华中科技大学

课 程 设 计 报 告

（题目）基于源代码的软件同源性分析与漏洞检测系统

课　　程　软件安全课程设计

院　　系　网络空间安全学院

专业班级　信安 1801

学　　号　U201814834

姓　　名　杨傲

指导教师　韩兰胜 付才 刘铭

2020 年　10 月　17 日

软件安全课程设计-功能完成情况

题目：基于源代码的软件同源性分析与漏洞检测系统

班级　信安 1801　　学号 U201814834　　姓名 杨傲

序号	内容及分值	得分标准	本人完成情况：完成项，填 Y；其他优点可备注；	分值	得分（教师填写）
必选	共 50 分				
R1	提供系统界面	图形化界面 GUI：4+；控制台界面 console：3+；设计合理、美观	Y console 美观、合理、清晰、体验预愉悦	5	
R2	利用字符串匹配进行同源性检测	能比对，并输出相似度 10；准确度 3；能对比展示样本不同行内容 2；	Y 能输出相似度，准确并生成 token 替代后的文件来对比不同	15	
R3	利用控制流程图 CFG 进行源代码同源性检测	生成 cfg 并保存 5；对比得出相似度 5；准确度 5	Y 能生成 cfg 保存到链表并输出，可以对比得出相似度，准确	15	
R4	栈缓冲区检测	给出可疑代码行数与列数	Y 给出行数及源代码	5	
R5	格式化字符串漏洞检测	给出可疑代码行数与列数	Y 给出行数及源代码	5	
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。	Y 在附件中 样本数量满足 每个代码行数不少于 100 行并且每种漏洞至少一个	5	

2 选 1	**共 5 分**				
A1	跨语言同源性检测验证	每额外增加一种语言 2 分（4 个样例）。5 分封顶		5	
A2	支持分布式任务调度	选定文件夹同时比对 3+，结果准确、快速 2+。	Y 可以选择多文件同时多线程对比，准确十分快速	5	
6 选 4	**共 40 分** 初步定位可能发生的位置，每个 5 分 （共 20 分） 对他人样本能完全、正确确定漏洞位置，每个 5 分，（共 20 分）				
B1	堆缓冲区检测	给出可疑代码行数与列数	Y 给出行数及源代码	10	
B2	整数宽度溢出检测	给出可疑代码行数与列数	Y 给出行数及源代码	10	
B3	整数运算溢出检测	给出可疑代码行数与列数	Y 给出行数及源代码	10	
B4	整数符号溢出检测	给出可疑代码行数与列数	Y 给出行数及源代码	10	
B5	空指针引用	给出可疑代码行数与列数		10	
B6	竞争性条件	给出可疑代码行数与列数		10	
2 选 1	**共 5 分**				
C1	同源性检测样本库	50 个样本(每个 100 行以上)每 10 个为相似的一组，1 分，最高 5 分	Y 见附件 满足组数和要求	5	
C2	漏洞检测样本库	R4+R5+B1-6 选 4，每组 7 个样本，共 6 组，每组		5	

		0.5 分，共 3 分；混合类型样本 18 个，1-2 分。			
合计	**满分 100**				

课程总评成绩

平时成绩	完成功能情况	课程报告		总评成绩
（10%）	（50%）	格式规范（20%）	内容完整（20%）	100%

目 录

1　课程设计任务书

1.1　课程设计目的

本课程题目为基于源代码的软件同源性分析与漏洞检测系统，在学习了 C 语言、汇编语言、数据结构、软件安全等先导课程后，要求自行完成完整的软件系统

1.2　课程设计要求

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。

1.3　系统环境

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。

我选择的系统环境为：　Win10　VS2019　QT

1.4　实验过程记录

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。

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。

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。

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。

界面部分可以 VS/QT/Python 等工具实现

2 绪言

2.1 标准

本课设鼓励对 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）。

我尽力进行到能够解决是什么以及多少问题的程度。

图形化界面计划采用 QT，然而在制作过程中完成了界面设计，可是涉及到 qt 中的信号与槽函数并没有在短时间内加以应用，最初的设想是使用 C++编写的程序可以在 click 界面中的按钮之后直接进行调用 C++函数并输出，后来通过学习 qt 教程发现并非如此，需要进行 qt 中内容的大量改写。鉴于验收时间临近，遂选择了 dos 界面，添加了美观的界面样式，并在前导提示字符串中包含姓名以防他人抄袭。

3　系统方案设计

3.1　利用字符串匹配进行同源性检测

任务 R1：图形化界面

任务 R6 以及 C1：提供样本库（以上二者已在文件中提供或呈现出来）

任务 R2：字符串匹配

通过代码有效字符串对比匹配，分析样本之间的拷贝比率。

思路：将需要进行比较的两个文件分别进行预处理，之后将两个预处理后的文件分别存入两个新文件之中，其中第一个文件直接一样的一行一行输出，而第二个文件中取消行之间的换行而变成一个很长的字符串，之后对于第一个文件中的每一行进行 KMP 算法，记录查找到相同的字符串的总长度以及文件本身的长度，从而得出相似度。

有一些在编程中注意到的注意事项：

1、关键字暂时支持到 99 标准，若更新仅需在数组中添加几个关键字即可。

2、由于读取方式是一次读一行，故需要检测样本默认源代码文件一行一个语句。

3、检测样本默认为使用 VS 编写的，使用默认代码美化格式的，即字符间空格严格存在。

4、为求简化，默认样本无头文件。如需对头文件替换为空，其实已在代码中实现。

4、以上默认对于后续的检测样本均适用。

3.2　利用控制流程图 CFG 进行源代码同源性检测

任务 R3：CFG

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。

思路：将需要进行比较的两个文件分别进行预处理，之后将两个预处理后的文件分别存入两个新文件之中，两个文件均为直接一样的一行一行输出。利用得到的两个文件进行构造函数调用关系图，计算两个关系图相似程度。

3.3　支持分布式任务调度

任务 A2：实现多进程并发检测，此处仅认为 CFG 较需要，故在 CFG 上实现多线程，其

他功能实现起来极为类似，不再赘述。

思路：在 CFG 的基础上进行，之前的 CFG 是两个文件之间进行 CFG 检测，此处使用多个进程并发执行，将一个文件同时与多个文件进行 CFG 检测。这里采用了五个线程，实际上可以使用很多但五个时已经能够直观的看出使用了多线程，并且便于输入和理解。输出未进行整理而是呈现出较为凌乱的语句，以便于老师检测多线程的实现状况。

3.4 栈缓冲区检测

任务 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。

思路：

构造代码中可能与栈溢出发生关联的变量们构成的链表，之后构造代码中与栈溢出有关的函数们构成的链表，从第一行遍历代码，寻找敏感函数出现的地方，并通过对于参数等的分类讨论来判断是否会发生溢出。

3.5 堆缓冲区检测

任务 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。

思路：

构造代码中可能与堆溢出发生关联的变量们构成的链表，之后构造代码中与堆溢出有关的函数们构成的链表，之后构造代码中与堆溢出有关的缓冲区设置以及边界审查们构成的链表，从第一行遍历代码，寻找敏感函数出现的地方，并通过对于参数等的分类讨论来判断是否会发生溢出。

3.6 整数宽度溢出检测

任务 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。

思路：

构造代码中可能与整数宽度溢出发生关联的变量们构成的链表，之后构造代码中与整数宽度溢出有关的函数们构成的链表，之后构造代码中与整数宽度溢出有关的缓冲区设置以及边界审查们构成的链表，从第一行遍历代码，寻找敏感函数出现的地方，并通过对于参数等

的分类讨论（例如发生赋值处两变量容纳大小宽度）来判断是否会发生溢出。

3.7 整数运算溢出检测

任务 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。

思路：

构造代码中可能与整数运算溢出发生关联的变量们构成的链表，之后构造代码中与整数运算溢出有关的函数们构成的链表，之后构造代码中与整数运算溢出有关的缓冲区设置以及边界审查们构成的链表，从第一行遍历代码，寻找敏感函数出现的地方，并通过对于参数等的分类讨论（例如发生运算处两变量容纳大小宽度）来判断是否会发生溢出。

3.8 整数符号溢出检测

任务 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。

思路：

构造代码中可能与整数符号溢出发生关联的变量们构成的链表，之后构造代码中与整数符号溢出有关的函数们构成的链表，之后构造代码中与整数符号溢出有关的缓冲区设置以及边界审查们构成的链表，从第一行遍历代码，寻找敏感函数出现的地方，并通过对于参数等的分类讨论来判断是否会发生溢出。

3.9 格式化字符串漏洞检测

任务 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。

思路：

构造代码中可能与格式化字符串漏洞发生关联的变量们构成的链表，之后构造代码中与格式化字符串漏洞有关的函数们构成的链表，之后构造代码中与格式化字符串漏洞有关的缓冲区设置以及边界审查们构成的链表，从第一行遍历代码，寻找敏感函数出现的地方，并通过对于参数等的分类讨论来判断是否会发生溢出。

4　系统实现

4.1　文件预处理

此处分为字符串匹配中的以及后面其他程序中的预处理两类：

（1）字符串匹配中的预处理：

对源代码文本进行一定层次的预处理,即采用一些记号来标记原来的源代码文本，即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。

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。

对源代码进行词法分析。该过程是按字符读入源代码信息,然后返回单词 Token，包括：标识符转化为_ID，如：name →_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。

代码中关键点在于：

采用逐行的读取文件方式，然后对读取出来的内容进行是否为行前回车或列表符、一个或多个空格、单行注释（即//）、多行注释（即/*，此时需要直到找到*/之前中间的所有代码全部置为空）、分号、左右大括号、左右括号、左右方括号、加减乘除号、等号、关键字以及标识符等进行判断，然后根据判断结果进行处理。

每次将一行存入一个数组中，便于进行处理，处理后再输出token到输出的文件中。对于实现行前多空格预处理，若有空格则仅留下一个；对于单行注释进行处理，注释均是单独的或行或块，直接置空；对于多行注释进行处理，需要直到*/的全置空；一行之中分号结尾，默认分号后没有语句，且没有空行无字符语句；有左大括号进行了替换，默认左右大括号单独成行；有右大括号进行了替换，默认左右大括号单独成行；有左括号进行了替换，从下一个字符继续进行预处理,；有右括号进行了替换，从下一个字符继续进行预处理，存在以右括号结尾的语句。

注意考虑到前一个和下一个字符间的空格，由于样本代码均为使用 VS 编写，在敲下分号后会进行自动的代码美化，导致符号间空格严格添加，故在进行字符间移动时切记这一点。

这里难以实现的其实仅仅只有一个多行注释的置空，着重解释一下：

```
if (instring[i] == '/' && instring[i + 1] == '*') //对于多行注释进行处理，需要直到*/的全消灭
{
    while (instring[i] != '*')
    {
        if (instring[i] == '\n')
        {
            fileout << outstring << endl;        //把字符串outstring保存到fileout;
            getline(filein, instring);           //默认/**/此种注释不会出现在末行
            i = 0;
            continue;
        }
        i++;
    }
    if (instring[i + 1] == '/')
    {
        i++; i++;   //将*/两个符号跳过
        continue;
    }
}
```

即当此行中读到/并且紧跟着下一个字符就是*，那么判定当前开始为多行注释，不断向后寻找再下一个字符*，若遇到回车就重新读入一行，直到读取到*并且其后一个字符为/，便可以判定多行注释结束。

（2）其他程序中的预处理

基本同上，但很大的区别是不再替换为 token，而只是去掉前面的头文件、单行注释以及多行注释、空格，剩下的原样输出到另一个文件中，便认为完成了预处理。此处仅需进行简单的预处理，因为重点已经不在预处理上而是其他的函数上。

4.2 字符串同源性匹配

进行基于单词（token）的比对。参考思路：首先对源代码文本进行一定层次的预处理,即采用自己定义的一些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 KMP 算法进行比对,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、关键字转化为 token,将源代码的比对技术转化成 token 序列的比对。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存储对比结果，最后计算相似度并进行评价。

将需要进行比较的两个文件分别进行预处理，之后将两个预处理后的文件分别存入两个新文件之中，其中第一个文件直接一样的一行一行输出，而第二个文件中取消行之间的换行而变成一个很长的字符串，之后对于第一个文件中的每一行进行 KMP 算法，记录查找到相同的字符串的总长度以及文件本身的长度，从而得出相似度。

```
while (getline(fileintemp, temp))          //从输入文件中读取文件内容存入临时字符串中
{
    if (KMP(instring, temp) != (-1))    //找到了相匹配的字符串
    {
        sameThing += temp.length(); //相同长度加上当前语句长度
    }
}
ratio = sameThing / total;                    //相同长度除以总长度即为比率
```

4.3 判断是函数体以及生成函数链表

由于需要将样本中的函数体信息连接存储到一个链表中，故需要先判断然后获取信息，然后再向链表节点进行存储。之后便可以通过这个链表进行函数体信息的的直接获取以及用来进行比对和添加子函数节点等操作。

需要判断是否找到函数体而不是函数声明，如下代码：

```
bool flag = 0;
if (strstr(s, "(") != NULL)
{
    if (strstr(s, "void") != NULL || strstr(s, "int") != NULL || strstr(s, "long") != NULL || strstr(s, "short") != NULL ||
strstr(s, "float") != NULL || strstr(s, "double") != NULL || strstr(s, "char") != NULL)
```

```
    {
        if (strstr(s, ";") == NULL)//当未找到分号时，即不为函数声明而是函数体
            flag = 1;
    }
}
return flag;
```

由于一个函数体首先会是返回值类型，比如 void、int、long、short、float、double、char

在使用 strstr（）这个十分好用的函数进行判断存在返回值类型后，再判断此行不存在分号，即不是函数声明，就可以断定当前来到了某个函数的函数体开头位置。如果去掉里面判断不含有分号的语句,则可以判断是否为函数声明或者函数体开头,而不是对于函数的使用，此处使用这个判断来找到入口以生成所有各不相同的函数链接形成的链表。这是由于函数声明和函数体开头其实一样，都能将该函数所有的重要信息提供，为了不重复记录相同的函数节点（比如为了避免同时将一个函数的声明和函数体记录为两个函数节点的情况），这里用：

```
if (strcmp(temp->Name,newNode->Name)==0)//用于判断是否这个函数已经写过了
{
    flag = 1;
    break;
}
```

这样可以在对所有的函数声明以及函数开头进行遍历后得出样本中所有不同的函数个数以及它们各自的重要信息记录在链表中。

之后是进行函数链表的生成：

首先判断时已经获得了返回值的类型，记录下来。返回值之后紧跟着的空格后的直到左括号前的部分均为函数的名字，记录下来。左括号后面的直到逗号前的为第一个参数，后面每两个逗号之间的也是参数，参数数量比逗号多一个，这样可以使用逗号的数量加一获得参数的数量，这里默认参数数量最多三个，并且此处注意由于逗号后面总是跟有空格，所以此处获得下一个字符需要移动两次。获得信息赋给函数节点并连接即成为记录函数的链表。

4.4 生成 CFG 图并进行 CFG 图的比较

需要建立函数内部子函数的调用链表以及各个函数连接起来的链表，以各个函数连接起来的链表来包含子函数链表的头指针，使用这些便可以实现获得某一个函数内部子函数调用情况。

此处有小技巧便是在同一个函数体内部，左右大括号的数量一定是相等的，以此作为判断同一个函数体内部的依据。

生成内部子函数的调用链表的方式为对于某一个函数体，一行行读入，在其中寻找是否存在已知的各种不同的函数其中之一，如果有存在则说明此处发生了子函数调用，记录在节点中并且重复多次，节点连接起来即可。

将各个函数信息以及内部子函数调用信息输出即形成简易的 CFG 图。

获得两个文件各自的各个函数连接起来的链表，以第二个链表为模板，将第一个中某个函数内各个子函数的类型与第二个链表中各函数类型进行比较，当遇到返回值、参数类型均相同的情况，则认为两个函数是同一个函数，相同函数数量加一。将第一个链表中所有函数的内部子函数均一个个的以第二个链表作为模板得出相同个数后再除以各个函数内部总子函数数量，即可获得各个函数的相似值，再将之取平均即为两个样本的相似值。

此处存在的一点讨论为：当有两个函数体但是内部都没有子函数，此时无法断定，默认此时两个函数体相同；当有两个函数体但是第一个内部没有子函数而第二个有，此时默认两个函数体不相同；没有和第一个函数体一样的第二个函数体，此时对于此第一个函数体对应的可能性为 0；当两个函数体一样而内部子函数不完全一样时计算相似度并进行记录。

4.5 多线程进行 CFG 图的比较

多线程这里我采用了五个线程并发，此处仅认为 CFG 较需要，故在 CFG 上实现多线程，其他功能实现起来极为类似，不再赘述。

在 CFG 的基础上进行，之前的 CFG 是两个文件之间进行 CFG 检测，此处使用多个进程并发执行，将一个文件同时与多个文件进行 CFG 检测。这里采用了五个线程，实际上可以使用很多但五个时已经能够直观的看出使用了多线程，并且便于输入和理解。输出未进行整理而是呈现出较为凌乱的语句，以便于老师检测多线程的实现状况。

通过使用 thread first(CFG_THR, infile1, infile2);这样的语句进行线程的创建，之后使用 first.join();来进行即可。

CFG 部分和前面基本相同。

4.6 栈溢出漏洞检测中的函数体链表形成

形成函数体链表，内部含有此函数开始和结束的行号，变量链表的头节点，敏感函数链表的头节点等。

一行行进行读取，若是函数体，则进入此函数体。

进行变量判断，若某行中不含有 printf 语句而含有 int、long、short、float、double、char 等，即不是输出语句而是变量定义语句时，判定需要进行变量节点的记录，具体记录方法为：能找到左方括号，说明含有数组类型，方括号中含有具体数值表示数组大小时，记录此数值（注意此时若在新建数组的同时进行了数组内容的初始化，则可以获得数组内存储的实际大小即字符串长度减去四，减去四是因为由于自动规范格式导致的空格出现）。方括号中不含有具体数值表示数组大小时，即方括号直接显示完整，此时进行了初始化数组操作，此时默认仅对字符串数组进行初始化，数组的大小和实际大小均为字符串长度减四。而当变量不是数组时，通过讨论变量的类型将其记录到变量节点的大小中，比如 int 型占有四个字节就将它的大小记作 4.以上便完成了变量节点信息的获取与记录。

进行敏感函数判断，若某行中含有 strcpy、strncpy、memcpy、memncpy、strcat、strncat、sscanf、vfscanf、fscanf、vscanf、vsscanf、sprintf、vsprintf、gets、getchar、fgetc、getc、read 等函数存在时，认为此行存在敏感函数，需要进行敏感函数链表节点的记录，具体记录方法为：读取敏感函数所在这一行，读取左括号前的字符串即为敏感函数的名称，如果是 strcpy 或者 strcat，则有两个参数，可以从逗号前后获得此函数的目的字符串以及源字符串，如果是 strncpy、memcpy、strncat 则还有一个参数 n 需要获取并记录，如果是 sscanf 则还有中间的一部分描述格式的语句需要跳过，不进行记录中间语句的操作。

以 strcpy 或者 strcat 为例：

```
sscanf(s, "%[^(]", mig->Name);
if (!strcmp(mig->Name, "strcpy") || !strcmp(mig->Name, "strcat"))
{
    s = strstr(s, "(");
    s++;
    sscanf(s, "%[^,]", mig->Mudi);
    s = strstr(s, ",");          //两个s++移动两位是因为VS中输入分号后自动规范格式使得逗号后面默认有一个空格
    s++;
    s++;
    sscanf(s, "%[^)]", mig->yuan);
```

}
仍然是通过极其好用的strstr函数来进行定位，以逗号为界即可很简单的获取到内部参数。

4.7 栈溢出漏洞检测中的漏洞的检测

如果某函数体内发现了会导致栈溢出漏洞的变量以及敏感函数，则进行分类讨论：
当敏感函数是strcpy或者sscanf时，如果源字符串实际长度大于目的字符串容积，
或者strncpy以及memcpy函数中目的字符串容积小于n，
或者strcat中目的字符串和源字符串实际大小之和小于目的字符串能容纳的大小，
或者strncat中n加上目的字符串实际大小大于目的字符串容积，
均记为存在漏洞，并输出行号以及样本中对应的代码。
同样地将各个函数体进行检测即可。

4.8 堆溢出漏洞检测

其中变量链表、函数体链表以及敏感函数链表的实现与上述栈溢出中的十分相似，不再赘述。
这里进行了对于缓冲区的信息记录以及边界审查条件的记录。
判断是函数体后进入此函数体。
进行缓冲区判断，若某行中不含有printf语句而含有int、long、short、float、double、char等，并且含有*符号和【符号（即存在指针以及数组），此时判定需要进行缓冲节点的记录，具体记录方法为：
若为数组则和之前变量记录方法相同。
若存在char*、short*、int*、long*等并且没有HeapAlloc时，通过strstr移动并且记录缓冲区。
若也存在HeapAlloc，则将此语句中的信息也通过strstr获取并记录在缓冲节点中。

进行边界审查判断，即是否存在if语句，以此作为是否超出边界的信息来源。同样获取信息并记录在边界节点中。

如果某函数体内发现了会导致堆溢出漏洞的变量以及敏感函数，则进行分类讨论：
当敏感函数是strcpy或者sscanf时，如果源字符串实际长度大于目的缓冲区容积，
或者strncpy以及memcpy函数中目的缓冲区容积小于n，
或者strcat中目的字符串和源字符串实际大小之和小于目的缓冲区能容纳的大小，
或者strncat中n加上目的字符串实际大小大于目的缓冲区容积，

均记为存在敏感函数中溢出的可能，将记号1标记。
之后对边界链表中各个节点和当前敏感函数中的变量名进行比较，如果存在相同，说明样本中有意的进行了边界审查，此时默认不存在堆溢出，否则仍可能存在堆溢出，标记记号2。
通过两个记号的分类讨论得出是否可能存在堆溢出。
同样地将各个函数体进行检测即可。

4.9 整数宽度溢出检测

对可能造成整数宽度溢出的不同情况进行分析，对每种可能造成整数宽度溢出的原因用一个记号标记，然后最后根据造成整数宽度溢出的情况对这几个记号进行分类讨论，输出原因、样本中代码以及发生溢出的位置。

其中大部分的链表生成均与前述堆溢出中类似，只是这里多了一个链表就是样本中出现了整数间赋值语句时均被记录形成的节点们连接成的链表。

```
while (XT != NULL) //赋值语句溢出
{
    xtflag = 1;

    strcpy(xtleft, XT->left);
    strcpy(xtright, XT->right);

    while (v != NULL)
    {
        if (!strcmp(xtleft, v->Name))
            lkind = v->kind;
        if (!strcmp(xtright, v->Name))
            rkind = v->kind;

        v = v->next;
    }
    if (lkind < rkind)//存在导致宽度溢出的情况
    {
        xtflag = 2;
```

```
                break;
        }
        if(xtflag!=2)
            XT = XT->next;
    }
```

即将记录赋值语句的节点内含有的左右变量的名字同已知的变量的名字对比，找到变量的名字后即可从变量链表中获得变量的大小信息，从而得知此赋值语句是否存在溢出。

存在导致宽度溢出的情况时将标号1标记，存在对变量进行边界检查则标记记号2，存在敏感函数目标数组和缓冲区相同则标记记号3。

通过此三者分类讨论得出是否整数宽度溢出：

赋值语句左边小于右边，敏感函数的目标数组与之相同且没有对其进行边界审查，溢出；

前两个成立但进行了边界审查，可能溢出；

无赋值语句或无敏感函数或有边界检查，不溢出；

4.10 整数运算溢出检测

对可能造成整数运算溢出的不同情况进行分析，对每种可能造成整数运算溢出的原因用一个记号标记，然后最后根据造成整数运算溢出的情况对这几个记号进行分类讨论，输出原因、样本中代码以及发生溢出的位置。

其中大部分的链表生成均与前述整数宽度溢出中类似，只是这里多了一个链表就是样本中出现了整数间计算语句时均被记录形成的节点们连接成的链表。

以下述代码判断是否存在整数运算：

```
bool PanCalB3(char s[MaxChar])//判断是否存在计算
{
    bool flag = 0;
    if (strstr(s, "+") != NULL)
    {
        flag = 1;
    }
    else
    {
        if (strstr(s, "*") != NULL)
```

```
        {
            if (strstr(s, "malloc") != NULL)
            {
                flag = 1;
            }
            else if (strstr(s, "=") != NULL)
            {
                flag = 1;
            }
            else
                flag = 0;
        }
    }
    return flag;
}
```

由于乘号和指针的符号是一样的所以要多加一层判定来确保当前语句中发生的是乘法而不是malloc等涉及指针的语句。存在整数间运算就返回1。

然后采用前面描述过的分析方法获取计算语句内部参数信息，包括几个参与运算的数的信息以及将结果赋值给的参数的信息。

```
void C_analysisB3(char s[MaxChar], struct Calcu* c)
{
    if (strstr(s, "+") != NULL)
    {
        sscanf(s, "%[^ =]", c->mudi);
        s = strstr(s, "=");
        s++;
        s++;
        sscanf(s, "%[^ +]", c->yuan1);
        s = strstr(s, "+");
        s++;
        s++;
        sscanf(s, "%[^;]", c->yuan2);
```

```
    }
    else if (strstr(s, "*") != NULL && strstr(s, "malloc") == NULL && strstr(s, "short*") == NULL && strstr(s,
"int*") == NULL && strstr(s, "long*") == NULL)
    {
        sscanf(s, "%[^ =]", c->mudi);
        s = strstr(s, "=");
        s++;
        s++;
        sscanf(s, "%[^ *]", c->yuan1);
        s = strstr(s, "*");
        s++;
        s++;
        sscanf(s, "%[^;]", c->yuan2);
    }
    else if(strstr(s, "*") != NULL && strstr(s, "malloc") != NULL)
    {
        s = strstr(s, "malloc");//跳过前面转换指针类型部分
        s = strstr(s, "(");
        s++;
        if (strstr(s, "*") != NULL)//仍存在乘号，即后面空间大小中存在乘号
        {
            sscanf(s, "%[^ *]", c->mudi);
            strcpy(c->yuan1, c->mudi);
            s = strstr(s, "*");
            s++;
            s++;
            sscanf(s, "%[^;]", c->yuan2);
        }
    }
}
```

此中不仅考虑了直接的整数间运算，还考虑了当存在malloc后仍然存在乘号（即后面分配空

间大小中存在乘号即存在整数间运算）的情况。

将记录运算语句的节点内含有的各个变量的名字同已知的变量的名字对比，找到变量的名字后即可从变量链表中获得变量的大小信息，从而得知此计算语句是否存在溢出。

存在导致计算溢出的情况时将标号1标记，敏感函数目标数组和缓冲区相同则标记记号2，通过此二者分类讨论得出是否整数运算溢出：

目标变量不为long型时并且敏感函数的目标数组与之相同则发生溢出。

无运算语句或无敏感函数或目标变量为long或敏感函数目标数组和缓冲区不相同，不溢出。

4.11 整数符号溢出检测

各个链表生成前面已经描述过。

当遇到memcpy等中第三个参数出现n出现负数，则存在符号溢出，输出溢出发生行数。

此时用到的对memcpy分析的语句如下：

```
void M_analysisB4(char s[MaxChar], struct MiGan* mig)
{
    char temp[20] = { 0 };
    if ( strstr(s, "strncpy") != NULL || strstr(s, "memcpy") != NULL || strstr(s, "memncpy") != NULL || strstr(s,
"strncat") != NULL)
    {
        sscanf(s, "%[^(]", mig->Name);
        s = strstr(s, "(");
        s++;
        sscanf(s, "%[^,]", mig->Mudi);
        s = strstr(s, ",");
        s++;
        s++;
        sscanf(s, "%[^,]", mig->yuan);
        s = strstr(s, ",");
        s++;
        s++;
        sscanf(s, "%[^)]", temp);
        mig->n = atoi(temp);
```

```
    }
}
```

可以把函数内三个参数依次存入节点中保存，第三个参数为n，当n为-1时发生整数符号溢出

4.12 格式化字符串漏洞检测

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。

首先对给出的C或C++的程序，能够判断出字符串的申请，使用；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；再判断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。最后输出相应的行号和结果。

此处对于printf内部参数的分析如下：

```
if (strstr(s, "printf") != NULL)
{
    sscanf(s, "%[^(]", mig->Name);
    while(strstr(s, "%") != NULL)
    {
        num++;
        s = strstr(s, "%");
        s++;
        sscanf(s, "%1c", &temp[num]);//从1到num记录了各个格式符号
    }

    s = strstr(s, "\"");
    s = strstr(s, ",");
```

```
s++;
while(strstr(s, ",") != NULL)
{
    realnum++;
    s = strstr(s, ",");
    s++;
}

if (num > realnum&& temp[num] == 'x')
{
    return 1;
}
else if (num > realnum&& temp[num] != 'x')
{
    return 2;
}
else
    return 3;
```

分为三种情况，第一种是输出参数数量大于参数个数并且最后一个格式化输出为%x，则存在格式化字符串漏洞，第二种是输出参数数量大于参数个数但最后一个格式化输出不为%x，则可能存在漏洞，第三种是输出参数数量不大于参数个数，此时不存在漏洞。

5　系统测试

5.1　字符串同源性检测

主界面：

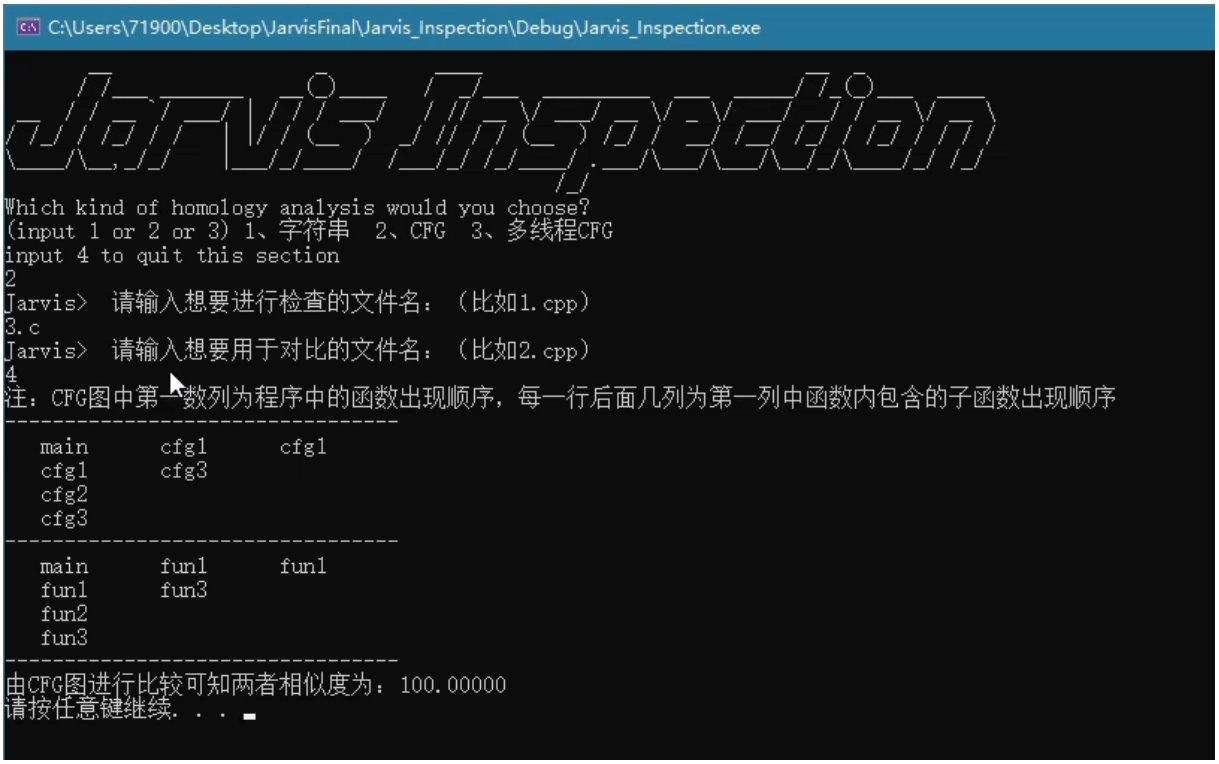
同源性检测界面：

字符串同源性检测：

此时是进行了对两个仅有一句代码不同的文件进行同源性匹配，可以看到检测出相似度极高，判定是抄袭。

两个源文件转化为 token 后可以在当前文件目录下找到对应的文件 temp1.cpp 和 temp2.cpp。

5.2 CFG 检测



这里进行了注释提醒，此处的 cfg 图中第一竖列为文件中函数体使用的顺序，之后各竖列为第一列函数体内部调用的子函数的顺序。这里用来对比的两个文件实际上是一样原理的文件但是仅仅把其中的函数名、变量名全部彻底替换掉了，仍然可以检测出是抄袭并且完全一样。

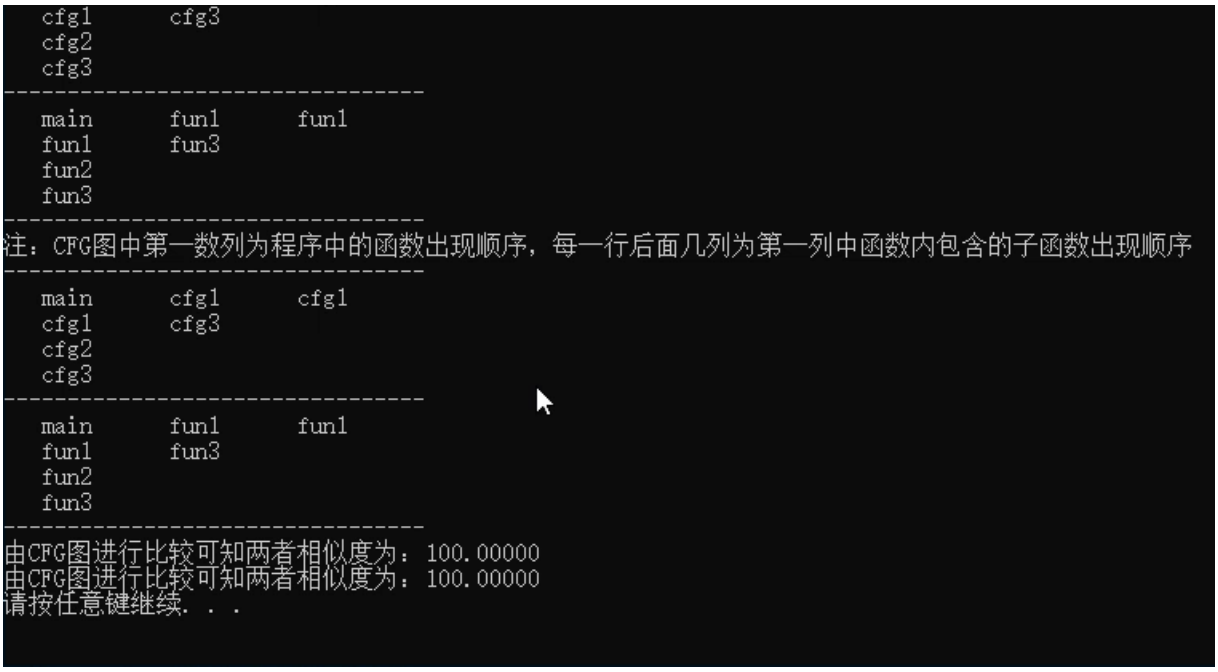
下面两图中便是这两个程序，可以看出除了变换名字其实内部调用关系完全相同，可以检测出相似度为 100%。

```c
int cfg1(int x, int y);
short cfg2(int x);
long cfg3( );


int main( )
{
    int n_1 = 1, n_2 = 2, n_3 = 3;
    short t_1;

    n_1 = cfg1(n_2, n_3);
    t_1 = cfg1(&n_1);

    return 0;
}

int cfg1(int x, int y)
{
    int z = x + y;
    cfg3( );
    return z;
}

short cfg2(int x)
{
    x = 4;
    return 4;
}

long cfg3( )
{
    long m;
    return tt;
}
```

```c
int fun1(int a, int b);
short fun2(int a);
long fun3( );


int main( )
{
    int m_1=1, m_2=2, m_3=3;
    short s_1;

    m_1 = fun1(m_2, m_3);
    s_1 = fun1(&m_1);

    return 0;
}

int fun1(int a, int b)
{
    int c = a+b;
    fun3( );
    return c;
}

short fun2(int a)
{
    a = 4;
    return 4;
}

long fun3( )
{
    long s;
    return s;
}
```

5.3 CFG 多线程检测

```
C:\Users\71900\Desktop\JarvisFinal\Jarvis_Inspection\Debug\Jarvis_Inspection.exe

Which kind of homology analysis would you choose?
(input 1 or 2 or 3) 1、字符串  2、CFG  3、多线程CFG
input 4 to quit this section
3
Jarvis>  输入想要将哪个文件作为比较模板
3.c
Jarvis>  输入四个将与之进行比较的文件
4 4 4 4
注：CFG图中第一数列为程序中的函数出现顺序，每一行后面几列为第一列中函数内包含的子函数出现顺序
--------------------------------
   main      cfg1      cfg1
   cfg1      cfg3
   cfg2
   cfg3
--------------------------------
   main      fun1      fun1
   fun1      fun3
   fun2
   fun3
--------------------------------
由CFG图进行比较可知两者相似度为：100.00000
注：CFG图中第一数列为程序中的函数出现顺序，每一行后面几列为第一列中函数内包含的子函数出现顺序
--------------------------------
```

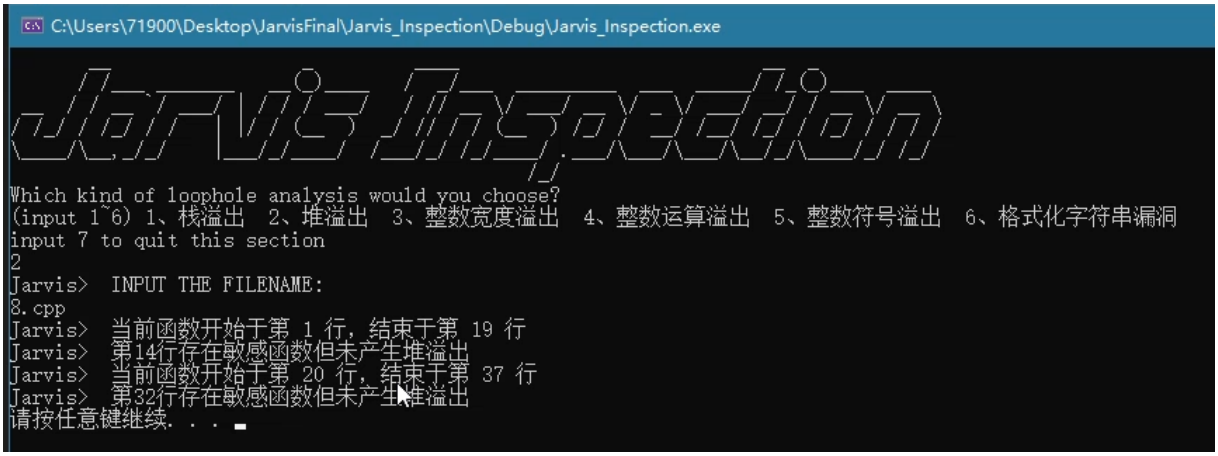


把 3.c 和四个 4 文件进行同时多线程 cfg 检测，可以发现输出有些凌乱，没有按照顺序进行，说明实现了多线程进行 CFG。

5.4 栈溢出漏洞检测

这里会进行代码审计并将发生栈溢出的情况进行输出，若有栈溢出发生会输出栈溢出所在行数，由于一行默认仅仅有一个语句，所以列的输出均为此行的开始，故列的表示没有意义，而选择进行将样本中对应发生溢出漏洞的原代码进行输出，方便使用者在样本中找到溢出漏洞对其进行更正。

5.5 堆溢出漏洞检测



进行检测的文件如下：

```
int main(int argc, char* argv[])
{
    char mybuf1[450];
    HANDLE hHeap;
    char* buf1;
    char* buf2;
    int i;

    for (int i = 0; i < 450; i++) {
        mybuf1[i] = 'a';
    }
    LoadLibrary("user32");
    hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x10000, 0xfffff);
    buf1 = (char*)HeapAlloc(hHeap, 0, 200);
    strcpy(mybuf1, buf1);
    buf2 = (char*)HeapAlloc(hHeap, 0, 200);
    HeapFree(hHeap, 0, buf1);
    HeapFree(hHeap, 0, buf2);
    return 0;
}

int main(int argc, char* argv[])
{
    char mybuf2[450];
    HANDLE hHeap;
    char* buf1;
    char* buf2;
    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;
}
```

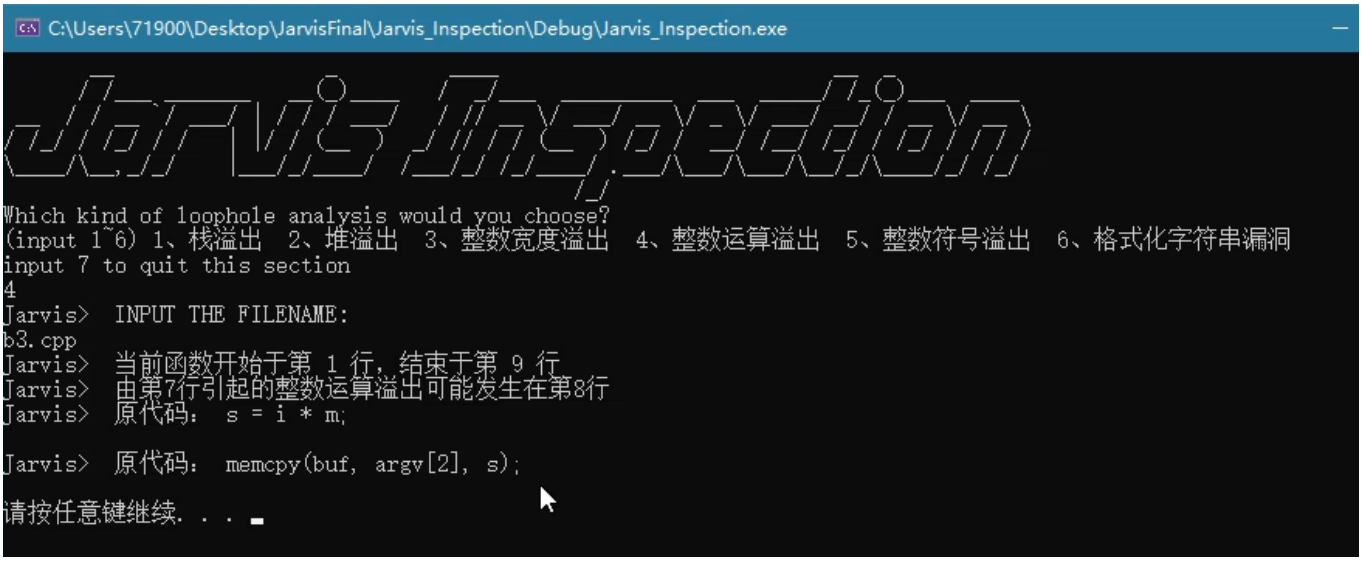
这里会进行代码审计并将发生堆溢出的情况进行输出，若有堆溢出发生会输出堆溢出所在行数，由于一行默认仅仅有一个语句，所以列的输出均为此行的开始，故列的表示没有意义，而选择进行将样本中对应发生溢出漏洞的原代码进行输出，方便使用者在样本中找到溢出漏洞对其进行更正。

这里对应的样例中虽然存在有敏感函数以及缓冲区说明，但实际经过判断并未溢出所以会输出某某行存在敏感函数但并没有溢出。

5.6 整数宽度溢出漏洞检测

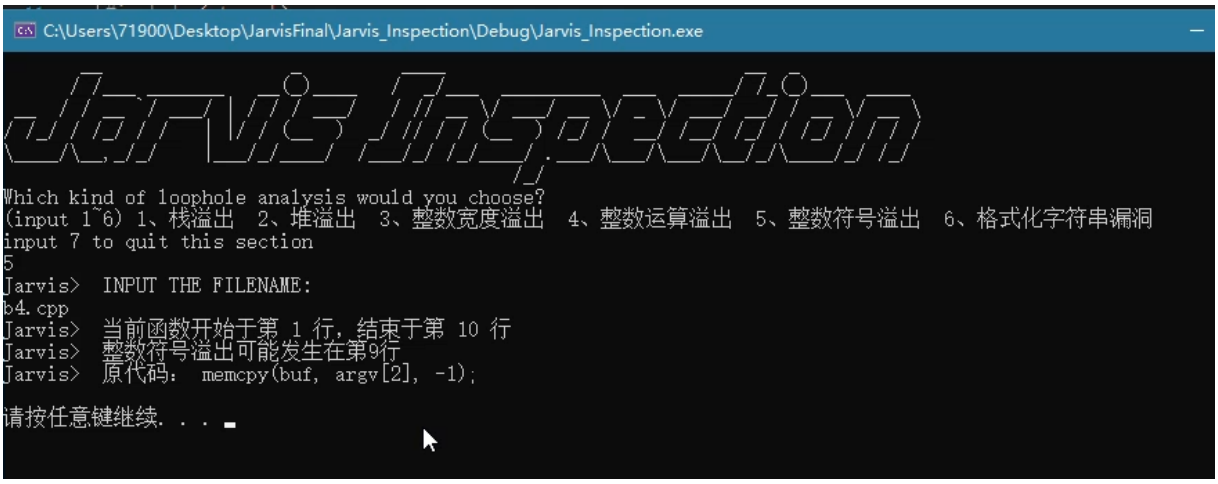
这里会进行代码审计并将发生整数宽度溢出的情况进行输出，若有整数宽度溢出发生会输出溢出所在行数，由于一行默认仅仅有一个语句，所以列的输出均为此行的开始，故列的表示没有意义，而选择进行将样本中对应发生溢出漏洞的原代码进行输出，方便使用者在样本中找到溢出漏洞对其进行更正。

5.7 整数运算溢出漏洞检测



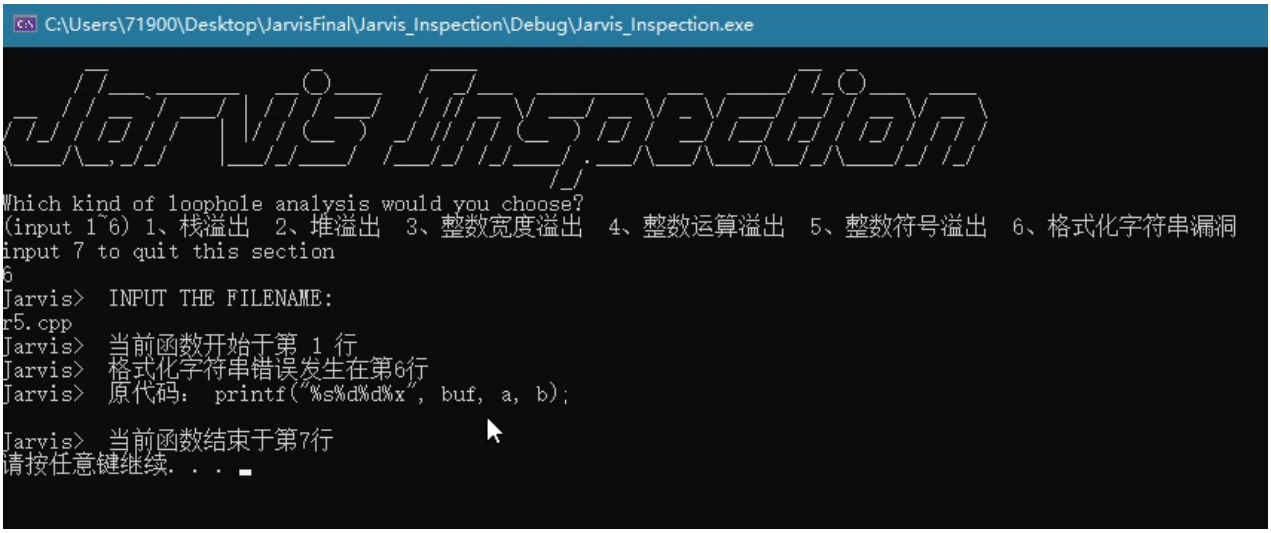
这里会进行代码审计并将发生整数运算溢出的情况进行输出，若有整数运算溢出发生会输出溢出所在行数，由于一行默认仅仅有一个语句，所以列的输出均为此行的开始，故列的表示没有意义，而选择进行将样本中对应发生溢出漏洞的原代码进行输出，方便使用者在样本中找到溢出漏洞对其进行更正。

5.8 整数符号溢出漏洞检测



这里会进行代码审计并将发生整数符号溢出的情况进行输出，若有整数符号溢出发生会输出溢出所在行数，由于一行默认仅仅有一个语句，所以列的输出均为此行的开始，故列的表示没有意义，而选择进行将样本中对应发生溢出漏洞的原代码进行输出，方便使用者在样本中找到溢出漏洞对其进行更正。

5.9 格式化字符串漏洞检测



这里会进行代码审计并将发生格式化字符串漏洞的情况进行输出，若有格式化字符串漏洞发生会输出格式化字符串漏洞所在行数，由于一行默认仅仅有一个语句，所以列的输出均为此行的开始，故列的表示没有意义，而选择进行将样本中对应发生溢出漏洞的原代码进行输出，方便使用者在样本中找到溢出漏洞对其进行更正。

6　总结与展望

6.1　总结与展望

本次实验自己的心路历程经历了多个阶段。

起初是对于 qt 初识的新鲜以及迷惑，界面和按钮这些都能明白，可是如何将编写好的字符串同源性检测程序和按钮联系起来，对我来说仍是难以理解的。于是决定先完成功能的 cpp 文件，满足老师的任务书上的要求。

之后是漫长的实现功能的历程，中间有着多次的将一个功能推倒重做的经历，原因便是在于有时一个功能按照自己的思路闭门造车会发现最后要么难以实现，要么大段的代码其实还不如一句话来的省事，这让我认识到了有时提前规划好代码框架再开始进行书写其实是一种事半功倍的方式，当能够将任务书中的需求完全理解并将自己将要完成的代码中的难点找到确定的解决方式（比如选择哪一个函数来实现某一个小功能），之后进行编写便很顺畅。

然而对于这样实际的课设要求以及有着明确的时间限制，还是要着手先编写着代码，一点点从基础的功能做起，不断加深需求，可以在一定时间内完成课设的要求。

在编写完成各个功能后，便是样本库的扩充以及图形化界面的实现。之前在 qt 上花费了两周左右的时间进行理解，但直到功能实现依然是三四周过去了，此时我才发现 qt 需要进行大量的工作将 cpp 文件进行改写，然而提交期限临近，遂放弃进行了许久的 qt 项目，转而实现尽量美观整洁的 console 界面，成功在提交检查前完成。

体会到完成第一个预处理以及第一个漏洞检测花费的精力是最多的，如何分析变量、函数、敏感函数、判断溢出等都需要全面的考虑。在经历困难完成了第一个程序后，剩下的程序的实现思路是十分相似的沿袭下来的。

体会到一定要和周围同学、老师多加交流，他们的想法、思路常常会是充满闪光点的，相比于自己闭门造车，这样一定是收获更多、学习更快、体会更深的一种学习方法。也要感谢周围同学们的悉心讲解以及韩兰胜老师、付才老师、刘铭老师的及时点拨。

7　对该课程的建议与意见(欢迎提交)

7.1　建议与意见

软件安全课程设计检查表给出的时间过晚，当从微助教上下载到这个文件后发现其中有些细节之处存在有自己对题目的理解和老师的要求存在偏差的地方，但是任务书上并没有对这些进行像表格给出的一样的细致表述，所以当学生按照任务书中的表述完成实验后，直到很晚临近提交才看到这个检查表，然而一点细节的地方和自己理解是有偏差的却又没有时间去修改源码了。

例如多线程这一点任务书中没有细致的描述，然而检查表格中却给出是需要对某个文件夹中的文件进行，然而我获得这个表格并按照要求的格式把表格添加进报告第二三页的时间已经是临近提交的时间了，没有余力再去修改。

另外，实验开始时提供的报告格式和最后微助教上要求的格式不一样，导致按照课程最初提供格式撰写的报告最后却又需要为了这样形式化的目的而重新排版。

希望老师们以后可以在课程开始提供检查表或者类似的细致要求，而不是仅仅给出模棱两可的任务书，并且尽量保证前后要求的报告格式统一，避免浪费时间在修改格式上。

8　参考文献

1、软件安全课程设计实验指导书 2020（整合）v1.6+　韩兰胜、付才、刘铭

2、https://blog.csdn.net/dark_cy/article/details/88698736　KMP 算法讲解

3、https://www.runoob.com/cprogramming/c-function-strstr.html　strstr()函数讲解

4、https://blog.csdn.net/larger5/article/details/78587076?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param

qt 应用讲解

5、https://blog.csdn.net/shijing_0214/article/details/53100992　字符串相似性度量方法

6、https://www.jianshu.com/p/d6fc0bb0689a　qt 信号与槽讲解

7、https://blog.csdn.net/joey_ro/article/details/104244648?utm_medium=distribute.pc_relevant.none-task-blog-blogcommendfrommachinelearnpai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-blogcommendfrommachinelearnpai2-1.channel_param

qt 引入图片背景

8、https://blog.csdn.net/cao269631539/article/details/62223388　qt 获取文件夹路径

9、https://blog.csdn.net/xiaowei_cqu/article/details/7760927/　使用 Lex 做词法分析

10、https://www.cnblogs.com/wangguchangqing/p/6134635.html　C++11 中的多线程

11、https://blog.csdn.net/lizun7852/article/details/88753218　C++多线程