

X 目 录

目 录.....	I
1 课程设计任务书	1
1.1 课程设计目的	1
1.2 课程设计要求	1
1.3 系统环境	3
1.4 实验过程记录	3
2 绪言	5
2.1 静态分析代码的相似度	5
2.2 堆/栈溢出	5
2.3 整数运算溢出	6
2.4 格式化字符串溢出	6
2.5 空指针引用	6
3 系统方案设计	8
3.1 总方案设计	8
3.2 文件组织设计	8
3.3 通用界面设计	9
4 系统实现	11
4.1 系统总框架实现	11
4.2 跳转结构实现	11
4.3 基于 Tkinter 库的界面实现	12
4.4 基于 token 的同源性匹配	13
4.5 基于 CFG 的静态调用关系分析	16
4.6 应用 Splint 的解决方案	17
4.7 整数符号漏洞的实现	19
4.8 多线程实现并发检测	20
5 系统测试	23
5.1 Launch	23
5.2 界面测试	23
5.3 Token 同源性检测测试	24

5.4	CFG 同源性测试	26
5.5	栈溢出测试	28
5.6	格式化字符串漏洞检测测试	30
5.7	堆溢出检测测试	30
5.8	整数宽度溢出检测测试	31
5.9	空指针解引用漏洞检测测试	32
5.10	整数符号溢出-单线程部分	32
5.11	整数符号溢出-多线程部分	33
6	总结与展望	36
6.1	Q&A	36
6.2	总结	36
7	参考文献	38

1 课程设计任务书

1.1 课程设计目的

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.

1.2 课程设计要求

序号	任务	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.

1.3 系统环境

系统:Ubuntu 18.04 LTS

虚拟环境: VMware® Workstation 15 Pro V15.1.0 build-13591040

运行环境: Python 3.6.8

开发工具: Visual Studio Code Insiders For Linux

版本: 1.39.0-insider

Electron: 4.2.10

Chrome: 69.0.3497.128

Node.js: 10.11.0

V8: 6.9.427.31-electron.0

OS: Linux x64 5.0.0-27-generic

第三方依赖:

- Tkinter
- Lex& Flex
- cflow
- splint

1.4 实验过程记录

1. 实验需求明确
2. 整体架构设计
3. 分项设计
 - a) 理论调研
 - b) 工具确定
 - c) 方案设计
 - d) 接口文档编写
 - e) 开发
 - f) 单元测试
4. 联合测试
5. 文件架构设计

6. 文件接入
7. 总测试
8. 总结
 - a) Readme 编写
 - b) 测试文件清理
 - c) 报告撰写

2 绪言

2.1 静态分析代码的相似度

静态分析代码的相似度,目前通用的技术是基于 token 的匹配技术和基于函数调用的 CFG 匹配技术.

2.1.1 基于 token 解析的匹配技术

该技术主要是使用词法分析工具对样本代码和模板代码生成 token 序列比对.

该技术使用 flex 进行规则编写和部署,缺点是耗时较长并对顺序调换和多语言兼容性较差.

2.1.2 基于 CFG 的匹配检测

该技术使用函数调用的检测匹配.主要步骤有:获取函数声明列表,生成函数调用图.整体匹配等.

该技术有效兼容顺序调换的同源样本,但主要关注于函数调用,需要和前述技术复合使用.

本实验使用 CFLOW 对 C-like 文件进行针对性分析.

2.2 堆/栈溢出

缓冲区溢出 (buffer overflow),是针对程序设计缺陷,向程序输入缓冲区写入使之溢出的内容 (通常是超过缓冲区能保存的最大数据量的数据),从而破坏程序运行、趁著中断之际并获取程序乃至系统的控制权.

缓冲区溢出原指当某个数据超过了处理程序限制的范围时,程序出现的异常操作.造成此现象的原因有:

- 存在缺陷的程序设计.
- 尤其是 C 语言,不像其他一些高级语言会自动进行数组或者指针的边界检查,增加溢出风险.
- C 语言中的 C 标准库还具有一些非常危险的操作函数,使用不当也为溢出创造条件.

因黑客在 Unix 的内核发现通过缓冲区溢出可以获得系统的最高等级权限,而成为攻击手段之一.也有人发现相同的问题也会出现在 Windows 操作系统上,以致其成为黑客最为常用的攻击手段.蠕虫病毒利用操作系统高危漏洞进行的破坏与大规模传播均是利用此技术.比较知名的蠕虫病毒冲击波蠕虫,就基于 Windows 操作系统的缓冲区溢出漏洞.¹

¹ Wikipedia, <https://zh.wikipedia.org/wiki/%E7%BC%93%E5%86%B2%E5%8C%BA%E6%BA%A2%E5%87%BA>

本实验使用业内知名的开源静态分析工具 `splint` 分析潜在漏洞。

2.3 整数运算溢出

该漏洞类型包括整数宽度溢出,整数运算溢出,整数符号溢出检测.主要利用的是整数的字长限制制造错误值来绕过一些检查.

C 语言中存在两类整数算术运算,有符号运算和无符号运算.在无符号运算里,没有了符号位,所以是没有溢出的概念的.

所有的无符号运算都是以 2 的 n 次方为模.如果算术运算符的一个操作数是有符号数,另一个是无符号数,那么有符号数

会被转换为无符号数(表示范围小的总是被转换为表示范围大的),那么溢出也不会发生.但是,当两个操作数都是有符号数

时,溢出就有可能发生.而且溢出的结果是未定义的.当一个运算的结果发生溢出时,任何假设都是不安全的²

2.4 格式化字符串溢出

`printf` 是 c 语言中为数不多的支持可变参数的库函数.根据 `cdecl` 的函数调用规定,函数从最右边的参数开始,逐个压栈.如果要传入的是一个字符串,那么就将字符串的指针压栈.这一切都井井有条的进行着.如果是一般的函数,函数的调用者和被调用者都应该知道函数的参数个数以及每个参数的类型.但是对于像 `printf` 这种可变参数的函数来说,一切就变得模糊了起来.函数的调用者可以自由的指定函数参数的数量和类型,被调用者无法知道在函数调用之前到底有多少参数被压入栈帧当中.所以 `printf` 函数要求传入一个 `format` 参数用以指定到底有多少,怎么样的参数被传入其中.然后它就会忠实的按照函数调用者传入的格式一个一个的打印出数据.任意内存写入需要用到 `%n` 这个不常用的参数,它的功能是将 `%n` 之前 `printf` 已经打印的字符个数赋值给传入的指针,通过 `%n` 我们就可以修改内存中的值了.³

2.5 空指针引用

空指针引用故障 (Null Pointer Dereference),也叫空指针解引用,是程序设计语言中一类常见的

² CSDN <https://blog.csdn.net/CoderBruis/article/details/80508666>

³<https://kevien.github.io/2018/04/07/%E6%A0%BC%E5%BC%8F%E5%8C%96%E5%AD%97%E7%AC%A6%E4%B8%B2%E6%BC%8F%E6%B4%9E/>

动态内存错误.指针变量可以指向堆地址、静态变量和空地址单元,当引用指向空地址单元的指针变量时,就会产生空指针引用故障,有可能产生不可预见的错误,导致软件系统崩溃.⁴

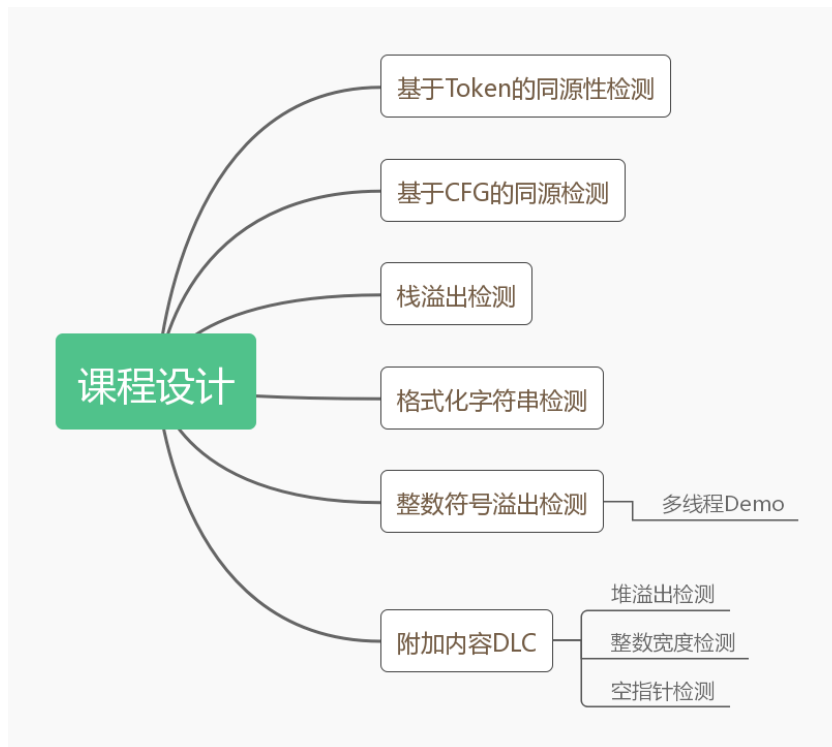
应对这类漏洞,应该注意对于每个指针的校验.许多函数也有错误检测机制.但在原生 C 上使用不多.

⁴<https://baike.baidu.com/item/%E7%A9%BA%E6%8C%87%E9%92%88%E5%BC%95%E7%94%A8%E6%95%85%E9%9A%9C/19452220?noadapt=1>

3 系统方案设计

3.1 总方案设计

系统的总设计如图所示:



每个模块使用一个界面,漏洞检测和同源性检测分别使用通用界面模板.

3.2 文件组织设计

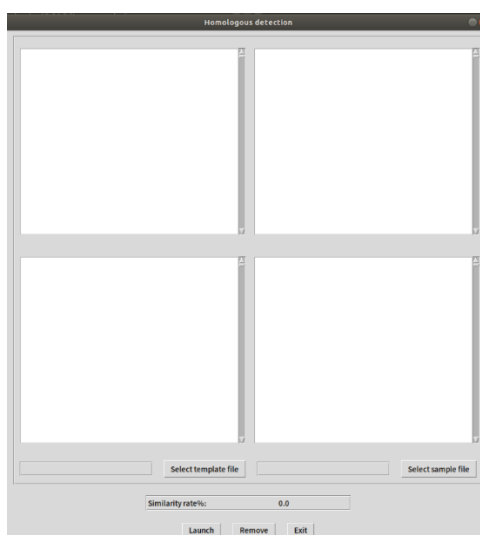
工程总文件存放在一个根文件夹下,下列几个子文件夹,,具体文件架构如下图所示.



每次触发[Launch]视作建立一次任务,每个任务以触发时间(精确到秒)为识别 id,并以任务为单元存放临时文件(输入,输出,中继等).

3.3 通用界面设计

同源性检测界面如下所示:



左边是模板文件,右边是样本文件(待检测).上方的文本框显示过滤注释后的源代码,下面是处理后的 token/CFG 序列.

文件路径显示在下面的 label 中.

按钮的功能:

- **Launch:**启动生成 TOKEN/CFG 并比对
- **Remove:**清除选中的文件和 token
- **Exit:**退出

漏洞检测通用界面如下图所示.



上方的文本框同步显示待测试代码

下方文本框显示漏洞信息.按钮功能同上.

添加了绑定的滚动条,在代码过长的时候可以下拉查看.

整数符号检测/多线程模块添加了一个选择文件夹按钮,可以选择对文件夹内所有.c 文件进行多线程检测.

4 系统实现

4.1 系统总框架实现

调用 Tkinter 库进行界面绘制和拼装.使用 Frame 容器封装,每个 Frame 包括一个文本框加一个绑定的滚动条或一组按钮.

窗口间的跳转通过按钮触发.按钮使用 `command` 绑定触发方法,由触发方法配置参数并唤起新窗口以实现跳转.

代码以类为单元封装各个功能,以实现继承和模块化开发,测试和调用.

下面是项目的类总计和说明,下文不再赘述:

<i>CLASS NAME</i>	<i>Comments</i>
Window(object)	主窗口的封装类.包含触发和跳转中间的代理方法.
Homologous_detection(object)	基于 Token 的同源性检测相关方法.包括界面封装,文件接口处理和功能函数的总控调度.
CFG(object)	CFG 比对项目的界面设置以及接口处理.
intsign()	整数符号溢出的界面设计.因为涉及多线程所以没有使用通用模板.
subwindow(object):	附加内容的窗口和绑定的跳转方法.
tkintertoolbox(object)	一些在操作 tk 界面时经常用到的方法,比如选择文件(夹),更新文本框显示等.
uniframe(object)	其他漏洞检测类的父类.其他类均由该类继承,包含界面和跳转逻辑,以及占位的中控方法.
Multi(object)	多线程专用类

4.2 跳转结构实现

Tkinter 中按钮的指令绑定按照以下格式:

```
Button(fr1,text="源代码审计",command=self.deployHomologousDetection,width=16).grid(row=0,column=0)
```

注意绑定的方法不能附加任何参数.所以在按钮触发和跳转执行之间应该设置一层自动参数配置方法.

代理方法大致如下:

```
def deployHomologousDetection(self):
```

```
s1=Homologous_detection.Homologous_detection()
```

每个方法实例化一个目标类.

每个类的__init__(self)方法中部署了窗口绘制和拉起功能.,当这个类实例化时窗口即绘制部署.

4.3 基于 Tkinter 库的界面实现

Tkinter 是 Python 的事实上的标准 GUI（图形用户界面）软件包.它是 Tcl/Tk 之上的薄的面向对象层.⁵

Tkinter 不是唯一的 Python 的 GuiProgramming 工具包.但是,它是最常用的一种.CameronLaird 称每年做出的保留 TkInter 的决定是 “ Python 世界的次要传统之一” .

Tk 由 John Ousterhout 开发为 Tcl 脚本语言的 GUI 扩展.第一次发布是在 1991 年.Tk 在 1990 年代被证明是非常成功的,因为它比其他工具包更易于学习和使用⁶

Tkinter 库包含了常用的界面绘制组建:文本框,按钮,标签(固定文本)等.下面是一些在开发过程中遇到的问题⁷:

➤ 窗口居中:

```
def center_window(self,w, h):#input width and height

    # 获取屏幕 宽、高

    ws = self.root.winfo_screenwidth()

    hs = self.root.winfo_screenheight()

    # 计算 x, y 位置

    x = (ws/2) - (w/2)

    y = (hs/2) - (h/2)

    self.root.geometry('%dx%d+%d+%d' % (w, h, x, y))
```

➤ 关闭窗口

使用 destroy()方法而不是 quit,否则在多窗口情况下会出现功能问题.

➤ 按钮的指令

在设置按钮功能时,要求是一个无输入的方法或函数(?)

⁵ Python 官方文档 <https://wiki.python.org/moin/TkInter>

⁶ Python Course https://www.python-course.eu/python_tkinter.php

⁷ 全文收录在个人原创博客文章,见 <https://blog.csdn.net/POTASSIUM711/article/details/101545600>

➤ 初始化

初始化一个新窗口对象时,要在按钮触发时实例化,否则一开始就会跳出所有窗口.

➤ 居中

封装的 frame 要居中,应该设置 side 属性.

➤ 浏览文件⁸

➤ Scrollbar

里面的 master 属性是目标的上一层容器.例如,要做一个 text 的 scrollbar,指定的 master 应该是上一层的 frame.

See:<https://www.cnblogs.com/tynam/p/8778338.html>

需要为一个文本框专门创建一个容器将 scroll 对准.

绑定需要 S.config(command=T.yview).

➤ 改变控件内容

See <https://segmentfault.com/a/1190000014280410>

4.4 基于 token 的同源性匹配

4.4.1 Flex 的安装和部署

在这里使用 flex 配合分析. flex 的部署方法见

下面进行一些摘录.

一个典型的 lex 规则文件(*.l)由下面几个部分:⁹

➤ 全局声明

第一个由%{ }%包括的部分为 脚本的全局声明部分,用于声明一些全局的变量,.由于一次检查匹配只对于一个词,所以这里主要谁一些计数器的声明.

➤ 匹配规则

匹配规则由两个%限定范围.这里一个正则表达式对应一个代码块.像这样:

```
/*正则部分*/ {Operation,... return /*int*/;}
```

...

⁸ More Information See <https://blog.csdn.net/zjiang1994/article/details/53513377>,also https://blog.csdn.net/Abit_Go/article/details/77938938

⁹ 该段出自个人原创文章 <https://blog.csdn.net/POTASSIUM711/article/details/100166253>,有删改.

注意:程序的主函数 `yylex()`返回值是 `int`,所以建议返回一个整型的 `token`,再查表得到字符串形式的 `token`.

➤ 主体部分

这个部分控制主要函数的调用形式,以及一些 I/O 操作(如果必要)

一般对于一个完整的文件扫描,使用 `while(token=yylex())`循环.

一般的 Linux 发行版都有预装 Lex 的一个版本 Flex(?).如果没有,使用下列指令安装.

```
sudo apt-get install flex bison
```

转到编写好的.l 文件路径,输入指令将.l 转化为.yy.c 文件准备编译

```
flex filename.l
```

再将生成的 yy.c 用 cc 编译出.out.

```
cc lex.yy.c -lfl
```

得到解析器的.out 文件.再将待分析的.c 文件输入.

```
a.out<filename.c
```

此时终端应该有返回信息了,如果文件编写正确的话.

4.4.2 规则编写思路

输入的代码应该是过滤了注释和非必要空字符的.

在 lex 的规则中,直接匹配关键保留字并转换为 `token`.这个在 lex 中科直接指定,不需要使用正则.使用正则会造成很多误报和漏报.这里体现了 lex 的准确和便捷性.

下面是脚本的片段:

```
auto {return 0;}
break {return 1;}
case {return 2;}
char {return 3;}
const {return 4;}
continue {return 5;}
default {return 6;}
do {return 7;}
double {return 8;}
else {return 9;}
```


每个返回数对应 list 中的 index,在后续处理函数里根据该数返回一个以\$开头的 token.¹⁰

4.4.3 部署文件的自动化应用

为了防止文件名或其他读写问题引发的崩溃以及方便导入,先进行文件处理,将去除非必要空字符的代码转移至固定的 temp.c 文件中.

规则文件编译完成后,在程序内部调用控制台命令

```
os.popen("../tool1/a.out<../tool1/temp.c")
```

token 化完毕的代码显示在终端中,按照如下方法读取:

```
r= os.popen("../tool1/a.out<../tool1/temp.c")
textlist =r.readlines()
final=""
for text in textlist:
    #print(text) # 打印 cmd 输出结果
    final=final+text
return final
...

TODO: Sort the document structure
...
```

获取之后使用 TkinterToolbox()类的 textupdate 方法在指定文本框上显示.

4.4.4 Token 序列的匹配

这里使用 difflib 库的 SequenceMatcher.ratio()方法来进行相似率的计算.

这是一个灵活的类,可用于比较任何类型的序列对,只要序列元素为 hashable 对象. 其基本算法要早于由 Ratcliff 和 Obershelp 于 1980 年代末期发表并以“格式塔模式匹配”的夸张名称命名的算法,并且更加有趣一些. 其思路是找到不包含“垃圾”元素的最长连续匹配子序列;所谓“垃圾”元素是指其在某种意义上没有价值,例如空白行或空白符. (处理垃圾元素是对 Ratcliff 和 Obershelp 算法的一个扩展.) 然后同样的思路将递归地应用于匹配序列的左右序列片段. 这并不能产生最小编辑序列,但确实能产生在人们看来“正确”的匹配.¹¹

SequenceMatcher 主要功能是比较文本的距离.该算法较为复杂,主要是相似子序列的匹配.这里

¹⁰ 完整的列表见个人博文 <https://blog.csdn.net/POTASSIUM711/article/details/100166253>

¹¹ Python3.7 官方文档 <https://docs.python.org/zh-cn/3.7/library/difflib.html>

主要关注 `ratio()` 方法.

该方法返回一个取值范围 $[0, 1]$ 的浮点数作为序列相似性度量. 计算所有匹配片段的长度之和 T , 然后 $2 * T / (\text{len}(a) + \text{len}(b))$ 结果在 $[0, 1]$, 相同的时候返回 1, 没有相同片段返回 0

其中 T 是两个序列中元素的总数量, M 是匹配的数量, 即 $2.0 * M / T$. 请注意如果两个序列完全相同则该值为 1.0, 如果两者完全不同则为 0.0.

这个方法可以便捷有效获取 `token` 序列的相似度. 但值得注意的是, 该解决方案主要针对代码复用的定性分析. 数值本身不能作为参数, 但当数值过高时就意味着存在代码复用的可能, 可以进行人工查验.

4.5 基于 CFG 的静态调用关系分析

4.5.1 CFLOW 的安装和部署¹²

`cflow` 是一款静态分析 C 语言代码的工具, 通过它可以生成函数的调用关系. 和 `calltree` 不一样, `cflow` 有独立的网页介绍它 (<https://www.gnu.org/software/cflow/#TOCdocumentation>). 而且在 Ubuntu 系统上, 我们可以不用去编译 `cflow` 的源码, 而直接使用下面命令获取

```
apt-get install cflow
```

该开源软件可以绘制 ASCII 码格式的函数调用图, 也可以生成 `dot` 或 `png` 格式的调用图. 但本次实验只使用生成 ASCII 图的功能.

4.5.2 文件接口设置

为了防止文件读写造成的非预期错误, 使用预读取技术在启动 `cflow` 前将文件读取到固定路径中.

```
cp -i "+self.samplefilepath+" "+path+current_time+"sample.c
```

4.5.3 cflow 调用和结果处理

`cflow` 返回 `ascii` 形式的调用图如下所示:

```
+--main() <int main (void) at timer.c:13>
  +-ev_timer_init()
```

¹² 该部分参考了 <https://blog.csdn.net/breaksoftware/article/details/75576878> https://blog.csdn.net/vivid_moon/article/details/81902905 的经验并做了自己的整理, 作为非完全原创成果发布在了 <https://blog.csdn.net/POTASSIUM711/article/details/100567490> 上.. 下面不再赘述.

```

+-timeout_cb() <void timeout_cb (EV_P_ ev_timer *w, int revents) at timer.c:7>
| +-puts()
| \-ev_break()
+-ev_timer_start()
\ -ev_run()

```

生成的调用图存储在临时文件夹中以时间命名的项目临时文件夹.具体情况见测试板块.

使用正则屏蔽参数信息后进行符号比对返回相似度.同样使用 `diff.lib.sequence.ratio()`.

4.5.4 存在的问题以及改进方案

但是值得注意的是,这种方案仍然存在问题—过长的函数名和过深的交叉调用会大幅度降低相似度,即使被认定为是具有高度相似度的样本.对此预想的解决方法是:

- 函数 id 化,即以特定的 id 识别函数
- 获取函数调用图后转为 dot 格式再加载进图中解析比对.

时间所限没有进一步优化但上述方案是一个可行的优化解决方案.

4.6 应用 Splint 的解决方案¹³

4.6.1 绪言

splint 是一个静态检查 C 语言程序安全弱点和编写错误的工具.splint 会进行多种常规检查,包括未使用的变量,类型不一致,使用未定义变量,无法执行的代码,忽略返回值,执行路径未返回,无限循环等错误.同时通过在源码中添加注记给出的附加信息,使其可以进行功能更加强大的检查.而注记,则是对文件中的函数、变量、参数以及类型进行假定的一种的程式化的注释.¹⁴

本项目中使用 splint 进行检测的模块有:

- 空指针解引用
- 格式化字符串
- 整数宽度溢出
- 缓冲区溢出(堆/栈)

下面对这些模块进行统一说明.

¹³ 本节主要参考 <https://www.cnblogs.com/bangerlee/archive/2011/09/07/2166593.html>,以及自己整理的博文.

¹⁴ linux 下 splint 检测 C 语言代码质量,<https://blog.csdn.net/xylt1/article/details/77895841>

4.6.2 Splint 的部署¹⁵

```
tar -xvf splint-3.1.2.src.tgz
cd splint-3.1.2
./configure --prefix=/usr/local/splint
# To avoid permission denied in make process, use sudo
sudo make && sudo make install
```

环境变量的部署

```
~/bashrc

export LARCH_PATH=/usr/local/splint/share/splint/lib
export LCLIMPORTDIR=/usr/local/splint/share/splint/import
export PATH=/usr/local/splint/bin:$PATH
```

下面是每个模块的参数配置.

<i>Module</i>	<i>parameters</i>
<i>Empty Pointer Dereference</i>	-weak +null -hints -varuse +posixlib
<i>Stack Overflow</i>	+weak +bounds -hints -varuse +posixlib
<i>Heap overflow</i>	+weak +bounds -hints -varuse +posixlib
<i>Format string overflow</i>	+weak +ignoresigns +formatconst
<i>Integer width overflow</i>	+weak -hints -varuse +posixlib

通过调整这些 flag 可以有效控制程序针对性的进行检测.检测的结果会返回在控制台上.

4.6.3 预处理和输出处理

这些模块继承了 uniframe 类的 filesort 方法并在部署 splint 之前进行整理.这一个方法源码如下:

```
if self.path != "":
    os.system("mkdir -p ../tool3")#Create the root folder if not exist
    os.system("mkdir -p ../tool3/temp")#Create the root folder if not exist
    now=int(round(time.time()*1000))
    current_time=time.strftime('%Y-%m-%d_%H:%M:%S',time.localtime(now/1000))#Get_current
time with format to name the file
    command="cp -i "+self.path+" ../tool3/temp/"+current_time+".c"#Copy to the file
```

¹⁵ 更多内容见本人原创整理博文 <https://blog.csdn.net/POTASSIUM711/article/details/101560750>

```
os.system(command)

self.path="../../tool3/temp/"+current_time+".c"#Refresh the path
```

1. 为了避免目录缺失引发的错误,尝试进行创建文件夹操作,如果存在则跳过;
2. 建立一个以当前时间命名的临时文件并复制内容.
3. 重置 path 变量

收到 splint 的输出之后,在输出中搜索关键词,以判断是否出现此类错误.

因为 splint 即使是在 weak 模式下报告的消息依然较多,需要进行关键词判断确认.当然这种策略依然存在误报风险.splint 的输出样例在测试部分会给出.

下面是各模式中需要捕获的关键词.虽然不一定准确,但也是经过了多次测试得出的.

<i>Module</i>	<i>Keywords(in Regular expression)</i>
<i>Empty Pointer Dereference</i>	[null NULL]
<i>Stack Overflow</i>	[out-of-bounds buffer overflow]
<i>Heap overflow</i>	[buffer maxSet[
<i>Format string overflow</i>	Type of parameter is not consistent with corresponding code in format string.
<i>Integer width overflow</i>	Assignment

确认结果后将结果回显在文本框中.具体效果见测试部分.

4.7 整数符号漏洞的实现

由于在测试中 splint 对此类漏洞的检测效果不佳,所以本节是自己开发的,没有调用其他程序.

由于涉及到多线程,界面没有继承通用模板.

多线程相关功能的实现在下一节详述.本节主要说明单线程下对单个文件的检测.

根据原理,当出现敏感函数(本项目选取了几个有代表性的函数,其余的和这几个区别不大)并且某些不能为负的关键参数存在经由运算有成为负数的可能时,判定有整数符号溢出的风险(当然这个并不准确,这里为了简化考虑使用了如上表述).

本项目选取的有代表性的敏感函数有:

三参数函数 strncat/strncpy/memcpy, 第三个参数要求为 unsigned 型.

二参数函数 strcpy/strcat, 第二个参数要求为 unsigned 型.

1. 通过正则表达式获取变量的声明,包含变量名和类型

2. 查找并定位敏感函数,获取整个调用行(包括名称,参数)
 - a) 三参数函数的正则表达式: `(strncat|strncpy|memcpy) *\((((*a-zA-Z1-9_0]*\s*),((*a-zA-Z1-9_0]*\s*),((*a-zA-Z1-9_0]*\s*))\)`
 - b) 二参数函数的正则表达式: `(strcpy|strcat) *\((((*a-zA-Z1-9_0]*\s*),((*a-zA-Z1-9_0]*\s*))\)`
 3. 对每个调用进行分析,在变量声明列表中查找该变量的类型
 4. 如果有潜在的风险(可认为没有指明是 `unsigned/unsigned int` 变量),则认为存在风险并输出报告信息.
 5. 报告信息的格式为”Error occurred in <function_name> ->(para.,para.,para.)”
- 最后处理报告信息并显示在文本框内,应用 `tkintertoolbox` 的 `textupdate` 类进行更新.

4.8 多线程实现并发检测

4.8.1 多线程的原理

同一时间,CPU 只能处理 1 条线程,只有 1 条线程在工作(执行).多线程并发(同时)执行,其实是 CPU 快速地在多条线程之间调度(切换).如果 CPU 调度线程的时间足够快,就造成了多线程并发执行的假象

4.8.2 阻塞和同/异步¹⁶

● 同步

所谓同步,就是在发出一个功能调用时,在没有得到结果之前,该调用就不返回

简单来说就是当前程序执行完才能执行后面的程序,程序执行时按照顺序执行,平时写的代码基本都是同步的.

● 异步

异步的概念和同步相对,当一个异步过程调用发出后,调用者不会立刻得到结果.实际处理这个调用的部件是在调用发出后,通过状态、通知来通知调用者,或通过回调函数处理这个调用.

简单来说就是程序没有等到上一步程序执行完才执行下一步,而是直接往下执行,前提是下面的程序没有用到异步操作的值,异步的实现方式基本上都是多线程(定时任务也可实现,但是情况少).

● 阻塞

¹⁶ See <https://blog.csdn.net/u013007900/article/details/79350407>

阻塞调用是指调用结果返回之前,当前线程会被挂起.函数只有在得到结果之后才会返回.对于同步调用来说,很多时候当前线程还是激活的,只是从逻辑上当前函数没有返回而已.

4.8.3 综述

本节主要使用 python 的 multiprocessing 库进行多线程操作.设计方案如下:

1. 使用文件夹选取以触发多线程
2. 获取文件夹路径后遍历文件夹内所有符合要求的文件(*.c),形成文件列表
3. 文件整理,原名称存放在以时间命名的文件夹内.
4. 异步执行每一个文件,执行方式大致和单线程相同.
5. 线程执行结束即返回结果

4.8.4 单/多线程控制代理函数

选择文件和选择文件夹均返回的是字符串.为了区分单/多线程指令并初始化线程池,需要编写控制方法 multiproxy.

4.8.5 区别单/多线程

单线程的路径用 path 成员存放,格式应该精确到文件名.

多线程用类中成员 paths 存放,精确到文件夹.

默认是使用单线程.当 path 成员为空时调用多线程.

4.8.6 线程池的初始化

为达到最大化利用,使用 multiprocessing.cpu_count()检查环境 CPU 核心个数.之后设立容纳同等数量的线程池并初始化:

```
multiprocessing.Pool(processes = cpus)
```

4.8.7 线程的部署

阻塞线程的部署:

```
pool.apply()
```

该方法部署一个阻塞线程,即在该线程关闭前不会部署下一个线程.

非阻塞(异步)线程的部署:

```
pool.apply_async()
```

该方法部署一个异步线程,即只要线程池还有空余的线程可用(即正在部署的线程数量小于线

程限制),立即部署一个线程.每个线程间是异步的,即执行进度可以不一样.

单线程和多线程的区别

为了避免混淆为多线程设立了一个专用的 `multi` 类.里面除了上述的 `multiproxy` 方法外还有多线程专用的 `multideploy` 方法.主要不同是内置调用的是类内部的 `filesort4multi` 方法,修改了文件组织方案.因为一次操作涉及到多个文件,所以将时间设置为**文件夹名**而不是文件名.文件夹内按原名保存了待检测的代码副本.

5 系统测试

5.1 Launch

安装好必须组件(清单和安装方法见随附 readme.md)之后,将终端切换至 root/Source/目录,使用 python3 命令运行:

```
Python3 <path>Software_SEC_V2.0.py
```

5.2 界面测试

界面是基于 GUI 库的 tkinter.在 Linux 下应为默认界面.



图 5-1 根界面

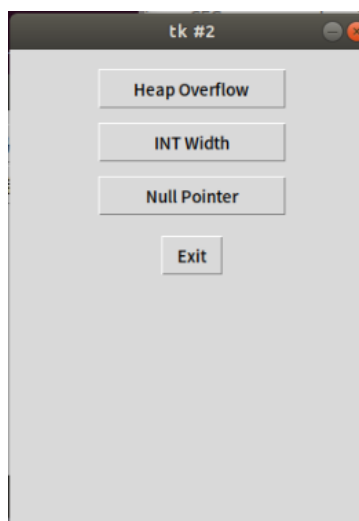


图 5-2 扩展内容选择界面

5.3 Token 同源性检测测试

触发[源代码审计]按钮,弹出如下所示界面:

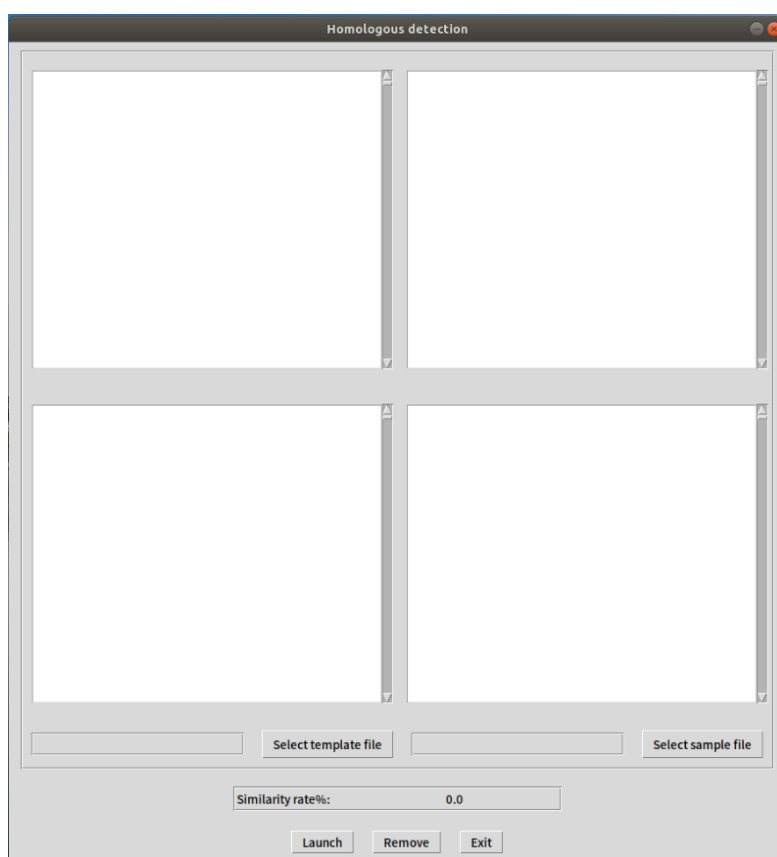


图 5-3 源代码审计界面

触发按钮选择文件.

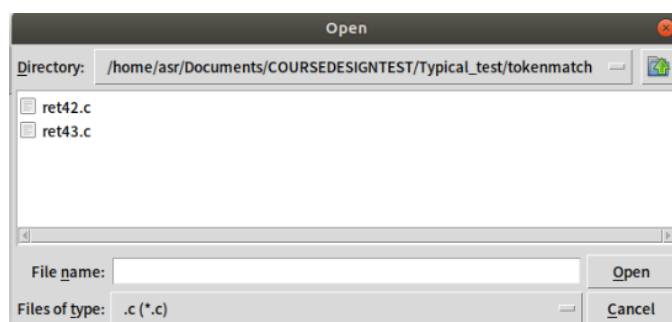


图 5-4 文件选择界面

分别选择文件之后源代码显示在文本框内.

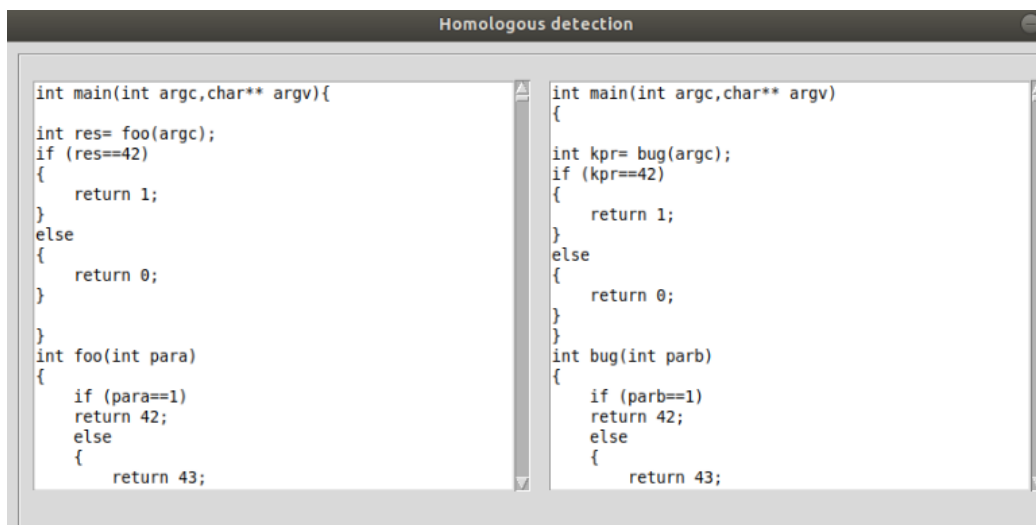


图 5-5 待测试的源代码

点击[Launch],开始生成 token 比对.

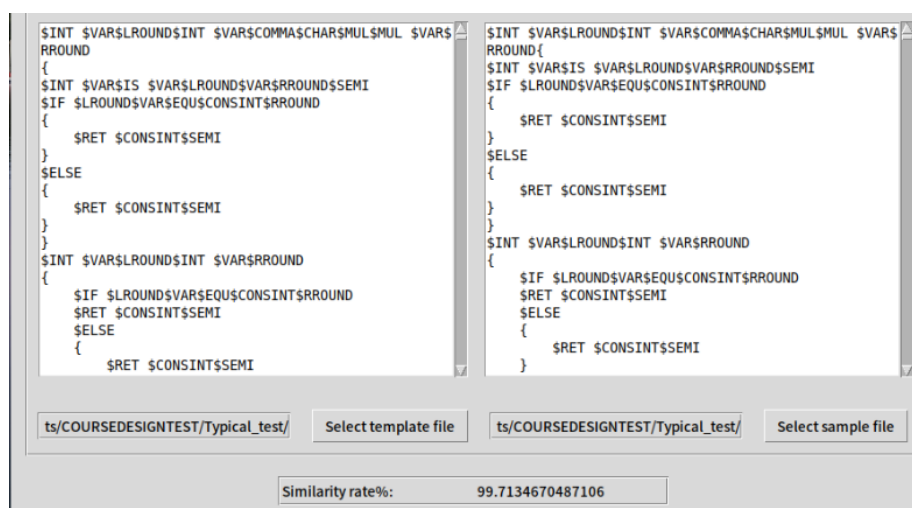


图 5-6 生成的 token 序列,以及比对的相似率

5.4 CFG 同源性测试

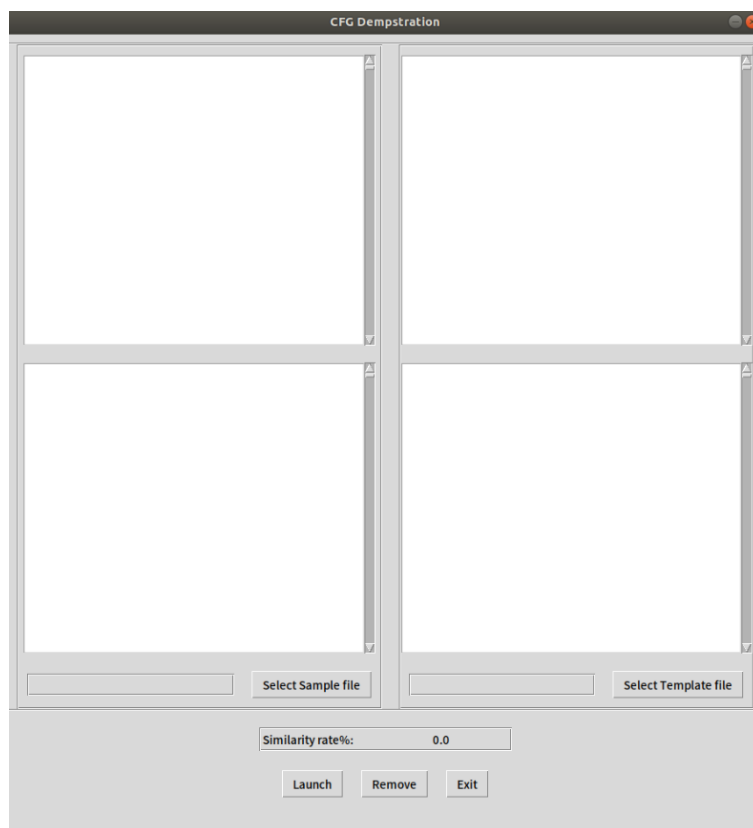


图 5-7 CFG 同源性检测的界面

选定文件进行比对.

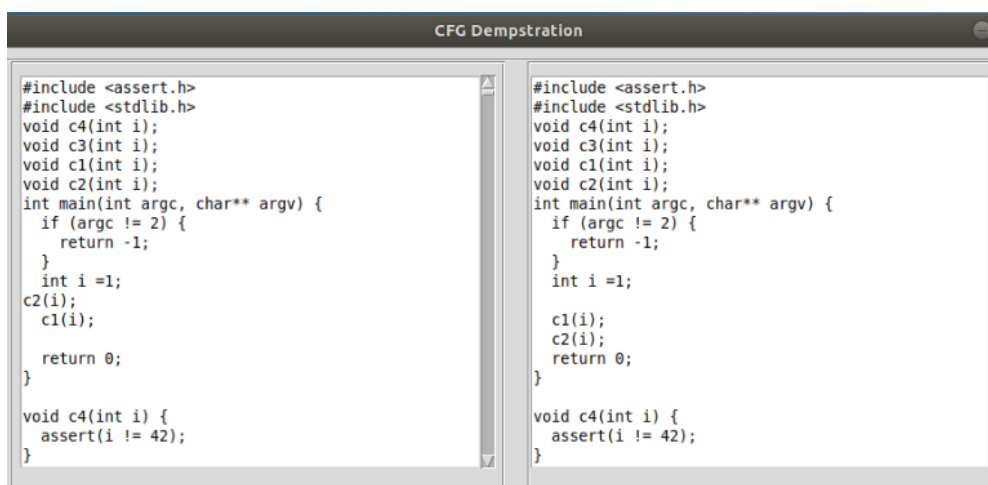


图 5-8 源代码显示情况

点击[Launch],调用 cflow 开始生成调用图并进行比对生成参考相似度.

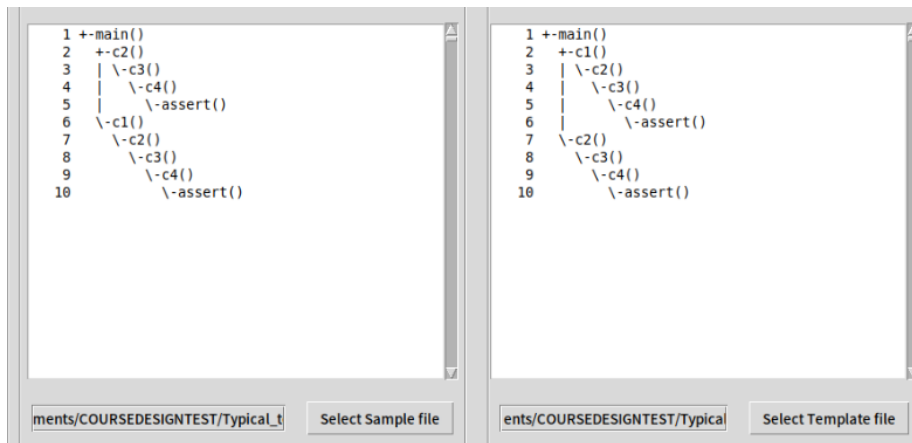


图 5-9 生成的调用图

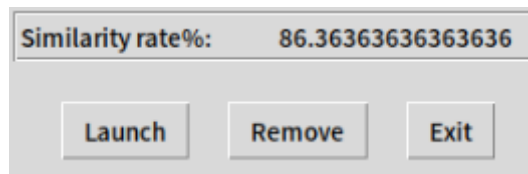


图 5-10 计算的参考相似度

./tool2/temp/<current_time>/目录下的临时文件

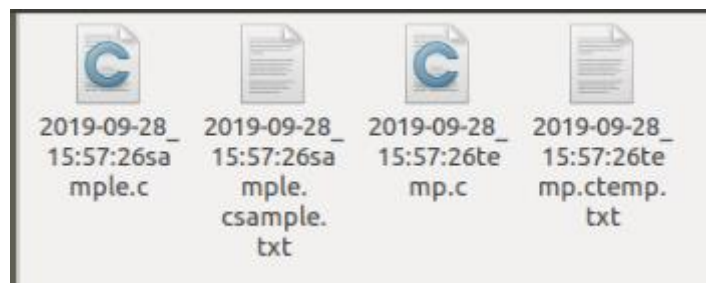


图 5-11 其中*.c 文件是文件的源码,*.txt 文件是调用图的存档.

其中一个源码的调用图存档如下:

```

1 +-main() <int main (int argc, char **argv) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:7>
2 +-c1() <void c1 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:30>
3 | \-c2() <void c2 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:26>
4 |   \-c3() <void c3 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:22>
5 |     \-c4() <void c4 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:18>
6 |       \-assert()
7 \-c2() <void c2 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:26>

```

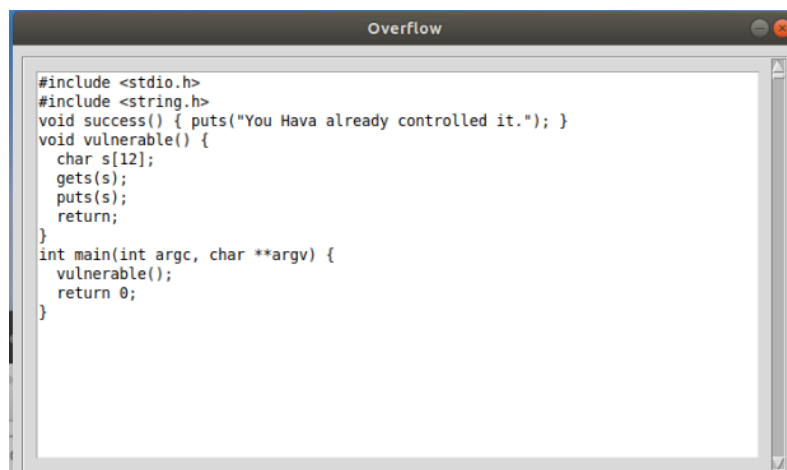
```
8  \-c3() <void c3 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:22>
9  \-c4() <void c4 (int i) at ../tool2/temp/2019-09-28_15:57:26/2019-09-28_15:57:26temp.c:18>
10  \-assert()
```

5.5 栈溢出测试



图 5-12 栈溢出检查使用的 uniframe 通用界面,下面不再赘述.

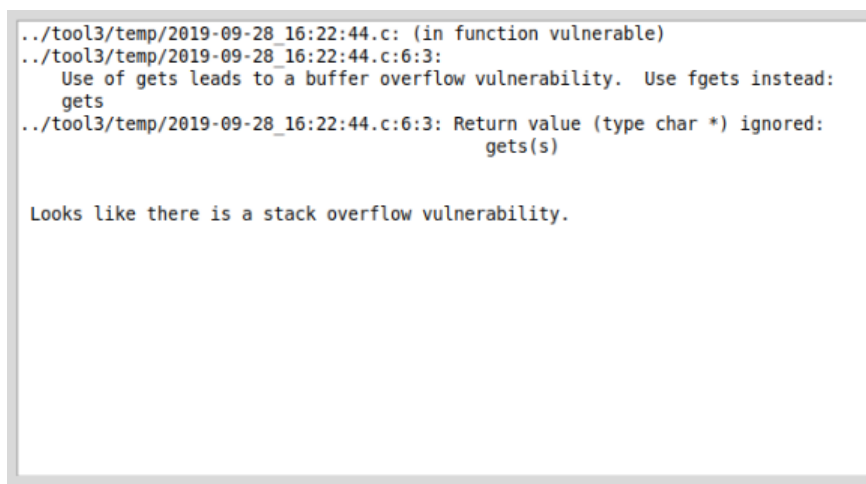
选择一个演示用典型存在该漏洞的文件



```
#include <stdio.h>
#include <string.h>
void success() { puts("You Hava already controlled it."); }
void vulnerable() {
    char s[12];
    gets(s);
    puts(s);
    return;
}
int main(int argc, char **argv) {
    vulnerable();
    return 0;
}
```

图 5-13 存在漏洞的源代码

点击[Launch]开始检测.

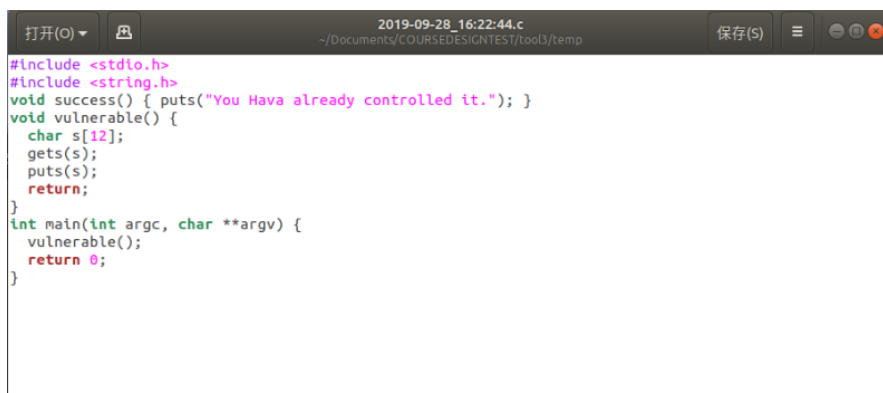


```
../tool3/temp/2019-09-28_16:22:44.c: (in function vulnerable)
../tool3/temp/2019-09-28_16:22:44.c:6:3:
    Use of gets leads to a buffer overflow vulnerability. Use fgets instead:
    gets
../tool3/temp/2019-09-28_16:22:44.c:6:3: Return value (type char *) ignored:
    gets(s)

Looks like there is a stack overflow vulnerability.
```

图 5-14 检测结果

检测到关键信息,返回报告为发现漏洞,位置在 6 行 3 列.结果符合预期.
临时文件的内容.



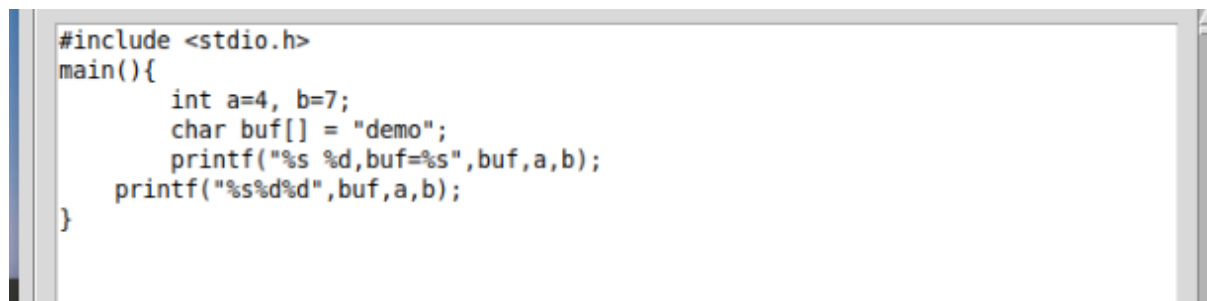
```
2019-09-28_16:22:44.c
~/Documents/COURSEDESIGNTEST/tool3/temp
保存(S)

#include <stdio.h>
#include <string.h>
void success() { puts("You Hava already controlled it."); }
void vulnerable() {
    char s[12];
    gets(s);
    puts(s);
    return;
}
int main(int argc, char **argv) {
    vulnerable();
    return 0;
}
```

图 5-15 临时文件的内容

5.6 格式化字符串漏洞检测测试

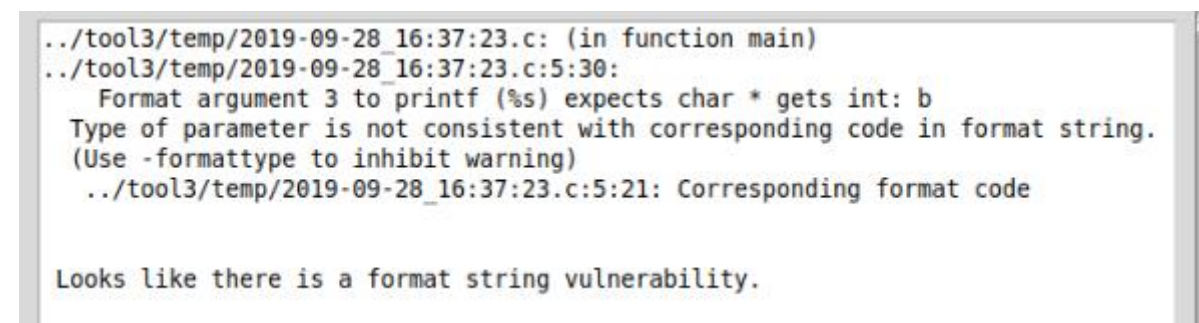
通用界面不再赘述.导入选择的文件



```
#include <stdio.h>
main(){
    int a=4, b=7;
    char buf[] = "demo";
    printf("%s %d,buf=%s",buf,a,b);
    printf("%s%d%d",buf,a,b);
}
```

图 5-16 选择的文件内容

检测结果显示存在此类漏洞.

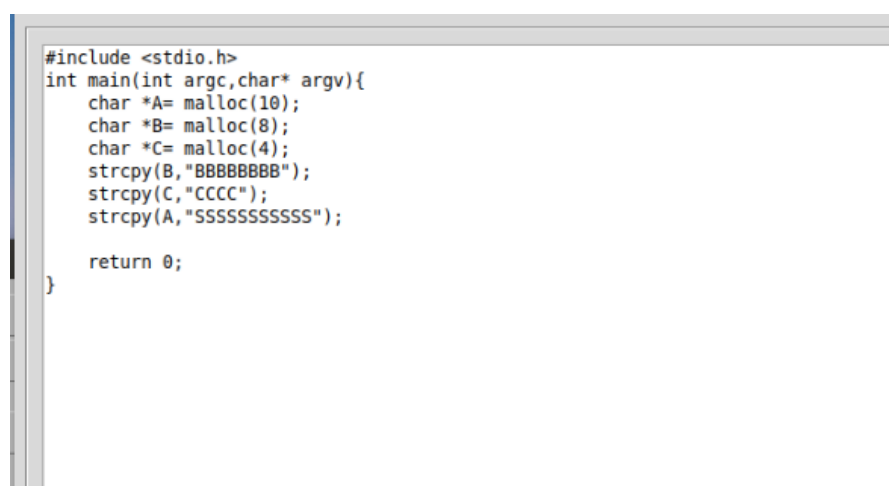


```
../tool3/temp/2019-09-28_16:37:23.c: (in function main)
../tool3/temp/2019-09-28_16:37:23.c:5:30:
    Format argument 3 to printf (%s) expects char * gets int: b
    Type of parameter is not consistent with corresponding code in format string.
    (Use -formattype to inhibit warning)
    ../tool3/temp/2019-09-28_16:37:23.c:5:21: Corresponding format code

Looks like there is a format string vulnerability.
```

图 5-17 返回的检测信息

5.7 堆溢出检测测试



```
#include <stdio.h>
int main(int argc,char* argv){
    char *A= malloc(10);
    char *B= malloc(8);
    char *C= malloc(4);
    strcpy(B,"BBBBBBBB");
    strcpy(C,"CCCC");
    strcpy(A,"SSSSSSSSSS");

    return 0;
}
```

图 5-18 测试源码


```
requires maxSet(B @ ../tool3/temp/2019-09-28_16:40:07.c:6:12) >= 8
  derived from strcpy precondition: requires maxSet(<parameter 1>) >=
requires maxSet(C @ ../tool3/temp/2019-09-28_16:40:07.c:7:12) >= 4
  derived from strcpy precondition: requires maxSet(<parameter 1>) >=
requires maxSet(A @ ../tool3/temp/2019-09-28_16:40:07.c:8:12) >= 11
  derived from strcpy precondition: requires maxSet(<parameter 1>) >=

Looks like there is a stack/heap overflow vulnerability.
```

图 5-19 测试结果

源码同样拷贝到临时文件中,此处不再赘述.

5.8 整数宽度溢出检测测试

```
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);
}
```

图 5-20 待检测源码

```
Assignment of int to unsigned short int: s = i

Looks like there is a truncated vulnerability.
```

图 5-21 检测结果

5.9 空指针解引用漏洞检测测试

```
int main()
{
    char* array=malloc(1024);
    char* array1=malloc(1024);
    memcpy(array,array1);
    return 0;
}
```

图 5-22 待检测的源码

```
../tool3/temp/2019-09-28_16:47:24.c: (in function main)
../tool3/temp/2019-09-28_16:47:24.c:5:5:
    Function memcpy called with 2 args, expects 3
../tool3/temp/2019-09-28_16:47:24.c:5:12:
    Possibly null storage array passed as non-null param: memcpy (array, ...)
../tool3/temp/2019-09-28_16:47:24.c:3:17: Storage array may become null
../tool3/temp/2019-09-28_16:47:24.c:5:18:
    Possibly null storage array1 passed as non-null param: memcpy (... , array1)
../tool3/temp/2019-09-28_16:47:24.c:4:18: Storage array1 may become null
```

Looks like there is a NULL pointer vulnerability.

图 5-23 检测结果 报告内容为存在错误

5.10 整数符号溢出-多线程部分

```
#include <stdio.h>
void kks(int a,int b,long c)
{
    memcpy(a,b,c);
}
int main(){
    int kksk;
    int pdse;
    long rous;

    strncpy(kksk,pdse,rous);
    strcat(pdse,rous);

    memcpy(kksk,pdse,rous);
    strcpy(pdse,kksk);
    strncat(kksk,pdse,rous);
    return 0;
}
```

图 5-24 测试源码

```
The following vulnerabilities may exist:  
Error occurred in strncpy->kksk,pdse,rous  
Error occurred in memcpy->kksk,pdse,rous  
Error occurred in strncat->kksk,pdse,rous  
Error occurred in strcat->pdse,rous  
Error occurred in strcpy->pdse,kksk
```

图 5-25 生成的错误报告,包括错误函数和涉及的参数.

5.11 整数符号溢出-多线程部分

测试环境 CPU 核数为 3 核.

测试样例中包含多个可能不含此类漏洞的样例.

```
/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign  
heap2.c  
1nts19n.c  
Formattest.c  
heap.c  
hea7.c  
intsigntest.c
```

图 5-26 选择的文件夹路径以及目录内的文件列表

下面是控制台输出的调试信息.

Pool set up

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/heap2.c

../tool4/temp/2019-09-28_17:00:05.c

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c

../tool4/temp/2019-09-28_17:00:07.c

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/Formattest.c

../tool4/temp/2019-09-28_17:00:09.c

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/heap.c

../tool4/temp/2019-09-28_17:00:11.c

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/hea7.c

../tool4/temp/2019-09-28_17:00:13.c

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c

../tool4/temp/2019-09-28_17:00:15.c

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/heap2.c:

The following vulnerabilities may exist:

None

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c:

The following vulnerabilities may exist:

Error occurred in

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c

strncpy->kksk,pdse,rous

Error occurred in

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c

memcpy->kksk,pdse,rous

Error occurred in

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c

strncat->kksk,pdse,rous

Error occurred in

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c

strcat->pdse,rous

Error occurred in

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/1nts19n.c

strcpy->pdse,kksk

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/Formattest.c:

The following vulnerabilities may exist:

None

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/heap.c:

The following vulnerabilities may exist:

None

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/hea7.c:

The following vulnerabilities may exist:

None

/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c:

The following vulnerabilities may exist:

Error occurred in

```
/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c
```

```
strncat->kksk,pdse,rous
```

```
Error occurred in
```

```
/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c
```

```
strncpy->kksk,pdse,rous
```

```
Error occurred in
```

```
/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c
```

```
memcpy->kksk,pdse,rous
```

```
Error occurred in
```

```
/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c
```

```
strcpy->pdse,kksk
```

```
Error occurred in
```

```
/home/asr/Documents/COURSEDESIGNTEST/Typical_test/intsign/intsigntest.c
```

```
strcat->pdse,rous
```

```
Exit
```

上述调试信息均成功显示在文本框内.结果符合预期.

最后顺利关闭线程池并退出.

以上测试一切符合预期.

6 总结与展望

6.1 Q&A

Q:为什么使用 Linux 系统?在虚拟机上开发会不方便吗?

A:

1. 因为在 Linux,特别是 Ubuntu 上进行 python2/3 的库的安装比较方便.虽然都可以用 pip/pip3 安装,但 Windows 上会发生很多未知错误导致库安装失败(比如权限问题).而 Linux 下这种问题就少得多,如果出现权限问题只需要在 pip 命令后加上—user 就可以解决.
2. 本项目所用到的很多开源工具都只能在 Linux 上运行.为了与其相匹配,代码中应用到控制台的指令都是 Linux 下的.
3. 在虚拟机上开发,由于 I/O 设备信号还需要转接,有延迟是难免的.隔着一层虚拟环境的确不方便.但好处是随用随删,随时可以重新搭建环境.

Q:为什么使用工具而不是自己写?

A:这些工具是在做前置的可行性研究时发现的.考虑到业内已经有很成熟的开源软件,稳定性和完备性均较高并且使用比较方便,所以最后决定使用这些开源软件.作为测试比较,整数宽度溢出(truncated.py)和整数符号溢出(intsign.py)部分自己也写了一份,在后来的对比测试中发现了两者之间的差距,决定整数宽度部分使用工具而整数符号溢出部分使用自己写的代码.同时还测试了多款开源工具,经过多次查证比较之后才确定的方案.而且掌握这些工具的使用方法并且进行自动化部署和启动也是一项不容易的工程,可以说工作量不比自己写小.

Q:存在哪些问题?还有哪些地方需要改进?

A:由于开源工具的功能和本课设的需求并不完全一致,一些极端样例并不能完全覆盖到.加之开源工具也存在一些错误,导致结果不一定准确(比如对于代码片段或不能运行的代码就不能识别,即使这个片段就存在错误).这个还需要进一步打磨.

文件组织上由于很多想法都是开发过程中出现的,早期开发的版本不一定都能修改,所以可能会出现一些问题.

6.2 总结

本次课程设计工作量较大,在开发过程中对自己的一些开发习惯也做了思考和调整.上一次的数据结构课设中编写了很多文档,说明等,本次因为时间紧迫就少了许多,更多的是工具的使用笔记和一些踩坑集锦,总体上更加精简了.

同时相较于以前,本次的工作成果更多地整理转化为了公开发布的成果(虽然也只是些只言片语和不成熟的想法),但对于有同样问题的人而言能让他们少走不少弯路.这也算是本次课设的附加成果.

此外,本次课设也是第一次使用 **Python** 和 **GUI** 编写大规模项目.在开发过程中查阅了大量官方文档和他人总结的资料.对代码能力还是有较大的提升.而且在 **Linux** 上开发也是第一次.同样获益不少.

时间和能力所限,没能做到最满意的地步,希望以后能更进一步提升代码能力和基础知识水平.

7 参考文献

[1] 王雅文, 姚欣洪, 宫云战, 杨朝红,. 一种基于代码静态分析的缓冲区溢出检测算法 [J]. , 2012, 49(4): 839-845. Wang Yawen, Yao Xinhong, Gong Yunzhan, and Yang Zhaohong. A Method of Buffer Overflow Detection Based on Static Code Analysis. Journal of Computer Research and Development, 2012, 49(4): 839-845.

[2] Python 官方文档, <https://docs.python.org/zh-cn/>

[3] Tkinter, Python Wiki, <https://wiki.python.org/moin/TkInter>

[4] argparse – Command line option and argument parsing. <https://pymotw.com/2/argparse/>
更多有参考价值的博文或文档在脚注中列出.