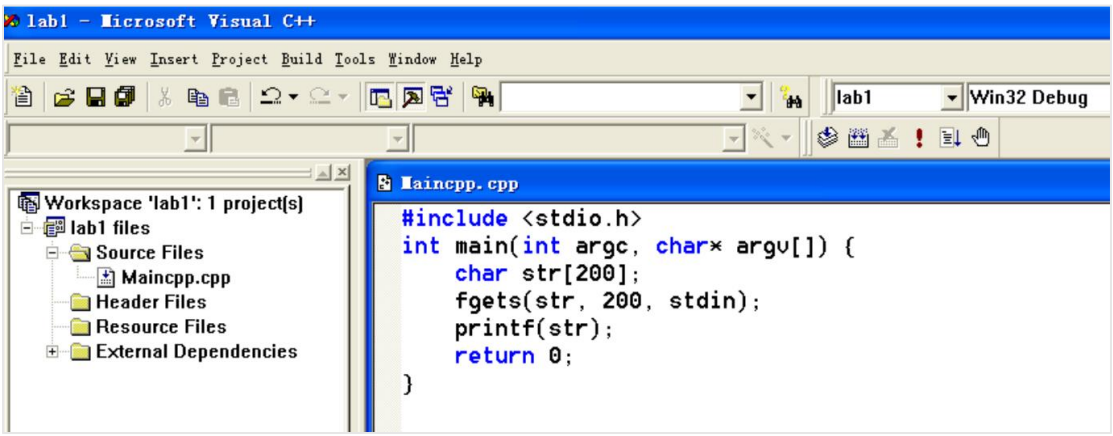
格式化字符串漏洞

二、实验要求

以第四章示例 4-7 代码，完成任意地址的数据获取，观察 Release 模式和 Debug 模式的差异，并进行总结。

三、实验过程

首先打开 VC6，编写如下图所示的具有潜在格式化字符串漏洞的程序，使用两种模式（Debug 模式和 Release 模式）运行。

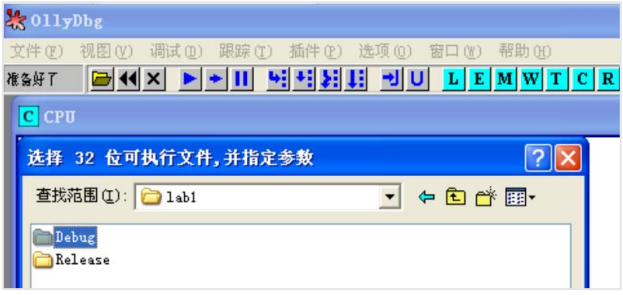


其中 fgets 是 C 语言中一个输入函数，用于从文件流（如标准输入 stdin）取字符串。相比 gets 函数，它通过明确指定缓冲区大小，避免了缓冲区溢出的风险。函数原型为：

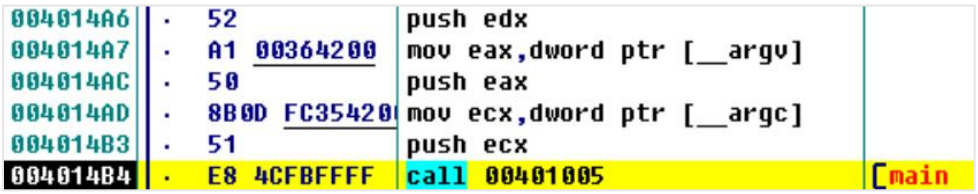
```
char* fgets(char* str, int n, FILE* stream);
// 参数：
// str：指向字符数组的指针，用于存储读取的内容（目标缓冲区）。
// n：最多读取的字符数（包括最后的空终止符\0，相当于实际最多只能读取 n-1 个字符）。
// stream：输入流（如 stdin 表示键盘输入，或文件指针）。
```

（一）Debug 模式

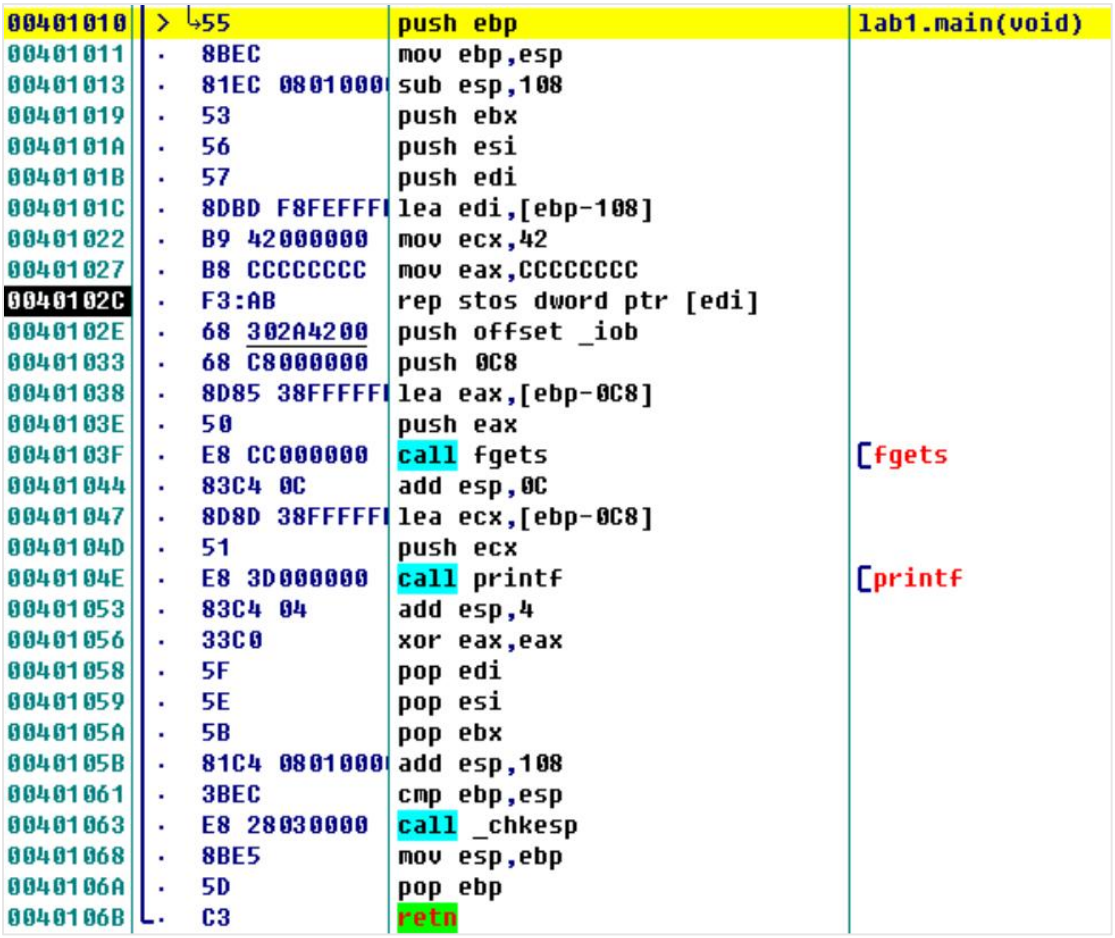
使用 Debug 模式生成 exe 文件，然后使用 OllyDbg 打开，进行调试。



找到 3 个 push 操作，并且发现参数 argv 和 argc，由此找到主函数的入口（地址为 00401005），按 F7 进入主函数内部。



进入 main 函数内部后，接着这几行首先保存函数状态（几个 push 操作），接着用 CC 填充未初始化的栈空间（rep stos 操作）。在之前的实验中已经遇到过多次，此处不再赘述。



接下来是重点：

```
0040102E    push    offset _iob     ; 参数 stdin 入栈
00401033    push    0C8             ; 参数 200（十六进制 C8）入栈
00401038    lea     eax,[ebp-0C8]   ; ebp（main 函数栈帧基址）减去传入 str 的大小
                                    ; 即为 str 地址
0040103E    push    eax             ; 参数 str（字符串的地址）入栈
0040103F    call    fgets           ; 调用 fgets 函数
```

调用 fgets 函数，此时输入以下包含格式控制符的字符串：AAAA%x%x%x%x（注意，按 F8 执行该语句之后才能输入）。

输入之后，通过右侧的栈区能够看到 0012FEB8 位置（即 str 的地址）已经被填充成字符串“AAAA%x%x%x%x”。（注意，下图最左边一列是栈区的地址，第二列是相应栈区地址存放的数据。例如，栈区 0012FE60 处存放的数据是 0012FEB8，这个数据的含义是指向用户输入字符串“AAAA%x%x%x%x”的地址。）

```
0012FE60  0012FEB8  ASCII "AAAA%x%x%x%x"
0012FE64  000000BB
0012FE68  00422A30  offset lab1._iob
0012FE6C  0012DA06
0012FE70  0012ADB4
0012FE74  7FFDA000
0012FE78  CCCCCCCC
0012FE7C  CCCCCCCC
```

查看 0012FEB8 位置的数据：

```
0012FEB8  41414141  AAAA
0012FEBC  78257825  %x%x
0012FEC0  78257825  %x%x
```

A、%、x 的 ASCII 码分别为 41、25、78（十六进制）。（注意是小端序）

```
00401044    add     esp,0C
00401047    lea     ecx,[ebp-0C8]
0040104D    push    ecx
0040104E    call    printf
```

接下来这几行首先恢复栈顶，然后相当于把 str 的地址赋给了 ecx 寄存器，作为函数 printf 的参数入栈。

```
寄存器 (FPU)
EAX 0012FEB8 ASCII "AAAA%x%x%x%x"
ECX 0012FEB8 ASCII "AAAA%x%x%x%x"
```

按 F8 执行 printf 函数，发现输出以下结果：

```
C:\Program Files\Microsoft Visual Studio\MyProjects\lab1\Debug\lab1.exe
AAAA%x%x%x%x
AAAA12da0612adb47ffda000cccccccc
```

这是由于传入的字符串%x 被当成格式控制符，意义是以十六进制形式输出整数。由于没有传入相应实际的参数，printf 函数直接打印栈中相应高地址的值。

```
0012FE68  0012FEB8  ASCII "AAAA%x%x%x%x"
0012FE6C  0012DA06
0012FE70  0012ADB4
0012FE74  7FFDA000
0012FE78  CCCCCCCC
0012FE7C  CCCCCCCC
0012FE80  CCCCCCCC
0012FE84  CCCCCCCC
```

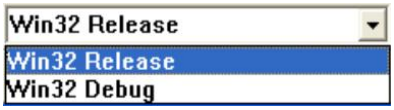
有 4 个%x，对应往高地址多读 4×4 个字节

储存“AAAA”的位置在 0012FEB8 处，与传入 printf 函数的参数之间隔了很长的 CC 填充的区域，这是由于 Debug 模式下开辟了足够大的栈帧。所以 Debug 模式下要读到“AAAA”的位置需要很多格式控制符%x。

```
0012FEA0  CCCCCCCC
0012FEA4  CCCCCCCC
0012FEA8  CCCCCCCC
0012FEAC  CCCCCCCC
0012FEB0  CCCCCCCC
0012FEB4  CCCCCCCC
0012FEB8  41414141
```

（二）Release 模式

使用 Release 模式生成 exe 文件，然后使用 OllyDbg 打开，进行调试。



找到 3 个 push 操作，并且发现参数 arg1、arg2、arg3，由此找到主函数的入口（地址为 00401005），按 F7 进入主函数内部。

```
00401213  .  50              push eax                 Arg3 => [406900] = ASCII "À■A"
00401214  .  FF35 F8684000   push dword ptr [4068F8]  Arg2 = 410E90
0040121A  .  FF35 F4684000   push dword ptr [4068F4]  Arg1 = 1
00401220  .  E8 DBFDFFFF     call 00401000            lab1.00401000
```

可以看到此时直接就进入了 main 函数内部，Release 模式栈帧的切换也和 Debug 模式不一样，栈的分配显得比之前要紧凑了一些，每个变量之间都靠在一起。

```
00401000  r$ 81EC C8000000   sub esp,0C8             lab1.00401000(guessed Arg1,Arg2,Arg3)
00401006  .  8D4424 00       lea eax,[esp]
0040100A  .  68 30604000     push offset 00406030
0040100F  .  68 C8000000     push 0C8
00401014  .  50              push eax
00401015  .  E8 47000000     call 00401061
0040101A  .  8D4C24 0C       lea ecx,[esp+0C]
0040101E  .  51              push ecx
0040101F  .  E8 0C000000     call 00401030
00401024  .  33C0            xor eax,eax
00401026  .  81C4 D8000000   add esp,0D8
0040102C  L. C3              retn
```

```
00401000    sub     esp,0C8          ; 将栈顶抬高 C8 字节（十进制 200）
; 无 ebp 等寄存器的变化，仅仅只是为局部变量 str 分配了栈空间
00401006    lea     eax,[esp]        ; 将栈顶位置的值（即 str 的地址）赋给 eax
0040100A    push    offset 00406030  ; 参数 stdin 入栈
0040100F    push    0C8              ; 参数 C8（十进制 200）入栈
00401014    push    eax              ; 参数 str（地址）入栈
00401015    call    00401061         ; 调用 fgets 函数
```

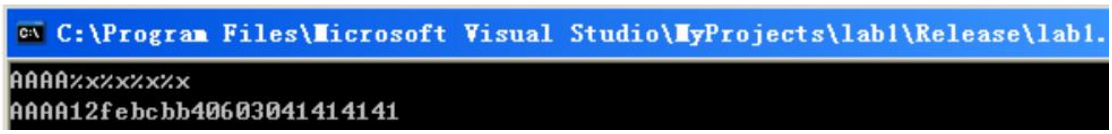
调用 fgets 函数，此时输入以下包含格式控制符的字符串：AAAA%x%x%x%x（注意，同样是按 F8 执行该语句之后才能输入）。

输入之后，通过右侧的栈区能够看到 0012FEB0 位置（即 str 的地址）已经被填充成字符串“AAAA%x%x%x%x”。

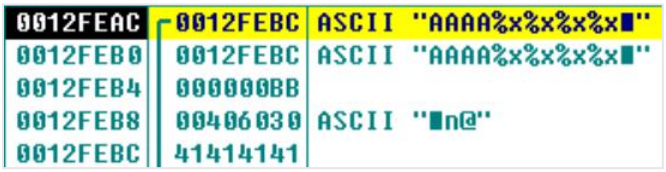
```
0012FEB0  0012FEBC  ASCII "AAAA%x%x%x%x■"
0012FEB4  000000BB
0012FEB8  00406030  ASCII "■n@"
0012FEBC  41414141
0012FEC0  78257825
0012FEC4  78257825
0012FEC8  0012000A
0012FECC  20202020
```

```
0040101A    lea     ecx,[esp+0C]
0040101E    push    ecx
0040101F    call    00401030
```

和 Debug 模式很相似，也是把 str 的地址赋给了 ecx 然后入栈，接着调用 printf 函数。发现输出了以下结果：



同样此时按照顺序直接打印了栈中越界的高地址储存的值。



如果将 AAAA 换成想要读取的地址，再将第 4 个“%x”换成“%s”，这样就能读到指定地址上的值了，展现了格式化字符串的漏洞。

（三）Debug 模式和 Release 模式的比较

以前我从来都不了解 Debug 模式和 Release 模式的区别，通过这次实验外加自行查阅的资料有了一个初步的了解。二者的主要区别体现在以下方面：

1.调试信息：

Debug 模式会保留完整的调试符号（如变量名、行号信息），并生成符号表供调试器使用；Release 模式则会移除所有调试信息以减小文件体积，并隐藏内部实现细节。

2.栈操作：

Debug 模式通过 **ebp 基址寄存器**维护栈帧结构，确保变量访问与栈平衡可追踪（例如通过[ebp-0xC8]定位局部变量）；Release 模式直接使用 **esp 栈顶指针**动态管理栈空间（如[esp+0xC]），省略 ebp 相关指令以减少性能损耗。

3.参数传递：

Debug 模式严格按照 cdecl 调用约定逐层压栈参数，且显式调整栈指针（如 add esp, 0xC）；Release 模式可能合并栈调整操作，甚至通过寄存器传递部分参数以提升效率。

4.优化策略：

Debug 模式禁用编译器优化（如函数内联、循环展开），确保代码与源码逐行对应；Release 模式启用多级优化（O1/O2/O3），通过指令重排、冗余代码消除等方式提升运行速度和资源利用率。

综上，Debug 模式以牺牲性能为代价提供可调试性和安全性，适用于开发阶段；而 Release 模式通过精简和优化实现高效执行，更适合作为最终交付版本。

四、心得体会

通过此次实验，我更加深刻地体会到了程序安全的重要性。在探究格式化字符串漏洞的过程中，我观察到通过简单的%x 或%s 等格式控制符，攻击者即可绕过正常的内存访问机制，直接读取甚至篡改栈中的敏感数据。这种漏洞的存在让我意识到，即使程序在功能逻辑上完全正确，若缺乏对输入数据的严格校验和内存访问的安全性设计，仍可能成为攻击者利用的入口。

实验中，最令我震惊的是“看似正常”的代码竟潜藏如此高风险。例如，使用 printf 直接输出用户输入时，未对格式字符串进行过滤，导致本应作为普通字符串处理的输入被解析为指令，进而引发越界内存访问。

此次实践也让我关注程序“正确性”与“安全性”的关系。一个能正确运行的代码可能在安全性上漏洞百出。尤其在大型项目中，模块间的复杂交互会放大单一漏洞的影响范围。因此，安全设计必须贯穿需求分析、编码、测试的整个过程。未来在参与开发时，我也会将安全性视为与功能实现同等重要的维度。