

前言

因为18年下半年自己的原因，RIPS源码精读这个系列文章的更新暂停了，因此在阅读本篇文章时，无可避免会有些滞涩感，可以再过一下我的前两篇该系列文章：

[RIPS源码精读\(一\):逻辑流程及lib文件夹大致说明](#)

[RIPS源码精读\(二\):扫描对象的实例化及token信息的生成](#)

在之前的文章中，RIPS通过Scanner类的初始化，完成对php源码的token分析处理，实现了格式的统一化。

RIPS根据token流信息生成AST树再生成CFG的方法来进行自动化分析,因此接下来文章的核心其实是: CFG的生成及污点分析。

CFG的生成主要通过使用token信息根据函数、变量进行分块，形成一级一级的调用链，进而生成CFG。

在RIPS中,通过对涉及危险函数的变量的不断回溯,来判断是否为用户可控的输入,进而完成污点分析。

考虑到之前大篇幅的粘贴代码，给流畅阅读带来了挺大的障碍，这一篇开始，将会尽量避免大幅的粘贴代码，转而以描述思路为主。

[一] 流程综述

整个生成过程以parse函数作为实现，整个的处理逻辑包裹于一个大循环中:

```
function parse(){
    for($i=0,$tokencount=count($this->tokens); $i<$tokencount; $i++, $this->tif++){
        {
            .....
        }
    }
}
```

在进行下一步之前，我们需要回顾一个知识点:

token信息通常可被表述为一个数组，分别为token■■■, token■■, token■

因此，基于上面这个知识点，在整个大循环中，主要由if组成，if的条件为:

```
if( is_array($this->tokens[$i]) )
```

在进行正式的解析过程之前，会进行对大文件的处理，进行缓冲处理，即:

```
$token_name = $this->tokens[$i][0];
$token_value = $this->tokens[$i][1];
$line_nr = $this->tokens[$i][2];

// add preloader info for big files
if($line_nr % PRELOAD_SHOW_LINE == 0)
{
    echo $GLOBALS['fit'] . ' ' . $GLOBALS['file_amount'] . ' ' . $this->file_pointer . ' (line ' . $line_nr . '
    @ob_flush();
    flush();
}
```

随后正式开始parse流程。

正式的解析流程通过if语句分割成多个块，大致功能经过整理分别如下:

```
/*VARIABLE*/
if($token_name === T_VARIABLE){

else if( in_array($token_name, Tokens::$T_FUNCTIONS) || (in_array($token_name, Tokens::$T_XSS) && ($_POST['vector'] == 'client'
/**STRING**/if($token_name === T_STRING && $this->tokens[$i+1] === '(')
/**FILE INCLUSION**/else if( in_array($token_name, Tokens::$T_INCLUDES) && !$this->in_function)
/**TAINT ANALYSIS**/if(isset($this->scan_functions[$token_value]) && $GLOBALS['verbosity'] != 5 && (empty($class) || (($this->i

/*CONTROL STRUCTURES*/
else if( in_array($token_name, Tokens::$T_LOOP_CONTROL) ){}
else if(in_array($token_name, Tokens::$T_FLOW_CONTROL)){}
}
```

```

/*FUNCTION*/
else if($token_name === T_FUNCTION)
else if($token_name === T_GLOBAL && $this->in_function){}
else if($token_name === T_RETURN && $this->in_function==1 ){}


/*CLASS*/
else if($token_name === T_CLASS){}
else if( $token_name === T_NEW && $this->tokens[$i-2][0] === T_VARIABLE ){}
else if($token_name === T_EXTENDS && $this->in_class){}


/*OTHER*/
else if($token_name === T_LIST){}
else if( $token_name === T_INCLUDE_END){}


/*BRACES*/
if($this->tokens[$i] === '{' && ($this->tokens[$i-1] === ')' || $this->tokens[$i-1] === ':' || $this->tokens[$i-1] === ';' ||
else if( $this->tokens[$i] === '}' && ($this->tokens[$i-1] === ';' || $this->tokens[$i-1] === '}' || $this->tokens[$i-1] === '

```

[二] 如何表示信息

这一部分我们掌握的是如何对已经分析出来的存在安全隐患的节点信息进行存储，以方便后续的分析。

RIPS中定义了一个节点类，专门用于存放相关信息，类名为VulnTreeNode，并且完成一次完整的分析，需要将许多个节点串起来，因此又定义了一个VulnBlock类用来关

在实际的审计过程中，我们会尤其关注危险函数中的可控参数，而在自动化审计中，同样如此。

当遇到危险函数存在可控参数后，需要对可控参数的来源进行不断的溯源，直到找到原始输入位置或者找到数据被过滤的位置才停下，而这种■■的过程便需要由Block与Tr

以下面这段源码作为示范:

```

index.php
<?php
    include 'vuln.php';
    $a = $_GET['a'];
    $b = $_GET['b'];
    $c = $a.$b;
    $c($_GET['d']);
?>

```


```


vuln.php
<?php
    function vuln($a){
        system($a);
    }


```

RIPS对变量继承关系的描述如下图所示:


▼  **var_declares_global** = {array} [3]


▶  **\$a** = {array} [1]


▶  **\$b** = {array} [1]


▼  **\$c** = {array} [1]


▼  **0** = {VarDeclare} [11]

 **id** = 24


▼  **tokens** = {array} [6]


▼  **0** = {array} [3]


 **0** = 320

 **1** = "\$c"


 **2** = 5


 **1** = "="


▼  **2** = {array} [3]


 **0** = 320


 **1** = "\$a"

 **2** = 5

 **3** = "."

▼  **4** = {array} [3]

 **0** = 320

 **1** = "\$b"

 **2** = 5

 **5** = ";

 **tokenscanstart** = 1

 **tokenscanstop** = 6

 **value** = ""

 **comment** = "vuln.php"

 **line** = 5

 **marker** = 0

 **dependencies** = {array} [0]

 **stopvar** = false

 **array_keys** = {array} [0]

使用另一个文件来演示Block与Node的关系:

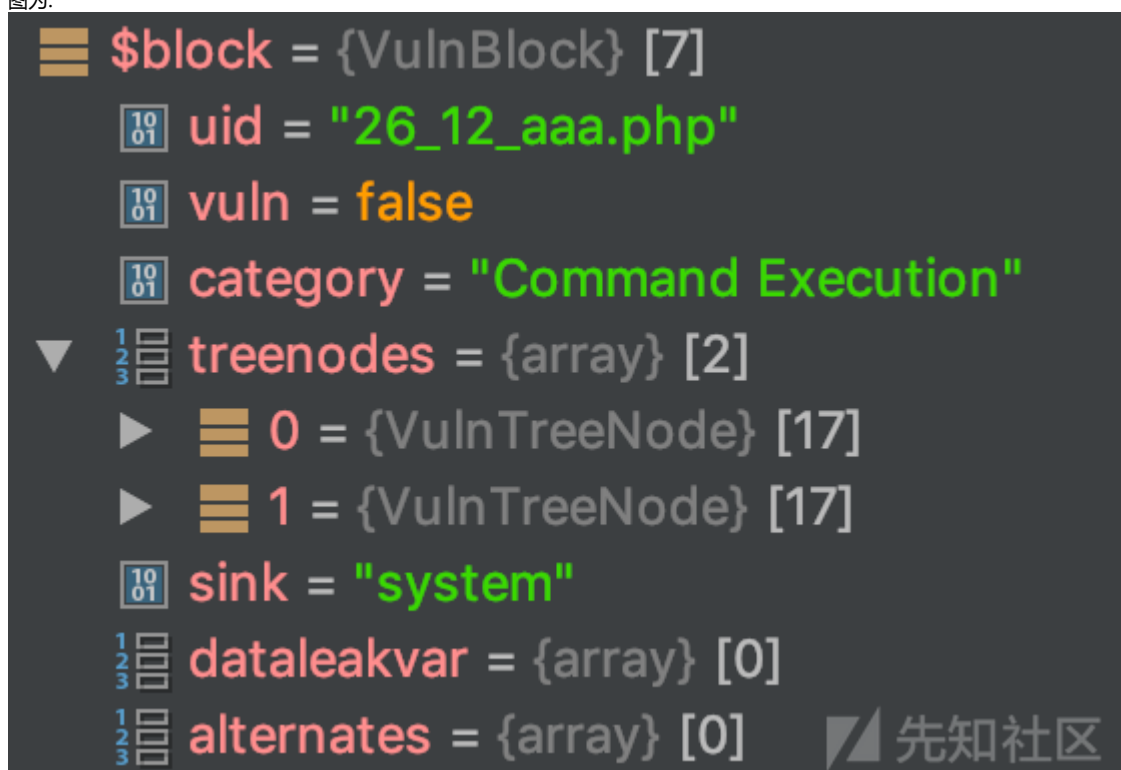
```

<?php
function c($c){
    d($c);
}
function b($b){
    c($b);
}
function a($a){
    b($a);
}
function d($d){
    system($d);
}
function e($e){
    f($e);
}
function f($f){
    d($f);
}

a($_GET['a']);
e($_GET['e']);
?>

```

图为:



[三] 功能详解 - 污点分析

在进行污点分析的讲解之前，要先理解前面提到的VARIABLE、STRING等多个模块有何作用。

从宏观上来说，VARIABLE、STRING、FUNCTIONS等部分都是为了相同的目的，即尽可能完整地描述出目标文件中变量继承关系、函数调用关系、各部分调用关系。

因此，在经过这些部分的处理后，我们能获得对目标文件的完整描述。而污点分析则是建立在这些描述之上的。

在自动化审计中，可能会面临这么一个问题，即如何重复扫描同一个Block。我们只需要在Block中设置一个字段，用以标识是否已被扫描过即可避免这个问题。在Block中，

```
$block = new VulnBlock($this->tif.'_'.$this->tokens[$i][2].'_'.basename($this->file_pointer), getVulnNodeTitle($token_value),
```

在确认当前块未被扫描后，首先会新建一个VulnTreeNode节点，后续对VulnTreeNode完善各种信息。随后对VulnTreeNode节点中的参数进行分析，找出是否存在用户

因此首先需要对危险函数的参数进行判断是否追溯，并得到该位置的参数是否被污染的结果，这两个在RIPS中的实现为:

```

■■■■■■■■■■
// parameter = 0 means, all parameters will be traced

```

```
$F_XSS = array(
    'echo'
    ...
)
```

■■■■■■■■

```
// trace back parameters and look for userinput, trace constants globally
$userinput = $this->scan_parameter(
    $new_find,
    $new_find,
    $this->tokens[$i+$c],
    $this->tokens[$i+$c][3],
    $i+$c,
    ($this->in_function && $this->tokens[$i + $c][1][0] === '$') ? $this->var_declares_local : $this->var_declares_global,
    $this->var_declares_global,
    false,
    $this->scan_functions[$token_value][1],
    false, // no return-scan
    $ignore_securing,
    ($this_one_is_secure || $in_securing)
);
```

其实到这里为止，后面的东西也没有什么需要过多描述的，由于在RIPS中没有明显的CFG构造过程，因此溯源过程看起来十分地凌乱。

[四] 最终

第三篇咕了半年，自己本地的第一版也是大篇幅的代码分析，但是年底的时候读了一遍，发现效果很差，于是采用了这种以思路为主的描述方式，希望对之后学习相关内容的人有所帮助。

点击收藏 | 0 关注 | 1

[上一篇：深入浅出Angr（一）](#) [下一篇：在内网中拿下DC的五种常用方法](#)

1. 1 条回复



[sera](#) 2019-02-25 15:47:13

支持

0 回复Ta

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)