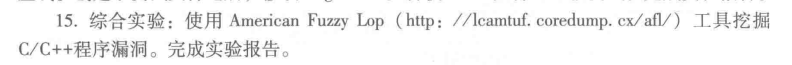
## 选取题目：

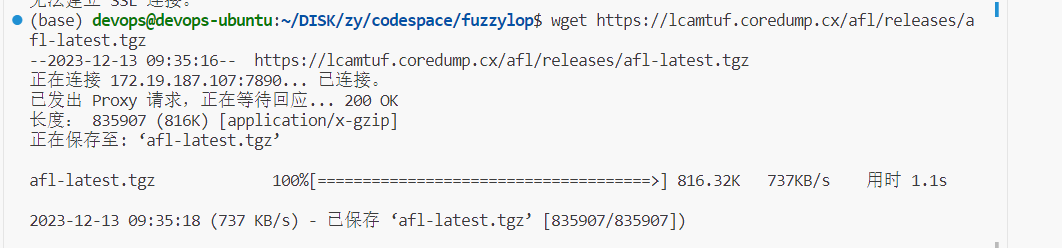
P239 15



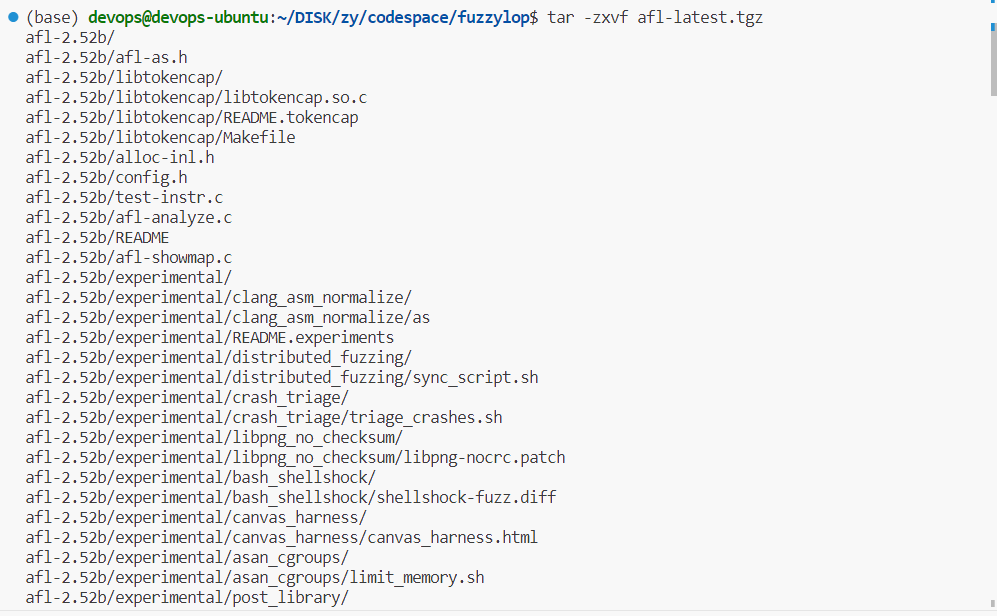
## 软件安装：

（这个软件在windows上貌似不是很好使用，所以选取实验室的linux服务器作为环境）

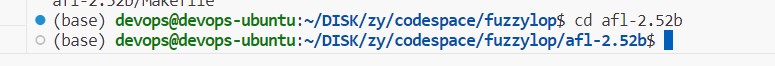
下载：



解压：

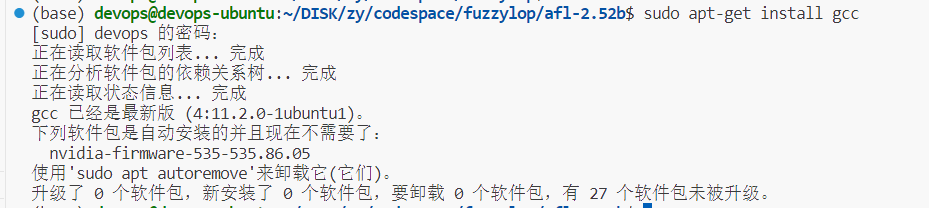


进入：

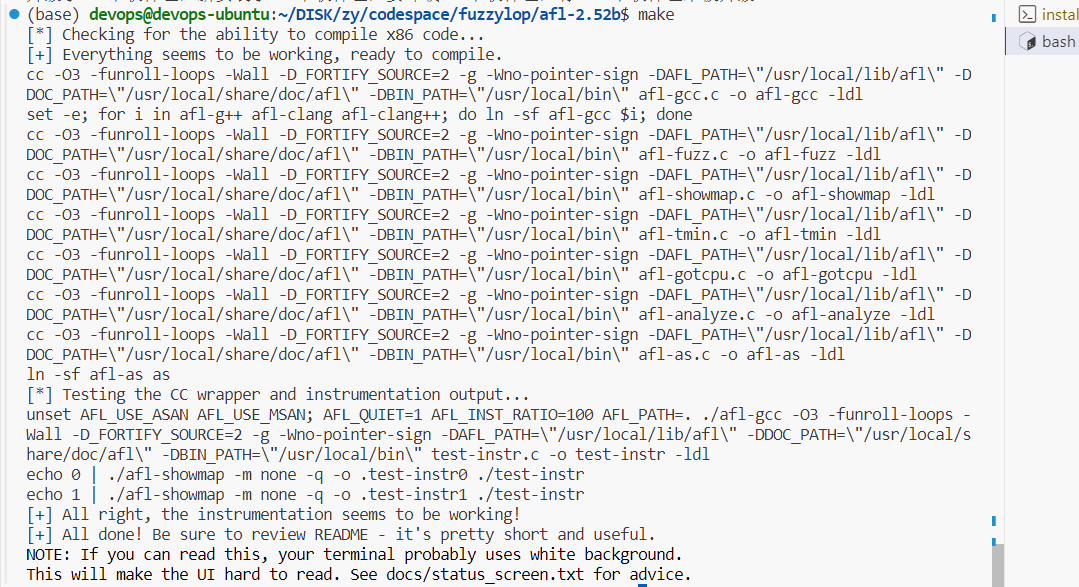


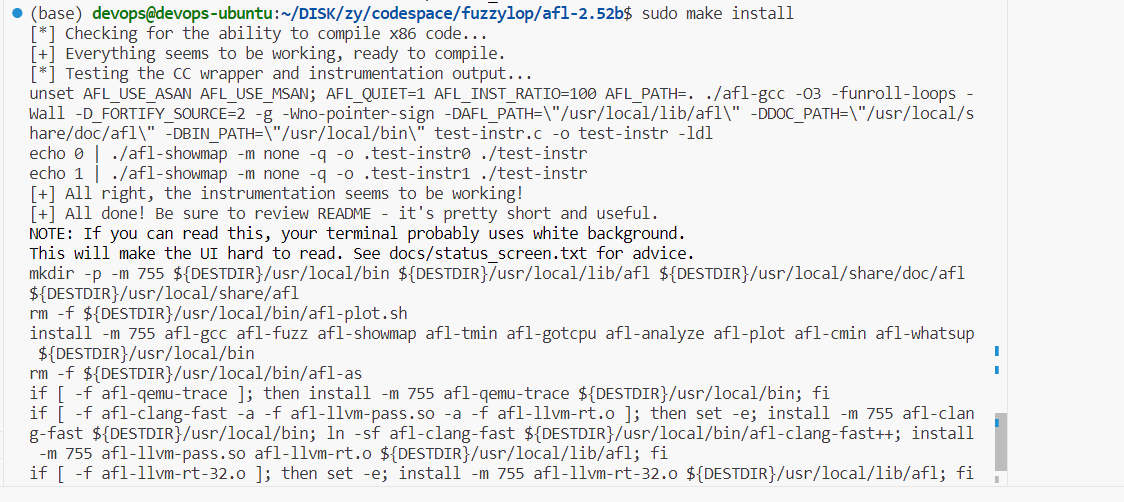
安装gcc编译器：

（发现已经安装好了）



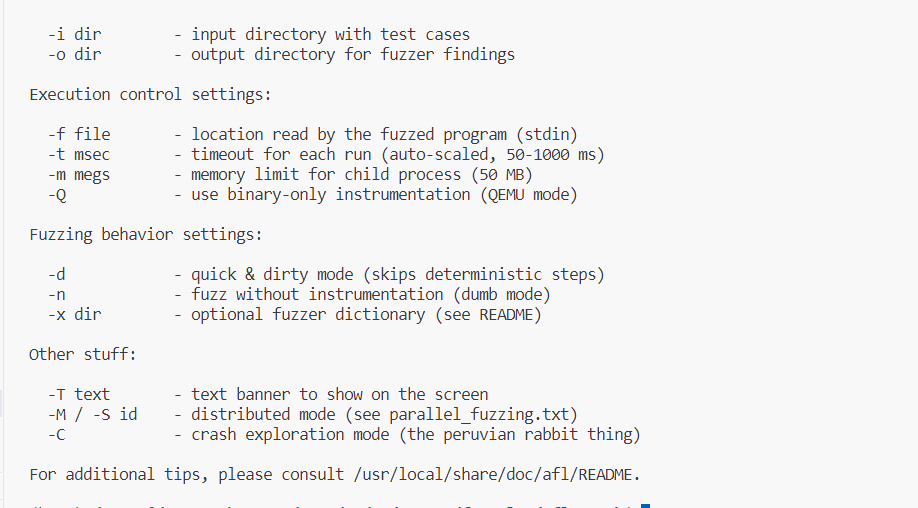
安装：



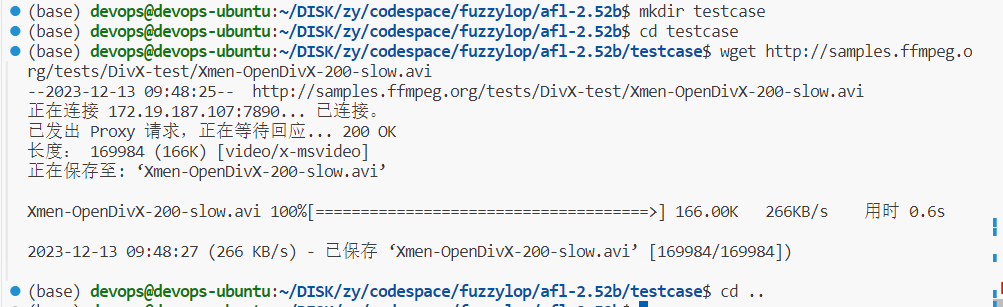


检验：

输入alf-fuzz检查是否安装成功

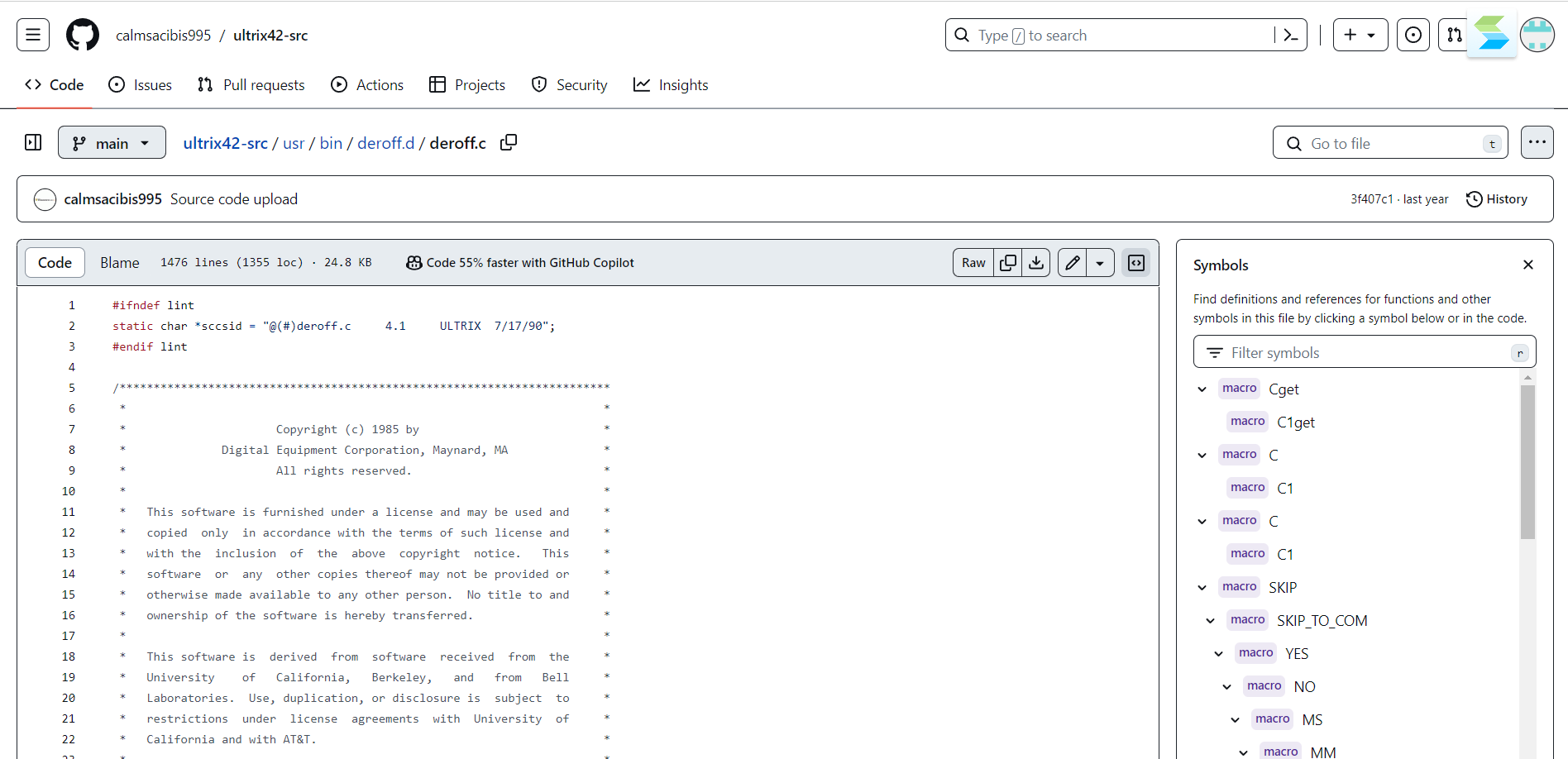


安装漏洞语料库、目标C程序：



### 静态分析过程

选取的代码是：



这是

使用Python选取项目中所有的.c文件，并将文件放入flawfinder中进行分析，产出对应的分析报告，删除分析结果为空的报告。

（flawfinder本身是可以以目录为输入自动寻找需要处理的文件的，但是由于编码问题我本地跑会报错，所以采用utf-8存到新文件再分析的过程）

代码截图：

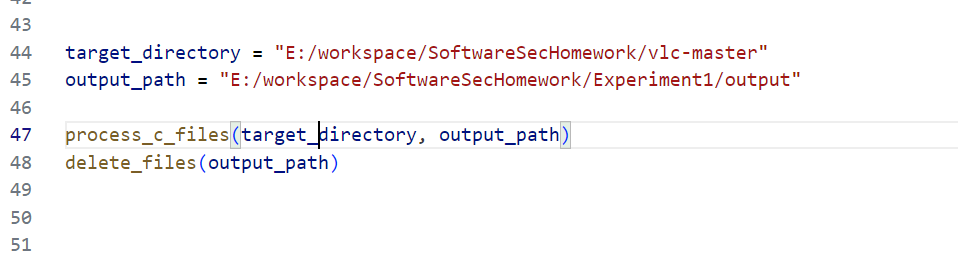
选取.c文件并用flawfinder分析部分代码：



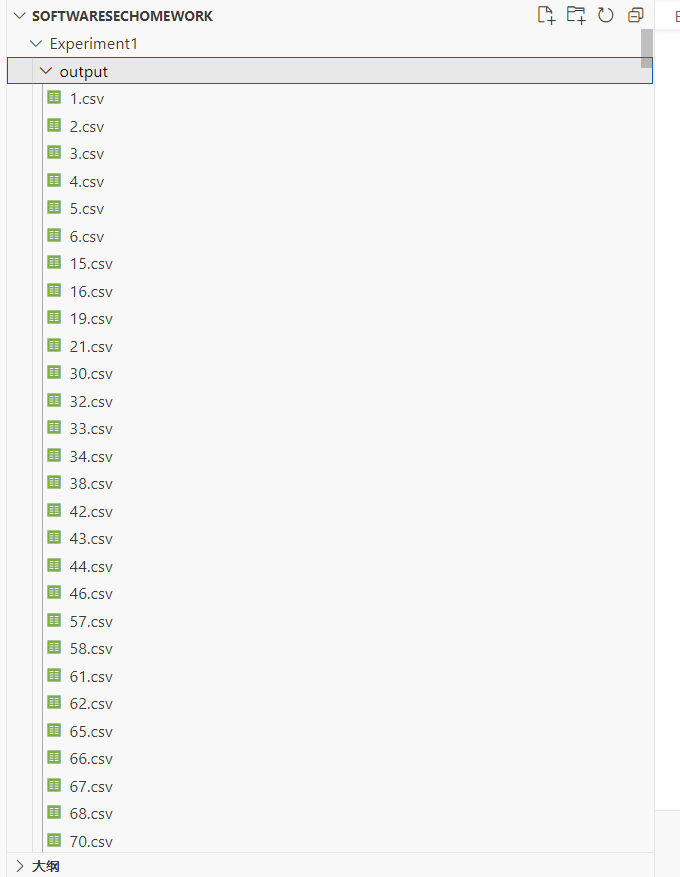
删除空报告部分代码：



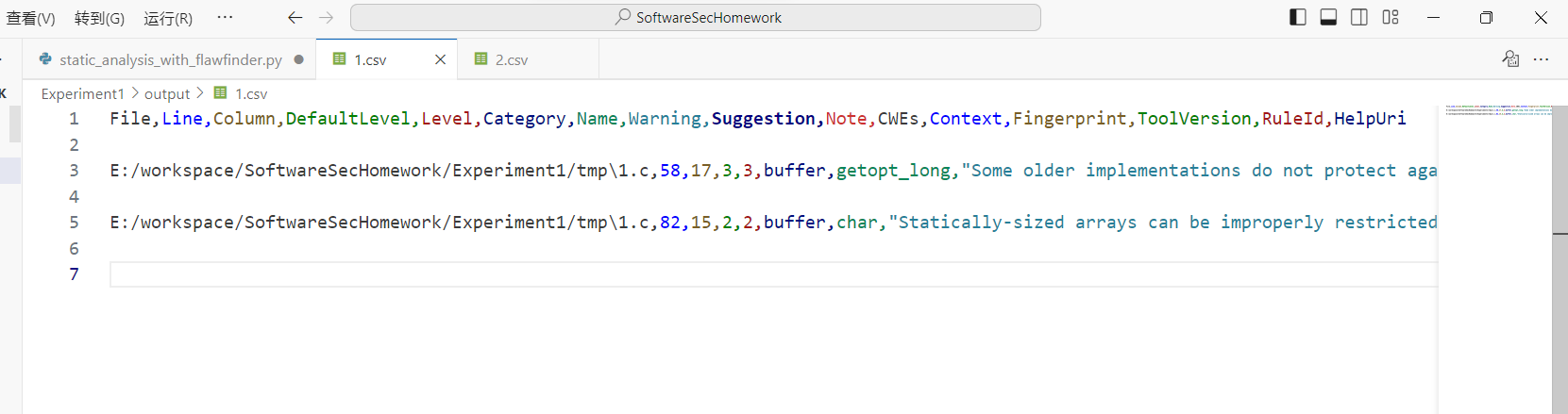
Main函数：



分析结果截图：



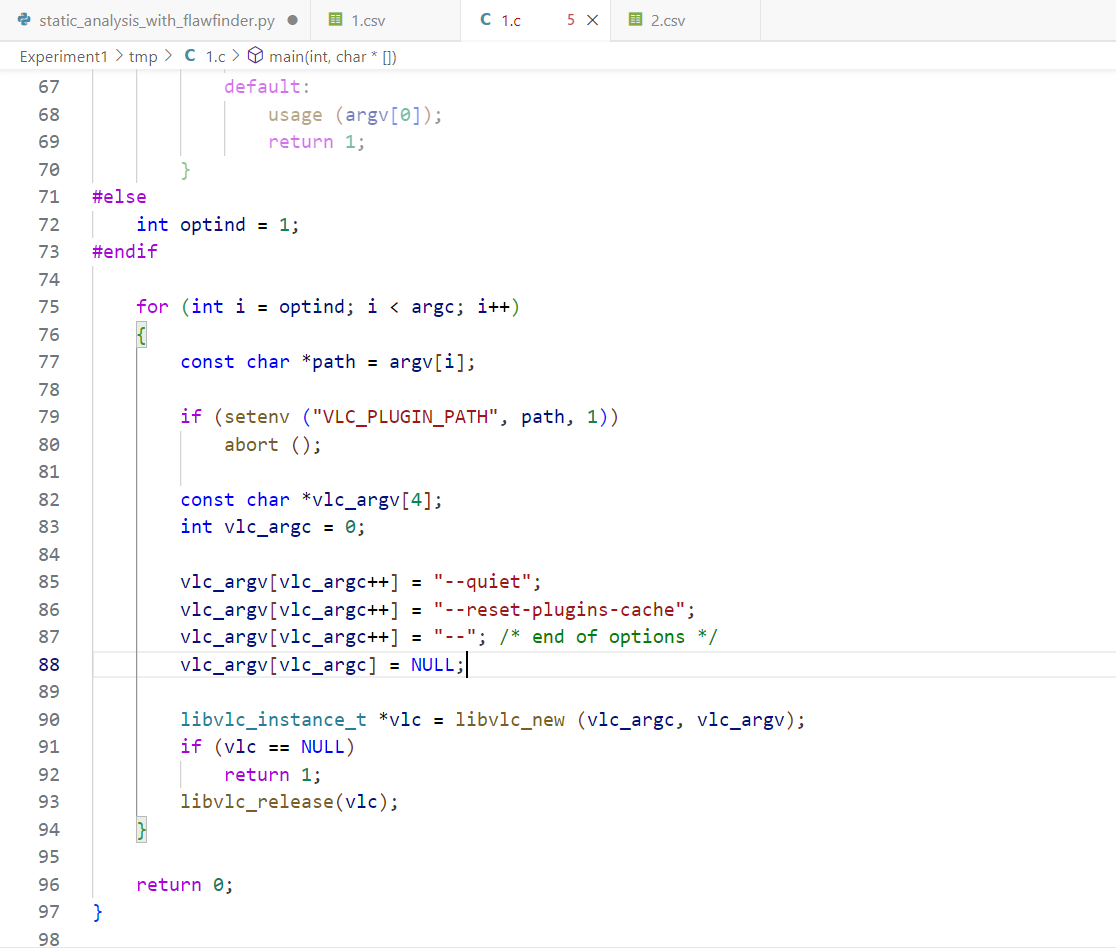
实例：



## 单个报告具体内容分析：

因为产出上百份报告无法逐个分析，选取其中1.csv进行单独分析。

对应的源代码函数部分截图：



分析结果内容：

（因为内容很长图放不下直接复制了过来）

File,Line,Column,DefaultLevel,Level,Category,Name,Warning,**Suggestion,**Note,CWEs,Context,Fingerprint,ToolVersion,RuleId,HelpUri

E:/workspace/SoftwareSecHomework/Experiment1/tmp\1.c,58,17,3,3,buffer,getopt\_long,"Some older implementations do not protect against internal buffer overflows (CWE-120, CWE-20).",**"Check implementation on installation, or limit the size of all string inputs.",**,"CWE-120, CWE-20","    while ((c = getopt\_long (argc, argv, ""hV"", opts, NULL)) != -1)",603d906030654b3ba3a72682fd08d821efcd8ec644c006f73db13e3b78f333b0,2.0.19,FF1027,https://cwe.mitre.org/data/definitions/120.html

E:/workspace/SoftwareSecHomework/Experiment1/tmp\1.c,82,15,2,2,buffer,char,"Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120).",**"Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.",**,CWE-119!/CWE-120,        const char \*vlc\_argv[4];,5754bc812b81e773366e7053e0206e36a58725845a535797857cf8bcbcc4f097,2.0.19,FF1013,https://cwe.mitre.org/data/definitions/119.html

翻译一下大概是：

在第58行的getopt\_long函数调用：

警告类别：Buffer

级别：3

描述：有些旧的实现可能不保护内部缓冲区溢出（CWE-120, CWE-20）。

建议：检查安装上的实现，或限制所有字符串输入的大小。

在第82行的静态数组定义：

警告类别：Buffer

级别：2

描述：静态大小的数组可能被不正确地限制，可能导致溢出或其他问题（CWE-119!/CWE-120）。

建议：进行边界检查，使用限制长度的函数，或确保大小大于最大可能的长度。

具体来说：

第一个错是getopt\_long这个库函数导致的，他是用于解析命令行参数的，比如-version这种参数。如果这个参数足够多确实可以让内部缓冲区溢出，导致buffer error。Flawfinder建议限制一下输入的字符大小，这样就不太可能造成缓冲区溢出了。

第二个错是



这行代码导致的，确实在不检查数组大小的情况下就访问数组第五个元素，万一数组大小小于等于4就会造成越界访问，在开发过程可能在其他地方限制了这个数组的大小，但是在这里还是加一下长度检验比较稳妥，这也是Flawfinder建议的做法。

## 静态分析工具对比

选取Flawfinder、Rats、Splint、Cppcheck这四个静态分析工具进行功能上的对比

（参考四个工具的官方文档）

### Flawfinder

Flawfinder 是一个简单的程序，用于检查C / C++源代码并报告可能的安全漏洞（“缺陷”），按风险级别排序。它对于在程序向公众广泛发布之前快速发现和消除至少一些潜在的安全问题非常有用。

Flawfinder是python编写的，运行速度不如C快，但是在漏洞检测任务上戳戳有余了，并且通过版本更迭，Linux上目前检查速度已经来到了平均45,126 行/秒。

Flawfinder的工作原理是基于一组已知的漏洞模式和规则进行扫描，使用这些规则来识别代码中的潜在漏洞。

Flawfinder使用简单，例如，在撰写本文时，Flawfinder 更易于使用，只需给 flawfinder 一个目录名称，flawfinder 就会递归进入目录，找出需要分析的内容，然后进行分析。

flawfinder 的其他优点是它可以处理国际化程序（它知道像 gettext（） 这样的特殊调用，不像 RATS），flawfinder 可以报告命中的列号（以及行号），并且 flawfinder 可以生成 HTML 格式的结果。自动递归和 HTML 格式的结果使 flawfinder 特别适合源代码托管系统。Flawfinder数据库包含许多不在 RAT 中的条目，因此Flawfinder会找到 RATS 不会找到的东西。

### RATS

RATS（Rough Auditing Tool for Security）是一种工具，用于扫描C，C++，Perl，PHP和Python源代码，并标记常见的与安全相关的编程错误，例如缓冲区溢出和TOCTOU（检查时间，使用时间）竞争条件。

RATS 扫描工具为安全分析师提供了需要关注的潜在故障点列表，以及描述了问题所在，并可能提出补救措施。它还提供了对每个问题潜在严重性的相对评估，以更好地帮助审计师确定优先级。该工具还执行一些基本分析，以尝试排除显然不是问题的情况。

RATS 可以处理C / C++以外的其他编程语言并且运行速度比flawfinder更快。这两个项目本质上都是自动化顾问，让两个顾问查看你的程序可能比只使用一个更好（这有点类似于让两个人审查你的代码以确保安全）。

### Splint

Splint 是 Secure Programming Lint 的缩写，是一种用于静态检查 C 程序是否存在安全漏洞和编码错误的编程工具。它以前称为 LCLint，是 Unix lint 工具的现代版本。

与其他工具相比Splint能处理的编程语言只有C，算是能处理语言最少的工具。但是，Splint 能够解释源代码的特殊注释，这使其比仅查看源代码的静态分析工具更强大。

Splint 被设计用来强调代码规范和良好的编码实践。它可以帮助发现代码中不符合规范的部分，从而改善代码的可读性和可维护性。其他工具更多倾向于找代码中的漏洞，splint还关注了编码规范这一块，即使没有漏洞，糟糕的编码习惯也会被Splint查出来。

### Cppcheck

Cppcheck 是 C/C++ 代码的静态分析工具。它提供独特的代码分析来检测错误，并专注于检测未定义的行为和危险的编码结构，目标是减少误报。Cppcheck 旨在能够分析您的 C/C++ 代码，即使它具有非标准语法（在嵌入式项目中很常见）。

在检测与分析内容上，Cppcheck 非常注重检测未定义的行为。

在使用上，Cppcheck应该是唯一一个拥有图形化界面的工具，这使得他的使用摆脱命令行，降低了学习成本和使用门槛。

原理上，Cppcheck 使用不健全的流敏感分析。其他几个分析器使用基于抽象解释的路径敏感分析，这也很好，但既有优点也有缺点。从理论上讲，路径敏感分析比流敏感分析更好。但在实践中，这意味着 Cppcheck 将检测到其他工具无法检测到的错误。

上面这段话有点抽象，具体来说，流敏感分析 侧重于跟踪数据和控制流的变化。它关注的是程序执行的实际路径，但并不会考虑所有可能的执行路径。这样不仅成本低，而且可以找到其他工具找不到的漏洞，最重要的是误报率会有降低。

特殊的是，Cppcheck中的检测流是双向的，简单来说，有的漏洞是需要逆向分析才能找到的。这个具体可以看这张图的介绍：

