一、课程设计题目

基于源代码的软件同源性分析与漏洞检测系统

二、课程编码：130681

课程性质：必修

三、学时学分

学时：2 周

学分：1 学分

四、先修课程

C 语言、汇编语言、数据结构、软件安全

五、课程设计任务与要求

下述任务可以自己选择 Windows 或者 Linux 平台实现,检测的源代码可以为 c/c++/Java.跨语言同源性为 c 语言与其它语言的对比检测。

(1)在 Linux 下面使用 gcc 编译.

(2)Win7/Win10,VC++(VS2013/VS2015/VS2019),QT,Python

本次课程设计要求设计与完成从界面、算法到系统优化等各个环节内容，形成完整的软件系统。在任务中，必做部分为 R1-R6,是所有

同学必须完成的部分，是本课程设计考核合格所需要的基本构成，选做部分根据自己的擅长做出最好的成果。

具体任务如下：

序号	任务（必须完成的）	要求
R1	提供系统界面	所有功能要有图形界面展示,形成完整的软件系统.可以使用 VS/QT/Python 等工具实现。
R2	利用字符串匹配进行同源性检测	通过代码有效字符串对比匹配，分析样本之间的拷贝比率
R3	利用控制流程图 CFG 进行源代码同源性检测	通过提取代码的调用关系图，检测样本之间各个函数调用关系图是否相似，得出相似的概率
R4	栈缓冲区检测	根据栈缓冲区原理分析分配的栈数据区是否存在溢出的问题，给出可疑代码行数与列数。
R5	格式化字符串漏洞检测	根据格式化字符串漏洞原理分析使用的格式化函数是否存在溢出的问题，给出可疑代码行数与列数。
R6	提供样本库	提供漏洞检测与同源性检测样本库，样本数量不少于 10 个，每个代码行数不少于 100 行；每种漏洞至

		少一个。
以下为 2 选 1:		
A1	跨语言同源性检测验证	在软件版权保护中，有时候需要检测是否参考了有版权的代码，换用一种语言实现同样的功能，本功能可以通过 CFG 检测实现，但需要给出 4 个以上不用语言的同源性分析样本。
A2	支持分布式任务调度	需要设计一个主控，多个进程/主机并发检测。
以下为 6 选 4:		
B1	堆缓冲区检测	根据堆缓冲区原理分析分配的数据区是否存在溢出的问题，给出可疑代码行数与列数。
B2	整数宽度溢出检测	根据整数宽度溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。
B3	整数运算溢出检测	根据整数运算溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。
B4	整数符号溢出检测	根据整数符号溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。

B5	空指针引用	根据课堂学习其它溢出原理是否存在空指针引用的问题，给出可疑代码行数与列数。
B6	竞争性条件	给出竞争性条件存在的代码位置。
以下 2 选 1:		
C1	同源性检测样本库	样本数大于等于 50 个，每个代码行数不少于 100 行,包含 1-100 行相同代码。
C2	漏洞检测样本库	样本数大于等于 50 个，每个代码行数不少于 100 行；每种漏洞至少一个。

六．实验过程

界面部分可以 VS/QT/Python 等工具实现，样例省略，下面为具体任务功能的例子供参考。

1.利用字符串匹配进行同源性检测

任务：利用字符串匹配进行同源性检测

要求：通过代码有效字符串对比匹配，分析样本之间的拷贝比率

输入样例 1A（原始代码）（C 语言）：

```
1. static int do_poll(struct pllfd *pfd,unsigned int nfds,int timeout)
2. {
3.     int ret;
4.     ret=poll(pfd,nfds,timeout);
5.     if (ret<0){
6.        if(errno==EINTR)
7.           return 0;
8.        log(LOG_ERR,”poll returned an error!\n”);
```

```
9.        return -1;
10.    }
11.    if (ret==0)
12.        return 0;
13.     return 1;
14.}
```

输入样例 1B（经过修改的代码）（C 语言）：

```
1.  typedef int INTEGER;
2.  typedef INTEGER int16;
3.  static int16 do_poll(struct pllfd *pfd,unsigned int nfds,int16
    timeout)
4.  {
5.     int16 ret;
6.     ret=poll(pfd,nfds,timeout);
7.     if (ret<0){
8.        if(errno==EINTR)
9.           return 0;
10.        log(LOG_ERR,” poll returned an error!\n”);
11.        return -1;
12.    }
13.    if (ret==0)
14.        return 0;
15.     return 1;
16.  }
```

输出：样例 1B 与样例 1A 的相同单词的单词及出现位置，计算样例 1B 拷贝样例 1A 的比率。

实验指导

软件的同源性分析是计算机编程语言研究的重要方面之一，主要有三类方向：基于文本的同源性比对、基于单词（token）的同源性比对、基于源代码语法结构的比对。

基于文本的同源性鉴别的目的,是查找出文本上的代码抄袭。常见文本抄袭手段有文本的*完全拷贝*、*增加删除行*、*顺序无关代码块调换*等。典型的基于文本相似性的鉴别技术有：基于子串匹配的方法,其基本思路是从文档中选取一些字符串,被称为“指纹”,然后把指纹映射到 hash 表中,一个指纹对应一个数字,最后统计 hash 表中相同的

指纹数目或者比率,作为文本相似度依据。这种只对源代码的部分特征进行统计，忽略源代码的结构信息,导致错误率较高；其次，参数化匹配方法（Brenda S.Baker）,解决了变量名替换问题。此外,还有典型的词频统计法用于同源性鉴别。源代码抄袭的过程一般都是整块复制,然后加以改动,比如替换变量名,在不影响程序功能的情况下打乱语句顺序,更改函数名或者函数位置等等。由于基于文本层次的鉴别完全忽略了源代码的语法含义,因而对于上述大部分改动情况都没有理想的效果。

基于单词（token）的比对（参考 CP-Miner，CCFinder）。**参考思路：首先对源代码文本进行一定层次的预处理,即采用该工具自己定义的一些一记号来标记原来的源代码文本,然后将这些经过处理得出的中间数据作为比对的对象,以多种规则进行比对,最后再对这些比对得出来的结果进行综合评价。**

具体实现：基于 token 比对的软件同源性检测技术的思想是将源代码中各种函数名称、变量、参数、类型等标识符、操作符、数字、关键字通过词法分析器转化为 token,将源代码的比对技术转化成 token 序列的比对。

1） 对源代码进行预处理。即对源代码中的注释、类型重定义、头文件包含语句、宏定义、内联函数等等进行相应处理。由于一些字符不影响语义,故在预处理中记录为空,如宏定义、注释、TAB、回车、空格等。类型重定义的情况都根据其语义进行处理,比如类型重定义语句 typedef int INTEGER 的处理就是将文件中的 INTEGER 都替换为 int,这样就能提高同源性检测的准确度。

2） 对源代码进行词法分析。该过程是用词法分析工具 Lex(Lexical compiler)实现,它生成的词法分析器按字符读入源代码信息,然返回单词 Token。Lex 转化字符为 token 包括：

标识符转化为_ID，如：mystringname->_ID，操作符转化为对应字母，如： + ->_PLUS，数字转化为_NUM,如：100 - >_NUM，关键字转化为对应字母，如：int->_INT。这样便于统一比较。利用词法分析工具 lex 编写词法分析程序,将源码转化成 token 序列。

下图说明源代码转换成 token 的过程：

```
17. int func（int x）
18. {
19.    int m;
20.    m=f1(x);
21.    return m
22. }
```

转换成 token 如下：

```
1. _INT _ID _LP _INT _ID _RP
2. _LSP
3. _INT _ID _SEMI
4. _ID _ASSIGN _ID _LP _ID _RP _SEMI
5. _RETRUN _ID _SEMI
6. _RSP
```

这样标识符的不同就被屏蔽了。

3） 将 token 序列输出，建立基准文件和目标文件的行链表，按行为单位进行对比并存入行链表，计算相似度。

这个方法在一定程度上考虑了语言特点，但也存在一些问题：其原理是找源代码中最长的相似子串,而不能应对代码顺序调换这样的抄袭情况。如何改进呢？可以参考下面的实验进行。

2.利用控制流程图 CFG 进行源代码同源性检测

实验内容:

利用控制流程图 CFG 进行源代码同源性检测（开始不要太复杂，由简单的程序，多加一些跳转，以便获取 CGF，注意图的表示方法，数组、链表等）

通过提取代码的调用关系图，检测样本之间各个函数调用关系图是否相似，得出相似的概率

输入样例 2A（原始代码）（C 语言）：

```
int main()
{
     int a,b,c;
        a=func_A(a);
        b=func_B(b);
        c=a+b;
           return func_C(c);
    }
```

输入样例 2B（经过修改的代码）（C 语言）：

```
int main(){
  int d,e,f;
    e=func_B(e); // calculate e
  d=func_A(d); // calculate d
  f=d+e;
    return func_C(f); // calculate the final result

}
```

下面是 C 语言编写的 DES 加密算法的一部分样例：

输入样例 3A（原始代码）（C 语言）：

```
int DES_Encrypt(char *plainFile, char *keyStr,char *cipherFile){
    FILE *plain,*cipher;
    int count;
    ElemType plainBlock[8],cipherBlock[8],keyBlock[8];
    ElemType bKey[64];
    ElemType subKeys[16][48];
    if((plain = fopen(plainFile,"rb")) == NULL){
        return PLAIN_FILE_OPEN_ERROR;
```

```
    }
     if((cipher = fopen(cipherFile,"wb")) == NULL){
         return CIPHER_FILE_OPEN_ERROR;
     }
     //设置密钥
     memcpy(keyBlock,keyStr,8);
     //将密钥转换为二进制流
     Char8ToBit64(keyBlock,bKey);
     //生成子密钥
     DES_MakeSubKeys(bKey,subKeys);

     while(!feof(plain)){
         //每次读 8 个字节，并返回成功读取的字节数
         if((count = fread(plainBlock,sizeof(char),8,plain
)) == 8){
             DES_EncryptBlock(plainBlock,subKeys,cipherBlo
ck);
             fwrite(cipherBlock,sizeof(char),8,cipher);

         }
     }
     if(count){
         //填充
         memset(plainBlock + count,'\0',7 - count);
         //最后一个字符保存包括最后一个字符在内的所填充的字符数
量
         plainBlock[7] = 8 - count;
         DES_EncryptBlock(plainBlock,subKeys,cipherBlock);

         fwrite(cipherBlock,sizeof(char),8,cipher);
     }
     fclose(plain);
     fclose(cipher);
     return OK;
 }
```

输入样例 3B（经过修改的代码）（C 语言）：

```
int DES_Encrypt(char *plainFile, char *keyStr,char *cipher
File){
    ElemType plainBlock[8],cipherBlock[8],keyBlock[8];
    ElemType bKey[64];
    ElemType subKeys[16][48];
    FILE *plain,*cipher;
```

```
    int count;
    if((plain = fopen(plainFile,"rb")) == NULL){
  return PLAIN_FILE_OPEN_ERROR; }
    if((cipher = fopen(cipherFile,"wb")) == NULL){
        return CIPHER_FILE_OPEN_ERROR;
     }  //设置密钥

     memcpy(keyBlock,keyStr,8);  //将密钥转换为二进制流
     Char8ToBit64(keyBlock,bKey);   //生成子密钥
     DES_MakeSubKeys(bKey,subKeys);

     while(!feof(plain)){
         //每次读 8 个字节，并返回成功读取的字节数
         if((count = fread(plainBlock,sizeof(char),8,plain
)) == 8){
             DES_EncryptBlock(plainBlock,subKeys,cipherBlo
ck);
             fwrite(cipherBlock,sizeof(char),8,cipher);

         }
     }
     if(count){
         //填充
         memset(plainBlock + count,'\0',7 - count);
     //最后一个字符保存包括最后一个字符在内的所填充的字符数量
         plainBlock[7] = 8 - count;
         DES_EncryptBlock(plainBlock,subKeys,cipherBlock);

         fwrite(cipherBlock,sizeof(char),8,cipher);
     }
     fclose(plain);
     fclose(cipher);
  return OK;
 }
```

实验指导

现在已经有一些从源代码语法结构的层次进行代码克隆检测和同源性比对的研究成果,比如利用欧式向量空间记录语法树信息进行比对（DECKARD)；用 Xml 记录语法树信息来比对。另外还有在线比对，如 Moss，JPlag 等（JPlag 在一定程度上考虑了语法）,所有的在线比

对系统需要提交源码,源代码的保密工作就可能无法保证。Baxter 等人提出了一种利用源代码抽象语法树进行代码克隆检测的算法,算法在整个比对过程中都保持了树形结构,需要进行多次树的遍历。

与第一个实验中的单词符号 token 比对类似，先通过预处理，可以去除程序中的注释、空白符号等，只保留语言有效的单词 token，对于获得的单词 token，按语言的词法规则将单词分类（参考编译程序的编写),比对中不关心单词的名称，而只注重单词的类型比较（当然，名称对于值传递有作用)。然后按该语言的语法规则，对于各种语法成分，用语法树形式存放并比较。这里我们因为课设时间有限，进行一定的简化，我们只保留*函数调用*的语法成分，在分析时只构造合适的存储结构（比如树形结构)，保留函数调用的相互关系（包括函数名、参数类型等)，形成*简化的 CFG*。将原始程序对应的 CFG 图与修改后的代码对应的 CFG 图进行比较，计算相似程度。这里需要同学们自己定义相似度的计算规则。

3.跨语言同源性检测验证

实验要求

任务：跨语言同源性检测验证

要求:在软件版权保护中,有时候需要检测是否参考了有版权的代码,换用一种语言实现同样的功能，本功能可以通过 CFG 检测实现，但需要给出 4 个以上不用语言的同源性分析样本。(通过开源软件库搜集典型样例进行比较)。下面给出比较简单的示范样例：

输入样例 4A（原始代码）(C 语言）

```
1.  #include “stdio.h”
2.   main()
3.   {
4.         fprintf(stdout,”hello world!\n”);
5.        exit(0);
```

```
6.     }
```

输入样例 4B（经过修改的代码）（python 语言）

```
1.  #!/usr/bin/env python
2.  print('Hello World!')
```

输入样例 4C（经过修改的代码）（java 语言）

```
1.  public class HelloWorld{
2.    public static void main(String[] args){
3.        System.out.println("Hello World!");
4.     }
5.  }
```

输入样例 4D（经过修改的代码）（perl 语言）

```
1.  #!/usr/local/bin/perl
2.  print "Hello World!";
```

3.2 实验指导

跨语言的检测较复杂，需要掌握不同语言的语法规则，通过词法分析及部分语法分析，得到语法树。类似上一个实验的简化方案，我们只想办法获得不同语言函数调用的简化 CFG，存储并比较即可。

4.栈缓冲区溢出检测

栈缓冲区检测主要是根据栈缓冲区原理分析分配的栈数据区是否存在溢出的问题，给出可疑代码行数与列数。

原理说明：

以下面代码为例：

```
int AFunc(int i,int j)
{
     int m = 3;
     int n = 4;
    char szBuf[8] = {0};
    strcpy(szBuf, “This is a overflow buffer!”);
    m = i;
    n = j;
```

```
    return 8;
}
```

上述代码栈缓冲区 szBuf 只有 8 个字节，在通过 strcpy 拷贝时候并没有检测边界，只要拷贝超过 8 字节，则会存在栈缓冲区溢出的问题。本任务要求是：

(1) 分析代码中是否存在可疑函数的使用；

(2) 分析可以函数调用参数存在溢出的可能性；

(3) 自己设计一个规则来判断溢出的可能性；

(4) 输出溢出的代码位置。

检测过程实现：

(1) 通过代码扫描确定函数的开始 FuncionBegin 与结束行 FuncionEnd；函数的开始可以根据 c 语言规则，从代码第一行开始扫描，以符号“{”， “}”判断函数的开始位置与结束位置,注意,开始位置与结束位置是指距离文件头的距离，而不是开始的行数与结束的行数，原因是有些代码中函数的开始与结束符号可能并不是独立的一行；

(2) 将所有函数的开始与结束行作为一个结构元素存入一个数组或者链表，也可以用 c++中的列表模板存储；

(3) 对每个函数开始与结束的代码块进行扫描；

(4) 扫描得到该函数所有的局部变量定义，得到变量名与该变量名所占用的空间大小，以结构形式存入一个数组或者链表；

(5) 扫描该函数每一行代码,检查代码中是否存在敏感函数调用，包 括 strcpy,strncpy,memcpy,memncpy,strcat,strncat,

sprintf(),vsprintf(),gets(),getchar()、fgetc()、getc()、read(),sscanf()、fscanf()、vfscanf()、vscanf()、vsscanf()等函数，如果存在，则记录其代码所在位置；

(6) 对上述函数的调用参数进行检查，是否与栈变量相关，判断存在栈溢出的可能性；

(7) 根据存在溢出的可能性，给出判断结论，并存入相应数据结构。

注意事项：(1)本课程设计一般不考虑预编译的情况。也就是如果存在预编译条件，代码扫描不予考虑。

（2）一般不考虑栈变量的传递，即：如果栈变量在父函数申请，在子函数使用存在溢出的情况可以不予考虑。

（3）溢出判断范围限定在扫描的函数范围之内，不考虑子函数是否存在溢出的情况。

（4）一般只考虑 c 语言溢出情况，c++作为可选分析对象。

(5)判断可疑溢出函数作为一个基本实现要求，溢出的可能性判断由自主设计的机制决定。

5.堆缓冲区溢出检测

堆缓冲区检测主要是根据堆缓冲区原理分析分配的堆数据区是否存在溢出的问题，给出可疑代码行数与列数。

原理说明：

以下面代码为例：

```
char mybuf2[450] ;
```

```
int main (int argc, char *argv[])
{
 HANDLE hHeap;
 char *buf1, *buf2;
 int i;

for(int i=0;i<450;i++){
 mybuf2[i]=' a' ;
}
LoadLibrary("user32");
 hHeap=HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x10000, 0xfffff);
 buf1 =(char*) HeapAlloc(hHeap, 0, 200);
 strcpy(buf1,mybuf2);
 buf2 =(char*) HeapAlloc(hHeap, 0, 200);
 HeapFree(hHeap, 0, buf1);
 HeapFree(hHeap, 0, buf2);
 return 0;
}
```

上述代码堆缓冲区 buf1 只有 200 个字节，在通过 strcpy 拷贝时候并没有检测边界，只要拷贝超过 200 字节，则会存在堆缓冲区溢出的问题。本任务要求是：

(1) 分析代码中是否存在可疑函数的使用；

(2) 分析可以函数调用参数存在堆溢出的可能性；

(3) 自己设计一个规则来判断溢出的可能性；

(4) 输出溢出的代码位置。

具体实现过程为：

(1) 进行函数扫描，与栈溢出相同。

(2) 扫描该函数每一行代码，检查代码中是否存在敏感函数调用，包括 strcpy,strncpy,memcpy,memncpy,strcat,strncat,sprintf(),vsprintf(),gets(),getchar()、fgetc()、getc()、read(),sscanf()、fscanf()、vfscanf()、vscanf()、vsscanf()

等函数，如果存在，则记录其代码所在位置；

(3) 对上述函数的调用参数进行检查，是否与堆变量相关，判断存在堆溢出的可能性；

(4) 根据存在溢出的可能性，给出判断结论，并存入相应数据结构。

注意事项：与栈溢出检测相同。

6.整数宽度溢出检测

根据整数宽度溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。

分析原理说明：

将一个较大整型转换为较小整型，并且该数的原值超出较小类型的表示范围，就会发生截断错误引起宽度溢出，原值的低位被保留而高位被丢弃。宽度溢出会引起数据丢失也可能会导致数据缓冲区溢出问题。

以下面代码为例：

```
void main(int argc,char* argv[])
{
    unsigned short s;
    int i;
    char buf[80];
    i = atoi(argv[1]);
    s = i;
```

```
    if(s >= 80)
        return;
    memcpy(buf,argv[2],i);
}
```

其中 s 是一个短整型，i 可能是一个很大的值，在赋值给 s 时候被截断，丢失高位数据，可能变成一个很小的值小于 80，这样会通过长度检查，导致缓冲区溢出。

本任务要求是：

(1) 分析代码中是否存在宽整型到短整型的转换；

(2) 自己设计一个规则来判断这种转换导致溢出的可能性；

(3) 输出整型转换与溢出的代码位置。

具体实现过程为：

(1) 进行函数扫描，与栈溢出相同。

(2) 扫描该函数每一行代码，检查代码中是否存在赋值操作，检查赋值操作左右两边类型，如果存在右边整型宽度大于左边，则记录其代码所在位置；

(3) 对上述函数的后续代码进行检查，是否与缓冲区变量相关，判断存在溢出的可能性；

根据存在溢出的可能性，给出判断结论，并存入相应数据结构。

7.整数运算溢出检测

根据整数运算溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。

分析原理说明：

当两个整数进行运算比如乘法或者加法的时候，可能运算的结果超出整数的表示范围，导致被截断变成较小的整数，这种运算结果被截断会引起数据丢失也可能会导致数据缓冲区溢出问题。

以下面代码为例：

```
void CopyIntArray(int *array,int len)
{
    int* myarray,i;
    myarray = malloc(len*sizeof(int));
    if(myarray == NULL)
        return;
    for(i=0;i<len;i++)
        myarray[i] = arrary[i];
}
```

其中 len 可能是一个很大的值，在与 sizeof(int)做乘法运算的时候，其结果可能超出整数表示范围导致被截断，丢失高位数据，可能变成一个很小的值小于 len,这样会通过长度检查,导致缓冲区溢出。

下面的代码也存在运算溢出的问题：

```
int main(int argc, char* argv[])
{
unsigned short total = strlen(argv[1]) + strlen(argv[2]) + 1;
char* buffer = (char*)malloc(total);
strcpy(buffer, argv[1]);
strcat(buffer, argv[2]);
free(buffer);
return 0;
  }
```

本任务要求是：

(1) 分析代码中是否存在整数的加法与乘法运算；

(2) 自己设计一个规则来判断这种转换导致溢出的可能性；

(3) 输出整数运算与溢出的代码位置。

具体实现过程为：

(1) 进行函数扫描，与栈溢出相同。

(2) 扫描该函数每一行代码，检查代码中是否存在整数运算操作，判定运算是否可能超出表示范围，有则记录其代码所在位置；

(3) 对上述函数的后续代码进行检查，是否与缓冲区变量相关，判断存在溢出的可能性；

根据存在溢出的可能性，给出判断结论，并存入相应数据结构。

8.整数符号溢出检测

根据整数符号溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。

分析原理说明：

当整数的符号处理不当，可能导致结果异常，比如：一个负数在被强制转换为无符号整数的时候，则会变成一个较大的整数，从而可能会导致数据缓冲区溢出问题。

以下面代码为例：

```
static char data[256];
int store_data(char *buf, int len)
```

```
{
    if(len > 256)
        return -1;
    return memcpy(data, buf, len);
}
```

其中 len 可能是一个负值，判断 len>256 的时候，因为是负数，所以该判断不通过，从而继续执行，当执行 memcpy(data, buf, len)的时候，由于 memcpy 第三个参数为 DWORD,无符号整数，则 len 会被转换为无符号的整数，一个负数变成一个很大的无符号整数，导致拷贝的目标缓冲区溢出。

本任务要求是：

(1) 分析代码中是否存在整数的有符号与无符号转换运算；

(2) 自己设计一个规则来判断这种转换导致溢出的可能性；

(3) 输出整数符号溢出的代码位置。

具体实现过程为：

(1) 进行函数扫描，与栈溢出相同。

(2) 扫描该函数每一行代码，检查代码中是否存在整数符号转换，有则记录其代码所在位置；

(3) 对上述函数的判定符号转换是否会导致缓冲区溢出问题，记录其溢出的位置；

根据存在溢出的可能性，给出判断结论，并存入相应数据结构。

9. 格式化字符串漏洞检测

熟悉原理：格式化字符串(Format String)漏洞是一种常见的软件漏洞。格式化字符串漏洞产生的原因是编程语言中数据输出函数中对输出格式解析的缺陷。格式化字符串函数按照特定格式（如%x，%s，%d 等格式化控制符）对数据进行输出。格式化字符串被黑客利用主要基于以下原因：

1. 编程语言 C 和 C++都没有检查格式化函数的参数个数和参数类型是否完全匹配的机制。

2. 控制符%n 可以把当前输入的所有数据写入到内存中，这就为黑客写入 shellcode 提供了方便。

实验指导：

根据格式化字符串产生的原因，实验也分为三个部分：

1）粗定位：对给出的 C 或 C++的程序，能够判断出字符串的申请，使用；

2）细定位：在粗定位的基础上进一步判定是否字符串格式化函数的参数个数、类型的匹配；

3）在细定位的基础上，程序中有无格式化字符串函数按照特定格式（如%x，%s，%d 等格式化控制符）对数据进行输出，并进一步判别相关具体值得操作。

输出相应的行号和结果；

例子

一个由 printf 函数不恰当使用产生格式化字符串漏洞的例子，如

图 9-1 所示。

```
#include “statio.h”
main(){
    int a=4, b=7;
    char buf[] = ‘demo’;
    printf(“%s %d d,buf=%s”,buf,a,b);
}
```

图 9-1　格式化字符串漏洞示例

此时打印出的结果为：demo,4,7。注意此时 printf 是按照%s %d %d 的顺序输入结果的，如图 2-6 所示。

当把 printf 语句换成：

printf(“%s%d%d%x”,buf,a,b);

此时 printf 中多了一个输出的格式化参数%x，输出结果为 demo,4,7, 64656D6F。把 64 65 6D 6F 转化成字符串得到“d e m o”，这恰好正是 buf 中存储的内容。

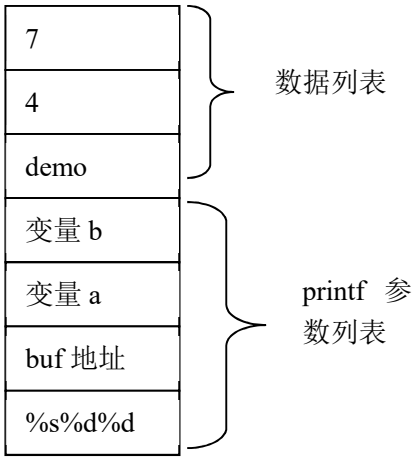

图 9-2　printf 函数调用时的内存布局

当 printf 中的格式控制符个数多于数据列表的个数时，程序中并没有检测参数的个数和类型是否匹配，导致多输出的格式控制符打印了原本不需要打印的数据。因此，当输入输出函数的格式化控制符受到外部输入影响时，黑客可以通过精心构造的格式化字符串来读取内存数据，甚至往内存写数据的方法来修改函数返回地址，引起进程劫持和 shellcode 植入和执行。

严格来讲，格式化字符串溢出不属于缓冲区溢出漏洞，是一种输入验证类漏洞，即程序没有在格式化字符串中过滤不被信任的外部输入数据。格式化字符串的静态分析方法也比较简单，大都是通过静态代码扫描可能会引起这种漏洞的格式化字符串系列函数，检查参数类型和个数是否匹配。比如传统的词法分析和 C 库增强技术等。近年来，又出现了通过制定各种规则，限制格式化字符串系列函数使用的静态分析技术。Ringenburg 等提出可写列表技术(white-lists)来检测和预防格式化字符串漏洞，设定内存的安全地址范围，当写地址指针在安全地址范围内时才允许写操作，效率较高。

10 空指针引用

原理：空指针引用(Null Pointer Dereference)是指当程序中方法或域对一个指针进行调用或引用时，该指针的值为 Null，从而导致程序崩溃或退出的一种程序缺陷。空指针引用包含两大类：调用为空的指针(CWE-476)和未检查函数返回值导致的空指针引用(CWE-690)。空指针引用产生的主要原因是程序中调用指针前没有对其进行非空的

判断。

实验指导：根据空指针的原理，当引用的指针指向了无效的地址，那么就可能会产生意想不到的错误。空指针引用会导致竞争条件(Race Condition)、拒绝服务攻击（DoS）等安全问题的发生。对所给出的 C 或 C++的源码编程语言中。

1）能够实现空指针的定位：定义的定位；使用的定位

2）对定位后的指针能根据简单的一层操作、引用等能判断是否存在空指针引用；

3）根据更广泛的使用，比如多层的嵌套，循环语句，依赖值得变动所形成的分支中的应用也能进行判断

一个空指针引用(CWE-476)的例子，如图 10-1 所示。

```
public void test() throws Throwable{
        int [] data;
        data = null;
        IO.writeLine("" + data.length);
}
```

图 10-1　空指针引用示例

在编码阶段，避免出现空指针引用缺陷的方法主要包括：

1. 在指针使用前对其进行安全检查，确保指针有效；

2. 检查所有函数的返回值，并在调用这些返回值之前，验证返回值是否为 null;

3. 检查所有外部输入的变量和数据，保证其是程序预期的值；

4. 在声明或第一次使用某个变量时，对其进行初始化。

相关的参照工具如 FindBugs、Saturn、Julia。

11 竞争条件

原理：竞争条件(Race Condition)是一种重要的常见软件安全漏洞，一般出现在多进程（线程）的程序中。多个进程并发访问和操作同一个共享资源，且执行结果与访问的特定顺序有关，这种情况称为竞争条件。

针对的是不正确同步并发执行的共享资源。该竞争条件产生的主要原因是在并发的前提下，当前对象在使用共享资源的过程中，其他对象同时也对该共享资源进行了修改操作，从而导致出现了资源竞争的现象。

判断依据，竞争条件违反了以下两个特征：

1. 排他性。在当前对象没有完全操作完共享资源时，其他对象不能操作修改该共享资源的属性。

2. 原子性。其他对象不能和当前对象同时使用共享资源。

实验指导：根据上述资源竞争条件，对给出的 C、C++和 Java 程序中。

1）判断是否存在资源共享；

2）判断竞争的资源的共享方的主体名称，输出名称；

3）判断对共享资源的修改操作。

```
void f(pthread_mutex_t *mutex){
        pthread_mutex_lock(mutex);
        /* access shared resource */
        pthread_mutex_unlock(mutex);
}
```

图 11-1　竞争条件示例

图 11-1 是 CWE 上提供的一个 C 语言中的竞争条件例子。

在图 11-1 中，所示代码是通过加锁和解锁来控制共享资源的操作，但代码中没有检查 pthread_mutex_lock()的返回值。于是，当由于某些原因导致 pthread_mutex_lock()发生错误时，此时共享资源没有加锁，就可能会出现竞争条件的情况。这只是一个简单的竞争条件的例子。在实际中，竞争条件由于和并发环境密切相关，往往会很复杂。

说明：本课设鼓励大家对 C,C++或 Java 语言进行识别判断，最好利用 C 语言编写，保证最基本的识别---有没有问题（及格）；其次解是不是问题（中等）；最好能解决是什么、多少的问题（良好、优秀）。