

0x01SS工作原理

首先了解ss协议在localserver端与client的交互所用协议socks协议。

socks协议

SOCK5 工作原理

首先大体说一下计算机网络的模型,有助于从总体上明白SOCKS工作原理，如下图

| TCP/IP模型 |
|----------|
| 应用层 |
| 传输层 |
| 网络层 |
| 链路层 |

| OSI模型 |
|-------|
| 应用层 |
| 展示层 |
| 会话层 |
| 传输层 |
| 网络层 |
| 数据链路层 |
| 物理层 |

上边两种模型，这是总所周知的两种计算机网络模型，TCP/IP模型的应用层对应OSI的前三层，网络接入层对应OSI的最后两层，计算机在进行网络连接，请求方会有一个数据封装的过程，接收方会有一个数据解封的过程。

SOCK5是属于TCP/IP模型中应用层的协议，因此从以上的网络连接过程，就可以理解基于SOCKS 5协议的请求由客户机到代理机的整个过程如下：

- 将用户数据添加SOCKS 5头部，发到传输层；
- 传输层将SOCKS 5协议数据分段，添加TCP/UDP协议数据发到网络层；
- 网络层将TCP/UDP协议数据添加IP协议头，发往链路层；
- 链路层添加帧头与尾，将数据封装成帧发往代理机。

基于TCP的SOCKS 5

SOCKS 5 协商

Note:

这个协商其实就是确认客户端与服务端确定验证方式的一次交互。

Client发送身份/方法选择，协议头格式：

| VER | NMETHODS | METHODS |
|-----|----------|---------|
| | | |

- VER: 版本, SOCKS 5为0x05
- NMETHODS: METHODS部分的长度
- METHODS: 是客户端支持的认证方式列表, 每个方法占1字节, 目前支持如下:
 - 0x00: 不需验证
 - 0x01: GSSAPI
 - 0x02: USERNAME/PASSWORD
 - 0x03~0x7F: IANA分配
 - 0x80~0xFE: 私人方法保留
 - 0xFF: 不接受的方法, 也就是未定义/错误

Note:

1. 一般来说具体的实现应该实现GSSAPI与USERNAME/PASSWORD
2. GSSAPI:通用安全服务应用程序层, 能够使程序员在编码过程中实现通用安全, 具体来说就是不用针对特定平台、特定安全机制、特定的保护形式以及特定的传输协议去进行安全实现, 也就是说程序支持GSSAPI就可以说该程序是满足网络安全的, 比如说支持公钥加密形式。
3. IANA: 互联网地址编码分配机构, 1)域名。IANA管理DNS域名根和.int, .arpa域名以及IDN (国际化域名) 资源.2)数字资源。IANA协调全球IP和AS (自治系统) 号并将它们提供给各区域Internet注册机构。3)协议分配。IANA与各标准化组织一同管理协议编号系统。

Server选择方法, 协议头格式:

| VER | METHOD |
|-----|--------|
| | |

- VER: 版本, SOCKS 5为0x05
- METHOD: Server从Client发送过来的METHODS中选择的方法

Note:

接着Client与Server进行具体方法的子协商, 但是若Server选择的方法为0xFF, 则Client必须断开连接

SOCKS 5 数据传输

Client请求, 协议头格式:

| VER | CMD | RSV | ATYP | DST.ADDR | DST.PROT |
|-----|-----|-----|------|----------|----------|
| | | | | | |

- VER: 协议版本, SOCKS 5为0x05
- CMD: CMD是命令码
 - 0x01 CONNECT请求
 - 0x02 BIND请求
 - 0x03 UDP转发
- RSV: 0x00 保留字段
- ATYP: 地址类型

- ○ 0x01 IPV4
- ○ 0x03 域名
- ○ 0x04 IPV6
- DST.ADDR: 目标地址(即想访问的地址)
- DST.PORT: 目标地址的端口

Note:

1. 代理及会根据DST.ADDR与DST.PORT这两个字段来请求目标主机
2. 关于CMD字段:

- CONNECT:是指TCP代理模式
- BIND:指双向连接, 比如FTP协议, 一个连接用于发送命令指令, 另外一个连接用于传输数据
- UDP:指UDP代理模式

Server回应, 协议头格式:

| VER | CMD | RSV | ATYP | BND.ADDR | BND.PORT |
|-----|-----|-----|------|----------|----------|
| | | | | | |

- VER: 版本, SOCKS 5为0x05
- REP: 回应字段
 - ○ 0x00 成功
 - ○ 0x01 socks服务器错误
 - ○ 0x02 未允许的连接
 - ○ 0x03 网络不可达
 - ○ 0x04 主机不可达
 - ○ 0x05 连接拒绝
 - ○ 0x06 TTL过期
 - ○ 0x07 命令码不支持
 - ○ 0x08 地址类型不支持
 - ○ 0x09~0xFF 未分配
- RSV: 0x00 保留字段
- ATYP: 地址类型
 - ○ 0x01 IPV4
 - ○ 0x03 域名
 - ○ 0x04 IPV6
- BND.ADDR: socks服务器绑定地址
- BND.PORT: socks服务器绑定端口

Note:

Client会根据BND.ADDR与BND.PORT这两个字段向代理Server发送请求
保留字段必须设置为0x00

基于UDP的SOCKS 5

基于UDP的请求和响应头是一样的, 唯一不同的是Client的请求地址和端口字段是BIND.ADDR和BIND.PORT, 格式如下:

| RSV | FRAG | ATYP | DST.ADDR/BIND.ADDR | DST.PORT/BIND.PROT | DATA |
|-----|------|------|--------------------|--------------------|------|
| | | | | | |

- RSV: 0x00 保留字段
- FRAG: 当前分片ID, 0x00表示不分片
- ATYP: 地址类型
 - 0x01 IPV4
 - 0x03 域名
 - 0x04 IPV6
- DST.ADDR/BIND.ADDR: 目标主机/绑定主机
- DST.PORT/BIND.PORT: 目标端口/绑定端口
- DATA: 用户数据

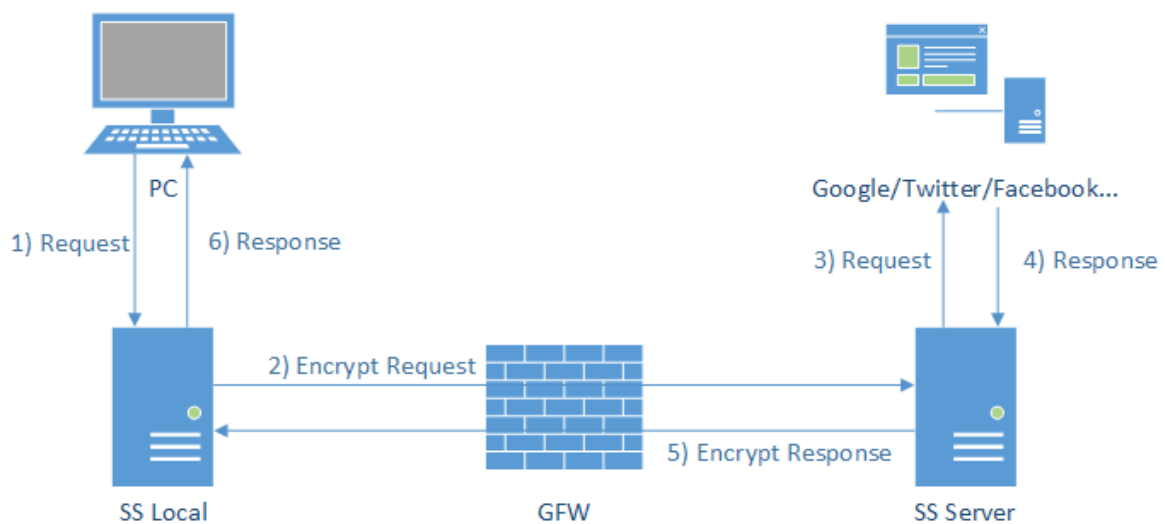
Note:

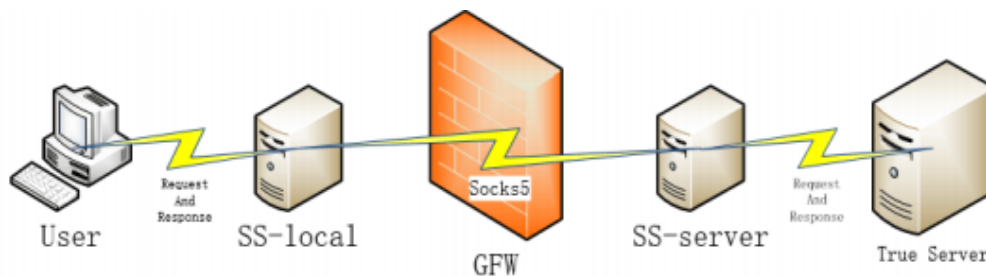
1. 代理及会根据DST.ADDR与DST.PORT这两个字段来请求目标主机, Client会根据BND.ADDR与BND.PORT这两个字段向代理Server发送请求。
2. FRAG字段用于标识是否进行分片, 如果进行分片, 数值越大表示分片排序越靠后, 如果值为0x00则表示不分片, 也就是说分片的order是从1开始的, 表示范围为1~127。如果分片的话, 每个接收者必须实现一个用于重新组长的队列 (REASSEMBLY QUEUE) 和一个用于标识过期的计时器 (REASSEMBLY TIMER) 当有分片被丢弃时, 队列应该重新初始化。
3. 地址类型不同, 每个UDP报文的大小应该有所限制, 以下说明都是方法独立的, 按每次来算:

- ATYP为0x01时, 小于10个字节
- ATYP为0x02时, 小于262个字节
- ATYP为0x03时, 小于20个字节

下面介绍ss整体架构与实现原理:

shadowsocks





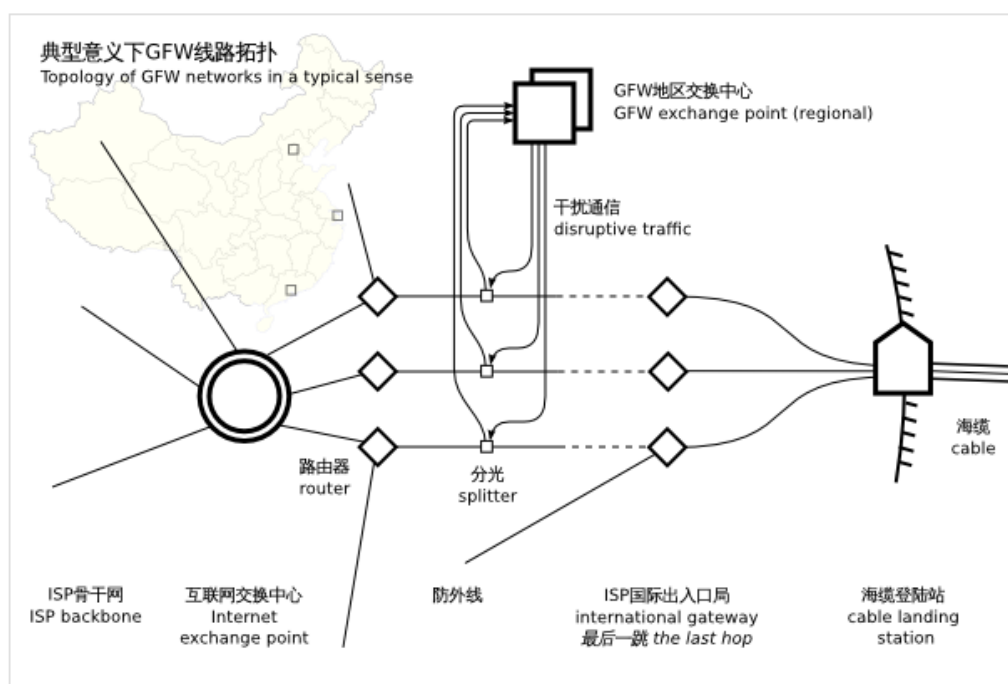
- 本地的sslocal: sslocal对于Socks 5客户端便是Socks 5服务器,对于Socks 5客户端是透明的, sslocal完成与Socks 5客户端所有的交互。
- 远程的ssserver: ssserver对于目标主机同样也是Socks 5服务器, 对于目标主机是透明的, 完成Socks 5服务器与目标主机的所有操作。
- sslocal-ssserver:sslocal接收到Socks 5客户端发送的数据, 会将数据加密, 并将配置信息发送到ssserver, ssserver接收到配置信息进行权限验证, 然后将数据进行解密, 然后将明文发往目标主机; 当目标主机响应ssserver, ssserver将接收到的数据进行解包, 并将数据加密, 发送到sslocal, sslocal接收到加密后的数据进行解密, 再发送给Socks 5客户端, 这就完成了一次交互。

SS协议中**没有任何控制流**, 本地代理获取用户原始TCP/UDP数据包获取之后会直接取出Data部分, **重新构造一个IP数据包**

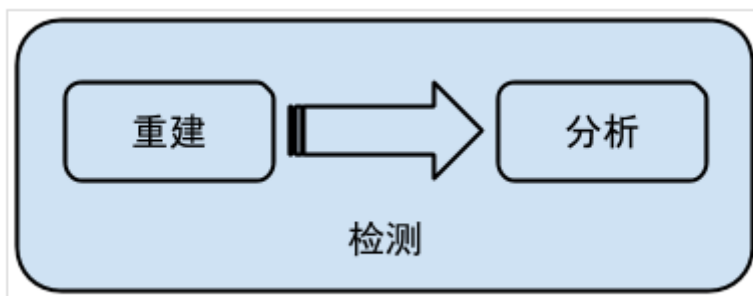
0x02GFW原理

黑盒系统 无从知晓 以下内容 纯属杜撰

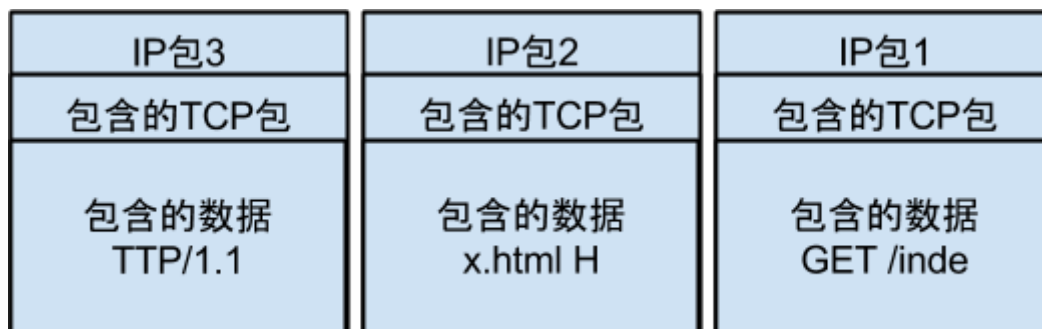
技术手段



GFW作为一个设备用“分光”的方式挂在主干路由器旁边做入侵检测主要经过重建和分析、处置三个步骤



重建



简单来说就是入侵检测系统中常用的包重建,从而获得用户完整的访问流量。

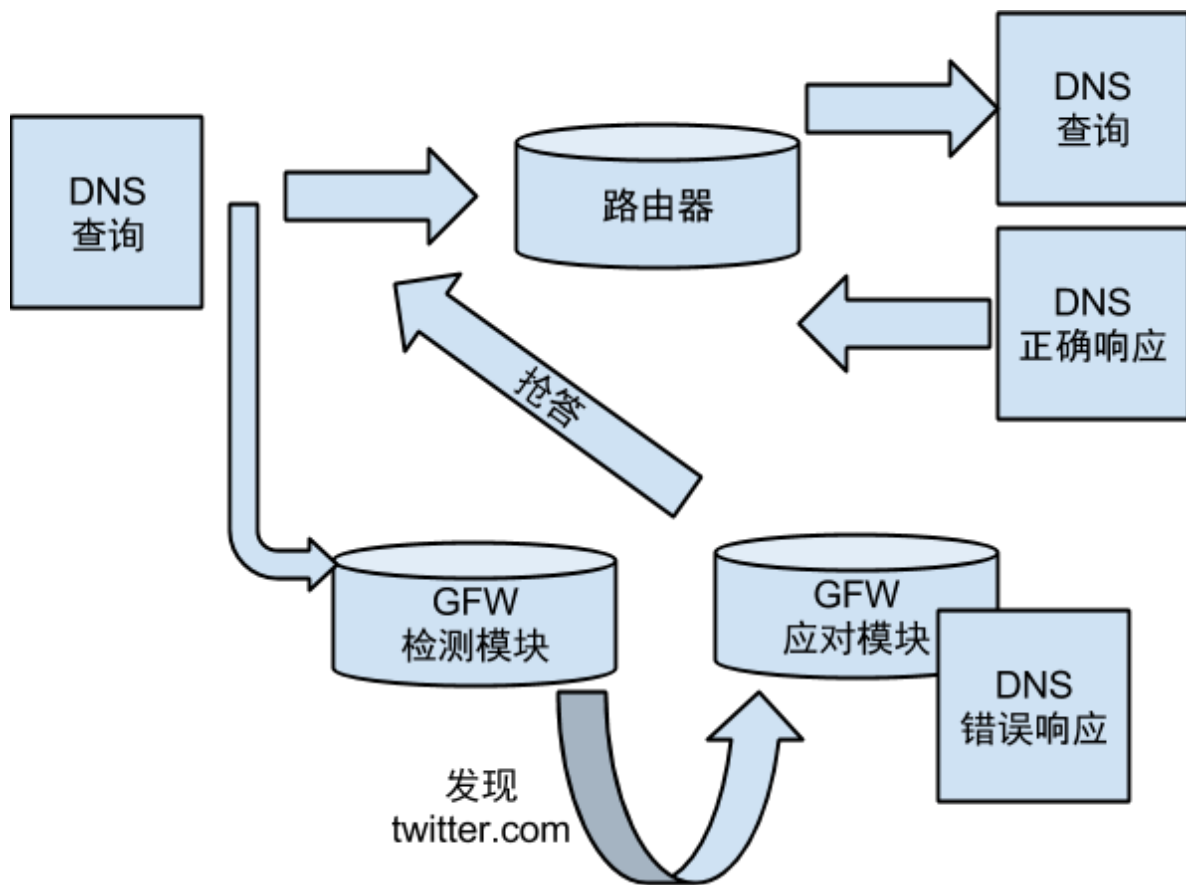
分析与处置



- DNS污染/劫持

在进行域名访问时，首先会将域名通过dns解析为对应的真实IP，然后通过IP进行HTTP访问，所谓DNS攻击手段，即通过某种手段使得客户机发起DNS查询但得到的却是错误的IP，导致客户机无法正常访问。

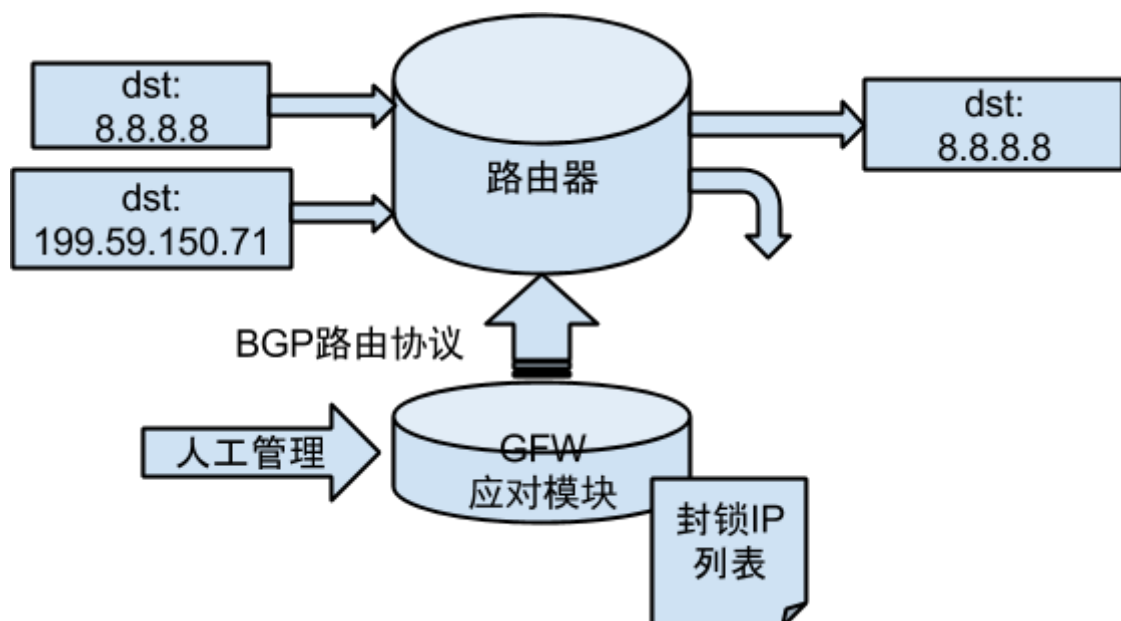
防火长城会在骨干网出口的53端口进行IDS入侵检测，检测到黑名单域名等，会伪装成域名服务器向客户机发送虚假的回应，由于DNS查询请求一般是基于UDP无连接传输层协议，该协议特征是无状态连接、不可靠传输，DNS查询会接收最先到达的请求，抛弃之后到达的请求，因此导致客户机被欺骗，请求被重定位到虚假IP。



- IP封锁

在客户机发送请求到服务器的过程中会经过一系列路由的转发，在路由器转发的过程中会根据路由表中存储的表项来决定下一跳的路由器或主机，选择的下一跳地址会根据路由协议来决定。

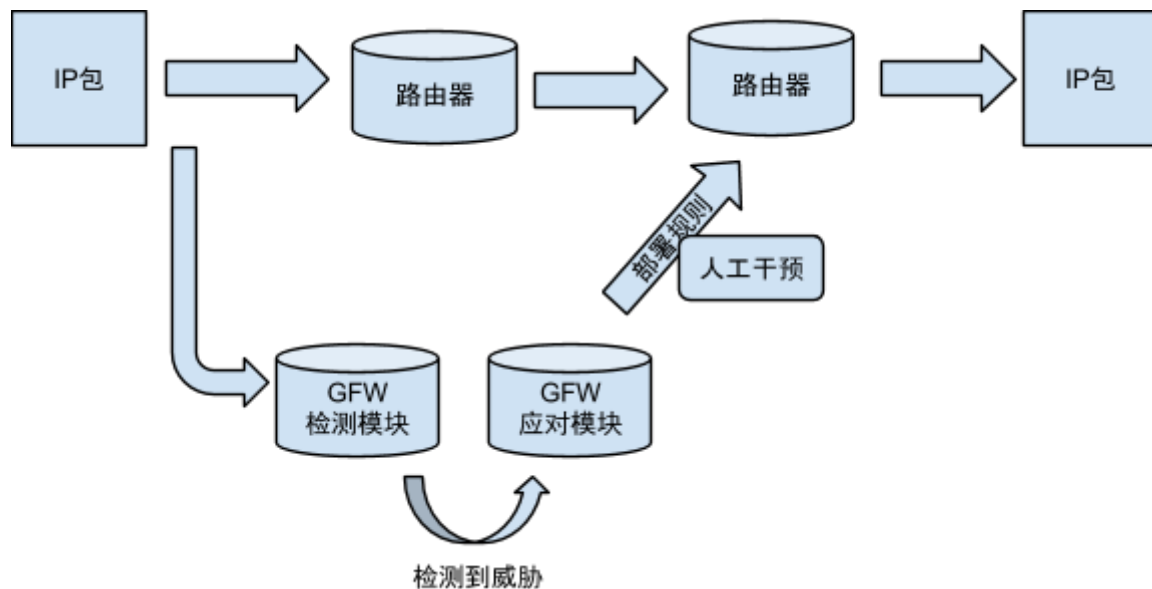
早期使用的是ACL（访问控制列表）来进行IP黑名单限制，现在更高效的路由扩散技术来进行对特定的IP进行封锁。早期路由器都是采用静态路由协议，每一条路由需要进行人工来配置路由表项，或者配置一些策略，在决定路由转发，这时可以通过检测，对相应要封锁的IP配置一条错误的路由，将之牵引到一个不做任何操作的服务器（黑洞服务器），此服务器所要做的就是丢包，这样便无声息封锁掉了。动态路由协议的出现可以更高效率的进行屏蔽，动态路由协议可以让路由器通过交换路由表信息来动态更新路由表，并通过寻址算法来决定最优化的路径。因此可以通过动态路由协议的路由重分发功能将错误的信息散播到整个网络，从而达到屏蔽目的。



- IP/端口黑名单

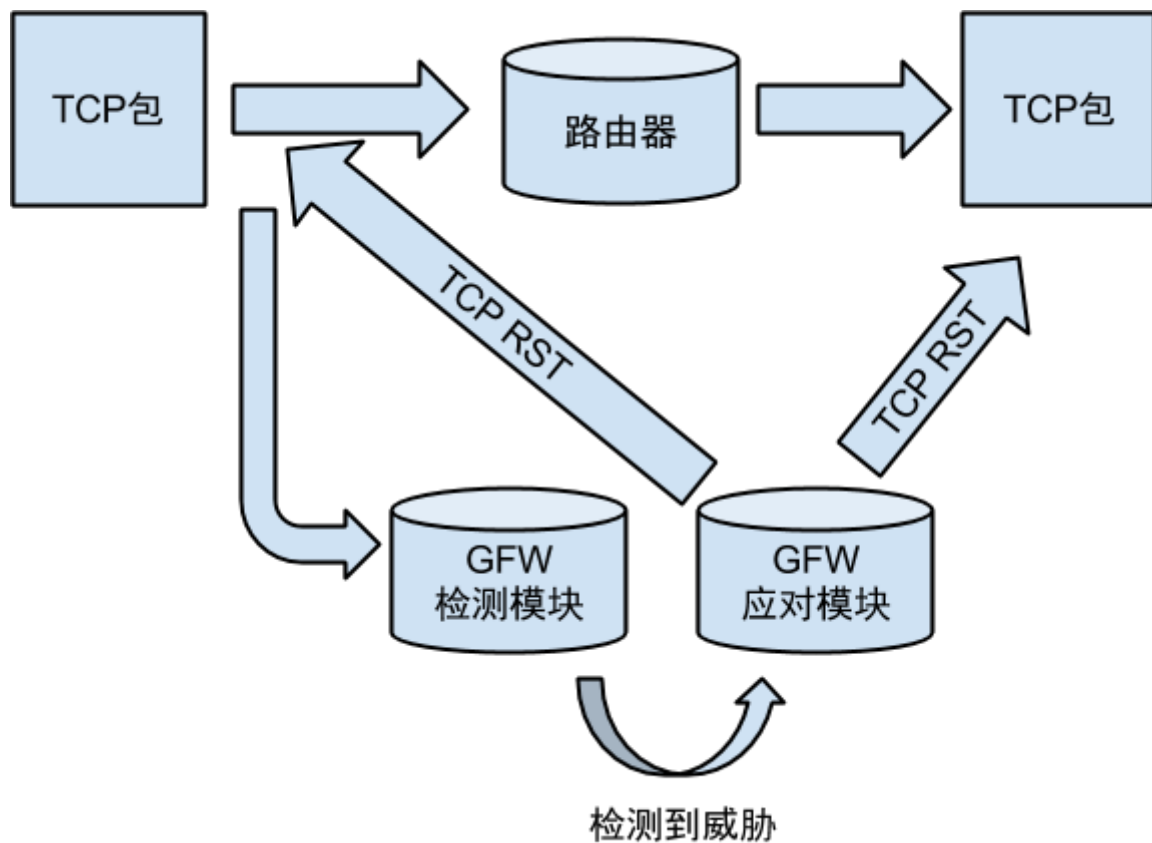
该手段可以结合上边提到的IP封锁技术，将封锁精确到具体的端口，使该IP的具体端口接收不到请求，从而达到更细粒度的封锁。经常被封锁的端口如下：

1. SSH的TCP协议22端口
2. HTTP的80端口
3. PPTP类型VPN使用的TCP协议1723端口，L2TP类型VPN使用的UDP协议1701端口，IPSec类型VPN使用的UDP协议500端口和4500端口，OpenVPN默认使用的TCP协议和UDP协议的1194端口
4. TLS/SSL/HTTPS的TCP协议443端口
5. Squid Cache的TCP协议3128端口



- 无状态TCP连接重置

TCP连接会有三次握手，此种攻击方式利用了该特点来进行攻击，gfw会对特定IP的所有数据包进行监控，会对特定黑名单动作进行监控（如TLS加密连接），当进行TCP连接时，会在TCP连接的第二部SYNC-ACK阶段，伪装成客户端和服务端同时向真实的客户端和服务端发送RESET重置，以很低的成本来达到切断双方连接的目的。与丢弃客户机的包相比，在丢包后客户机不断的发起重试，这样会加重黑洞服务器的负担，利用TCP连接重置来断开连接，客户机也不必发送ACK来确认，这样成本就要低得多。



- TCP协议关键字阻断
该手段在无状态TCP连接重置手段之上，加入了关键字过滤功能，当协议的头部包含特定的关键字便对其连接进行重置，比如HTTP协议、ED2K协议等等。
- 深度包检测
深度数据包检测（Deep packet inspection,DPI）是一种于应用层对网络上传递的数据进行侦测与处理的技术，被广泛用于入侵检测、流量分析及数据挖掘。就字面意思考虑，所谓“深度”是相对于普通的报文检测而言的——相较普通的报文检测，DPI可对报文内容和协议特征进行检测。基于必要的硬件设施、适宜的检测模型及相应的模式匹配算法，gfw能够精确且快速地从实时网络环境中判别出有悖于预期标准的可疑流量，并对此及时作出审查者所期望的应对措施。
- 机器学习、深度学习等
- 服务器主动探测

设配探测



<https://github.com/fqrouter/qiang>

ttl探测，发现主节点有可能和GFW联动进行阻断的处置，上图为节点或GFW分布情况。

SS阻断

- 机器学习算法 如随机森林检测等 (The Random Forest based Detection of Shadowsocks Traffic)
- 深度学习 神经网络 (专利：基于长短期记忆网络的V2ray流量识别方法 申请号：201910225762.4)
- 特征

```
#!/usr/bin/env python

from scipy.stats import entropy
from scapy.all import *
import numpy as np
import dpkt

def conn(ip1, ip2, port1, port2):
    swap = False

    if ip1 > ip2:
        ip1, ip2 = ip2, ip1
        port1, port2 = port2, port1
        swap = True

    if ip1 == ip2 and port1 > port2:
        port1, port2 = port2, port1
        swap = True

    return (ip1, ip2, port1, port2), swap

def dist(str):
    p = np.zeros(256)
    for i in str:
        p[ord(i)] += 1
```

```

        return p

score = {}
blocked = {}
thres = 15
def add_score(c, x):
    if blocked.has_key(c):
        return
    if not score.has_key(c):
        score[c] = x
    else:
        score[c] += x
    if score[c] >= thres:
        print c
        blocked[c] = True

def add(c, x):
    add_score((c[0], c[2]), x)
    add_score((c[1], c[3]), x)

track = {}
def sniffer(pkt):
    ip = pkt.payload
    tcp = ip.payload
    c, s = conn(ip.src, ip.dst, tcp.sport, tcp.dport)

    if tcp.flags & dpkt.tcp.TH_SYN != 0:
        track[c] = []
    if not track.has_key(c):
        return

    if tcp.flags & dpkt.tcp.TH_FIN != 0 or tcp.flags & dpkt.tcp.TH_RST != 0:
        del track[c]
        return

    if tcp.flags & dpkt.tcp.TH_PUSH != 0:
        track[c].append((entropy(dist(str(tcp.payload))), s))
        if len(track[c]) >= 4:
            if track[c][0][0] > 4.8 or \
                (track[c][0][0] > 4.4 and track[c][1][0] > 4.2) or \
                (track[c][0][0] > 4.2 and track[c][2][0] > 4.2 and \
                 track[c][0][1] == track[c][2][1]) or \
                track[c][0][1] == track[c][1][1]:
                add(c, 1)
            else:
                add(c, -1)
            del track[c]

sniff(filter='tcp', store=False, prn=sniffer)

```

0x03 SS代码研读

tcp

以下为关键代码：

```
@shell.exception_handle(self_=True, destroy=True)
def handle_event(self, sock, event):
    # handle all events in this handler and dispatch them to methods
    if self._stage == STAGE_DESTROYED:
        logging.debug('ignore handle_event: destroyed')
        return
    # order is important
    if sock == self._remote_sock:
        if event & eventloop.POLL_ERR:
            self._on_remote_error()
            if self._stage == STAGE_DESTROYED:
                return
        if event & (eventloop.POLL_IN | eventloop.POLL_HUP):
            self._on_remote_read()
            if self._stage == STAGE_DESTROYED:
                return
        if event & eventloop.POLL_OUT:
            self._on_remote_write()
    elif sock == self._local_sock:
        if event & eventloop.POLL_ERR:
            self._on_local_error()
            if self._stage == STAGE_DESTROYED:
                return
        if event & (eventloop.POLL_IN | eventloop.POLL_HUP):
            self._on_local_read()
            if self._stage == STAGE_DESTROYED:
                return
        if event & eventloop.POLL_OUT:
            self._on_local_write()
    else:
        logging.warn('unknown socket')
```

udp

简化tcp链接即为udp，不再赘述

epoll

```
def run(self):
    events = []
    while not self._stopping:
        asap = False
        try:
            events = self.poll(TIMEOUT_PRECISION)
        except (OSError, IOError) as e:
            if errno_from_exception(e) in (errno.EPIPE, errno.EINTR):
                # EPIPE: Happens when the client closes the connection
                # EINTR: Happens when received a signal
                # handles them as soon as possible
```

```

        asap = True
        logging.debug('poll:%s', e)
    else:
        logging.error('poll:%s', e)
        traceback.print_exc()
        continue

    for sock, fd, event in events:
        handler = self._fdmap.get(fd, None)
        if handler is not None:
            handler = handler[1]
            try:
                handler.handle_event(sock, fd, event)
            except (OSError, IOError) as e:
                shell.print_exception(e)
    now = time.time()
    if asap or now - self._last_time >= TIMEOUT_PRECISION:
        for callback in self._periodic_callbacks:
            callback()
        self._last_time = now

```

循环运行关键代码如下

local

```

config = shell.get_config(True)
daemon.daemon_exec(config)
dns_resolver = asyncdns.DNSResolver()
tcp_server = tcprelay.TCPRelay(config, dns_resolver, True)
udp_server = udprelay.UDPRelay(config, dns_resolver, True)
loop = eventloop.EventLoop()
dns_resolver.add_to_loop(loop)
tcp_server.add_to_loop(loop)
udp_server.add_to_loop(loop)
daemon.set_user(config.get('user', None))
loop.run()

```

以上为除去了配置加载、日志和信号处理之后的关键代码

server

和local大同小异，不再赘述

dns

```

def _handle_data(self, data):
    response = parse_response(data)
    if response and response.hostname:
        hostname = response.hostname
        ip = None
        for answer in response.answers:
            if answer[1] in (QTYPE_A, QTYPE_AAAA) and \
                answer[2] == QCLASS_IN:
                ip = answer[0]
                break
        if not ip and self._hostname_status.get(hostname, STATUS_SECOND) \

```

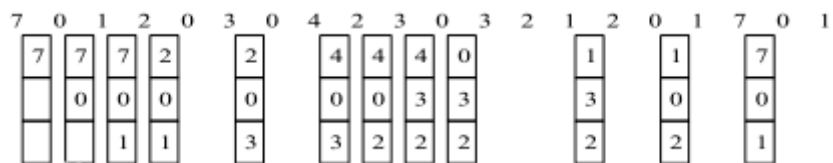
```

        == STATUS_FIRST:
        self._hostname_status[hostname] = STATUS_SECOND
        self._send_req(hostname, self._QTYPES[1])
    else:
        if ip:
            self._cache[hostname] = ip
            self._call_callback(hostname, ip)
        elif self._hostname_status.get(hostname, None) \
            == STATUS_SECOND:
            for question in response.questions:
                if question[1] == self._QTYPES[1]:
                    self._call_callback(hostname, None)
                    break

def handle_event(self, sock, fd, event):
    if sock != self._sock:
        return
    if event & eventloop.POLL_ERR:
        logging.error('dns socket err')
        self._loop.remove(self._sock)
        self._sock.close()
        # TODO when dns server is IPV6
        self._sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM,
                                    socket.SOL_UDP)
        self._sock.setblocking(False)
        self._loop.add(self._sock, eventloop.POLL_IN, self)
    else:
        data, addr = sock.recvfrom(1024)
        if addr[0] not in self._servers:
            logging.warn('received a packet other than our dns')
            return
        self._handle_data(data)

```

lru 算法



LRU是Least Recently Used的缩写，即最近最少使用，是一种常用的[页面置换算法](#)，选择最近最久未使用的页面予以淘汰。该算法赋予每个[页面](#)一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。

epoll模型

epoll是[Linux内核](#)为处理大批量[文件描述符](#)而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量[并发连接](#)中只有少量活跃的情况下的系统CPU利用率。另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。epoll除了提供select/poll那种IO事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll_wait/epoll_pwait的调用，提高应用程序效率。

https://blog.csdn.net/mango_song/article/details/42643971

0x04二次开发

- 伪装为http/https服务 确保伪装效果 不被探测到

local: http client port:4864->80 httpclient header:firefox hostserverip post:__VIEWSTATE:
data

server : http server port:80->4864 httpserver header :serverip + iis8 response:data

gfw->return response

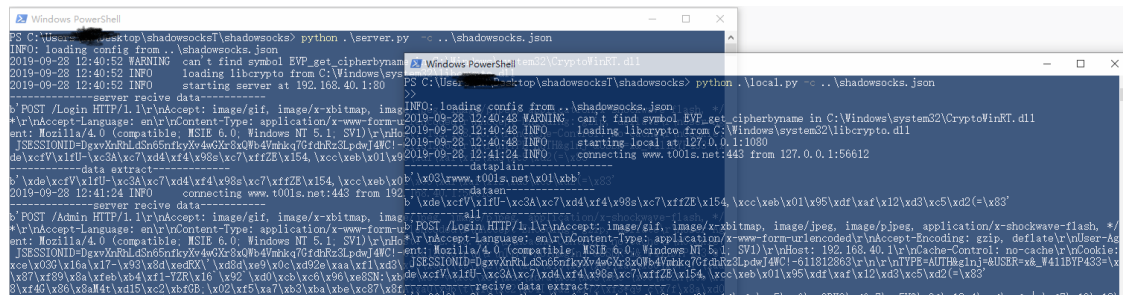
- ssr v2ray

<https://github.com/trojan-gfw/trojan>

<https://github.com/shadowsocks/shadowsocks-backup>

<https://github.com/2dust/v2rayN>

- 简单demo实现（对于https站点不友好）



<https://github.com/cn-leowong/shadowsocks-analyse>

0x05参考

ihmsc-2017-132

<http://peachey.blog/2017/12/28/py-shadowsocks/>

pam2011.gatech.edu/papers/pam2011--Xu.pdf

<http://allenn.cn/articles/2016-10/2016-10-20-learn-gfw/>

<http://gfwrev.blogspot.tw/2010/02/gfw.html>

<https://en.greatfire.org>