Lab 4 Program Analysis | CodeQL & Fuzzing

老师您好QWQ~我选择 **静态分析** 部分为作业；**动态分析** 部分为额外尝试的内容。因此我将两部分同时放在该报告文档中，静态分析部分放在前面。

本文同时记录了我一些学习的笔记记录，为了避免带来困扰，我提交时将使用 灰色 字体来表示这些内容与作业本身无关。感谢您！

程序分析 就是在编译 / 运行时进行的自动化的、检测可能存在的漏洞的分析的总称。笼统来说，可以分为：

- 动态分析：运行时进行检测或者响应；好处是检测到的一定是漏洞，坏处是不完备；
- 静态分析：编译时检查；都能找到，但是容易误报；
  - 结构化静态分析：不考虑语义的分析；
  - 控制流 / 数据流分析。

Lab 4 用两个实验尝试程序分析。

1 静态分析 | HW–B: Static Analysis with CodeQL

1.1 什么是 CodeQL / QL 语法入门

> 该部分内容参考了：
>
> - 实验指导
> - QL Language Reference
> - CodeQL for C and C++
>
> 该部分内容只是 QL 语法的摘录，并不完备

CodeQL 是 GitHub 开发的静态分析框架；将源代码的语法、语义、数据类型、数据流图、控制流图等相关信息提取到数据库中，然后通过编写 QL 代码查询数据库的方式找到可能存在的漏洞。

QL 是一种 query language，也是 logic language，长得有点像 SQL。在 QL 里，字面量有 Boolean (`true` , `false`), Integer, Float, String 这几种。另外 QL 还支持 ranges，如 `[0 .. 2]` ，以及 sets，如 `[1, 3, 9]` 。

Select

显然，一个 QL 程序的核心就是 queries。Select clauses 的格式如下：

```SQL
from /* ... variable declarations ... */
where /* ... logical formula ... */
select /* ... expressions ... */
```

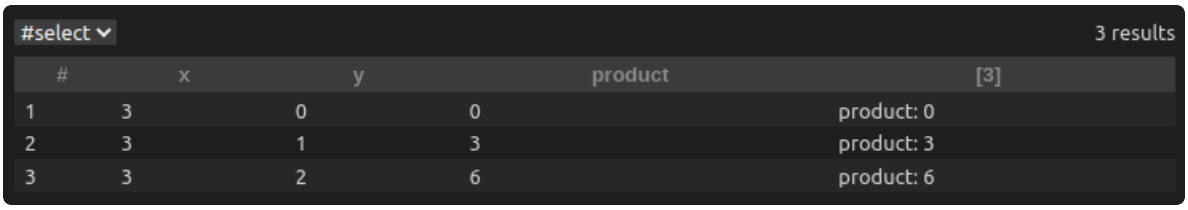
（语雀竟然没有 QL 的高亮）

其中，`from` 和 `where` 是可选的。类似 SQL，这里也可以有 `as` 和 `order by`（`asc`，`desc`）。

例如：

```SQL
from int x, int y
where x = 3 and y in [0 .. 2]
select x, y, x * y as product, "product: " + product
```

可以得到如下结果（实际上运行这种和 database 无关的 query 也需要一个 database，我们稍后讨论如何建立 database）：



Aggregations

和 Aggregations 聚集函数 | SQL 一样，QL 也有 Aggregations，但是和 SQL 的语法不太一样。QL 的 Aggregations 的语法为 `<aggregate>(<variable declarations> | <formula> | <expression>)`。

例如，`select sum(int i, int j | i = [0 .. 2] and j = [3 .. 5] | i * j)` 的结果即为：

更多的 Aggregations 可以在 Aggregations | QL Language Reference 中找到。

如果只有一个 Aggregation variable，那么 `<expression>` 可以省略。例如如下两个表达式等价：

PL/SQL | 复制代码

```
avg(int i | i = [0 .. 3] | i)
avg(int i | i = [0 .. 3])
```

Formulas

Aggregations 里面有 formula，它主要包括这样几种：

- Comparisons
  - `>, >=, <, <=, =, !=`
  - 需要说明的是，`A = B` holds 当且仅当 A 中的**一个**值和 B 中的**一个**值相等；对应地，`A != B` holds 当且仅当 A 中的**一个**值和 B 中的**一个**值不相等；因此 `not A = B` holds 当且仅当 A 和 B 中**没有**共同值，这与 `A != B` 是不同的。例如以下 Comparisons 为真：`1 != [1 .. 2]`，`1 = [1 .. 2]`，`[2 .. 5] != [1 .. 2]`；以下 Comparisons 为假：`not 1 = [1 .. 2]`。
    - 这里当然也有 `not`，`and`，`or` 这些关键字。
- `<expression> instanceof <type>`
- `<expression> in <range>`
- Calls to Predicates
  - Predicates, 谓词，是用来简化判断的。例如下面两个 predicates，其意义是明显的：

```
predicate isCountry(string country) {
  country = "Germany"
  or
  country = "Belgium"
  or
  country = "France"
}

predicate isSmall(int i) {
  i in [1 .. 9]
}
```

SQL | 复制代码

    - 还有一些内置的 predicates，比如 `any()` 永真， `none()` 永假。
- Quantified formulas
    - `exists(<variable declarations> | <formula>)`，含义是显然的；
        - `exists(<variable declarations> | <formula 1> | <formula 2>)` 等价于 `exists(<variable declarations> | <formula 1> and <formula 2>)`
    - `forall(<variable declarations> | <formula 1> | <formula 2>)`，它 holds 当且仅当所有满足 `formula 1` 的 values 都满足 `formula 2`
        - 值得注意的是，如果没有任何 value 满足 `formula 1`，那么该语句始终 holds
    - `forex(<variable declarations> | <formula 1> | <formula 2>)` 就是 `forall` 和 `exists` 的结合；它 holds 当且仅当 `forall(<vars> | <formula 1> | <formula 2>) and exists(<vars> | <formula 1> | <formula 2>)`。

Classes

我们也可以用 classes 来进一步提高重用。下面三段代码的效果是一致的：

SQL 复制代码

```
// - 1 -
from FunctionCall call
where call.getTarget().hasName("free")
select call

// - 2 -
predicate isFreeCall(FunctionCall call) {
  call.getTarget().hasName("free")
}

from FunctionCall call
where isFreeCall(call)
select call

// - 3 -
class FreeCall extends FunctionCall {
  FreeCall() {
    this.getTarget().hasName("free")
  }
}

from FreeCall call
select call
```

建立数据库和查询示例

现在我们将这种查询应用到具体的工程中。我们用

此处为语雀内容卡片，点击链接查看：https://www.yuque.com/xianyuxuan/coding/ssec_lab1?view=doc_embed

的第一题 `bof-baby` 创建一个数据库：

Shell 复制代码

```
codeql database create /home/ssec2022/Desktop/ql_databases/bof-baby --source-root=/home/ssec2022/ssec22spring-stu/hw-01/01_bof_baby --language=cpp --command="gcc /home/ssec2022/ssec22spring-stu/hw-01/01_bof_baby/bof-baby.c -o /home/ssec2022/ssec22spring-stu/hw-01/01_bof_baby/bof-baby -fno-stack-protector -fno-pie -no-pie -mpreferred-stack-boundary=4"
```

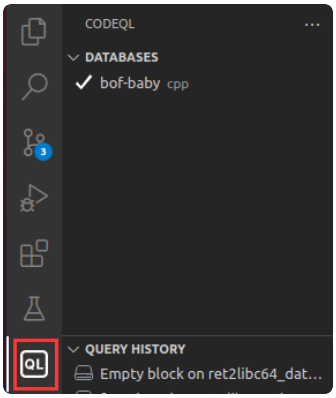
即，创建数据库的语法即是：

`codeql database create <输出数据库路径> --source-root=<目标源代码所在路径> --language=<目标源代码语言> --command="<编译命令>"`

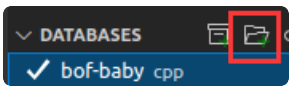
可以看到如下的输出，即建立好了数据库：

```
Initializing database at /home/ssec2022/Desktop/ql_databases/bof-baby.
Running build command: [gcc, /home/ssec2022/ssec22spring-stu/hw-01/01_bof_baby/bof-baby.c, -o, /home/ssec2022/
ssec22spring-stu/hw-01/01_bof_baby/bof-baby, -fno-stack-protector, -fno-pie, -no-pie, -mpreferred-stack-bounda
ry=4]
Finalizing database at /home/ssec2022/Desktop/ql_databases/bof-baby.
Successfully created database at /home/ssec2022/Desktop/ql_databases/bof-baby.
```

在 VSCode 的 QL Extention 这里：

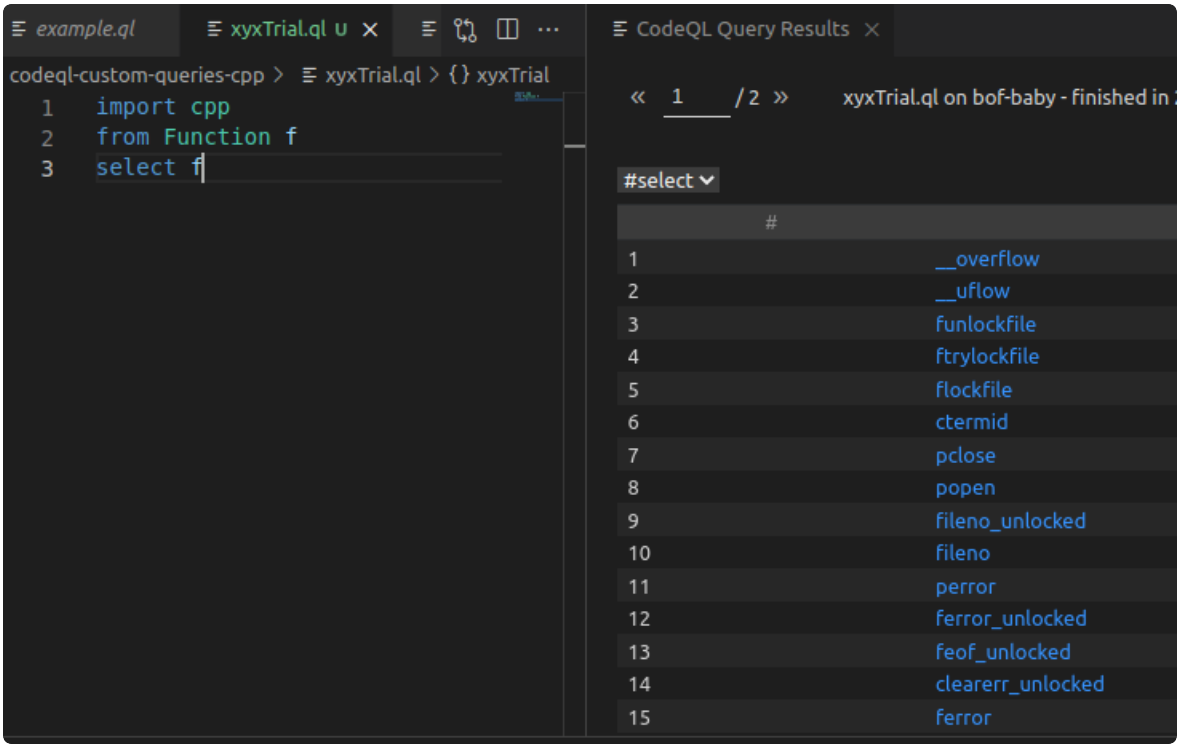


点选从文件夹导入数据库，选择刚刚导入的文件夹，可以看到成功导入；左边白色的勾说明我们当前的 query 正在该 database 中进行。



CodeQL for C and C++

为了具体地应用到某个代码工程，我们需要引入一些能够处理相应语言特性的库。我们的工程是 C 语言的，因此在这里我们 `import cpp` 。使用下面的 query，我们可以找到这个工程中的所有 `Function` ，包括所调用的库中的函数：



这里的 `Function` 类定义于我们 import 的 `cpp`，即 `cpp.qll` 。点开可以看到里面引用了许多 class。我们可以在 CodeQL for C and C++ 里面详细学习摘录一下（暂时只看 C 语言相关的了）：

- 常用的 `Declarations` 的子集：
  - `GlobalVariable, LocalVariable, Function`
  - Function | CodeQL Library for C / C++
- 常用的 `Stmt` 的子集：
  - `BlockStmt, ExprStmt, IfStmt`
- 常用的 `Expr` 的子集：
  - `FunctionCall`

下面就开始尝试具体玩一玩啦！

1.2 尝试找一找 FSB 漏洞

根据实验指导的提示，只要搜到 printf 只有一个参数而且参数非常量即可。但是感觉有多个参数也有可能有 FSB 漏洞；因此我们的思路是，**找第一个参数不是字符串字面量的调用 `printf` 的 FunctionCall**。

我们用这样的代码来构造数据库：

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("You can type exactly 256 charecters ...\n");
    char buffer[256];
    read(STDIN_FILENO, buffer, 256);
    printf(buffer);
    printf("\ndone\n");
    const char *msg = "This is a const string.\n";
    printf(msg);
    return 0;
}
```

这里包含 4 个 `printf` ；理论上来说，只有第 9 行的 `printf` 是有风险的；但是如果检查字符串字面量的话，第 12 行也会被检查出来（实际上，用 gcc 编译这段代码的时候第 12 行就会报 warning）。

我们编写这样的查询代码（含义已经在本节开头说明了）：

```
import cpp
from FunctionCall fc
where fc.getTarget().hasName("printf") and not fc.getArgument(0)
instanceof StringLiteral
select fc, fc.getArgument(0), fc.getArgument(0).getType()
```

运行看看结果：

#	fc	[1]	[2]
1	call to printf	buffer	char[256]
2	call to printf	msg	const char *

和我们预想的一样！这里是存在 false positive 的；在 1.5.1 中，我们学习了污点分析的方式进行优化；即我们从单纯的语法层面上的上下文无关的检查跨越到了语义层面上的检查。

1.3 尝试找一找 BOF 漏洞

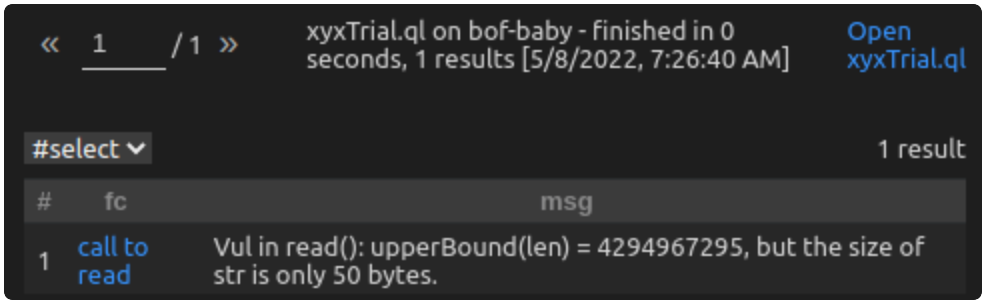
根据实验指导的提示和对源码的分析，我们主要识别这样两种情况：

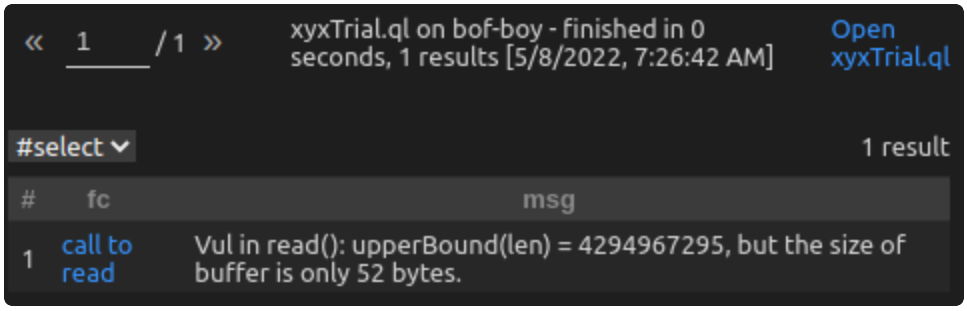
- 对于 `read()`，其第二个参数，即接收读入的数组定义时的字节数小于第三个参数，即读取字符个数的最大可能值。这种情况我们用 predicate `isOverRead()` 识别，核心的判断在如下代码的 5~7 行；
- 对于 `strcpy()`，其第一个参数，即接收复制的数组定义时的字节数小于第二个参数，即源数组定义时的字节数。这种情况我们用 predicate `isOverCopied()` 识别，核心的判断在如下代码的 16~18 行；

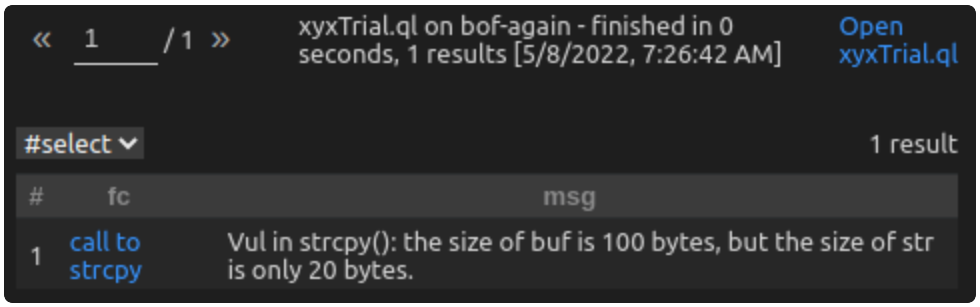
SQL | 复制代码

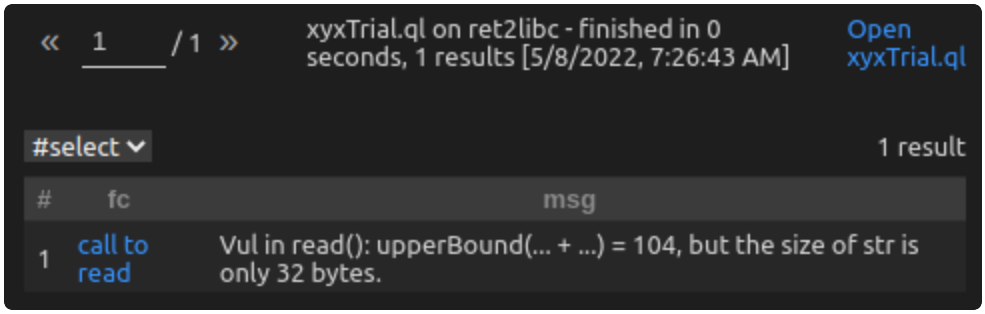
```
import cpp
import semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis

string isOverRead(FunctionCall fc) {
  fc.getTarget().getName() = "read"
  and
  fc.getArgument(1).getType().(ArrayType).getByteSize() <
upperBound(fc.getArgument(2))
  and
  result = "Vul in read(): upperBound(" + fc.getArgument(2).toString()
        + ") = " + upperBound(fc.getArgument(2))
        + ", but the size of " + fc.getArgument(1).toString() + " is only
"
        + fc.getArgument(1).getType().(ArrayType).getByteSize() + "
bytes."
}

string isOverCopied(FunctionCall fc) {
  fc.getTarget().getName() = "strcpy"
  and
  fc.getArgument(0).getType().(ArrayType).getByteSize()
        < fc.getArgument(1).getType().(ArrayType).getByteSize()
  and
  result = "Vul in strcpy(): the size of " + fc.getArgument(1).toString()
+ " is "
        + fc.getArgument(1).getType().(ArrayType).getByteSize()
        + " bytes, but the size of "
        + fc.getArgument(0).toString() + " is only "
        + fc.getArgument(0).getType().(ArrayType).getByteSize() + "
bytes."
}

from FunctionCall fc, string msg
where msg = isOverRead(fc) or msg = isOverCopied(fc)
select fc, msg
```

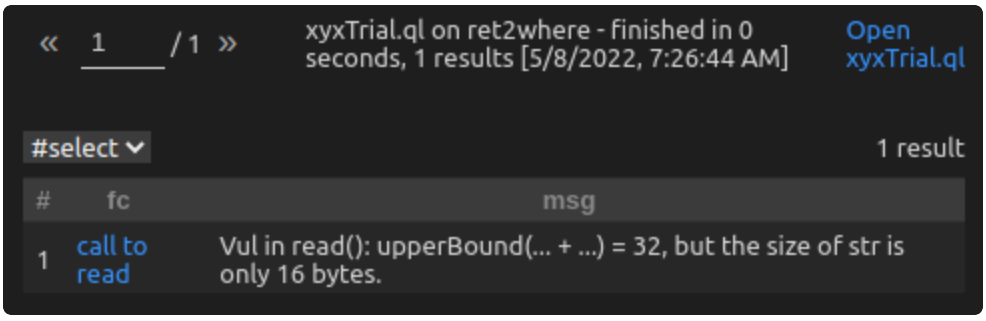
对 HW 1 和 HW 2 中的 5 个相关代码建立数据库，一并查询得到如下结果。可以看到，这份查询可以发现这 5 个代码中的 BOF 漏洞：











1.4 学习数据流的使用

略微调整一下顺序~先学一下 workshop 然后再看范例，不然好像看不太懂QWQ

这里记录一些看 Workshop 2 – Finding security vulnerabilities in C/C++ with CodeQL 的笔记

Workshop 的文件在 这里

污点分析 (Taint Analysis) 可以抽象成一个三元组 `<sources, sinks, sanitizers>` 。 `sources` 就是污染源，以 use-after-free 为例， `sources` 就是将一个指针 free 掉，这时这个指针变量保存的值就不再是一个安全的值了，再对它解引用就会引发安全问题；而 `sinks` 就是危险操作，例如这里就是在 free 之后再使用或者解引用它； `sanitizers` 就是无害化处理，比如在这里就是将这个指针赋值为 `NULL` 。

我们首先找到 use-after-free 的 sources，用一个 `isSource` predicate 来找到它：

SQL 复制代码

```
predicate isSource(Expr arg) {
  exists(FunctionCall fc |
         call.getTarget().hasGlobalOrStdName("free")
         and call.getArgument(0) = arg
        )
}
```

CodeQL 提供了数据流分析的能力，需要 `import semmle.code.cpp.dataflow.DataFlow` 。 `DataFlow::node` 是数据流分析中的结点，即任何可能有值的语义单元。它与 `Expr` 之类的东西不一样；前者是语义单元而后者是语法单元，可以通过 `asExpr()` 等 predicate 将其转化回它对应的表达式，即：

SQL 复制代码

```
predicate isSource(DataFlow::node argNode) {
  exists(FunctionCall fc |
         call.getTarget().hasGlobalOrStdName("free")
         and call.getArgument(0) = argNode.asExpr()
        )
}
```

在我们刚刚谈到的 use-after-free 的例子中，我们希望追踪 `free` 之后的变量，这才是真正需要追踪的污点。这种情况 (data flowing *out of* an expression)，我们使用 `asDefiningArgument()` 而不是 `asExpr()` ：

```
predicate isSource(DataFlow::Node argNode) {
  exists(FunctionCall fc |
         fc.getTarget().hasGlobalOrStdName("free")
         and fc.getArgument(0) = argNode.asDefiningArgument()
        )
}

```

事实上，存在 `asDefiningArgument()` 和 `asExpr()` 的区分是因为存在 pass by reference 和 pass by value 的区分。对于 C 语言来说，传递指针虽然在实现上也是 pass by value 的，但是从数据流的分析上仍然可以认为是 pass by reference。

下面我们希望找到 sink，即可能出现的危险操作。我们找到所有对指针解引用的语句：

```
predicate isSink(DataFlow::Node sink) {
  dereferenced(sink.asExpr())
}
```

这里的 `dereferenced()` 是 C++ 库里给的一个 predicate，考虑了所有可能的解引用情形。

下一步就是将 `source` 和 `sink` 连接起来！ `import DataFlow::PathGraph` 给我们提供了这种能力；因为数据流分析实际上就是将程序看成结点（有值的语义单元）和边（传递值的操作）组成的图。

为了使用这种能力，我们需要做一些配置：

SQL | 复制代码

```sql
class Config extends DataFlow::Configuration {
  Config() { this = "Just to happy the compiler~ " }

  override predicate isSource(DataFlow::Node argNode) {
    exists(FunctionCall fc |
           fc.getTarget().hasGlobalOrStdName("free")
           and fc.getArgument(0) = argNode.asDefiningArgument()
          )
  }

  override predicate isSink(DataFlow::Node sink) {
    dereferenced(sink.asExpr())
  }
}

from Config config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink, source
```

这种分析并不会捕获那些被 `free` 后被赋值为 0 的指针，因为它追踪的是语义的值而不仅是变量本身。这和污点分析存在一定不同。

数据流分析还提供了 `barrier` 的功能，也即污点分析中的 `sanitizer` ；Workshop 中并没有具体介绍，但这是减少 false positive 的方法之一。

Workshop 中还提到了 CodeQL 也支持 `TraintTracking` 即污点分析，数据流分析和污点分析的区别是，数据流分析跟踪的是确切的值，而污点分析将使用这个值进行运算的情况也同样视为污点；因此如果涉及到使用污点进行计算或者字符串拼接等操作的时候，适合使用污点分析。污点分析更加通用。在下一节中，我们可以看到污点分析的具体例子QWQ

> Workshop 中还提到了 Metadata 的相关内容。
>
> 在 AST Viewer 里面可以看到 AST，从中可以了解到源代码各个语法单元的类型
>
> poke around the sources~
>
> I don't know off the top of my head and I will not pretend to

1.5 学习一些高级的 Query 范例

从 这里 学习几个范例！结果的话就只看第一个的了）

1.5.1 CWE-134 FSB

（省略了 metadata 和 import）

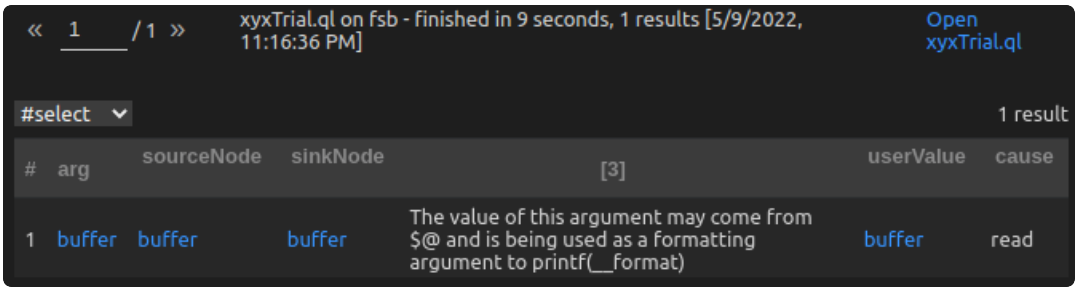
SQL | 复制代码

```
class Configuration extends TaintTrackingConfiguration {
  override predicate isSink(Element tainted) {
    exists(PrintfLikeFunction printf |
printf.outermostWrapperFunctionCall(tainted, _))
  }
}

from
  PrintfLikeFunction printf, Expr arg, PathNode sourceNode, PathNode
sinkNode,
  string printfFunction, Expr userValue, string cause
where
  printf.outermostWrapperFunctionCall(arg, printfFunction) and
  taintedWithPath(userValue, arg, sourceNode, sinkNode) and
  isUserInput(userValue, cause)
select arg, sourceNode, sinkNode,
  "The value of this argument may come from $@ and is being used as a
formatting argument to " +
    printfFunction, userValue, cause
```

这段代码使用了 `semmle.code.cpp.security.TaintTracking`，是之前提到（但是并未展开）的污点分析。查看 TaintedWithPath | Library 可以知道，extend `TaintTrackingConfiguration` 之后可以使用 `taintedWithPath` predicate 来实现污点分析。

具体而言，这里判断的语句中 `printf.outermostWrapperFunctionCall(arg, _)` 就是选出 `PrintfLikeFunction` 调用的 `arg`，这是一个可能的 taint（这里的 `_` 是 don't care expressions, 参见 这里）；然后 `isUserInput(userValue, _)` 就是选出所有用户输入的 `userValue`，这是可能的 source； `Configuration extends TaintTrackingConfiguration` 中的 `isSink` 定义了 sink，即在 `PrintfLikeFunction` 中调用 taint。（逻辑上的）最后，使用 predicate `taintedWithPath` 来找到 FSB 漏洞。

在我们 1.2 中编写的文件中做查询，可以发现它只找到了一个漏洞，即我们讨论中唯一可能的漏洞：




```c
int main()
{
    printf("You can type exactly 256 charecters ...\n");
    char buffer[256];
    read(STDIN_FILENO, buffer, 256);
    printf(buffer);
    printf("\ndone\n");
    const char *msg = "This is a const string.\n";
    printf(msg);
    return 0;
}
```

1.5.2 CWE-835 Infinite Loop

（省略了 metadata, import 和一些注释）

这段 query 检查是否存在死循环；之前在用 CLion 之类的 IDE 的时候就很好奇这种东西的检查方式：

```
SQL | 复制代码
```

```
import cpp
import semmle.code.cpp.controlflow.BasicBlocks
private import semmle.code.cpp.rangeanalysis.PointlessComparison
import semmle.code.cpp.controlflow.internal.ConstantExprs

predicate impossibleEdge(ComparisonOperation cmp, boolean value,
BasicBlock src, BasicBlock dst) {
  cmp = src.getEnd() and
  reachablePointlessComparison(cmp, _, _, value, _) and
  if value = true then dst = src.getAFalseSuccessor() else dst =
src.getATrueSuccessor()
}

BasicBlock enhancedSucc(BasicBlock bb) {
  result = bb.getASuccessor() and not impossibleEdge(_, _, bb, result)
}

predicate impossibleEdgeCausesNonTermination(ComparisonOperation cmp,
boolean value) {
  exists(BasicBlock src |
    impossibleEdge(cmp, value, src, _) and
    src.getASuccessor+() instanceof ExitBasicBlock and
    not enhancedSucc+(src) instanceof ExitBasicBlock and
    exists(EntryBasicBlock entry | src = enhancedSucc+(entry))
  )
}

from ComparisonOperation cmp, boolean value
where impossibleEdgeCausesNonTermination(cmp, value)
select cmp, "Function exit is unreachable because this condition is
always " + value.toString() + "."
```

这个 query 定义和使用了 **`impossibleEdgeCausesNonTermination`** 这个 predicate，它接受一个比较操作 `cmp` 以及一个布尔值 `value`，它 holds 当这个 `cmp` 的值一直是 `value`。在其中，它寻找是否存在这样一个基本块 `src`，它满足：

1. `impossibleEdge(cmp, value, src, _)`，即因为 `cmp` 是 `src` 的出口，且由于 `cmp` 的值永远是 `value`，因此该 `value` 的相反情况引发的后继都是不可达的；

具体而言，在 `impossibleEdge(ComparisonOperation cmp, boolean value, BasicBlock src, BasicBlock dst)` 这个 predicate 内：

a. 它调用 `getEnd()` 来判断当前 `cmp` 是否是 `src` 这个基本块的最后一个控制流结点，亦即是否是这个基本块的出口；

b. 然后调用 `reachablePointlessComparison(cmp, _, _, value, _)` 来判断这个 `cmp` 是否是一个可达的无意义的比较，即取值永远是 `value` ；

`reachablePointlessComparison` 是 `rangeanalysis.PointlessComparison` 中提供的一个 predicate，我们可以进一步查看其实现：

```
predicate reachablePointlessComparison(
  ComparisonOperation cmp, float left, float right, boolean value, SmallSide ss
) {
  pointlessComparison(cmp, left, right, value, ss) and
  // Reachable according to control flow analysis.
  reachable(cmp) and
  // Reachable according to range analysis.
  not exprWithEmptyRange(cmp.getAChild+())
}
```

```
predicate pointlessComparison(
  ComparisonOperation cmp, float left, float right, boolean value, SmallSide ss
) {
  alwaysLT(cmp.(LTExpr), left, right, ss) and value = true
  or
  alwaysLE(cmp.(LEExpr), left, right, ss) and value = true
  or
  alwaysGT(cmp.(GTExpr), left, right, ss) and value = true
  or
```

```
private predicate alwaysLT(ComparisonOperation cmp, float left, float right, SmallSide ss) {
  ss = LeftIsSmaller() and
  left = upperBoundFC(cmp.getLeftOperand()) and
  right = lowerBoundFC(cmp.getRightOperand()) and
  left < right
}
```

可以看到，它调用了 `pointlessComparison` ，而 `pointlessComparison` 调用了一系列 `alwaysXX` ，而每一个 `alwaysXX` 会找到 `cmp` 两边的 `upperBound` 或 `lowerBound` 。因此 `reachablePointlessComparison(cmp, _, _, value, _)` 事实上会自动帮我们找到上下界并判断可能的无意义比较。

c. 然后根据这个 `value` 的真值来判断相反的 `dst` ，即出口的基本块。

概括而言，这个 predicate holds 如果存在一个比较 `cmp` 是基本块 `src` 到 `dst` 的条件，但是由于 `cmp` 的值始终是相反的，因此 `dst` 始终不可达。

2. `src.getASuccessor+() instanceof ExitBasicBlock and not enhancedSucc+(sr`

`c) instanceof ExitBasicBlock`，即 `src` 存在后继到该函数的出口，但进一步分析得知所有的后继都不可达；

> 具体而言，`enhancedSucc(BasicBlock bb)` 其实就是用 `bb.getASuccessor() and not impossibleEdge(_, _, bb, result)` 获取基本块的**并非不可达的**后继；这里的 `getASuccessor+()` 中的 `+` 是递归查询的意思，也就是说 `src.getASuccessor+()` 无视可达性取到 `src` 的所有后继，其中存在后继可以到达 `ExitBasicBlock`，即退出函数的基本块；而 `enhancedSucc+(src)` 考虑可达性取到 `src` 的所有**可达**后继，其中不存在后继可以到达 `ExitBasicBlock`；也就是说，因为 `impossibleEdge` 的原因，这个函数永远不可能停止；这样就能找到一个死循环了。

3. `exists(EntryBasicBlock entry | src = enhancedSucc+(entry))`，即当前 `src` 是从 `entry` 可达的。

   这样，这个 query 就能分析出死循环，并能一定程度上避免 false positive。

1.5.3 其他范例

还有一些（能看懂的）例子，有空的时候可以再看看QWQ

- 119 OverflowBuffer
- 129 数组下标检查
- 131 字符串结束符
- 468 数组偏移
- 676 Function Overflow

2 动态分析 | HW-A: Fuzzing libxml2

> 这一部分是额外完成的方向~并没有做的很完整

2.1 什么是 Fuzzing

测试就是尝试在运行中找错误；容易观察的错误是 crash，不过逻辑错误也需要考虑。Fuzzing 就是做随机的测试；当然要生成一个合法的随机初始输入，然后做若干突变。需要考虑 coverage。

但是遇到 bug 有的时候不会立刻 crash，甚至不会 crash，怎么办呢?

Sanitizers 的思路是基于编译器在编译期做一些插桩 (instrumentation)，这样当检测到一些 bug 的时候就发信号或者直接 crash。

例如，Address Sanitizer (ASAN) 可以检测 heap, stack 和 global 的越界访问、free 后使用、scope 外使用等问题，通过 clang 编译指令 `-fsanitize=address` 启用；一般会使程序耗时变成原来的 2 倍。

Undefined Behavior Sanitizer (UBSAN) 可以检测除以 0、符号数溢出、对未对齐的指针或空指针解引用等未定义行为，可以通过编译指令 `-fsanitize=undefined` 启用。

Memory Sanitizer (MSAN) 可以检测未初始化的读取（这种读取有可能会泄露一些之前栈上的信息，导致泄露 offset 等内容），可以通过 `-fsanitize=memory` 启用；一般会使程序耗时变为原来的 3 倍。

Thread Sanitizer (TSAN) 可以检测 data races，可以通过 `-fsanitize=thread` 启用；会慢 5~15 倍。

AFL 是目前广泛使用的 fuzzer；它在编译器做插桩从而来跟踪 coverage；通过一些策略做突变。coverage 发生变化时，触发新覆盖的测试用例将保存为测试用例队列的一部分。

2.2 配置运行 AFL++

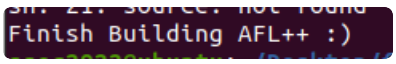
2.2.1 配置

1. Setup AFL++：

```
sh -c "$(wget https://gitee.com/ret2happy/ssec22_fuzzing_script/raw/master/setup_aflpp.sh -O -)"
```

setup_aflpp.sh Shell 复制代码

```shell
#! /bin/bash

# You could replace it with the origin AFL++, i.e.,
https://github.com/AFLplusplus/AFLplusplus.git
git clone https://gitee.com/ret2happy/AFLplusplus.git

# Install dependency
sudo apt update
sudo apt install -y llvm-10 clang-10 llvm-10-dev
sudo apt-get install -y build-essential python3-dev flex bison
libglib2.0-dev libpixman-1-dev python3-setuptools

sudo update-alternatives --install /usr/bin/llvm-config llvm-config
/usr/bin/llvm-config-10 10


# build it
cd AFLplusplus
make afl-fuzz
make afl-showmap
make llvm
sudo make install
cd -
source ~/.bashrc

echo "Finish Building AFL++ :)"
```

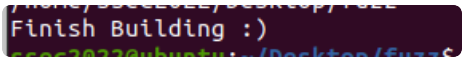


2. Setup libxml2：

```
sh -c "$(wget https://gitee.com/ret2happy/ssec22_fuzzing_script/raw/master/setup_libxml2.sh -O -)"
```

Shell 复制代码

```shell
#! /bin/bash

# fetch source code
git clone https://gitee.com/ret2happy/libxml2.git libxml2-2.9.4

# build libxml2, make sure afl-clang-fast is available
cd libxml2-2.9.4

# install dependency
sudo apt install automake-1.15

CC=afl-clang-fast CXX=afl-clang-fast++ CFLAGS="-fsanitize=address" CXXFLAGS="-fsanitize=address" LDFLAGS="-fsanitize=address" ./configure --disable-shared --without-debug --without-ftp --without-http --without-legacy --without-python LIBS='-ldl'

make -j `nproc`
cd -

echo "Finish Building :)"
```



2.2.2 运行

1. 下载 AFL++ 预设的 xml dictionary：

```
wget https://raw.githubusercontent.com/AFLplusplus/AFLplusplus/stable/dictionaries/xml.dict
```

它定义了一些基本的格式，从而方便 AFL++ 生成相关的输入

2. 创建初始语料库：

```
git clone https://gitee.com/ret2happy/libxml2_sample.git corpus
```

3. 创建主 fuzzer 来运行：

```
sudo bash -c "echo core >/proc/sys/kernel/core_pattern"
```

```
afl-fuzz -M master -m none -x xml.dict -i ./corpus -o output -- ./libxml2-2.9.4/xmllint --valid @@
```

实验指导给出了这样的解释：

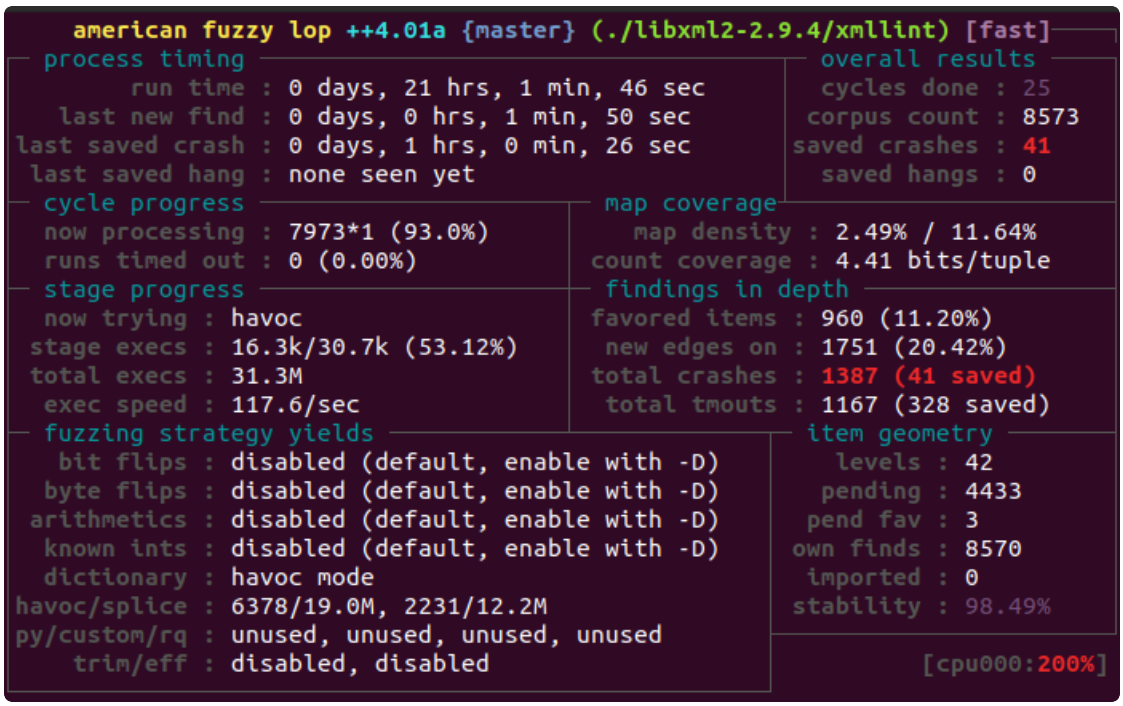
其中

- `-M master` 表示将当前fuzzer尽可能的被指定为master fuzzer。master fuzzer会对各个slave fuzzer进行语料的管理与调度。
- `-m none` ASan开启时需要大量虚拟内存，此处让fuzzer不对内存进行限制。
- `-x xml.dict` 指向之前获取的xml.dict，用于提供变异时的候选词。
- `-i /path/to/corpus` 指定了初始的输入语料库
- `-o output` 指定fuzzer的输出目录，其中包含了已变异待执行的testcase，fuzzer目前的状态、寻找到的crash等各类信息。
- `xmllint --valid @@` 指定了被fuzz程序的命令执行方式，其中 `@@` 表示占位的文件名，在运行时AFL会将输入文件的文件名替换 `@@` ，以便被测程序能够读取真实的输入。

> tips: 对于master fuzzer，我们可以不使用 `-D` 的参数，而通过仅在slave fuzzer中加入 `-D` 参数以增加各个fuzzer策略的多样性。这对于寻找crash有很大帮助。

2.2.3 AFL++ 运行截图及释义

刚开始我并不清楚这个东西不会自动结束，而是会一直运行下去，所以我放着他跑了二十多个小时……效果如下：



我们来简单查阅一下资料，分析上面各部分的含义：

- `process timing`，即运行时间，记录总运行时间、距离上次发现问题的时间、距离上一次保存的 crash 和挂起的时间；
- `cycle progress`，当前的进度以及超时的个数和比例
- `stage progress`，当前正在执行的 fuzz 的变异策略以及进度和速度
- `fuzzing strategy yields`，变异的模式；包括：
  - `bit flips`，bit 翻转
  - `byte flips`，byte 翻转
  - `arithmetics`，一些算术的加减
  - `known ints`，尝试一些特殊的内容，例如边界情况等
  - `dictionary`，根据生成或用户提供的字典
  - `havoc`，结合上述方法进行大量变异
  - `splice`，拼接文件形成新的文件
- `overall results`，总体的结果，包含完成的循环次数、语料库计数、保存的 crash 和挂起的数目；
- `map coverage`，即对分支的覆盖率；
- `findings in depth`，路径和发现问题的信息；
- `item geometry`，包括测试等级、待测数量、待测项目中比较可能的数量、找到的个数、导入的个数、以及被测试程序的稳定性

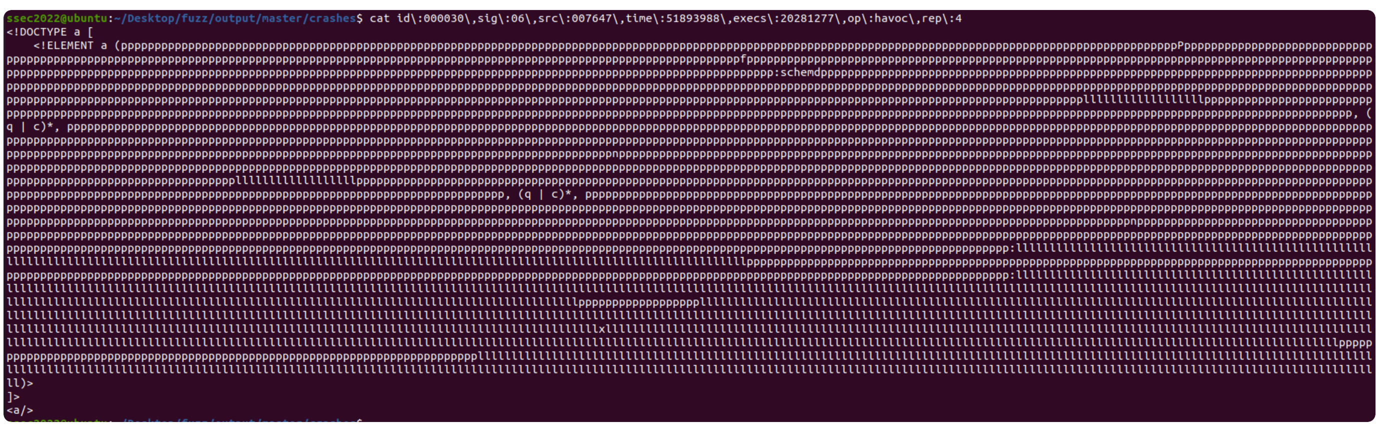
2.3 触发 AddressSanitizer 崩溃的样例和截图

可以看到，这里有若干 crash 的样例记录：

```
ssec2022@ubuntu:~/Desktop/fuzz/output/master/crashes$ ls
id:000000,sig:06,src:000001,time:117394,execs:43894,op:havoc,rep:4
id:000001,sig:06,src:000808,time:1828257,execs:738311,op:havoc,rep:4
id:000002,sig:06,src:001967,time:14335107,execs:1465479,op:havoc,rep:4
id:000003,sig:06,src:000740,time:26112652,execs:7697842,op:havoc,rep:8
id:000004,sig:06,src:000740,time:26115651,execs:7699348,op:havoc,rep:2
id:000005,sig:06,src:000776,time:26118097,execs:7700695,op:havoc,rep:2
id:000006,sig:06,src:005118,time:26748056,execs:8039074,op:havoc,rep:2
id:000007,sig:06,src:005214,time:27046823,execs:8187320,op:havoc,rep:4
id:000008,sig:06,src:000740,time:36821248,execs:13156467,op:havoc,rep:4
id:000009,sig:06,src:000740,time:36821340,execs:13156516,op:havoc,rep:2
id:000010,sig:06,src:002624,time:36861415,execs:13178319,op:havoc,rep:2
id:000011,sig:06,src:002624,time:36867577,execs:13181464,op:havoc,rep:4
id:000012,sig:06,src:002624,time:36880245,execs:13188277,op:havoc,rep:4
id:000013,sig:06,src:006630,time:37524686,execs:13511367,op:havoc,rep:2
id:000014,sig:06,src:006630,time:37525398,execs:13511704,op:havoc,rep:2
id:000015,sig:06,src:006630,time:37527748,execs:13512828,op:havoc,rep:2
id:000016,sig:06,src:006630,time:37529431,execs:13513629,op:havoc,rep:2
id:000017,sig:06,src:006630,time:37531362,execs:13514631,op:havoc,rep:2
id:000018,sig:06,src:006630,time:37532417,execs:13515180,op:havoc,rep:4
id:000019,sig:06,src:006630,time:37536446,execs:13517228,op:havoc,rep:2
id:000020,sig:06,src:006630,time:37549020,execs:13523612,op:havoc,rep:2
id:000021,sig:06,src:006630,time:37556876,execs:13527563,op:havoc,rep:16
id:000022,sig:06,src:006640,time:37635333,execs:13568511,op:havoc,rep:4
```

查看其中两例：

```
ssec2022@ubuntu:~/Desktop/fuzz/output/master/crashes$ hexdump -C id\:000000\,sig\:06\,src\:000001\,time\:117394\,execs\:43894\,op\:havoc\,rep\:4
00000000  3c 21 44 4f 43 54 59 50  45 20 61 20 5b 0a 20 20  |<!DOCTYPE a [.  |
00000010  20 20 3c 21 45 4c 45 4d  45 4e 54 20 61 20 28 70  |  <!ELEMENT a (p|
00000020  70 70 70 70 70 70 70 70  70 70 70 70 70 70 70 70  |pppppppppppppppp|
*
000006f0  70 70 70 70 70 70 6c 6c  6c 6c 6c 6c 6c 6c 6c 6c  |ppppppllllllllll|
00000700  6c 6c 6c 6c 6c 6c 6c 6c  6c 6c 6c 6c 6c 6c 6c 6c  |llllllllllllllll|
00000710  70 70 70 70 70 70 70 70  70 70 70 70 70 70 70 70  |pppppppppppppppp|
*
00000dd0  70 70 3a 6c 6c 6c 6c 6c  6c 6c 6c 6c 6c 6c 6c 6c  |pp:lllllllllllll|
00000de0  6c 6c 6c 6c 6c 6c 6c 6c  6c 6c 6c 6c 6c 6c 6c 6c  |llllllllllllllll|
*
000013a0  6c 6c 6c 6c 6c 6c 6c 6c  6c 29 3e 0a 5d 3e 0a 3c  |lllllllll)>.]>.<|
000013b0  61 2f 3e 0a                                       |a/>.|
000013b4
```



尝试复现 crash：

```
ssec2022@ubuntu:~/Desktop/fuzz$ ./libxml2-2.9.4/xmllint --valid output/master/crashes/id\:000000\,sig\:06\,src\:000001\,time\:117394\,execs\:43894\,op\:havoc\,rep\:4
=================================================================
==19500==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff0da6fe88 at pc 0x00000048037b bp 0x7fff0da6ea30 sp 0x7fff0da6e1d8
WRITE of size 1495 at 0x7fff0da6fe88 thread T0
    #0 0x48037a in strcat (/home/ssec2022/Desktop/fuzz/libxml2-2.9.4/xmllint+0x48037a)
    #1 0x5c7c6a in xmlSnprintfElementContent /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/valid.c:1279:3
    #2 0x5e5ce6 in xmlValidateElementContent /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/valid.c:5445:6
    #3 0x5e5ce6 in xmlValidateOneElement /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/valid.c:6152:12
    #4 0x84c7f5 in xmlSAX2EndElementNs /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/SAX2.c:2467:24
    #5 0x545bd4 in xmlParseElement /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/parser.c:10203:3
    #6 0x555d8f in xmlParseDocument /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/parser.c:10953:2
    #7 0x573fba in xmlDoRead /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/parser.c:15432:5
    #8 0x573fba in xmlCtxtReadFile /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/parser.c:15677:13
    #9 0x4caf07 in parseAndPrintFile /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/xmllint.c:2391:9
    #10 0x4c7b3d in main /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/xmllint.c:3767:7
    #11 0x7f732c33b0b2 in __libc_start_main /build/glibc-sMfBJT/glibc-2.31/csu/../csu/libc-start.c:308:16
    #12 0x41c58d in _start (/home/ssec2022/Desktop/fuzz/libxml2-2.9.4/xmllint+0x41c58d)

Address 0x7fff0da6fe88 is located in stack of thread T0 at offset 5128 in frame
    #0 0x5e1d9f in xmlValidateOneElement /home/ssec2022/Desktop/fuzz/libxml2-2.9.4/valid.c:5943

  This frame has 5 object(s):
    [32, 82) 'fn.i' (line 5288)
    [128, 5128) 'expr.i' (line 5441)
    [5392, 10392) 'list.i' (line 5442) <== Memory access at offset 5128 partially underflows this variable
    [10656, 10660) 'extsubset' (line 5950)
    [10672, 10722) 'fn' (line 6063)
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/home/ssec2022/Desktop/fuzz/libxml2-2.9.4/xmllint+0x48037a) in strcat
Shadow bytes around the buggy address:
  0x100061b45f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b45f90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b45fa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b45fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b45fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x100061b45fd0: 00[f2]f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2
  0x100061b45fe0: f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2
  0x100061b45ff0: f2 f2 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b46000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b46010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100061b46020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==19500==ABORTING
```

可以看到，虽然发生了栈溢出，但是被 `AddressSanitizer: stack-buffer-overflow on address 0x7fff0da6fe88 at pc 0x00000048037b bp 0x7fff0da6ea30 sp 0x7fff0da6e1d8` 检测到了，所以程序被 abort 了。