SyntaxFlow Cookbook - 高级静态代码分析

本文将会描述 SyntaxFlow 的基础定义和使用案例以及一些高级用法。在阅读本文中，请确保你已经正常对 Yaklang 的命令行工具有一些理解，可以通过 `github.com/yaklang/syntaxflow-zero-to-hero` ，来学习基础的操作。

github.com/yaklang/syntaxflow-zero-to-hero

README

除了安装最基础的执行环境之外，你还需要事先对 "程序" 有一些基本认知，当然，如果你有 PHP / JS / Java 的一点点基础是最好的，方便我们在后面

0x01 从零开始

假设现在的你现在并不知道 SSA 是什么，也不知道 SyntaxFlow 的基础语法，那么我们就从真的 "Zero" 开始吧！

0x01.1 Clone 本项目到本地

```
git clone https://github.com/yaklang/syntaxflow-zero-to-hero
cd syntaxflow-zero-to-hero/lesson-1-hello-world/
```

0x01.2 编译 Hello World 程序

当然，SyntaxFlow 并不能被证明是图灵完备的，也并不适合像其他语言一样 `Println("Hello World")` 。所以 Yaklang SyntaxFlow 的 Hello World 要相对特殊很多。

我们要先把要审计的代码编译成特定的 SSA 格式，才能开始执行 SyntaxFlow。编译命令非常简单，在确保你 clone 到本地之后，进入 `lession-1-hello-world` 仓库，执行如下命令：

```
yak ssa -t . --program lesson1
```

> 注意，你设置 --program lesson1 之后，在后续使用中，都需要使用到这个程序名称，分析其才知道你要分析到底是哪个程序。

执行完应该会看到

```
→ lesson-1-hello-world git:(main) ✗ yak ssa -t . --program lesson1
[INFO] 2024-06-25 22:57:36 [ssacli:131] start to compile file: .
[INFO] 2024-06-25 22:57:36 [ssacli:147] compile save to database with program name: lesson1
[INFO] 2024-06-25 22:57:36 [ssa:42] init ssa database: /Users/v1ll4n/yakit-projects/default-ya
```

当你完成了 `lesson-1` 的训练之后，接下来需要学习的是 SyntaxFlow 规则文件的编写。在后续的使用中，我们已所有的审计内容和输出几乎都依赖 SyntaxFlow 规则文件，这类文件通常以 `.sf` 结尾

语言介绍

SyntaxFlow 是一个 Yak Project 研发的一个编译器级声明式的高级程序分析语言。它旨在分析由 Yaklang SSA 编译器编译后存储于数据库中的程序代码（IrCode）。这种语言专门设计用于解决静态分析中的各种挑战，例如：精确搜索、模糊搜索和特定方法搜索，以及数据流分析、控制流处理等。SyntaxFlow 提供了代码容错功能，能够针对不完整的代码进行审计。

语言的一大特点是能够处理包括 Java、PHP、JavaScript 等多种编程语言的代码，并支持对象编程中的方法跟踪和上下文敏感分析。SyntaxFlow 还能编译这些语言为 SSA 形式，并支持基于 SSA 的查询，例如追踪变量的定义和使用，进而帮助开发者理解代码中的数据流和潜在的安全漏洞。

为使用 SyntaxFlow，开发者首先需要设置 Yaklang 环境，编译目标代码为 SSA 形式，然后通过编写和执行 SyntaxFlow 规则来进行代码审计。这些规则利用语言特定的查询表达式来定位和分析代码中的潜在问题，例如寻找和追踪代码中的命令执行操作。

规则文件结构

在使用 SyntaxFlow 技术过程中，理解规则文件（.sf 文件）的结构至关重要。这些文件包含特定的语句和表达式，用于定义如何在代码中搜索特定模式和行为。本章节将通过几个实际案例来展示规则文件的编写方法，并解释每个组成部分的功能和作用。

> 在后面的叙述中，我们通常说 SyntaxFlow Rule 可能会简述成 SF 文件或者，SF 规则等描述。

1. 通用的规则文件架构

SF 规则文件的结构通常遵循以下模式：

- **描述性说明** (`desc`): 提供规则的概览和目的。
- **审计语句**：定义特定的代码模式或行为来捕捉和分析。
  - **过滤器**：通过条件表达式过滤和选择代码的特定部分。
  - **变量命名** (`as`): 用于后续引用的结果命名。
- **条件检查** (`check`): 根据审计语句的结果来断言和输出相应的信息；当然也可以通过 `alert` 来告诉报告生成器需要重点关注的或者有漏洞的变量信息。

在上面的描述中， `desc` 和 `check` 尽管不是必须的，但是我们还是强烈推荐用户在编写规则的时候，输出这两个语句，这可以让你的规则的输出更容易让人理解。在上述所有的内容中，“审计语句”是最核心的。

通过这种结构化的方式，SyntaxFlow 规则文件能够高效地指导开发者识别和解决代码中的潜在问题。每个组件都是构建高效、准确静态分析规则不可或缺的一部分。在撰写规则时，清晰地定义每个部分的作用和逻辑关系，将有助于提高规则的可读性和维护性。

2. 规则文件案例与解读

XXE 漏洞检测规则 (`xxe.sf`)

```
1 desc("Description": 'checking
  setFeature/setXIncludeAware/setExpandEntityReferences in
  DocumentBuilderFactory.newInstance()')
2 DocumentBuilderFactory.newInstance()?{!((.setFeature) ||
  (.setXIncludeAware) || (.setExpandEntityReferences))} as $entry;
3 $entry.*Builder().parse(* #-> as $source);
4
5 check $source then "XXE Attack" else "XXE Safe";
```

解读：

- `desc` 语句描述了规则的目的，即检查是否在 `DocumentBuilderFactory.newInstance()` 方法调用中避免了设置某些可能导致 XXE 漏洞的特性。
- 审计表达式中的 `?{}` 结构用于确保没有调用 `setFeature`、 `setXIncludeAware` 或 `setExpandEntityReferences` 方法。
- `#->` 运算符追踪从 `parse` 方法传入的参数的顶级定义，以识别可能的攻击载体。
- `check` 语句基于 `$source` 的存在与否来判定是否可能存在 XXE 攻击。

URL 请求检测规则 (`url-open-connection.sf`)

```
1 URL(* #-> * as $source).openConnection().getResponseMessage() as $sink;
2
3 check $sink then "Request From URL" else "No Response From URL";
```

解读：

- 此规则追踪 `URL` 对象创建到发起网络请求的完整流程。
- `$source` 表示 URL 的来源，而 `$sink` 表示从该 URL 获得的响应。
- 使用 `check` 语句来确定是否成功从 URL 获取了响应。

本地文件写入检测规则 (`local-file-write.sf`)

```
1 Files.write(, * #-> as $params) as $sink;
2 $params?{.getBytes()} as $source;
3
4 check $source then "Local Files Writer" else "No Files Written";
```

解读：

- 此规则检查 `Files.write` 方法调用，并追踪写入操作中使用的参数。
- 通过检查是否调用了 `getBytes` 方法来确认是否有数据被写入。

编写入门 SyntaxFlow 规则

在学习编写 SyntaxFlow 规则之前，为了方便用户理解使用，我们使用 XXE 这个漏洞来进行教学，用户可以在手动实现对这个漏洞的分析检测过程中，掌握 SyntaxFlow 的编写技术。

1. 准备要审计的代码

我们直接把这段存在 XXE 漏洞的 Java 代码保存为 `XXE.java`，存放在

```
1  package com.vuln.controller;
2
3  import org.springframework.web.bind.annotation.RequestMapping;
4  import org.springframework.web.bind.annotation.RequestParam;
5  import org.springframework.web.bind.annotation.RestController;
6
7  import javax.xml.parsers.DocumentBuilder;
8  import javax.xml.parsers.DocumentBuilderFactory;
9  import java.io.ByteArrayInputStream;
10 import java.io.InputStream;
11
12 @RestController(value = "/xxe")
13 public class XXEController {
14
15     @RequestMapping(value = "/one")
16     public String one(@RequestParam(value = "xml_str") String xmlStr) throws Exception {
17         DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
18         InputStream stream = new ByteArrayInputStream(xmlStr.getBytes("UTF-8"));
19         org.w3c.dom.Document doc = documentBuilder.parse(stream);
20         doc.getDocumentElement().normalize();
21         return "Hello World";
22     }
23 }
```

这段代码位于一个使用 Spring Framework 构建的 Web 应用中，定义了一个处理 XML 数据的控制器 XXEController。控制器中的 one 方法用来处理通过 HTTP 请求传递的 XML 字符串。以下是代码的具体行为和存在的安全问题：

代码解释：

1. `@RestController(value = "/xxe")` 注解定义了一个 RESTful 控制器，其所有请求的基础 URL 是 /xxe
2. `@RequestMapping(value = "/one")` 注解表明，one 方法将处理对 `/xxe/one` 的 HTTP 请求。
3. 方法 one 接收一个名为 xml_str 的请求参数，这个参数通过 @RequestParam 注解获得。这个参数预期包含 XML 格式的数据。
4. `DocumentBuilderFactory.newInstance().newDocumentBuilder()` 创建了一个 `DocumentBuilder` 实例，用于解析 XML 数据。
5. `new ByteArrayInputStream(xmlStr.getBytes("UTF-8"))` 创建了一个 InputStream，它从传入的字符串 xml_str 中读取数据。
6. `documentBuilder.parse(stream)` 解析这个流，尝试构建一个 DOM 树。
7. `doc.getDocumentElement().normalize()` 规范化文档结构，确保 DOM 树的结构正确。

存在的 XXE 漏洞：

这段代码存在 XML 外部实体 (XXE) 漏洞，原因如下：

1. **默认的解析器设置**：DocumentBuilderFactory 默认配置不禁用外部实体的处理。这意味着如果 XML 输入包含对外部实体的引用，解析器将尝试解析这些实体。
2. **安全风险**：攻击者可以利用 XML 输入中的外部实体，引导服务器解析恶意内容。例如，攻击者可能会引入指向敏感文件（如 /etc/passwd）的实体，导致敏感信息泄露。此外，恶意的外部实体还可以用来触发拒绝服务攻击（DoS）等。

编译源码

我们进入 `XXE.java` 所在的目录，直接执行下面代码即可编译：

```
1 yak ssa -t . --program xxe
```

编译完成之后，你就会在输出中看到以下日志，看到 `finished comiling` 则说明编译完成了。

```
1 [INFO] 2024-06-26 11:52:38 [ssacli:132] start to compile file: .
2 [INFO] 2024-06-26 11:52:38 [ssacli:148] compile save to database with program
  name: xxe
3 [INFO] 2024-06-26 11:52:38 [ssa:42] init ssa database: /Users/v1ll4n/yakit-
  projects/default-yakssa.db
4 [INFO] 2024-06-26 11:52:38 [language_parser:46] parse project in fs:
  *filesys.LocalFs, localpath: .
```

```
5 [INFO] 2024-06-26 11:52:38 [language_parser:152] file[XXE.java] is supported
  by language [java], use this language
6 [WARN] 2024-06-26 11:52:38 [visit_package:31] Dependencies Missed: Import
  package [org springframework web bind annotation RequestMapping] but not found
7 ...
8 ...
9 ...
10 [INFO] 2024-06-26 11:52:38 [language_parser:68] compile XXE.java cost:
  194.70225ms
11 [INFO] 2024-06-26 11:52:38 [language_parser:72] program include files: 2 will
  not be as the entry from project
12 [WARN] 2024-06-26 11:52:38 [reducer:51] Compile error: parse file xxe.sf
  error: file[xxe.sf] is not supported by any language builder, skip this file
13 [INFO] 2024-06-26 11:52:38 [ssacli:163] finished compiling..., results: 1
```

2. 编写描述

在大致了解了 SF 规则文件之后，我们可以来学习尝试构建自己的 SF 文件，首先我们创建一个文件 `xxe.sf`，并在文件中写入：

```
1 desc(title: "审计因为未设置 setFeature 等安全策略造成的XXE漏洞")
```

显然，这段代码只是增加一点文件描述，并不会产生实际审计含义，但是我们添加这个会让结果输出的时候，包含对结果的解读信息。因此还是比较有必要的

SPEC：语句 `desc`

```
1 // descriptionStatement will describe the filterExpr with stringLiteral
2 descriptionStatement: Desc ('(' descriptionItems? ')') | ('{'
  descriptionItems? '}');
3 descriptionItems: descriptionItem (',' descriptionItem)*;
4 descriptionItem
5     : stringLiteral
6     | stringLiteral ':' stringLiteral
7     ;
```

在 SyntaxFlow 规则文件中，`desc` 语句用于为规则提供描述性的文本，这有助于理解规则的目的和应用场景。此语句可以包含一条或多条描述项，这些项可以单独列出或配对（键和值）。下面是关于如何编写 `desc` 语句的详细教程，以及如何通过案例来实际应用这些语句。

语法结构

`desc` 语句可以采用以下两种形式之一：

1. 使用圆括号 `()` 封装描述项。
2. 使用花括号 `{}` 封装描述项。

描述项可以是单个字符串字面量，也可以是一对键和值（均为字符串字面量），用冒号 `:` 分隔。

语法定义

- **DescriptionStatement**:
  - \`Desc\` 关键词后跟括号内的描述项。括号可以是圆括号 \`()\` 或花括号 \`{}\`。
- **DescriptionItems**:
  - 一个或多个 \`descriptionItem\`，通过逗号 \`,\` 分隔。
- **DescriptionItem**:
  - 单个字符串字面量，或
  - 一对字符串字面量，形式为 `key: value`。

示例解释

考虑以下 `desc` 语句示例：

```
1 desc(title: "审计因为未设置 setFeature 等安全策略造成的XXE漏洞")
```

这个例子中，`desc` 语句使用圆括号包含了一个键值对描述项：

- **Key** (`title`): 说明描述的类别或主题。
- **Value** (`"审计因为未设置 setFeature 等安全策略造成的XXE漏洞"`): 具体描述规则的目的，即审计由于未设置某些 XML 安全特性（如 `setFeature` ）而可能导致的 XXE 漏洞。

编写教程

编写有效的 \`desc\` 语句时，请遵循以下最佳实践：

1. **明确目的**：确保描述清楚地阐述了规则的审计目的和背景。
2. **使用键值对**：当需要明确区分多个方面的描述时，使用键值对格式可以增加清晰度。
3. **简洁表达**：尽管描述需要完整，但也应避免冗长。精简的文本更易于理解和维护。

应用案例

假设你需要编写一个规则来审计使用了不安全配置的数据库连接。一个有效的 `desc` 语句可能是：

```
1 desc(
```

```
2   title: "审计数据库连接安全配置",
3   detail: "检查数据库连接是否使用了加密和正确配置的认证机制"
4 )
```

在这个案例中，我们使用了两个描述项，`title` 和 `detail`，通过花括号 `{}` 分隔，以清晰地说明规则的目的和具体的审计焦点。这种结构化的描述方式不仅有助于规则的编写者，也方便其他开发者或安全分析师理解和使用该规则。

优化描述

根据上面的解释，我们可以优化我们的描述信息为：

```
1 desc(
2     "Title": "审计因为未设置 setFeature 等安全策略造成的XXE漏洞",
3     "Fix": "修复方案: 需要用户设置 setFeature / setXIncludeAware /
setExpandEntityReferences 等安全配置"
4 )
```

当我们不做任何审计的时候，直接执行这个语句将会直接输出描述信息

```
1  yak sf --program xxe xxe.sf
2 [INFO] 2024-06-26 11:53:36 [ssacli:221] start to use SyntaxFlow rule: xxe.sf
3 [INFO] 2024-06-26 11:53:36 [ssa:42] init ssa database: /Users/v1ll4n/yakit-
projects/default-yakssa.db
4 [INFO] 2024-06-26 11:53:36 [ssacli:272] syntax flow query result:
5 rule md5 hash: 2b0aaa151a9f3c58a08487f38185ad47
6 rule preview: desc(     "Title": "审计因为未设置 setF...tExpandEntityReferences
等安全配置" )
7 description: {Title: "Title", Fix: "Fix"}
8
9
```

3. 编写审计规则

审计规则是 SyntaxFlow 的核心，我们观察有漏洞的代码，发现漏洞集中在下面三行：

```
1 DocumentBuilder documentBuilder =
  DocumentBuilderFactory.newInstance().newDocumentBuilder();
2 InputStream stream = new ByteArrayInputStream(xmlStr.getBytes("UTF-8"));
3 org.w3c.dom.Document doc = documentBuilder.parse(stream);
```

在这三行中，我们发现，最重要的其实是 `documentBuilder.parse(stream)`。我们的审计可以先从这个地方开始。

如何编写 SyntaxFlow 规则找到 `documentBuilder` 的 `parse` 调用？当然用户可以直接在“规则文件结构中”找到 `xxe.sf` 的实现直接得到答案，但是编写规则的具体细节，仍然需要用户学习，接下来我们将抽丝剥茧，逐步由浅入深地为用户解读核心规则编写的步骤。

从变量名方法名开始审计

如果要找到 `documentBuilder.parse(...)` 这个函数调用位置，用户需要找到 `documentBuilder` 这个变量和 `parse` 成员。在 SyntaxFlow 中，你可以直接输入 `documentBuilder` 来找到这个位置。

SPEC: 词法与符号搜索

```
1 filterItemFirst
2     : nameFilter                              # NamedFilter
3     | '.' lines? nameFilter                   # FieldCallFilter
4     ;
5
6 nameFilter: '*' | identifier | regexpLiteral;
```

在 SyntaxFlow 中，通过词法搜索能够直接定位到特定的变量名、方法名或者函数名。这是一个非常有用的特性，特别是在进行代码审计或安全分析时，快速精确地定位到感兴趣的代码片段至关重要。下面是如何在 SyntaxFlow 中利用词法搜索的一些具体案例：

1. 搜索特定的变量名

如果你想找到代码中所有使用 `documentBuilder` 这个变量的地方，可以使用以下 SyntaxFlow 查询：

```
1 documentBuilder;
```

这条规则会匹配代码中所有的 `documentBuilder` 变量实例。

2. 搜索方法调用

要找到所有调用 `parse` 方法的位置，你可以使用以下查询：

```
1 .parse;
```

这条规则利用了 `.` 前缀来指定我们正在搜索的是一个方法或属性名，而不是变量名。

3. 结合变量名和方法调用

要精确找到 `documentBuilder.parse(...)` 的调用位置，可以结合变量和方法名进行搜索：

```
1 documentBuilder.parse;
```

这样的查询将定位到所有 `documentBuilder` 对象上调用 `parse` 方法的代码位置。

4. 使用正则表达式进行模糊搜索

如果你不确定具体的方法名，或者想要查找包含某个模式的所有方法，可以使用正则表达式进行搜索。例如，要找到所有以 "get" 开头的方法调用，可以使用：

```
1 .get*;
```

或者使用更精确的正则表达式：

```
1 /(get[A-Z].*)/;
```

这将匹配所有以 “get” 开头并且紧跟一个大写字母的方法，例如 `getName` 、 `getInfo` 等。

5. 使用 glob 格式

对于更加模糊的搜索，比如想要找到所有含有 “config” 字眼的变量或方法，可以使用 glob 格式：

```
1 *config*;
```

这条规则会匹配所有包含 “config” 的标识符，不论其前后如何。

6. 搜索常量

对于某些硬编码的常量、敏感信息、配置信息等，希望 `精准匹配` / `模糊匹配` 。常量搜索集成了三种格式 `regexp` 、 `glob` 、 `exact` 。可以通过前缀 `r` 、 `g` 、 `e` 的格式来使用，支持 `heredoc` 和 `QuotedString` ：

```
1 g"a*"
```

```
2
3
e<<<CODE
password
CODE
```

7. 实战中注意事项

- 在审计具体的代码的时候，如果你想让你的 SF 规则可以审计通用代码，尽量不要具体指明参数名。
- 不需要担心赋值语句会中断数据流，你可以链式调用，结果一般不会有啥影响，因为在 SSA 中，不存在赋值操作，因为数据流并不会因为重新赋值一个变量而被切断

> AI 解读：
>
> 1. **审计通用代码时避免具体指明参数名**
>
> 当你编写 SyntaxFlow 规则以审计代码时，建议尽量不要指定具体的参数名。这是因为在多种编程实践中，参数的命名可能会有所不同，尤其是在处理多个项目或多种技术栈时。如果规则中包含了具体的参数名，那么该规则的适用性就会受到限制，只能在特定的命名约定下有效。相反，使用通用的匹配模式（如使用 * 或正则表达式）可以提高规则的灵活性和适用范围。
>
> 2. **赋值语句不会中断数据流**
>
> 在传统的程序分析中，变量的重新赋值可能会使跟踪变量的数据流变得复杂。然而，在使用 SSA 形式表示的代码中，每个变量在其生命周期内只被赋值一次。这种特性简化了数据流的分析，因为变量的值在其被定义之后就不会再改变，即使在代码中出现了看似重新赋值的操作。
>
> 在 SSA 中，如果一个变量需要重新赋值，会引入新的变量来代替。这意味着在审计或分析过程中，不需要担心常规的赋值操作会中断或改变数据流的追踪。因此，即使是复杂的链式调用或多次赋值操作，也不会影响到最终的分析结果。
>
> 这种特性使得使用 SyntaxFlow 进行静态分析时，能够更加直接和清晰地追踪数据流和变量之间的关系，即使在面对复杂的代码逻辑时也能保持高效和准确。
>
> 总结来说，以上的注意事项强调了在使用 SyntaxFlow 进行代码审计时，应采取灵活通用的规则编写策略，并充分利用 SSA 形式的优势，以提高分析的效率和覆盖范围。这些建议对于那些希望深入理解并有效利用 SyntaxFlow 进行安全审计的开发者和安全专家来说非常有价值。

作为函数调用审计

SEPC: 寻找函数调用参数审计

寻找函数调用特性是 SyntaxFlow 中常见的操作方式，例如我想找到所有 `.parse` 作为成员被调用的指令，直接执行 `.parse()` 就可以找到，如果想要审计函数调用的参数，则分两种情况，一种是审计全部参数（不使用逗号分隔，例如 `.parse(* as $source)`），另一种是审计特定位置的参数（使用逗号分隔 `.parse(* as $source,)`），具体的语法定义如下：

```
filterItemFirst
    : nameFilter                                 # NamedFilter
    | '.' lines? nameFilter                      # FieldCallFilter
    ;

filterItem
    : filterItemFirst                            #First
    | '(' lines? actualParam? ')'                # FunctionCallFilter
    ...
    ...
    ...
    ;

actualParam
    : singleParam    lines?                   # AllParam
    | actualParamFilter+ singleParam? lines?  # EveryParam
    ;

actualParamFilter: singleParam ',' | ',';

singleParam: ( '#>' | '#{' (recursiveConfig)? '}' )? filterStatement ;
```

在 SyntaxFlow 中，审计函数调用及其参数是一项非常重要的功能，特别适用于安全分析和代码审计。通过精确地捕捉函数调用和审查其参数，审计员可以识别潜在的安全风险，如不当的数据处理或可能的漏洞利用。本教程将详细介绍如何在 SyntaxFlow 中查找和审计函数调用参数。

1. 搜索函数调用

要找到所有使用 `.parse` 方法的调用，您可以简单地使用以下查询：

```
.parse();
```

这条规则匹配所有调用 `.parse` 方法的地方，而不考虑它被调用的上下文或参数。

2. 审计所有参数

如果您想要审计 `.parse` 方法调用时传递的所有参数，可以使用如下格式：

```
1  .parse(* as $source);
```

这里的 `*` 代表匹配任何参数， `as $source` 将匹配到的参数赋予变量名 `$source` ，方便后续进一步分析。

3. 审计特定位置的参数

如果您只关心 `.parse` 方法调用中特定位置的参数，比如仅第一个参数，您可以这样写：

```
1  .parse(* as $source,);
```

这里的逗号 `,` 表示分隔参数， `* as $source` 指定只匹配第一个参数。如果需要匹配第二个或后续参数，可以根据需要添加逗号和对应的匹配模式。

4. 语法详解

- **filterItemFirst**:
  - `nameFilter` : 匹配具体的函数名或方法名。
  - `.` + `nameFilter` : 指定函数或方法调用。
- **filterItem**:
  - `(` `actualParam?` `)` : 匹配函数调用时的参数列表。
- **actualParam**:
  - `singleParam` : 匹配所有参数。
  - `actualParamFilter+` `singleParam?` : 匹配特定的参数。
- **singleParam**:
  - 表达式用于捕获并操作函数调用中的参数。

5. 使用场景示例

假设我们要审计一个安全敏感的函数 `loadData` ，它可能从外部源加载数据：

```
1  .loadData(* as $data);
```

此规则将捕获所有 `loadData` 函数调用的参数，并将其存储在变量 `$data` 中。这可以帮助开发者或安全专家进一步分析这些参数，确定是否存在安全隐患。

6. 实战使用：结合变量名、方法名链与方法调用逻辑进行审计

结合变量名、方法名链与方法调用逻辑进行审计是一种高效的策略，可以帮助审计员深入理解代码的功能和潜在风险。通过精确地追踪变量和函数调用，可以揭示代码中的复杂交互和潜在的安全漏洞。下面我们将通过几个案例来展示如何实现这种审计，并提出相应的注意事项。

案例 1：追踪特定方法调用并审计其参数

假设我们需要审计一个 web 应用中所有涉及 XML 解析的位置，特别是 `parse` 方法的调用。我们的目标是确定是否正确地设置了 XML 解析器，以防止 XXE 攻击。

SyntaxFlow 查询：

```
1  DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(* as $source);
```

解释：

- 这个查询首先定位到所有 `DocumentBuilderFactory.newInstance().newDocumentBuilder()` 的调用。
- 然后它追踪到 `parse` 方法的调用，并捕获所有传递给 `parse` 方法的参数。
- 参数被存储在变量 `$source` 中，以便进一步分析是否存在潜在的风险。

案例 2：审计敏感函数的使用情况

考虑一个场景，我们需要找到所有使用敏感函数 `exec` 的地方，这对于发现潜在的命令注入攻击非常重要。

SyntaxFlow 查询：

```
1  Runtime.getRuntime().exec(* as $cmd);
```

解释：

- 该查询寻找所有 `Runtime.getRuntime().exec` 的调用。
- 它捕获传递给 `exec` 的所有参数，并将其赋值给变量 `$cmd`，用于后续分析命令的内容和安全性。

实战中的注意事项：

1. **避免过于具体的参数名：**

- 尽量使用通用的匹配模式，如 `*`，以提高规则的适用性和灵活性。
- 这有助于规则适用于不同的编码风格和技术栈。

2. **深入理解数据流**：
   - 利用 SyntaxFlow 的 SSA 架构优势，理解变量赋值和数据流是如何在代码中传递的。
   - 了解 SSA 可以帮助你更准确地追踪变量的历史和变化，尽管存在复杂的赋值和引用关系。
3. **重视方法链的完整性**：
   - 在编写规则时，尽可能追踪完整的方法调用链。这不仅有助于精确地定位问题，还可以提供调用上下文，有助于深入分析潜在问题。
   - 方法链的完整追踪也有助于防止漏报，特别是在涉及多层方法调用和对象创建的复杂系统中。

通过以上案例和注意事项，用户可以更好地理解和掌握在实际审计活动中应用变量名、方法名链和方法调用逻辑的技术。这些技术不仅提高了审计的效率，还大大增强了审计的准确性和深度。

审计中间结果暂存

SPEC：审计中间变量

在审计的过程中，你可以把任何步骤审计出的值作为一个 “变量” 暂存在 SyntaxFlow 的上下文中，例如： `as $source` 或 `.parse(* as $params)` 具体定义如下：

```
1 filterStatement
2     : refVariable filterItem*  (As refVariable)? # RefFilterExpr
3     | filterExpr  (As refVariable)?             # PureFilterExpr
4     ;
```

在 SyntaxFlow 中，审计中间结果的暂存是通过使用 `as` 关键字将审计结果赋值给变量来实现的。这种机制使得在复杂的代码审计过程中能够方便地引用、分析及进一步处理这些中间结果。下面我将详细介绍这个语法的使用方法和重要性。

1. 语法结构

在 SyntaxFlow 中，可以通过两种基本的表达式来存储和引用审计结果：

1. **RefFilterExpr**:
   - **形式**: `refVariable filterItem* (As refVariable)?`
   - 这种形式允许从一个引用变量开始，经过一系列的过滤操作，最终可能将结果再次存储到一个新的引用变量中。
2. **PureFilterExpr**:
   - **形式**: `filterExpr (As refVariable)?`

- 这种形式直接从一个过滤表达式开始，可选地将结果存储到一个引用变量中。

2. 使用 `as` 关键字

`as` 关键字用于将某个过滤表达式或操作的结果存储到一个变量中，便于后续的引用和操作。这在处理复杂的数据流或多步骤的代码审计中尤为重要。

示例说明

假设我们需要追踪函数 `parse` 被调用时传递的参数，并希望进一步分析这些参数。

查询示例:

```
1 .parse(* as $params);
```

在这个例子中， `*` 捕获了 `parse` 方法的所有参数， `as $params` 将这些参数存储到变量 `$params` 中。之后，你可以使用 `$params` 在其他查询或过滤中引用这些参数。

3. 实战应用

在实际的代码审计或安全分析中，这种能力极大地增强了规则的灵活性和表达力。例如，如果我们要分析一个可能受到 SQL 注入攻击的数据库查询：

```
1 DriverManager.getConnection().createStatement().executeQuery(* as $sql);
```

- 这里， `as $sql` 捕获了传递给 `executeQuery` 的所有参数，并将其存储在变量 `$sql` 中。随后，我们可以对 `$sql` 进行进一步的安全性检查，比如检测是否包含潜在的危险SQL命令或模式。

4. 重要性

使用 `as` 关键字来存储中间结果，为编写高效且易于管理的审计规则提供了以下几个优势：

- **模块化**：可以将复杂的审计任务分解成多个简单、模块化的步骤。
- **重用性**：存储的变量可以在多个不同的审计表达式中重复使用，减少重复劳动。
- **清晰性**：明确标记和存储关键审计点的结果，使得审计过程更加透明和易于理解。

通过有效地使用 `as` 关键字和上述结构，你可以提升审计的准确性和效率，更好地管理和分析在复杂代码环境中捕获的数据。

递归检查链式调用

深度遍历链式调用（Deep Chain Invocation Tracing）是一种递归检查链式调用的技术，允许分析工具或程序员快速追踪到某个特定方法调用，无论这个调用是直接发生还是间接通过多层链式方法调用

发生的。这种方式特别适用于处理那些包含多级链式调用的代码，如 Java 中常见的各类构造器和工厂方法。

SPEC：深度遍历链式调用

```
filterItem
    : ...
    | '...' lines? nameFilter                    # DeepChainFilter
    ...
    ;
```

这里的 '...' 符号用于指示从当前位置开始，向下递归追踪直到找到指定的方法或函数（通过 nameFilter 指定）。这种语法允许开发者不必明确每一级调用的具体细节，而是侧重于关心的特定函数或方法。

案例分析与语法应用

定义如上，结合代码理解是最快的，大家看如下案例：

```java
DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
InputStream stream = new ByteArrayInputStream(xmlStr.getBytes("UTF-8"));
org.w3c.dom.Document doc = documentBuilder.parse(stream);
doc.getDocumentElement().normalize();
```

这段案例出现了多次，我们在“递归检查链式调用”的语法中，如果想要快速定位 `parse` 这个调用，但是并不愿意加入任何“变量名”，按之间的定义应该如何编写定位语句?

```
DocumentBuilderFactory.newInstance().newDocumentBuilder().parse()
```

通过这个“词法名追踪+函数调用审计”可以轻松的找到 `parse()` ，但是他也暴露了一些不优雅的地方，例如，我们必须把完整的路径写出来才行。“递归链式调用”就可以解决这个问题，我们可以使用一些简单的写法：

```
DocumentBuilderFactory...parse()
```

使用 `...` 可以直接省略到其中 `newInstance().newDocumentBuilder()` 的定义，直接让 SyntaxFlow 自动寻找。

重要性和应用

这种递归检查链式调用的方法在安全审计和代码维护中非常有价值。它不仅减少了必须编写的代码量，还提高了代码审计的效率，特别是在面对复杂且嵌套的方法调用时。通过省略中间层，审计员可以更快地定位关键方法调用，并集中精力分析这些调用的安全性和正确性。

追踪某个值的使用链和定义链

SPEC：Use-Def 链追踪运算定义

简单来说，SyntaxFlow 是支持 UD 和 DU 链的追踪的，这个技术十分有用，可以充分发挥 SSA 的技术优势，精准追踪到想要的数据流。在 SyntaxFlow 设计中：

1. `->` 表示向下追踪一级使用链的节点，`-->` 表示向下追踪使用链节点直到链结束。
2. `#>` 表示向上追踪一级定义链的节点，`#->` 表示向上追踪支配（定义）链直到链结束。
3. `{}` 表示追踪设置追踪的时候的上下文或者参数，例如 `-{depth: 5}->` 表示向下追踪定义链，追踪深度为5表示最多追踪5层。

```
1  filterItem
2      : filterItemFirst                                    #First
3      ...
4      | '->'                                               # NextFilter
5      | '#>'                                               # DefFilter
6      | '-->'                                              # DeepNextFilter
7      | '-{' (recursiveConfig)? '}->'                      # DeepNextConfigFilter
8      | '#->'                                              # TopDefFilter
9      | '#{' (recursiveConfig)? '}->'                      # TopDefConfigFilter
10     ...
11     ;
```

1. 基本教程：追踪 Use-Def 链运算符在 SyntaxFlow 中的使用

在 SyntaxFlow 中，对变量的使用（Use）和定义（Def）链的追踪是通过特定的运算符实现的，这些运算符使得在静态单赋值（SSA）形式的代码中追踪数据流变得异常精确和高效。这一部分教程将解释如何使用这些关键的符号来追踪变量和函数的使用和定义链。

2. 运算符概览

1. **`->` 和 `-->` （使用链追踪）**
   - `->`：追踪到下一个使用该变量或函数的地方。这是追踪变量在代码中“一级”使用的基本方式。
   - `-->`：追踪直到使用链的结束。这个运算符将继续追踪，穿过所有使用点，直到没有更多的使用，即完全展开整个使用链。

2. `#>` 和 `#->` （定义链追踪）

- `#>`：追踪到变量或函数的直接定义点。这通常是变量被赋值或函数被声明的地方。
- `#->`：追踪直到定义链的开始。使用这个运算符可以追踪到变量或函数的最初定义，穿过所有中间的定义点。

3. `-{}` 和 `{}` 内的设置（定制追踪深度或上下文）

- `-{}->`：允许你定义追踪的深度或其他参数。例如，`-{depth: 5}->` 表示追踪使用链，但追踪的深度限制为5层。

3. 使用实例与解释

案例一：审计针对 ProcessBuilder 的 RCE

审计代码示例：

```
 1 import java.io.*;
 2 
 3 public class TestRCE {
 4     public static void main(String[] args) {
 5         String cmd = "ping example.com";
 6 
 7         // 这行代码将被上面的规则捕获
 8         ProcessBuilder pb = new ProcessBuilder(cmd.split(" "));
 9         pb.start();
10     }
11 }
```

SyntaxFlow 规则案例

```
1 // 审计潜在的远程代码执行风险
2 desc(title: "rce")
3 // 捕获创建 ProcessBuilder 对象时的所有参数
4 ProcessBuilder(* as $cmd) as $builder
5 // 追踪 ProcessBuilder 对象启动进程的方法调用
6 $builder.start() as $execBuilder
7 // 检查是否成功执行了start方法，如果未执行，则可能存在漏洞
8 check $execBuilder then "fine" else "rce 2 SyntaxFlow error"
```

执行效果

我们可以把上述文件保存到 `RCE.java`，然后执行如下代码，来观察结果输出：`yak ssa -t . --program rce && yak sf --program rce rce.sf`

```
[INFO] 2024-06-26 14:11:10 [ssacli:132] start to compile file: .
[INFO] 2024-06-26 14:11:10 [ssacli:148] compile save to database with program name: rce
[INFO] 2024-06-26 14:11:10 [ssa:42] init ssa database: /Users/v1ll4n/yakit-projects/default-yakssa.db
[INFO] 2024-06-26 14:11:10 [language_parser:46] parse project in fs: *filesys.LocalFs, localpath: .
...
...
[INFO] 2024-06-26 14:11:10 [ssacli:272] syntax flow query result:
rule md5 hash: d04b7fc1476e8957cdad9f8ba36214a6
rule preview: // 审计潜在的远程代码执行风险 desc(title: "rc...e" else "rce 2 SyntaxFlow error"
description: {title: "title", $execBuilder: "fine"}
Result Vars:
  cmd:
    t1283661: Undefined-ProcessBuilder
        Sample.java:8:32 - 8:62
    t1283666: Undefined-cmd.split(" ")
        Sample.java:8:51 - 8:61
  builder:
    t1283662: Undefined-ProcessBuilder
        Sample.java:8:32 - 8:62

```

案例二：GroovyShell 代码执行漏洞

审计代码示例：

```
package org.vuln.javasec.controller.basevul.rce;
import groovy.lang.GroovyShell;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/home/rce")
public class GroovyExec {

    @GetMapping("/groovy")
    public String groovyExec(String cmd, Model model) {
        GroovyShell shell = new GroovyShell();
        try {
            shell.evaluate(cmd);
            model.addAttribute("results", "执行成功！！！");
```

```
18          } catch (Exception e) {
19              e.printStackTrace();
20              model.addAttribute("results", e.toString());
21          }
22          return "basevul/rce/groovy";
23      }
24  }
25
```

SyntaxFlow 规则案例

```
1  // 审计通过 GroovyShell 实例执行代码的情况
2  desc(title: "groovy shell eval")
3
4  // 捕获 GroovyShell 的 evaluate 方法调用时传递的所有参数
5  GroovyShell().evaluate(* as $cmd)
6  // 追踪参数 $cmd 的定义来源，直到最初的定义点
7  $cmd #-> * as $target
8
9  // 检查是否能追踪到 $cmd 的源头，若无法追踪，则可能存在代码注入风险
10 check $target then "fine" else "not found groovyShell.evaluate parameter"
```

执行效果：

同样的方式，使用 `yak ssa -t . --program groovy && yak sf --program groovy rce.sf` 执行审计动作，将会得到如下结果：

```
1  ...
2  ...
3  ...
4  [INFO] 2024-06-26 14:24:11 [ssacli:272] syntax flow query result:
5  rule md5 hash: 17476c2166f26d3dfc5fe3f1c116451b
6  rule preview: // 审计通过 GroovyShell 实例执行代码的情况 de...
   groovyShell.evaluate parameter"
7  description: {title: "title", $target: "fine"}
8  Result Vars:
9    cmd:
10     t1323355: Parameter-cmd
11         Groovy.java:13:29 - 13:39
12   _:
13     t1323363: Undefined-shell.evaluate(valid)
14         Groovy.java:16:18 - 16:31
15   target:
16     t1323355: Parameter-cmd
```

```
17          Groovy.java:13:29 - 13:39
18
19
```

这些规则和示例展示了如何使用 SyntaxFlow 来追踪和审计可能执行外部代码的调用，进而帮助检测潜在的安全漏洞。这种方法不仅能有效识别代码中的危险操作，还可以帮助开发和安全团队预防未经授权的代码执行。

4. 实战中的注意事项

- **上下文敏感**：理解当前代码的上下文非常关键，特别是在处理复杂的逻辑或大型代码库时。使用 `-{}->` 运算符来设置适当的追踪参数可以帮助维持追踪的可管理性。
- **性能考虑**：在大型项目中，使用 `-->` 或 `#->` 可能会导致性能开销。在可能的情况下，限制追踪的深度或明确追踪的起始点和终点。
- **追踪的精确性**：使用这些运算符时，需要确保理解每个符号的具体含义，以便精确地捕捉到想要的数据流和定义点。

附魔：分析值的筛选过滤

过程进行到这里，我相信用户已经基本掌握了 SyntaxFlow 的执行和基本规则的编写，接下来我们将会带领大家进入一个可以给上述审计过程“附魔”的技术：“分析值筛选过滤”。即通过 `?{...}` 来过滤掉不想要的审计值，或者不正确的（没有漏洞）的值。这个特性十分强大，具体定义如下：

SPEC：分析值筛选过滤定义

```
1  filterItem
2      : filterItemFirst                               #First
3      ...
4      | '?{' conditionExpression '}'                  # OptionalFilter
5      ...
6      ;
7
8  conditionExpression
9      : '(' conditionExpression ')'                        # ParenCondition
10     | filterExpr                                         # FilterCondition
// filter dot(.)Member and fields
11     |  Opcode ':' opcodes (',' opcodes) * ','?           # OpcodeTypeCondition
// something like .(call, phi)
12     |  Have  ':' stringLiteralWithoutStarGroup           #
StringContainHaveCondition // something like .(have: 'a', 'b')
13     |  HaveAny ':' stringLiteralWithoutStarGroup         #
StringContainAnyCondition // something like .(have: 'a', 'b')
14     |  VersionIn ':' versionInExpression                 # VersionInCondition
```

```
    | negativeCondition conditionExpression
                # NotCondition
    ...
    | conditionExpression '&&' conditionExpression         # FilterExpressionAnd
    | conditionExpression '||' conditionExpression         # FilterExpressionOr

    ;

Opcode: 'opcode';
Have: 'have';
HaveAny: 'any';
VersionIn: 'version_in';
```

在 SyntaxFlow 中， `?{}` 结构提供了一种强大的方式来应用条件表达式对审计数据进行过滤。这种功能对于精确控制哪些数据流继续参与进一步的审计分析非常重要。下面的表格介绍了几种常见的条件表达式，这些表达式可以帮助您根据特定的需求筛选数据。

1. 可用的过滤方式

<table>
<tr><th>表达式类型</th><th>描述</th><th>示例</th></tr>
<tr><td>嵌套语句</td><td>确定方法成员或属性等嵌套语句的执行是否存在</td><td>$vars?{.setFeature} as $new</td></tr>
<tr><td>! (逻辑非)</td><td>排除特定操作，用于否定条件</td><td>$vars?{!((.setFeature) || (.setXIncludeAware))} as $new</td></tr>
<tr><td>&& (逻辑与)</td><td>同时满足多个条件</td><td>$vars?{(.setFeature) && (.setXIncludeAware)} as $new</td></tr>
<tr><td>|| (逻辑或)</td><td>满足任一条件</td><td>$vars?{(.setFeature) || (.setXIncludeAware)} as $new</td></tr>
<tr><td>Opcode :</td><td>检查特定类型的操作，常用于操作码过滤</td><td>$vars?{opcode: 'call', 'phi'} as $new</td></tr>
<tr><td>Have :</td><td>检查是否包含指定字符串（无通配符）</td><td>$vars?{have: 'abc'} as $new</td></tr>
<tr><td>HaveAny :</td><td>检查是否包含任一指定字符串（无通配符）</td><td>$vars?{any: 'abc', 'def'} as $new</td></tr>
<tr><td>In:</td><td>检查依赖版本是否在某个版本区间，可以使用 || 表示区间并集</td><td>__dependency__.*fastjson.version as $ver;
$ver?{version_in:(1.2.3,2.3.4]} as $vulnVersion</td></tr>
<tr><td>conditionExpression</td><td>结合多种条件进行复杂的逻辑过滤</td><td>$vars?{(.setFeature) && !(.setXIncludeAware)} as $new</td></tr>
</table>

这些表达式可以结合使用，形成更复杂的过滤逻辑，以确保只有符合特定安全或业务逻辑的数据流被选中用于进一步分析。

2. 实战：XXE 漏洞检测

让我们通过一个具体的例子来看看如何在实践中应用这些过滤表达式来审计潜在的 XML 外部实体（XXE）攻击：

示例：审计 DocumentBuilderFactory 的配置

```
desc(
    "Title": "审计因为未设置 setFeature 等安全策略造成的XXE漏洞",
    "Fix": "修复方案: 需要用户设置 setFeature / setXIncludeAware / setExpandEntityReferences 等安全配置"
)

// 审计 DocumentBuilderFactory 是否已经设置了防止 XXE 攻击的安全属性
$factories = DocumentBuilderFactory.newInstance();
$unsafeFactories = $factories?{!((.setFeature) || (.setXIncludeAware) || (.setExpandEntityReferences))} as $entry;
$entry.*Builder().parse(* #-> as $source);

// 检查 parse 方法的调用源是否安全
check $source then "XXE Attack" else "XXE Safe";
```

解释：

- 首先创建 `DocumentBuilderFactory` 的新实例，并将其存储在 `$factories`。
- 使用 `?{...}` 过滤出那些没有调用 `.setFeature`、`.setXIncludeAware` 或 `.setExpandEntityReferences` 的实例，并将这部分结果存储为 `$entry`。
- 追踪 `$entry` 中的所有 `.parse` 方法调用，并检查其参数源头是否安全。

通过这个例子，可以看到 `?{...}` 结构如何帮助您精确控制审计过程中的数据流，并专注于那些最有可能展示出安全风险的部分。这种方法提高了审计的效率和准确性，是高级安全分析中不可或缺的工具。

执行结果

我们对最一开始提到的 XXE 漏洞进行分析，可以查看：

```
package com.vuln.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.ByteArrayInputStream;
```

```
10 import java.io.InputStream;
11
12 @RestController(value = "/xxe")
13 public class XXEController {
14
15     @RequestMapping(value = "/one")
16     public String one(@RequestParam(value = "xml_str") String xmlStr) throws
Exception {
17         DocumentBuilder documentBuilder =
DocumentBuilderFactory.newInstance().newDocumentBuilder();
18         InputStream stream = new ByteArrayInputStream(xmlStr.getBytes("UTF-
8"));
19         org.w3c.dom.Document doc = documentBuilder.parse(stream);
20         doc.getDocumentElement().normalize();
21         return "Hello World";
22     }
23 }
```

保存为 `XXE.java` 然后把上述编写的 SF 文件保存成 `xxe.sf` ，命令行执行 `yak ssa -t . --program xxe && yak sf --program xxe xxe.sf` 执行的结果为：

```
 1 [INFO] 2024-06-26 14:58:24 [ssacli:272] syntax flow query result:
 2 rule md5 hash: f3dde2cbbb200606c3361adb0f276c0e
 3 rule preview: desc(     "Title": "审计因为未设置 setF...en "XXE Attack" else
"XXE Safe";
 4 description: {Title: "审计因为未设置 setFeature 等安全策略造成的XXE漏洞", Fix: "修复
方案: 需要用户设置 setFeature / setXIncludeAware / setExpandEntityReferences 等安
全配置", $source: "XXE Attack"}
 5 Result Vars:
 6   factories:
 7     t1325817: Undefined-DocumentBuilderFactory.newInstance(valid)()
 8         XXE.java:17:65 - 17:78
 9   entry:
10     t1325817: Undefined-DocumentBuilderFactory.newInstance(valid)()
11         XXE.java:17:65 - 17:78
12   source:
13     t1325822: Undefined-ByteArrayInputStream
14         XXE.java:18:33 - 18:79
15     t1325821: Undefined-ByteArrayInputStream
16         XXE.java:18:33 - 18:79
17     t1325824: ParameterMember-parameter[1].getBytes
18         XXE.java:18:61 - 18:78
19     t1325801: Parameter-xmlStr
20         XXE.java:16:22 - 16:68
21     t1325825: "UTF-8"
```

```
22          XXE.java:18:70 - 18:77
23    _:
24      t1325829: Undefined-documentBuilder.parse(valid)
25          XXE.java:19:51 - 19:64
26
27
```

根据结果我们实际可以发现：`description` 中包含了 `XXE Attack` 字段说明漏洞结果已经被检出。并且对结果的描述中也写清楚了修复方案以及检测的结果理由。

集合运算：`交集:&`，`差集:-`，`并集:+`

SEPC: 集合运算定义

我们把查询过程当成集合的话，SyntaxFlow 定义了一套集合运算的运算符作为简单支持：

```
1 filterItem
2     : filterItemFirst                         # First
3     ...
4     ...
5     | '+' refVariable                         # MergeRefFilter
6     | '-' refVariable                         # RemoveRefFilter
7     | '&' refVariable                         # IntersectionRefFilter
8     ;
```

在 SyntaxFlow 中，集合运算是一种强大的工具，用于处理和分析代码中的数据流。通过定义三种基本的集合运算符：交集（`&`）、差集（`-`）、并集（`+`），SyntaxFlow 允许用户以灵活的方式组合和修改数据集，以满足特定的分析需求。下面详细解释这些运算符的用法和含义。

交集运算符: `&`

交集运算符 `&` 用于找出两个集合中共有的元素。在代码分析的上下文中，这可以用来确定两个不同数据流路径中共同的数据点或函数调用。

示例：

```
1 $callPoint & $filteredCall as $vuln;
```

这里，`$callPoint & $filteredCall` 表示求 `$callPoint` 集合与 `$filteredCall` 集合的交集。结果集 `$vuln` 将包含同时存在于 `$callPoint` 和 `$filteredCall` 中的元素。这种操作通常用于识别两个不同分析过程中共同的安全漏洞或关键点。

差集运算符: `-`

差集运算符 `-` 用于从一个集合中移除存在于另一个集合中的元素。这在需要排除特定数据点或排除已处理（例如已经过滤或验证）的数据点时非常有用。

示例:

```
1  $callPoint - $filteredCall as $vuln;
```

在这个例子中，`$callPoint - $filteredCall` 表示从 `$callPoint` 集合中移除那些也出现在 `$filteredCall` 集合中的元素。结果集 `$vuln` 将包含只存在于 `$callPoint` 中，而不在 `$filteredCall` 中的元素。这可以用来识别潜在的未过滤或未验证的危险数据流。

并集运算符: `+`

并集运算符 `+` 用于合并两个集合的元素，结果集包含两个集合中的所有元素（重复的元素只保留一份）。这在需要组合来自不同源的数据点时特别有用。

示例:

```
1  $paramDirectly + $paramIndirectly as $vuln;
```

这里，`$paramDirectly + $paramIndirectly` 表示将 `$paramDirectly` 集合和 `$paramIndirectly` 集合合并。结果集 `$vuln` 将包含两个集合中的所有元素。这通常用于创建一个更全面的数据集，以便进行更广泛的分析。

通过这些集合运算符，SyntaxFlow 提供了一种简洁而强大的方式来操作和分析代码中的数据流。这些操作使得用户能够根据特定的需求灵活地组合、排除或扩展数据集，从而更有效地识别和处理潜在的安全问题。

原生扩展（NativeCall）机制

SyntaxFlow 的高级关键特性之一是使用 **NativeCall** 函数，这些函数是预先定义的，可在语言内部提供各种实用功能。本教程将介绍 NativeCall 函数的概念，解释其用法，并提供可用函数的完整列表及其描述。

NativeCall 是预封装的函数，可调用以对代码中的值执行特定操作。这些函数用于操作、检查和转换数据结构，促进高级代码分析和转换任务。

SPEC: NativeCall 语法定义

```
1  <nativeCallName(arg1, argName="value", ...)>
```

其中：

- `<`：标记 NativeCall 的开始。
- `nativeCallName`：要使用的 NativeCall 函数名称。
- `(...)`：包含函数参数的圆括号。
- `>`：标记 NativeCall 的结束。

其完整的 eBNF 描述为：

```
nativeCall
    : '<' useNativeCall '>'
    ;

useNativeCall
    : identifier useDefCalcParams?
    ;

useDefCalcParams
    : '{' nativeCallActualParams? '}'
    | '(' nativeCallActualParams? ')'
    ;

nativeCallActualParams
    : lines? nativeCallActualParam (',' lines? nativeCallActualParam)* ','? lines?
    ;

nativeCallActualParam
    : (nativeCallActualParamKey (':' | '='))?  nativeCallActualParamValue
    ;

nativeCallActualParamKey
    : identifier
    ;

nativeCallActualParamValue
    : identifier | numberLiteral | '`' ~'`'* '`' | '$' identifier | hereDoc
    ;
```

NativeCall 函数列表

以下是 SyntaxFlow 中所有可用的 NativeCall 函数的表格，以及它们的描述：

函数名称	描述
<getReturns>	获取函数或方法的返回值。
<getFormalParams>	检索函数或方法的形式参数。
<getFunc>	查找包含特定指令或值的当前函数。
<getCall>	获取一个值的调用，通常用于获取与操作码相关的调用。
<getCaller>	查找包含特定值的调用指令。
<searchFunc>	在整个程序中搜索对某个函数的调用。如果输入已经是调用，则搜索该方法（函数）的其他调用。
<getObject>	获取与值关联的对象，通常用于面向对象的上下文中。
<getMembers>	获取对象或类的成员变量或方法。
<getSiblings>	检索代码结构中兄弟节点或值。
<typeName>	获取值的类型名称（不含包路径）。
<fullTypeName>	获取值的完整类型名称（包括包路径）。
<name>	获取函数、方法、变量或类型的名称。
<string>	将值转换为其字符串表示形式。
<include>	包含一个外部文件或资源中的 SyntaxFlow 规则。
<eval>	评估一个作为字符串提供的新 SyntaxFlow 规则。
<fuzztag>	评估一个 Yaklang 的 fuzztag 模板，利用 SFFrameResult 中的变量。
<show>	输出值而不执行任何操作（用于调试）。
<slice>	从值中提取一个切片，类似于数组或字符串的切片操作。使用示例：<slice(start= 。
<regexp>	对字符串执行正则表达式匹配或提取。支持捕获组。示例：<regexp(..., group: 1)>
<strlower>	将字符串转换为小写。
<strupper>	将字符串转换为大写。
<var>	将值赋给变量以供后续使用。
<mybatisSink>	在代码分析中识别 MyBatis 的 Sink（特定于 Java MyBatis 框架）。
<freeMarkerSink>	识别 FreeMarker 的 Sink（特定于 FreeMarker 模板引擎）。
<opcodes>	获取与值关联的操作码（opcode）。
<sourceCode>	获取值的源代码表示形式。
<scanPrevious>	扫描相对于给定值的前一个操作码或指令。
<scanNext>	扫描给定值之后的下一个操作码或指令。
<delete>	从当前上下文中删除变量或值。
<forbid>	将值标记为禁止；如果该值存在，将报告严重错误。
<self>	获取当前值本身（用于链式操作中）。

<dataflow>	获取与值相关的数据流信息。在流操作符 --> 或 #-> 之后使用。
<const>	在代码中搜索常量值。
<versionIn>	判断依赖版本是否在某个版本区间。

SyntaxFlow 的 NativeCall 一般来说是逐步演进的，上述列表可能并不能包含全部的 NativeCall，但是实际上覆盖了绝大多数常用的 NativeCall。如果有未覆盖的 NativeCall，用户可以参考源码和相关规则了解使用用法。在后续的案例中，我们会逐步为大家讲解 NativeCall 中的内容究竟都是如何使用的。

依赖版本信息检测

SyntaxFlow能够解析依赖包的版本信息，同时也能够编写规则分析版本信息。

获取相应依赖信息

解析的版本信息存储在SyntaxFlow内置变量名为 `__denpency__` 的变量中，因此可以通过筛选名称的方式获取其信息。

```
1  __dependency__.*fastjson.version as $ver; // 获取依赖名以fastjson结尾的依赖版本
2  __dependency__.*fastjson.filename as $ver; // 获取依赖名以fastjson结尾所在的依赖文
   件
```

筛选依赖版本

有了依赖信息，可以对依赖进行筛选。比如某个依赖的版本在一定版本区间可以认为其存在漏洞。一般对依赖版本的筛选使用 `version in` 语法或者 `分析值筛选过滤` 的 `VersionInCondition` 。

SPEC: 筛选依赖版本语法定义

```
1  filterItem
2     ...
3     | In versionInExpression                    # VersionInFilter
4     ;
5
6  versionInExpression: versionInterval ('||' versionInterval)*;
7  versionInterval: ( '[' | '(') vstart? ',' vend? (  ']'| ')' ) ;
8  vstart: versionString;
9  vend: versionString;
10 // unless ',' ']' ')'
11 versionBlockElement: Number versionSuffix* ;
12 versionSuffix: '-' | Identifier;
13 versionBlock:  versionBlockElement ('.' versionBlockElement )*;
14 versionString
15     : stringLiteral
```

```
16      | versionBlock
17      ;
18
19 In: 'in';
```

```
1 conditionExpression
2     ...
3     |  VersionIn ':' versionInExpression            # VersionInCondition
4     ...
5     ;
6
7 VersionIn: 'version_in';
```

以 `version in` 语法为例，在 `in` 关键字前面需要一个表示版本号的变量，而在后面需要添加 `版本区间` 。版本区间的表示方式使用通用的区间记号，圆括号表示“排除” ，方括号表示“包括” 。同时，区间还能使用 `||` 连接，表示并集。例子如下：

```
1 $version in (1,2] // 版本号是否在大于小于等于2区间内(1< version <=2)
2 $version in (1.0.0,2.0.0] //版本号是否在大于1.0.0小于2.0.0区间内(1.0.0 <version <=2.0.0)
3 $version in (1.2.3-beta,2.2.1-beta] //版本号是否在大于1.2.3-beta小于等于2.2.1-beta区间内(1.2.3-beta < version <= 2.2.1-beta)
4 $version in [1.1,1.3] || [2.2,2.3] || [3.2,3.3]//版本号是否在1.1~1.3或者2.2~2.3或者3.2~3.3区间内
```

同时，也能使用 `分析值筛选过滤` 进行版本筛选。

```
1 $version ?{version_in:(1,2]}// 版本号是否在大于小于等于2区间内(1< version <=2)
2 $version ?{version_in:(1.0.0,2.0.0]}//版本号是否在大于1.0.0小于2.0.0区间内(1.0.0 <version <=2.0.0)
3 $version ?{version_in:(1.2.3-beta,2.2.1-beta]}//版本号是否在大于1.2.3-beta小于等于2.2.1-beta区间内(1.2.3-beta < version <= 2.2.1-beta)
4 $version ?{version_in:[1.1,1.3] || [2.2,2.3] || [3.2,3.3]}//版本号是否在1.1~1.3或者2.2~2.3或者3.2~3.3区间内
```

实战:分析具有漏洞版本的fastjson

依赖信息（maven）：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>vulnerable-fastjson-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Fastjson dependency with known vulnerabilities -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <!-- An example version with known vulnerabilities, make sure to check for specific vulnerable versions -->
            <version>1.2.24</version>
        </dependency>
    </dependencies>
</project>
```

规则：

```
__dependency__.*fastjson.version as $ver;
$ver?{version_in:(1.2.3,2.3.4]}  as $vulnVersion
alert $vulnVersion for {
    title:"存在fastjson 1.2.3-2.3.4漏洞",
};
```

高级程序分析：SyntaxFlow 高阶用例

当读者读完前置的内容，已经基本对 SyntaxFlow 的使用风格有了入门印象了，SyntaxFlow 实际上是一个非常具有深度的技术，我们将在本章节为大家讲解 SyntaxFlow 的高级用法和高级特性，以便帮助用户解决非常难的技术问题。

高级静态代码分析技术：数据流路径敏感分析

在静态代码分析领域，我们经常遇到一系列复杂的概念，如指针敏感性（pointer sensitivity）、数据流敏感性（data flow sensitivity）和路径敏感性（path sensitivity）等。大多数相关文献充斥着晦涩

难懂的数学公式和LaTeX算法，这些表述方式往往给人以高深莫测的印象。令人遗憾的是，这些所谓的"高级技术"在实际工程实践和漏洞挖掘中的应用效果往往难以体现。更有甚者，相关的实现代码常常被刻意或无意地隐藏在文献的角落，使得这些技术与实际应用之间产生了巨大的鸿沟。

过分依赖理论推导和数学模型，而忽视了实际应用的重要性。这种做法不仅使得相关知识难以被广大工程师所理解和应用，更是阻碍了整个领域的发展。我们需要的是更加实用、直观、易于理解的教学和研究方法。

数据流分析与路径

在古早的教材和中文互联网材料中，数据流的自顶向下传播和影响，往往被认为是另一个概念“污点传播”，一般被定义为用户输入从某一个输入点，会逐步影响后续的函数执行，并且传播数据，进入Sink（污染槽）之类的概念。

实际上污点传播就是一个非常狭义的数据流的描述。如果以它来概括代码中的数据流就过分以偏概全了。实际上数据流我们可以大致分为自顶向下，或者自底向上的两种流动方式。这两种流动方式基本可以适配常见可以通过数据流发现的程序行为。

Java 大家都看腻了，我们以大家更容易读得懂的 PHP 来为大家介绍这个概念，可能会更加容易让人理解：

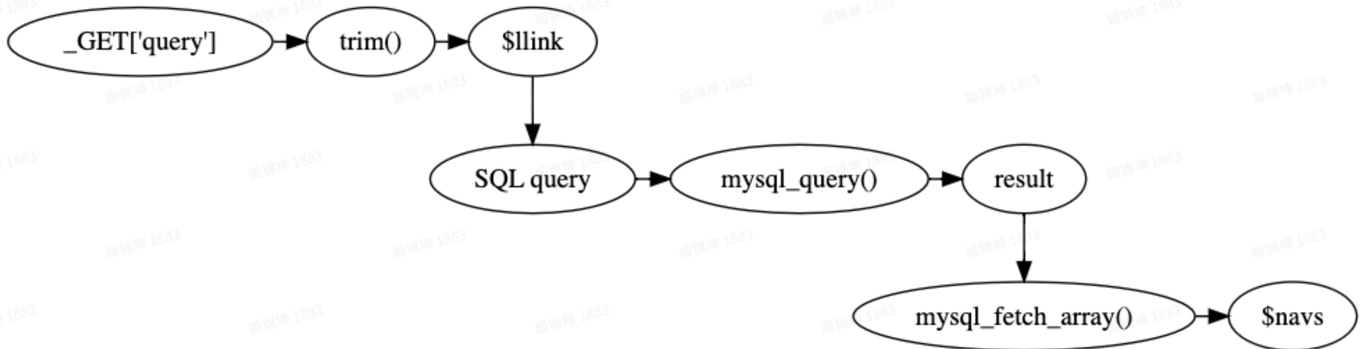
```php
1 <?php
2     $llink=trim($_GET['query']);
3     $query = "SELECT * FROM nav WHERE link='$llink'";
4     $result = mysql_query($query) or die('SQL语句有误：'.mysql_error());
5     $navs = mysql_fetch_array($result);
```

这段 PHP 代码主要用于从数据库中检索数据，其具体功能和流程如下：

1. **获取 URL 参数**：
   `$link=trim($_GET['query'])` 这行代码从 URL 的查询字符串中获取名为 `query` 的参数值，并使用 `trim()` 函数去除其前后的空白字符。这个值被存储在变量 `$llink` 中。
2. **构建 SQL 查询**：
   `$query = "SELECT * FROM nav WHERE link='$llink'";` 这里使用了一个 SQL 查询语句，目的是从 `nav` 表中选取所有列，条件是其 `link` 列的值等于 `$llink`。这个 SQL 语句将被用于查询数据库。
3. **执行 SQL 查询**：
   `$result = mysql_query($query) or die('SQL语句有误：'.mysql_error());` 使用 `mysql_query()` 函数执行前面构建的 SQL 查询。如果查询失败（例如，因为 SQL 语法错误或数据库连接问题），则脚本执行将终止，并显示错误信息 `SQL语句有误：` 后跟具体的错误原因。

4. **处理查询结果：**
   `$navs = mysql_fetch_array($result);` 使用 `mysql_fetch_array()` 函数从结果集中获取一行数据，并将其保存在数组 `$navs` 中。这个数组将包含 `nav` 表中对应行的所有列数据。

自顶向下的数据流



自顶向下的数据流我们发现，基本就是污点传播的路径问题，从 `_GET[query]` 开始，经过 `trim`，经过 `mysql_query`，最后查询出来的结果放在了 `$navs` 中。从数据传播的角度上来看，用户输入的参数，直接影响到了 `$navs` 变量的结果。中间经过了数据库查询。

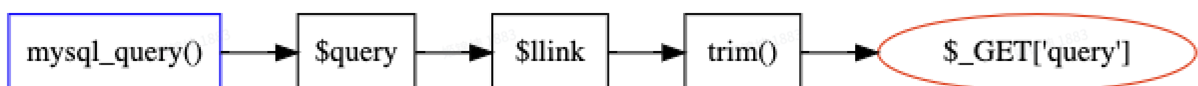
这是数据流的最基本研究方法，也是广为人知的“污点传播”的理论基础。然而我们很容易发现一些基本事实：

1. 我们关心的实际上并不是 `$navs` 而是 `mysql_query` 这个函数，因为他可能会导致 SQL 注入；
2. 我们不关心“污点”最后是谁，只要他经过了 `mysql_query` 按理说就会有问题；

因此，我们需要一整套的算法或者机制让数据流在自顶向下的过程中可以根据一定条件停止，或筛选出我们想要的值。这个意义十分重要。

自底向上的数据流

在我们的理念中，数据流是可以自底向上进行探索传播的，意味着，我们可以直接找到 `mysql_query` 函数，直接去探索它的参数的向上的数据流有没有经过用户输入的控制。



上图表示了自底向上的数据流分析的实际效果，我们发现通过这种方式反而更容易的定位问题。

这些基本思路和基本研究方法我们早在之前的文章中早已给大家介绍，接下来将讨论如何利用数据流分析的结果进行漏洞分析和行为限制。

数据流路径妙用：识别过滤函数

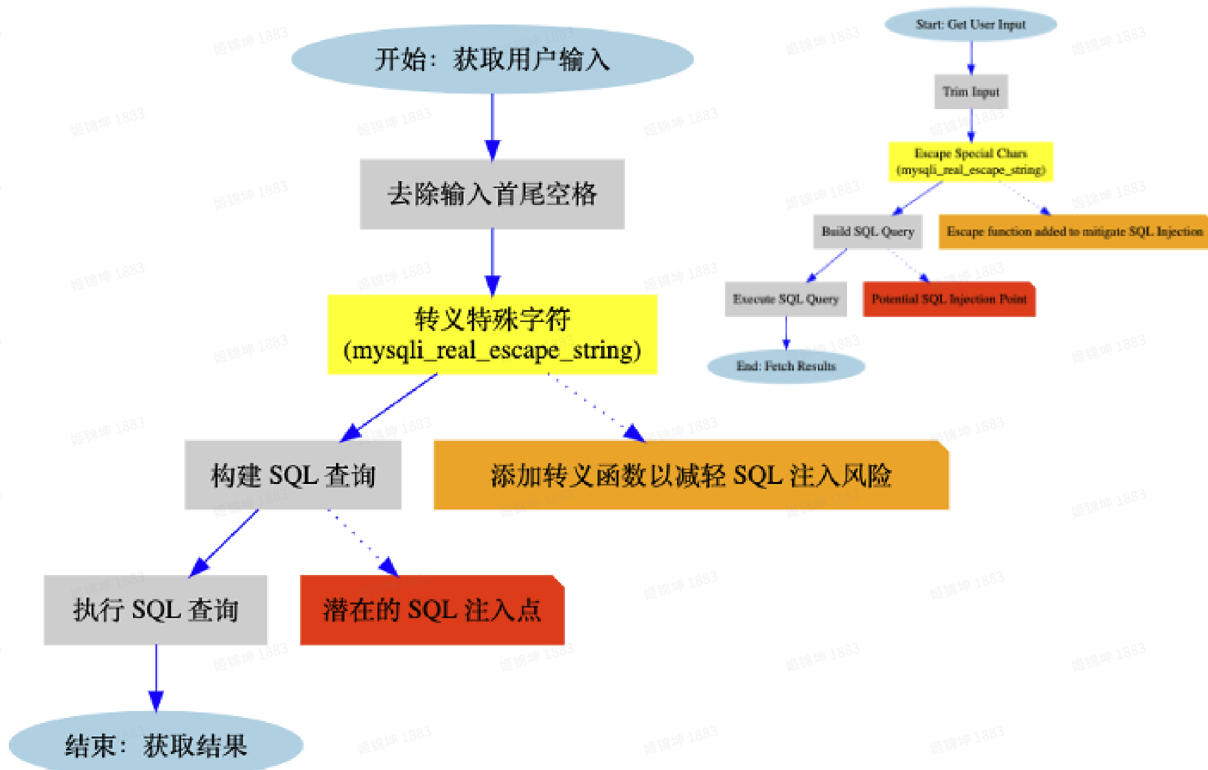
在上面的代码中，我们似乎能识别出这是一个 SQL 注入漏洞了，不需要任何限制，只需要分析数据流就可以得到，那么我们再对上述例子进行修改，增加一个 `escape` 函数来过滤一些危险字符，一般遇到这种情况，注入将会变的困难：

```php
<?php
    $llink=mysqli_real_escape_string(trim($_GET['query']));
    $query = "SELECT * FROM nav WHERE link='$llink'";
    $result = mysql_query($query) or die('SQL语句有误: '.mysql_error());
    $navs = mysql_fetch_array($result);
```

那么，从数据流的角度上来看，增加一个过滤函数并不会破坏数据流的流动，数据流的路径上会增加一个 `call mysqli_real_escape_string` 的节点（指令）。

最简单的思路是我们仍然需要找到数据流。并且把在数据流路径过程所有的指令都整理出来，检查一下这个指令是否包含一些过滤函数，如果包含过滤函数，说明这个 SQL 注入增加了过滤特殊字符的函数调用，就应该调整它的漏洞级别。我们如何表示这种过程?

过程展示



如何使用规则自动识别路径敏感的漏洞?

在前面的描述中，我们极力避免出现除了 PHP 之外的代码，为了主要是讲解思路和基本研究过程。那么当基本思路都清楚之后，我们将会讲解一下如何利用我们的技术和基础设施完成这种漏洞的识别。

实战案例：在PHP中使用 `-->` 识别参数最终影响的数据流位置

我们还是针对这一段代码：

```php
<?php
    $llink=mysqli_real_escape_string(trim($_GET['query']));
    $query = "SELECT * FROM nav WHERE link='$llink'";
    $result = mysql_query($query) or die('SQL语句有误：'.mysql_error());
    $navs = mysql_fetch_array($result);
```

保存为 `code.php` 之后，执行 `yak ssa -l php -t . -p dataflow` 命令进行编译，编译后我们就可以对 `dataflow` 这个程序名对应的程序进行测试并且，编写 SyntaxFlow 规则。在编译过程中会有日志输出，当你看到:

```
...
...
...
...
[INFO] 2024-09-26 13:43:54 [language_parser:72] parsed file: [code.php]
[INFO] 2024-09-26 13:43:54 [language_parser:77] program dataflow finish
[INFO] 2024-09-26 13:43:54 [ssacli:189] finished compiling..., results: 1
[INFO] 2024-09-26 13:43:54 [database_profile:26] SSA Database SaveIrCode Cost: 23.121462ms
[INFO] 2024-09-26 13:43:54 [database_profile:27] SSA Database SaveIndex Cost: 5.212542ms
[INFO] 2024-09-26 13:43:54 [database_profile:28] SSA Database SaveSourceCode Cost: 359.583µs
[INFO] 2024-09-26 13:43:54 [database_profile:29] SSA Database SaveType Cost: 4.127001ms
[INFO] 2024-09-26 13:43:54 [database_profile:30] SSA Database CacheToDatabase Cost: 23.195708ms
```

之类的内容输出的时候意味着编译完成了，随后创建扫描规则文件内容：

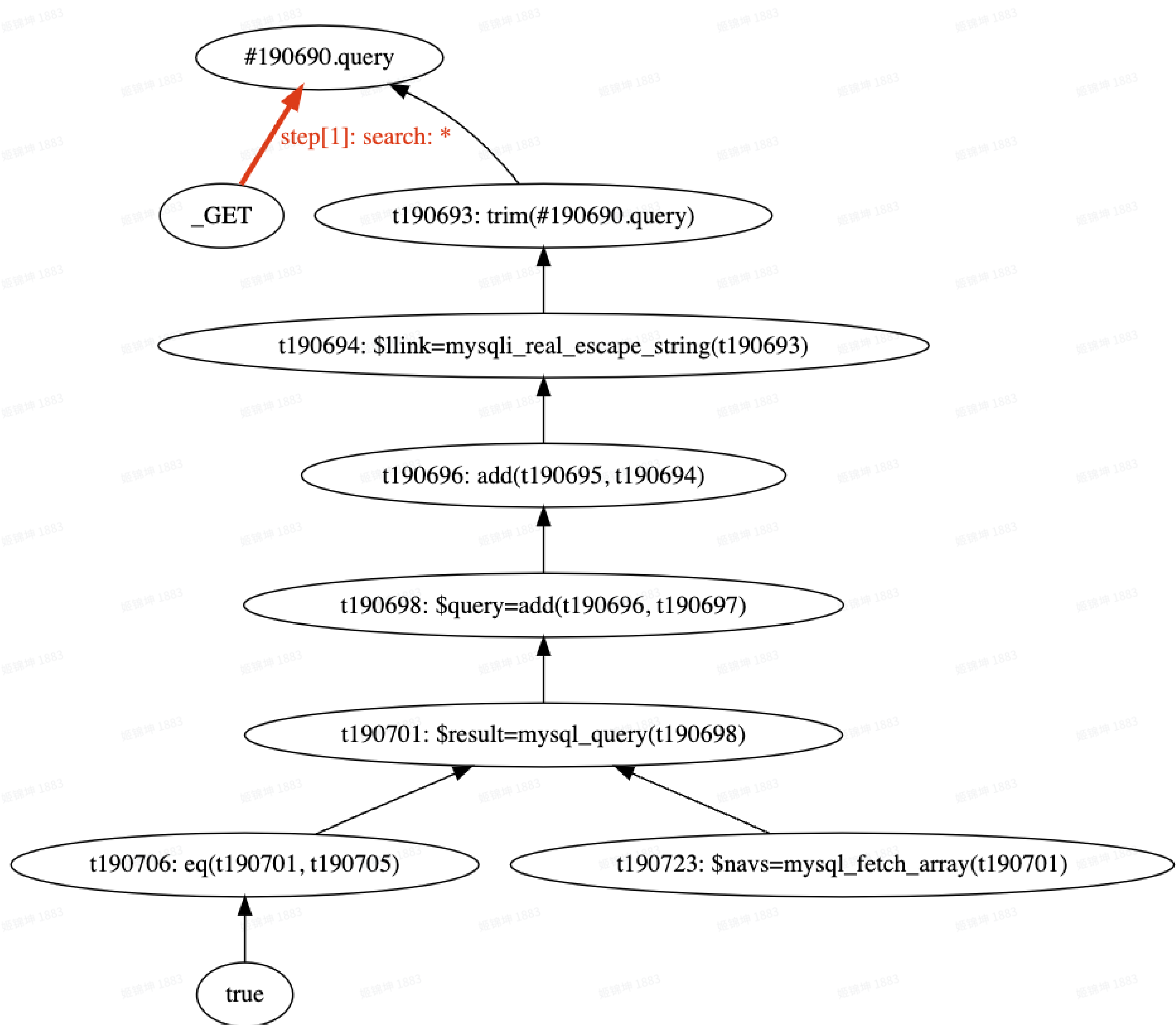
```
_GET.* as $params;
$params --> * as $sink;
alert $sink;
```

把上述规则保存为： `rule.sf` 然后再命令行中执行 `yak sf rule.sf -p dataflow` 将会看到输出为：

```
[INFO] 2024-09-26 13:48:03 [ssacli:539] start to use SyntaxFlow rule: rule.sf
...
...
...
[INFO] 2024-09-26 13:48:03 [ssacli:688] syntax flow query result:
rule md5 hash: 468a9fc888219cc95c0771b76b4ee88a
rule preview: _GET.* as $params;  $params --> * as $sink;  alert $sink;
description:
{"desc":"","lang":"","level":"","title":"","title_zh":"","type":""}
Result Vars:
  $sink:
    t190614: eq(Undefined-mysql_query(add(add...ET.query(valid)))), "''"),
true)
        code.php:4:5 - 4:67
    t190631: Undefined-mysql_fetch_array(Unde...ned-_GET.query(valid))),
"''")))
        code.php:5:13 - 5:39
```

看到上述数据，基本说明我们获取到了数据流的两个关键最终点，一个是 `mysql_fetch_array` 的调用结果，另一个是 `mysql_query` 执行结果和 `true` 的比较函数（这是由 or 运算来产生的，不要惊慌）。

我们在这里已经成功获取到了数据流最终终结的位置，在这里看他还是相当准确的，把上述结果整理起来就可以看到真正的结果所有数据流相关的结果：



在途中我们可以很清楚的发现，最终的结果其实是 `eq` 和一个 `$navs` 变量。那么光得到这一步，我们还不能说可以过滤出想要的结果，那么如何在规则中编写过滤语句呢？

获取数据流路径并过滤结果：检查 SQL 注入

我们经过上述操作可以得到数据流最终的点，但是需要检查过滤的步骤，这个时候应该怎么做呢？我们在 SyntaxFlow 中可以通过 `<dataflow()>` 这个功能来实现：我们把这个复杂的数据流捋清的过程封装成了一个叫 `<dataflow>` 的NativeCall。用户可以调用这个指令，把数据流的所有路径整理成在一起，一起进行检查，直到过滤出自己想要的数据。

继续案例中 PHP 的程序规则：

```
1 _GET.* as $params;
2 $params --> * as $sink;
3
4 $sink<dataflow(<<<CODE
```

```
5 *?{opcode: call && <getCaller><name>?{have: mysqli_real_escape_string }} as
  $__next__
6 CODE)> as $filtered;
7 $sink - $filtered as $vuln;
8
9 alert $vuln;
```

这段审计代码看起来就复杂了不少，我们直接把这段代码执行一下：`yak sf -p dataflow rule.sf` 发现他无法输出原来的信息了，因为我们过滤了 `mysqli_real_escape_string` 这个函数。我们创建一个有漏洞的代码：

```php
1 <?php
2     $llink=trim($_GET['query']);
3     $query = "SELECT * FROM nav WHERE link='$llink'";
4     $result = mysql_query($query) or die('SQL语句有误：'.mysql_error());
5     $navs = mysql_fetch_array($result);
6
```

在上述代码中移除了 `mysqli_real_escape_string` 函数，发现重新执行规则将会检查出来，那就发生了什么？接下来我们将解释一下这段代码究竟怎么回事儿：

SyntaxFlow 检测 SQL 注入的规则解释

1. `_GET.* as $params;`
   - 这行代码的意图是从 PHP 的全局 `$_GET` 数组中获取所有参数，并将这些参数存储到一个名为 `$params` 的变量中。在实际的 PHP 代码中，这通常是通过遍历 `$_GET` 数组完成的。
2. `$params --> * as $sink`：
   - 这条指令可能意味着将 `$params` 中的数据流向任何可能的“汇点” （sink），这里的“汇点”指的是数据可能被用于敏感操作的地方，如数据库查询。`$sink` 变量代表这些潜在的危险使用点。
3. `$sink<dataflow(<<<CODE`
   - 这部分开始定义一个数据流查询，用来跟踪 `$sink` 中的数据流。`<<<CODE` 表示接下来是一段内嵌的代码或查询。
4. `*?{opcode: call && <getCaller><name>?{have: mysqli_real_escape_string }} as $__next__`
   - 这段查询检测任何函数调用（`opcode: call`），并且特别查找调用者名称中包含 `mysqli_real_escape_string` 的情况。如果存在这样的函数调用，这个调用点被存储为 `$__next__`。

5. `CODE)> as $filtered;`
    - 这表示上述代码块的结束。查询的结果，即所有经过 `mysqli_real_escape_string` 处理的数据点，被存储在变量 `$filtered` 中。
6. `$sink - $filtered as $vuln`
    - 这行代码从 `$sink` 中排除那些已经被标记为 `$filtered` 的数据点。换句话说，它寻找那些潜在未经过滤直接用于敏感操作的数据点，并将这些点存储在 `$vuln` 中。
7. `alert $vuln;`
    - 最后，这行代码发出警告或报告关于 `$vuln` 的信息，即所有潜在的未经过滤的危险数据使用点。

这段代码是用于静态代码分析，特别是用来检测和报告潜在的 SQL 注入漏洞。它通过跟踪从用户输入到可能的敏感操作的数据流，并检查这些数据是否经过了适当的过滤（如使用 `mysqli_real_escape_string`），来帮助识别和预防安全风险。这种分析对于确保 Web 应用的安全性至关重要。

NativeCall 中的 <dataflow> 支持路径检测

`<dataflow>` 是一个高级的 NativeCall 函数，用于在代码中进行数据流分析。它的参数是一段专门用于检查特定变量的数据流路径的代码。这段代码帮助识别和过滤数据流路径，从而发现潜在的安全漏洞。为了确保分析的准确性，进入 `<dataflow>` 的变量必须至少经过 Use-Def 运算符的处理，这样可以确保变量的来源和使用都被妥善追踪和记录。

示例代码解释

```
1 _GET.* as $params;
2 $params --> * as $sink;
3
4 $sink<dataflow(<<<CODE
5 *?{opcode: call && <getCaller><name>?{have: mysqli_real_escape_string }} as
  $__next__
6 CODE)> as $filtered;
7 $sink - $filtered as $vuln;
8
9 alert $vuln;
```

`<<<CODE` 与 `CODE` 语法

`<<<CODE` 到 `CODE` 的语法类似于 PHP 中的 HereDoc 语法。它用于定义一个多行的字符串或代码块，在 `<dataflow>` 的上下文中，这种语法用于编写复杂的查询逻辑。 `<<<CODE` 标记代码块的开始，而 `CODE` 标记代码块的结束。这允许开发者在 NativeCall 函数中嵌入较长或较复杂的代码片段。

`$__next__` 变量的作用

在 `<<<CODE` 块中定义的查询逻辑中，`$__next__` 是一个特殊的变量，用于存储满足查询条件的代码点。在上述示例中，任何调用 `mysqli_real_escape_string` 函数的地方都会被捕获并赋值给 `$__next__`。如果 `$__next__` 不为空，即存在至少一个满足条件的点，那么当前路径对应的点将会被保留。这种机制允许 `<dataflow>` 分析函数细致地筛选和确定哪些数据流路径是安全的，哪些可能包含潜在的安全风险。

产品化工程化的 SyntaxFlow 规则

我们通过数据流路径敏感技术分析了 SQL 注入漏洞的检测，并介绍了高级的 SyntaxFlow 规则。虽然我们探索了基础的数据流分析方法，这些方法有效地帮助我们识别和防范常见的安全漏洞如 SQL 注入，然而我们所展示的仅是冰山一角，SyntaxFlow 的能力远超这些基础示例，它能够应对更加复杂和隐蔽的代码场景。

为了激发读者对学习更多高级 SyntaxFlow 规则的兴趣，我们提供了一个更为复杂的规则示例，这个示例使用 SyntaxFlow 内置的输入点识别并且可以检测常见的 php 过滤函数，可以直接按不同级别找到不同风险的 SQL 注入漏洞，可以在一条规则中识别多种模式的 SQL 注入情况：

```
desc(
    title: "mysql inject",
    type: audit,
    level: low,
)
<include('php-param')> as $params;
<include('php-filter-function')> as $filter;

mysql_query(* as $query);
$query #{
    until: `* & $params`,
}-> as $root;
$root?{!<dataflow(<<<CODE
*?{opcode: call} as $__next__;
CODE)>} as $result;
alert $result for {
    title: "Direct mysql injection",
    title_zh: "直接的mysql注入不经过任何过滤",
    type: 'vuln',
    level: 'high',
};

$root?{<dataflow(<<<CODE
*?{opcode: call && <self> & $filter} as $__next__;

CODE)>} as $filter_result;

```

```
alert $filter_result for {
    title: 'Filtered sql injection, filter function detected',
    title_zh: '经过过滤的sql注入，检测到过滤函数',
    type: 'low',
    level: 'low'
};

$root?{<dataflow(<<<CODE
*?{opcode: call && !<self> & $filter} as $__next__;
CODE)>} as $seem_filter;

alert $seem_filter for {
    title: 'Filtered sql injection, but no filter function detected',
    title_zh: '经过过滤的sql注入，但未检测到过滤函数',
    type: 'mid',
    level: 'mid'
};
```

这段 SyntaxFlow 规则的精妙之处在于其能够通过高级的数据流分析技术，综合利用路径敏感性、函数调用分析和条件判断，来识别和区分不同级别的 SQL 注入风险。以下是对这些规则的详细解释，帮助用户和读者更好地理解其工作原理和应用场景：

1. **参数识别与过滤函数检测**
    - `<include('php-param')> as $params;` 和 `<include('php-filter-function')> as $filter;` 这两行代码分别用于引入 PHP 参数和过滤函数的定义。这意味着规则可以自动识别 PHP 代码中的输入参数和常见的过滤函数，如 `mysqli_real_escape_string`。
2. **基本 SQL 注入检测**
    - `mysql_query(* as $query);` 这行代码标记了对 `mysql_query` 函数的调用点，这是执行 SQL 语句的关键位置。
    - `$query #{ until:` * & $params `, }-> as $root;` 这行代码定义了一个从 SQL 查询中追溯到参数输入的数据流路径。这里的 `until` 关键字用于限定追踪到包含参数的点为止，确保数据流的起点是用户输入。
3. **直接 SQL 注入检测**
    - `$root?{!<dataflow(<<<CODE *?{opcode: call} as $__next__; CODE)>} as $result;` 这段代码检测是否有直接从用户输入到 SQL 查询的数据流，而中间没有任何函数调用。如果存在这样的数据流，即认为存在高风险的直接 SQL 注入。
4. **过滤函数检测**

    - `$root?{<dataflow(<<<CODE *?{opcode: call && <self> & $filter} as $__next__; CODE)>} as $filter_result;` 这段代码用于检测数据流中是否存在过滤函数的调用。如果数据流中包含了过滤函数，那么认为 SQL 注入的风险较低。

5. **过滤函数未检测到的情况**
    - `$root?{<dataflow(<<<CODE *?{opcode: call && !<self> & $filter} as $__next__; CODE)>} as $seem_filter;` 这段代码检测数据流中是否虽然有函数调用，但并非是已知的过滤函数。如果是这种情况，那么将其认为是中等风险的 SQL 注入。

通过这样的规则设置，SyntaxFlow 能够在单一的规则文件中，根据数据流的不同特点，区分出不同级别的 SQL 注入风险。这种细致的风险分级有助于开发者更精确地定位潜在的安全问题，并采取相应的防护措施。这不仅提高了代码的安全性，也优化了安全审计的效率。