

# FreeRouter V2

---

完全手册

[www.lifetyper.com](http://www.lifetyper.com)

## 关于一切

这本手册的基本定性是FreeRouterV2 项目的完全手册, 包含项目的背景介绍, 涉及的原理, 使用到的技术, 以及具体的实现方法. 但也可以作为普通网民了解 GFW<sup>1</sup>工作原理, 以及与之对抗的入门手册.

我希望的结果是这本手册写完之后可以解释项目中的所有问题, 让所有人都有能力自己去做翻墙路由器, 并且解决碰到的问题, 所以我可能不得不去解释一些在“专业人士”看来很简单的基本概念, 借着 latex 的优势, 我会把文章的章节分的很细, 你可以很轻松的跳过你觉得自己很清楚的概念, 不过, 你最好确定你真的清楚.

但是考虑到”文科生的电脑知识, 基本来自大一的《计算机基础》; 而大部分非计算机专业的理工科学生的电脑知识, 来自他们从来没看懂的讲了一堆他们这辈子都没用上 X86 汇编的《计算机原理》这个残酷的事实 (这是个长句), 如果你们在阅读全书后觉得有什么内容依然是不可理解, 完全没有概念的, 欢迎反馈.

和老罗不同的是, 我不怎么爱这个世界, 所以, 本手册不推荐以下人类阅读:

1. 绝对不肯在翻墙上花一分钱, 同时还没朋友肯帮你的人, FreeRouter 里的 Free= 自由!= 免费
2. 过去三年内都没有完整看完过一本书的人, 因为你可能也看不完本书
3. 觉得我欠你很多钱或者觉得我爱你, 但实际上我根本不认识你的人

因为翻墙路由本身拥有对你网络的绝对控制能力, 而我并没有借此项目进行任何盈利的目的, 更不想把她变成另一套私人 GFW, 所以一切可能涉及到商业利益和主观判定的数据/选择/方法, 我最多可能会作为选项给你们介绍, 但不会导入 FreeRouter V2 项目本身. 基于同样的原因, 我不会向我不认识的任何人, 发布任何现成的路由器固件, 我只会教你们方法.

本手册的版权归毕勤所有, 任何人都可以自由分发和传播, 但不得做任何修改, 亦不可用于任何商业目的. 我知道在中国谈版权没什么现实意义, 但对于尊重版权的人来说, 给他们一个明确的说明是一种基本的尊重.

如果你还不知道 FreeRouter V2 的地址, 同时还不小心没注意本文开始时的链接, 那么 FreeRouter V2 当前的项目地址是:

[https://github.com/lifetyper/FreeRouter\\_V2](https://github.com/lifetyper/FreeRouter_V2)

我说了这是当前地址, 如果我觉得这个项目不再适合公开发布, 或者有其他计划都可能导致地址变更, 但是作者的 Blog 在 10 年内都不会变更地址,

<http://www.lifetyper.com>

在 Blog 上你们可以找到我的联系方式.

---

<sup>1</sup>China's Great FireWall, 中国国家防火墙简称

# 目录

<b>第一章 网络基础知识</b>	<b>1</b>
1.1 你的上网过程	1
1.2 什么是 IP?	2
1.3 什么是端口?	4
1.4 什么是 DHCP?	5
1.5 什么是域名?	6
1.6 什么是 DNS?	8
1.7 什么是 TCP 和 UDP?	9
1.8 什么是 VPN?	10
1.9 网卡/接口/适配器?	12
1.10 什么是网关?	13
<b>第二章 GFW 的工作方式</b>	<b>14</b>
2.1 DNS 劫持和污染	14
2.1.1 虚假 IP 劫持	17
2.1.2 空包劫持	18
2.1.3 轻松的扩散污染	19
2.2 敏感词过滤	20
2.3 IP 阻断	21
<b>第三章 常用的辅助工具</b>	<b>23</b>
3.1 Ping 和 TCping 命令	23
3.2 traceroute 命令	24
3.3 route 命令	25
3.4 Dig 命令	27
3.5 正则,SED,AWK	28

<b>第四章 FreeRouter V2 的技术原理</b>	<b>31</b>
4.1 IP 命令	31
4.1.1 table 概念	31
4.1.2 把数据添加到 table	32
4.1.3 让 table 数据走 VPN 接口	33
4.2 IPTables 防火墙	34
4.2.1 IPSET 概念	35
4.2.2 iptables 的 mark 功能	36
4.2.3 iptables 的 m32 模块	38
4.2.4 iptables 的 string 模块	43
4.3 Dnsmasq 解析器	47
4.3.1 IPSET 功能	47
4.3.2 为 Dnsmasq 指定 DNS	49
4.3.3 添加多个 DNS	50
4.3.4 屏蔽运营商的 DNS 广告	52
4.3.5 限定范围的解析	54
4.4 DNSSEC 的原理和实用性	55
4.4.1 简介	55
4.4.2 传统 DNS 系统的弱点	55
4.4.3 公钥/私钥加密的基本原理	56
4.4.4 DNSSEC 中公钥私钥的应用	56
4.4.5 DNSSEC 的最大问题, 是域名签名部署的缺失	59
4.4.6 对未签名的域名,DNSSEC 几乎不能提供任何保护	59
4.4.7 对签名的域名, 中间人攻击依然有可能进行	60
4.4.8 国内支持 DNSSEC 的 DNS 服务器极度缺乏	60
4.5 给 Dnsmasq 启用 DNSSEC	61
4.5.1 域名	61
4.5.2 客户端	61
4.5.3 服务器	61
4.5.4 Dnsmasq 的配置:	61
4.5.5 DNSSEC 的效果:	63
<b>第五章 FreeRouter 实战</b>	<b>64</b>
5.1 FreeRouter V2 的工作流程	64
5.2 OpenWRT 基础	66
5.2.1 OpenWRT 是什么?	66

5.2.2	我的路由能刷 OpenWRT 吗?	66
5.2.3	我的路由需要多大 ROM?	67
5.2.4	我应该下载选哪个版本的固件?	67
5.2.5	如何刷固件?	67
5.2.6	为什么刷了固件后打不开路由管理界面?	67
5.2.7	如何安装各种 package?	68
5.3	FreeRouter V2 需要的组件	69
5.4	部署 FreeRouterV2 的文件	70
5.4.1	系统的基本设置	70
5.4.2	VPN 接口设定	71
5.4.3	部署文件	73
5.4.4	调试和检查	75
5.5	自己打包固件	75

# 第一章 网络基础知识

“最强大的防火墙, 是十三亿中国人的自我审查与自我阉割”

## 1.1 你的上网过程

我需要在这里把你的上网过程先描述一遍, 以打开一个网页 (例如 `twitter.com`) 为例子, 这中间可能涉及到很多你不了解的专业术语, 但没关系, 这些我们都会在后面慢慢解释.

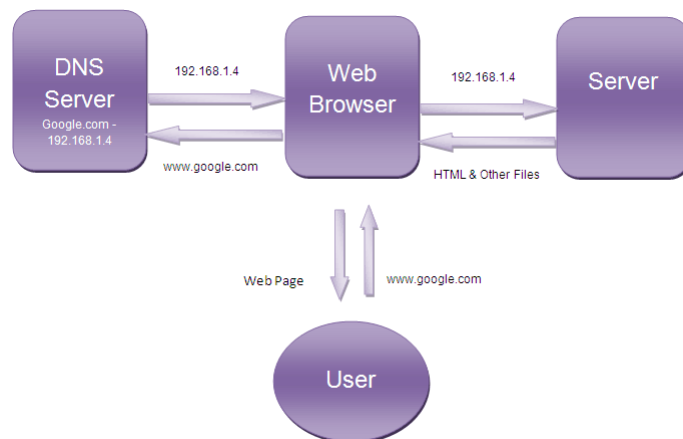


图 1.1: 浏览器如何访问网站

1. 在你打开浏览器之后, 在地址栏输入 “<https://twitter.com/star1000song>” 这个地址<sup>1</sup>
2. 浏览器会自动先识别出这是位于域名 `twitter.com` 下的一个页面, 需要访问 `twitter.com` 这个域名对应的服务器

---

<sup>1</sup>你们早晚会知道这是什么地址

3. 接着, 浏览器会调用相应的操作系统的 API 去解析这个域名, 然后你的 Windows 系统会向电脑连着的路由发送解析请求
  4. 路由器上如果之前有个域名的解析结果, 并保留在了缓存里, 就可以直接返回解析结果了
  5. 如果路由器没有关于这个域名的缓存, 那路由就会向之前在路由器里设置好的 DNS 服务器请求解析这个域名
  6. 一般的 DNS 服务器会经过一个递归解析过程, 把解析到的 twitter.com 的 IP 返回给路由, 路由再返回给你的电脑
  7. 最后, 你的电脑就可以通过 IP 地址, 直接访问 twitter.com 的服务器了, 然后向服务器的 80 端口请求 “<https://twitter.com/star1000song>” 这个地址的内容
  8. 如果网络通畅的话, twitter.com 的服务器在收到请求后就会把结果返回给你的电脑, 网页就打开了.
- 专业词汇太多看不懂?OK, 一个个解释.

## 1.2 什么是 IP?

我想这是困住很多人的第一个问题, 我觉得你们知道 twitter 是什么的可能性都比这个高.

IP, 准确的说是 IP 地址, 解释过来就是 Internet Protocol Address, Internet 协议地址. 你可以把它假想成各家各户的門牌号, 你要找一个地址, 最靠谱的方法就是知道准确的門牌号.

IP 地址又分为公网地址和内网地址两种, 这个怎么理解呢? 你可以认为长安街 1 号这种地址是公网地址, 因为通过所有人都可以走<sup>2</sup>的公共道路, 人人都可以到达这里. 而长安街 1 号里面, “2 号楼 3F302” 这个地址就是内网地址了, 因为那是别人的私有领地, 你可以通过长安街到他家门口, 但未必可以进入这个私有领地. 但私有领地的人却完全可以走出来访问公共道路上的任何其他地址, 就像你可以从你家的路由后面访问微博一样, 但他们也同样不能直接进入别人的私有领地.

实用点说的话, 你通过 ADSL 拨号, 光纤猫接入, 这些方法分配到的 IP 一般都是公网 IP, 直接接在电脑上, 别人通过这个 IP 可以访问到你的电脑了. 但是一般你都会在猫后面接个路由, 然后分配出一些类似 192.168.1.XXX 的地址, 你的电脑上往往是通过这些地址接入互联网的, 这些都是内网 IP, 隔着路由别人是无法直接访问到你的电脑的, 因为这个内网地址对外面的人来说根本不可见.

内网地址的范围是有规范的, 而公网地址绝对不能使用这些地址, 范围如下:

1. 10.0.0.0/8 – 10.0.0.0 ~ 10.255.255.255

---

<sup>2</sup>好吧, 到后面你会知道公共道路也未必人人都能走

2. 172.16.0.0/12 –172.16.0.0~172.31.255.255

3. 192.168.0.0/16 –192.168.0.0~192.168.255.255

上面的表格中, 右边的表示也许比较直观, 而左边用的掩码表示方法看起来有点费解, 但这却是实际网络应用中最常用的格式. 首先我们要知道, 一个 IP 地址是计算机中的一个 32 位的 2 进制数, 分成 4 段, 每段 8 位, 所以你看 IP 地址每段的最大值是 255, 但实际上 255 是用于表示广播地址不会分配给任何主机的, 而 0 是用于表示整个网络号的, 也不会分配给主机, 所以一般你路由能分配的 IP 最后一段肯定是 1~254(在 DHCP<sup>3</sup>支持的情况下).

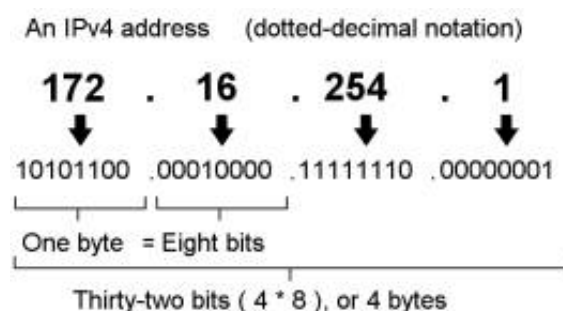


图 1.2: IPV4 的 IP 地址有 32 位

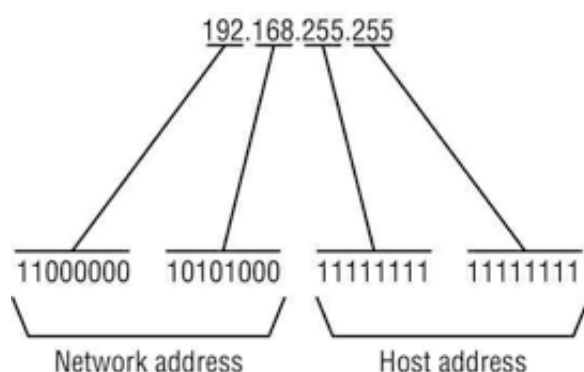


图 1.3: IP 地址包含网络号和主机号

一个 IP 地址看起来是 4 段, 其实是两个部分, 分别是网络号和主机号, 按网络号范围的不同, IP 地址分为 ABCDE5 类, DE 类为特殊地址类我们不考虑, 现在看看 A 类地址是怎么定义的:

A 类地址的网络号由一段 (8 位) 数字表示, 网络地址最高位必须是 0, 所以 A 类地址有  $2^7 - 2$  (126) 个网段; 剩下的 3 段 (24 位) 表示主机号, 每个网段有  $2^{24} - 2$  (16777214) . 类似的可以知道 B 类和 C 类地址的计算方法, 如果还不清楚可以去参考百度百科的[相关页面](#).

<sup>3</sup>这个概念我们后面会说



大概知道了 IP 的格式和分类, 掩码表示法要怎么理解呢? 我们看一下 “172.16.0.0/12” 的解释过程

1. 我们说了 0 表示网络号, 准确的说应该是 0 提示网络号, 前面的 172.16 就是网络号, 这段是固定的值
2. 最后的 12 表示采用的是 12 位掩码. 这个掩码是什么意思?
3. 12 位掩码表示 IP 地址的高 12 位和 1 进行与 (AND) 操作 (也就是不变), 剩下低 20 位的空间和 0 进行与操作, 清空出来分配 IP.
4. 也就是说, 从 172.16.0.0 开始, 分配  $2^{20}$  个 IP 地址, 算一下你就知道刚好到了 172.31.255.255
5. 我们说了 0 和 255 分别表示网络号和广播地址, 是不能分配给具体设备的, 所以实际可用范围是两者中间的值.

以上是关于 IPV4 体系下的 IP 地址的介绍, 也是我们现在基本都在用的体系. 但是你可能也看到了, IPV4 体系下 IP 地址的范围非常有限 (理论值也就是 30 多亿), 加上一些大型企业和机构占用大量地址, 全球的 IPV4 地址其实已经分配完毕 (不一定全部被使用了, 但已经被分配完毕了). 所以就有了新的 IPV6 体系, IPV6 体系下的 IP 格式大致如下:

```
2a03:2880:2110:df07:face:b00c:0:1
```

可以看到 IPV6 地址高达 128 位, 总数就是  $2^{128}$  个, 这个数量已经绝对不需要担心耗尽的可能了. 具体的关于 IPV6 的问题, 在我们的翻墙过程中只有一小部分涉及到, 这里就不再讲更多了.

## 1.3 什么是端口?

在讲上网过程 1.1 的时候, 我们提到了最后浏览器要访问的是 twitter.com 的服务器的 80 端口, 那么什么是端口呢? 专业上讲这是传输层的 TSAP 寻址方式, 不过你肯定不满意这个解释.

我们知道一台电脑要联网的软件 (进程) 其实是很多的, 这就需要给不同的软件分配不同的通道以区别他们的数据, 否则所有的进程都在一条通道上收发数据, 那么不同协议不同规范的数据毫无规则的混在一起, 就完全没有意义了, 或者代价就是每个进程都要把所有数据都先截获下来, 再慢慢提取出属于自己的那部分数据, 这显然是不可能的.

那么端口就起了这样的作用, 端口为不同的进程/协议分配了不同的端口, 例如针对浏览器的 http 服务器协议是 80 端口, ftp 数据传输是 21 端口, ssh 远程登录是 22 端口, telnet 是 23 端口, DNS 解析是 53 端口, 不同的协议使用不同的端口, 数据就很清晰了, 对于一个端口上的数据就肯定有一个统一的协议来解析了. 这就像紫禁城的 N 个城门, 每个门都有自己特殊的用途, 给不同的人 and 不同的车走一样, 这些门就是紫禁城的端口.

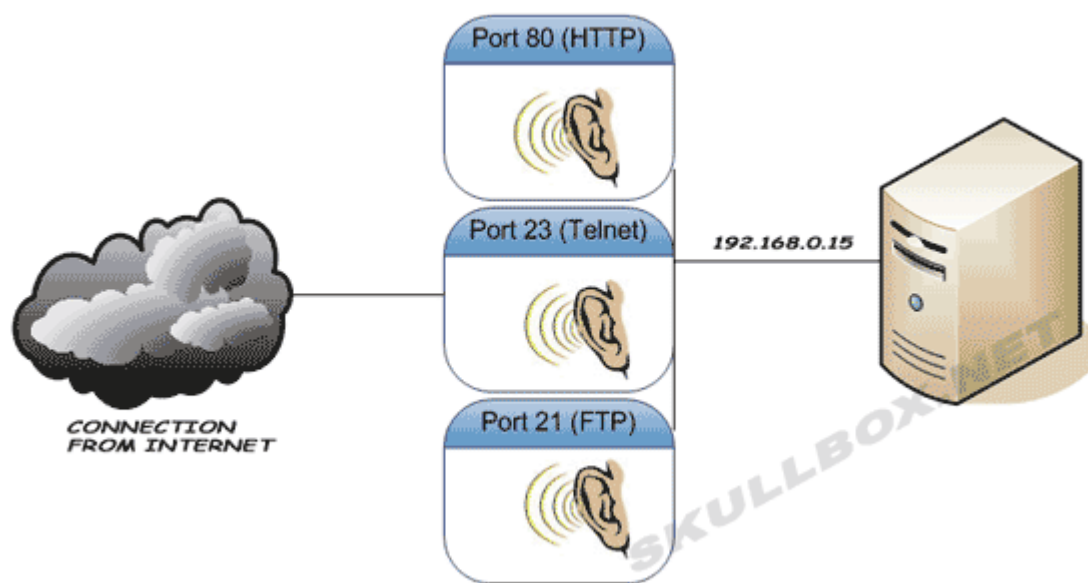


图 1.4: 不同的服务对应不同的端口

注意, 以上列出的标准端口都是针对服务器而言的, 因为很多通信协议默认都是使用这些标准端口, 你如果擅自改成别的端口, 客户端并不知晓, 就不知道该向哪里请求了. 例如有些网站给自己的内部管理服务器改成了 8080 端口或者别的端口, 这样可以避免别人发现后台的地址 (其实别人要扫描端口也可以发现), 只有知道端口的人才能访问. 按照规定 0~1023 端口都是预留给特定协议的端口范围, 其他的应用和私有的协议不应该使用这个范围内的端口, 否则可能和现有的服务协议造成端口冲突.

端口的连接是双方的, 服务器有一个端口, 客户端要连接也必须有一个对应的端口. 服务器的端口一般要固定, 以便让别人连接; 而在客户端这里, 就完全可以用各种不同的端口了, DNS 协议为了增强安全性, 客户端的端口甚至应该鼓励增强随机性, 最好每次都不同. 好比皇帝出去祭天要走特定的门不能变, 可是要微服私访就得灵活一点了, 不然行踪就容易暴露了.

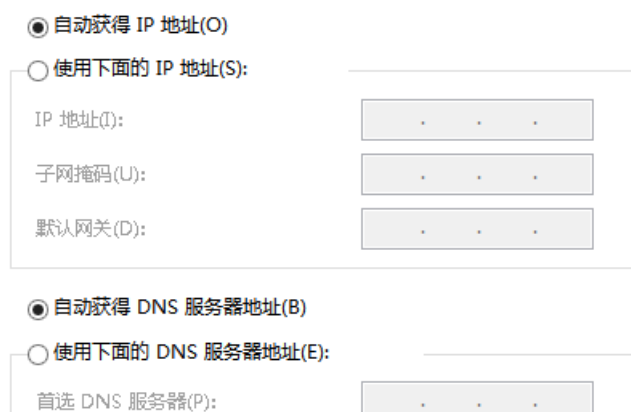
## 1.4 什么是 DHCP?

我们前面说了一个问题, 一般路由能分配的最后一段 IP 一般是 1~254, 这是在 DHCP 允许的条件下, 那么 DHCP 是什么东西?

简单来说, DHCP 是一个自动分配 IP 的系统. 例如你的电脑在不联网的时候是不会有 IP 的, 在接入一个路由器之后, 你可以自己手动设置 IP, 但这很繁琐, 而且万一两个人设置了一样的 IP, 就会导致网络冲突. 所以普遍的做法是让路由自动分配 IP 给机器.

DHCP 的实现过程大概是这样的:

1. 你的电脑接入路由后, 就向路由器下的整个网络 (所有主机和设备) 发送一个 DHCP Discover 请求, 查询看有没有 DHCP 服务器
2. 你的路由收到这个请求之后, 就反馈给你的电脑一个 DHCP Offer, 告诉它 “老子就是 DHCP 服务器”
3. 也许网络中还有别的也提供 DHCP 服务的主机 (这个情况一般比较少), 但一般来说你的电脑会选择响应最快的那个 Offer, 然后发送一个 DHCP Request 请求给他.
4. 路由收到这个 DHCP Request 之后, 就会开始和主机进行一些列握手协议, 然后发送相应的 IP 配置给他.



The image shows a Windows network configuration window. It has two main sections. The first section is for IP address configuration, with the radio button '自动获得 IP 地址(O)' (Obtain an IP address automatically) selected. Below it, the '使用下面的 IP 地址(S):' (Use the following IP address) option is unselected, and its corresponding input fields for IP address, subnet mask, and default gateway are empty. The second section is for DNS server configuration, with the radio button '自动获得 DNS 服务器地址(B)' (Obtain DNS server address automatically) selected. Below it, the '使用下面的 DNS 服务器地址(E):' (Use the following DNS server address) option is unselected, and its corresponding input field for preferred DNS server is empty.

图 1.5: Windows 通过 DHCP 自动获取 IP 和 DNS

其实呢,DHCP 不光是分配 IP 的, 他同时还会分配 DNS 服务器, 网关地址这些东西给你. 所以你看一个网卡的 IPV4 或者 IPV6 协议属性的时候,IP 和 DNS 都是可以自动获取的, 这就是通过 DHCP 实现的.

## 1.5 什么是域名?

你可能也看到了,IP 地址其实是很难记忆的一个东西,30 多亿的地址对人类的记忆来说根本就是开玩笑. 如果你要访问一台服务器, 难道还先打个电话去问他们 IP 是什么吗? 当然不可能, 所以就有了域名? 没这么快, 首先有的是 HOST 系统.

对于 windows 用户来说,c:/windows/system32/etc/drivers/hosts 文件里一般只有一行内容, 就是

“127.0.0.1 localhost”. 这个是告诉计算机, 当访问 localhost 这个名称的主机的时候, 实际就是要访问 127.0.0.1, 也就是计算机的本机 IP(访问自己). 类似的, 你可以把 “192.168.1.105 mailserver” 写入 host 文件, 这样就告诉了计算机, 邮件服务器 mailserver 的 IP 是 192.168.1.105, 这样人类的记忆就稍微轻松了一些.

可是随着互联网的发展, 很快人们就发现 hosts 文件根本就不够用, 而且很多主机你更根本不知道他的存在, 或者哪台服务器换了 IP 也没通知别人, 旧的 host 文件就一直让你去访问错误的 IP 了. 终于, 域名系统出现了.



图 1.6: 各种不同的顶级域名

基本概念上你可以把域名和 Host 文件等效起来理解, 但是域名是一个多级系统, 它有着不同的域. 例如 mail.lifetyper.com 表示 lifetyper.com 这个域下面的一台叫 mail 的主机, 这样一个域就可以有了很多主机记录, 这些主机记录我们可以叫做 lifetyper.com 这个父域的子域. 而 lifetyper.com 其实也是 com 这个父域下面的一个子域. 你不知道的可能是, com 的完整写法应该是 “com.”, 表示 com 是 “.” 这个父域下的一个子域. 这个 “.” 就是我们称之为根域的东西, .net, .org, .cn 这些顶级域名域其实都是根域的子域.

就像你用文件搜索器查找一个文件一样, 它除了告诉你文件在哪里, 一般还会告诉你文件的大小, 最近的修改日期, 文件的所有者等等. 同样的, 一个域名中往往包含了很多条记录, 们常用的几个记录有:

1. A 记录, 告诉你这个域名对应的 IPV4 IP 地址
2. AAAA 记录, 对应的是域名的 IPV6 IP 地址 (当然, 目前很多域名并没有 IPV6 地址)
3. NS 记录, 告诉你这个域名的名称服务器
4. CNAME 记录, 也叫别名记录. 这个记录是把一个域名指向另一个域名, 例如给 lifetyper.com 设置一个 abc.lifetyper.com 的 CNAME 记录, 把这个记录设置为 www.lifetyper.com, 那么访问 abc.lifetyper.com 实际上就是要去访问 www.lifetyper.com. 注意这个和单纯的跳转不同, 应该理解为转译.

上面提到了一个 Name Server 名称服务器 (以后简称 NS) 的概念, 这个 NS, 就是负责设置一个域名几乎全部记录的服务器, 并且在别人查询的时候把这些记录告诉别人, 就像一个传达室大爷. 例如他可以设置

A 记录为 8.8.8.8, 这样就把域名指向了 8.8.8.8 这个 IP 上的服务器 (服务器鸟不鸟它就是另一个问题了).

为什么说是几乎全部记录, 而不是全部呢? 因为 NS 记录他设置不了, 相反的, NS 记录设置了他. 这个 NS 记录一般只有在域名的注册商那里才可以设置, 就好像传达室大爷可以负责告诉你哪个教室美女多教务处在哪儿, 但谁当传达室大爷那是村长校长说了算的. 而且 NS 记录会同时存放在更高一级域的 NS 服务器中, 例如 lifetyper.com 的 NS 记录会存放在 com 域的 NS 服务器上, 否则一旦 lifetyper.com 的 NS 服务器自己都挂了, 你哪里查 NS 记录呢? 找不到 NS 记录就找不到 NS 服务器, 那又怎么查域名的其他记录呢?

除了多级域概念之外, 域名系统最重要的一点是不需要人们自己去记录这些域名对应什么 IP, 而是通过 DNS 系统去查询. 就好像你打开 windows 的文件搜索, 通过文件名就可以查到一个文件的所在位置一样. 那么下一节的标题你也猜到了.

## 1.6 什么是 DNS?

这是我们翻墙过程中涉及到的一个非常重要的问题, 这里只做基本介绍, 具体的很多内容后面会展开. 本书中发起最初 DNS 解析请求的设备或软件, 我们统称为 DNS 解析器, 例如电脑就是一个解析器.

我们说了, 想要知道一个域名对应的 IP, 就要向 DNS 服务器查询. DNS 就是 Domain Name Server 的简称 (域名称服务器), 你们熟知的 Google DNS(8.8.8.8) 和 114DNS(114.114.114.114) 就是这类服务器. 我们还是用一个流程例子来描述 DNS 的工作方式 (这里以递归 DNS 方式为准)

1. 你的浏览器调用 windows 的相关函数接口, 让 windows 发出了解析 twitter.com 的请求
2. 这个请求先发送到了你的路由器, 但路由器没有关于 twitter.com 的记录缓存, 就继续向路由器上设定的 DNS(假设是 8.8.8.8) 请求解析这个域名
3. 8.8.8.8 收到解析请求, 发现他也没有这个域名的记录 (虽然现实中不太可能), 他就会向根域 (也就是“.”域) 的 NS 服务器查询
4. 根域的 NS 服务器说, “我不知道, 不过 com 域的 NS 服务器知道, 这是 com 域的 NS 服务器地址 (例如是 2.2.2.2), 你去问他”
5. 然后 8.8.8.8 就向 com 域的 NS 服务器 2.2.2.2 发出解析请求, 2.2.2.2 说 “妈蛋, 我也不知道, 不过我知道 twitter.com 的 NS 是啥 (3.3.3.3), 你去问他吧”
6. 8.8.8.8 灰溜溜的跑去问 3.3.3.3, 这是 twitter.com 的 NS, 当然知道 twitter.com 的 IP 是什么, 终于大喊一声 “你咋才来啊!” 然后告诉了它 twitter.com 的 IP.

7. 然后 8.8.8.8 告诉路由, 路由告诉你的电脑, windows 告诉浏览器, 不出意外的话, 你还是没打开 twitter.com(谁让你生在中国!)

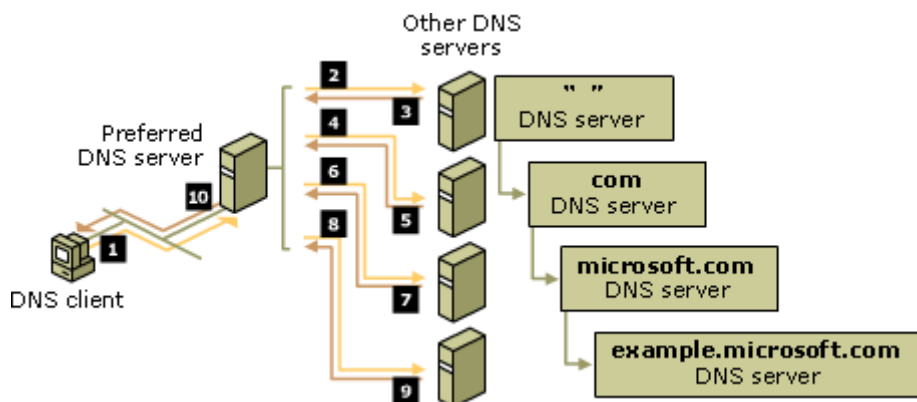


图 1.7: DNS 递归解析过程

上面这个递归解析方式, 是目前最常见的 DNS 服务器工作方式, 大部分工作都由 DNS 服务器完成. 还有一种很蛋疼的解析方式叫做迭代解析, 那么这个过程有什么不同呢?

1. 你的浏览器调用 windows 的相关函数接口, 让 windows 发出了解析 twitter.com 的请求
2. 这个请求先发送到了你的路由器, 但路由器没有关于 twitter.com 的记录缓存, 就继续向路由器上设定的 DNS(假设是 8.8.8.8) 请求解析这个域名
3. 8.8.8.8 收到解析请求, 发现他也没有这个域名的记录,, 然后蛋疼的过程开始了
4. 8.8.8.8 说 “那个啥, 地址我不知道, 这个是根域名称服务器的地址, 你自己那个啥吧....”
5. 然后你就得自己把上面 8.8.8.8 干的那些活, 那一次次被人戏弄的过程自己重复一遍, 直到查到 IP 为止

怎么样, 这活挺好干吧, 返回个根域的名称服务器地址就行了, 这样的 DNS 服务器到底有啥用? 我觉得没啥用, 因为根服务器就那几台, 我还需要你来告诉我? 所以基本已经没几个 DNS 服务器是迭代查询了, 基本都是递归解析.

## 1.7 什么是 TCP 和 UDP?

哎呀, 怎么忽然跳到这么专业的两个词汇上了? 好吧, 这可能是本章最后两个概念了, 你先忍忍吧, 我尽量用不专业的语句描述 (太专业的我也不会啊). TCP 和 UDP 都是网络传输层的概念, 是两种机制不同

的传输协议, 他们的主要区别是 TCP 是面向连接的, 而 UDP 是无连接的, 怎么理解呢?

就好像你要在 AB 两地之间互相送货, 如果你先派人去在两地之间开辟一条路线, 然后东西只会沿着这条道路上传输, 这次传输完成之后这条线路就作废, 如果还有下次送货, 还要重新开辟一条道路, 这就是有特定连接线路进行传送的, 这种方式就类似 TCP.

当然你也可以把货物交给最近的的一个人, 然后让他交给理他最近的一个人, 然后一站一站这样递交下去, 最终达到另一端. 但下次再送东西的时候, 每一次的“最近的那个人”可能就不是上次的那个人了, 整个送货的路线就可能发生了变化了, 这就是没有固定连接路线的, 这种方式就类似 UDP.

那么很明显了, 当有既定的传输路线的时候, TCP 数据会经过哪些节点是很清楚的, 要进行数据的追踪, 管理和保密也是可行的, 数据的到达也是有序的, 数据的安全性就有了保障. 而 UDP 就好像你找个快递寄东西一样, **这中间跟你玩掉包, 丢包什么的太容易了**, 虽然大部分情况下这不会发生, 但如果有人想要这么干还是很容易的.

你也许会问为什么 UDP 这么不安全还要这种协议呢? 因为 UDP 有 TCP 不能比拟的一个好处就是效率高. 它可以在每次传输时根据实际情况选择最快的路径进行送达, 而不是墨守成规走特定的路线. 就好像一般情况下你走京沪线可能很快, 可是如果一辆车走到山东的时候洪水把京沪线摧毁了, TCP 就还是傻了吧唧的继续等京沪线修复, 而 UDP 可能就会选择绕道河南从山西去北京, 虽然走一条陌生的道路可能有风险, 但如果这风险没发生的话, UDP 很可能就比 TCP 先到了.

这里我们归纳一下 TCP 和 UDP 的区别:

特点	TCP	UDP
可靠性	面向连接, 高	无连接, 低
按序传输	按先后顺序到达	随机
实现难度	需要进行数据重组恢复, 复杂	不在乎顺序, 简单
传输单位	数据流 (水管)	数据包 (包裹)
应用范围	HTTP, FTP, MAIL	DNS, TFTP, VOIP

## 1.8 什么是 VPN?

不是说好最后两概念了吗? 怎么又来一个! 我不是说了可能嘛, 章节规划这种事我从来都不擅长. 其实你想想, 做个翻墙路由如果连 VPN<sup>4</sup>是啥都不先说清楚, 可能吗?

如果你经常玩海外服的游戏, 你可能已经用过 VPN 了, 很多游戏加速器什么的其本质就是 VPN, 只不过一般他们会控制一下路由表 (算了, 路由表以后再说吧), 让你的机器只对游戏服务器走 VPN 线路, 而

<sup>4</sup>本文默认只讨论 PPTP VPN 这一种类型, 其他类型的原理不同, 但在 FreeRouter 上的应用方式基本一样



对其他服务器的访问还是走你原来的线路, 这其实已经类似一个 FreeRouter 了, 只不过翻墙的范围不是你自己决定的.

如果你认为 VPN 就是一个代理那就错了, 纯粹的 VPN 本身不能提供任何代理作用. VPN 的全名是 Virtual Private Network, 就是虚拟私有网络. 中间的实现原理我们就不管了, 其结果就是: 在客户端建立了一个虚拟的网卡, 这个虚拟的网卡和服务器的虚拟网卡之间建立了一条虚拟的专用通道.

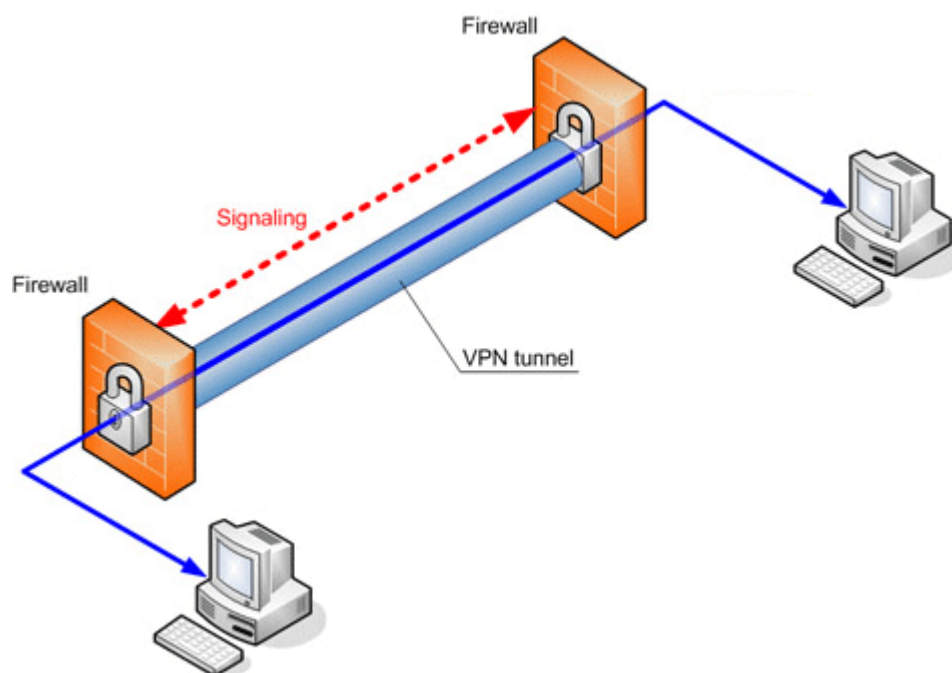


图 1.8: VPN 等效结构

所有经过这个通道传输的数据, 都会经过不同的 VPN 协议加密打包最后封装成可以被现实网络接收的数据格式, 通过现实的网络接口传达到对方之后<sup>5</sup>, 再以之前建立连接时商量好的 VPN 协议, 对数据进行拆包和解密. 那你能看出这个过程哪里像个代理吗? 当然没有.

纯粹的 VPN 配置, 就是让你和服务器之间建立了一个连接, 如果其他人再连进服务器来, 你和其他人就类似处于同一个局域网内了. 那么通过 VPN 远程访问办公室的服务器资源, 或者实现翻墙, 游戏代理是怎么实现的呢? 在 Linux 系统上, 这个是借助系统防火墙的转发功能实现的.

VPN 建立之后相当于一个局域网, 而 VPN 服务器就是这个局域网的网关, 所有局域网客户端的数据都会先发送到服务器这个虚拟网关上. 那么在防火墙规则里, 就可以把所有来自这个虚拟局域网 IP 的数据, 全部转发到一个现实的网卡上去, 如果这个网卡连接了办公室的网络, 那么你就可以通过 VPN 访问办

<sup>5</sup> 虚拟的始终是虚拟的, 只是让操作系统看起来以为有一个网卡而已, 实际在物理上并不存在



公室网络了。

如果你还是觉得 VPN 的概念太玄幻, 你就把 VPN 当做你计算机/路由上接入的**另一条**网线好了。

谈到这里我觉得又不得不再补充两个概念了, 因为你可能已经对上一段话的一些语句感到迷茫了, 放心, 这两个概念都会说得很简洁 (因为真的不简单)。

## 1.9 网卡/接口/适配器?

网卡是现实世界的概念, 是你看的见摸得着的东西。

也许你也听说过, 对于 Linux 系统来说一切事物都是文件, 同样的, Linux 系统对网卡的操作, 包括数据收发也都是通过一个网卡的文件形式化身来实现的, 这个化身在操作系统里就被叫做接口。在 Linux 终端里输入 “ifconfig” 命令就可以看到你的机器有哪些接口了:

```
TX packets:1081 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:124918 (121.9 KiB) TX bytes:124918 (121.9 KiB)

pptp-VPN Link encap:Point-to-Point Protocol
inet addr:10.10.10.4 P-t-P:10.10.10.1 Mask:255.255.255.255
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:25878 errors:0 dropped:0 overruns:0 frame:0
TX packets:18512 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:3
RX bytes:25429117 (24.2 MiB) TX bytes:2249026 (2.1 MiB)

wlan0 Link encap:Ethernet HWaddr 10:0D:7F:45:9E:58
inet6 addr: fe80::120d:7fff:fe45:9e58/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:79454 errors:0 dropped:0 overruns:0 frame:0
TX packets:452027 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:8458962 (8.0 MiB) TX bytes:284715262 (271.5 MiB)

wlan1 Link encap:Ethernet HWaddr 10:0D:7F:45:9E:5A
inet6 addr: fe80::120d:7fff:fe45:9e5a/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:473272 errors:0 dropped:0 overruns:0 frame:0
TX packets:1312275 errors:0 dropped:0 overruns:0 carrier:0
```

图 1.9: OpenWRT 路由下的接口

可以看到, 建立 VPN 连接之后, 我的路由里有一个叫 pptp-VPN 的接口, 这对路由的系统来说就是一个可用的网卡。在 windows 里面, 这个一般被叫做网络适配器, 可以通过 “ipconfig /all” 来查看。

你可以把网卡/接口/适配器看做是护照, 当你有了一国的护照, 你至少就有了在这个国家行走的权利, 在这个国家行走的权利就是这本护照对应的网络。

## 1.10 什么是网关？

我们速战速决解决这个概念, 因为真的展开起来将非常麻烦.

汉子汉字是非常美妙的 (一不小心差点出柜了), 网关这两个字其实就已经告诉了你们它的含义: 一个网络的关卡.

我们说了接口好像护照, 颁发护照的国家就是对应的网络, 但你如果想要出国, 就必须通过出入境关卡才行, 哪怕你要去的国家是免签的, 这个出入境关卡就是网关.

默认网关, 顾名思义, 就是当一个访问网络的请求没有被指定要走哪个特定网关的时候, 就走默认网关, 就好像日本人进出国门一般都走成田机场一样. 但那并不是强制要求, 谁还禁止你从东京飞洛杉矶吗.

一般来说, 当你连接上路由之后, 路由就是你的默认网关, 你所有的数据都是通过路由收发的. 如果有人进行 ARP 欺骗<sup>6</sup>让你误以为他的机器是默认网关, 那么你所有的网络数据都会通过他的机器, 对于 FTP 和 SMTP 这种连密码都是明文的协议, 他就可以在自己机器上完全截获, 轻松获得你的密码.

我们也说了, 默认网关是当网络请求没有指定网关的时候走的通路, 如果你愿意, 你也可以要求走其他的网关. 例如你可以设定访问某个 IP 的时候, 不走默认网关而走特定的网关. 当然, 同时伴随着的是, 你肯定要使用这个网关所对应的接口来使用这个网关. 例如你现在在太平洋中间的公海, 你要去看兵马俑, 你就可以拿着中国护照 (接口) 从北京出入境关卡 (网关) 进入中国; 而如果你要去看自由女神, 你就可以拿着美国护照 (另一个接口) 从纽约 (另一个网关) 入境.

你也可以在使用一个接口访问网络的时候不指定这个接口的网关, 这个时候就会使用这个接口的默认网关, 我们一般面对的都是这个情况. 而针对不同的目的地, 选择性的使用网关/接口, 这个其实就是 FreeRouter 实现针对网站 (准确的说是域名) 选择性翻墙的关键所在.

---

<sup>6</sup>这个真不能再讲了, 自己搜

## 第二章 GFW 的工作方式

“你可以很轻易的欺骗全世界，但你这辈子也骗不了自己”

在有了一些基本的网络知识之后，我们现在讲讲 GFW 的工作方式，所谓知己知彼百战不殆嘛。不过 GFW 本身是一个人人都知道它存在，却从来不会被官方承认的机构，就好像秘密警察部门一样。所以它的工作方式，各方也都只能是通过分析和模拟来大致猜测，也许已经接近真相，但永远都不等于真相。

首先你要明白 GFW 的目的是什么，它不是要阻止你访问所有的境外网站，也不是允许你访问所有的境内网站。如果一个国内网站出现它不想看到的東西，基于现在中国宽带网络的现状，网警要动手清理一个境内网站实在太容易了，不过这就不在我们的关心范围里了。话说回来，在国内开反动网站的话，这种智商的确令人心痛。

对于境外的网站，GFW 的封锁重点是：新闻，社交，政治，色情，文件共享类网站。其他几种很好理解，社交类的原因是因为阿拉伯之春的很多串联都是通过 facebook 完成的，而 twitter 又是实际上的最大的即时新闻发布站点，文件共享类网站可以让人们很方便的传播一些它不想看到的東西。

对于 GFW 来说，最有效的方式是什么？请参考第一章的题注——。

### 2.1 DNS 劫持和污染

如果从技术实现上来说，GFW 做有效的方式当然是 DNS 劫持和污染了，也许你对这个东西完全没概念，但实际上你这么多年来可能一直在被这东西祸害着。在这里先教你一个简单的命令：nslookup。

开始菜单，运行（win8 用户 win+R 键），输入“cmd”，回车。

分别用 nslookup 查一下 facebook.com(nslookup facebook.com) 和 twitter.com(nslookup twitter.com) 的域名，重复几次。是不是发现点问题？这两个网站怎么 IP 是一样的？facebook 准备收购 twitter 了？你再试试 youtube.com，嗯，你总不会觉得 Google 要把 youtube 卖给 facebook 吧。之所以会看到这种奇怪的景象，就是 GFW 的 DNS 污染在作怪。

我们前面已经说了, 要访问一个网站, 首先的步骤是通过 DNS 服务器解析这个网站域名对应的 IP, 可是如果 DNS 服务器返回给你的 IP 都是错的, 你又怎么可能访问到网站呢? 好比你想从广州去北京, 阻止你的最好办法不是半路截杀你, 而是告诉你上船一路往南走。

GFW 的 DNS 劫持是造成劫持和污染的根源!

很多人可能对 GFW 的 DNS 污染的概念始终停留在 DNS 缓存投毒这个概念上, 什么叫缓存投毒? 首先要了解什么是 DNS 缓存. 因为 DNS 系统设计的时候, 就考虑到这是一个需要频繁响应的系统, 所以一条域名记录被查询到之后, 都会在 DNS 服务器的缓存中驻留一段时间, 这样下次查询的时候就直接从缓存中读取记录而不需要再次进行递归查询了。

如果错误的记录进入了 DNS 服务器, 这个错误的记录就会被保持一段时间<sup>1</sup>, 如果在缓存过期后又有一次错误的信息进入, 那么还是继续保留这错误的记录, 这个过程如果一直重复, 服务器就永远无法返回正确的记录了, 这就成了缓存中毒. 而一般意义上的黑客的缓存投毒, 是通过生日攻击<sup>2</sup>这种方式进行的, GFW 需要用这么低级的手段吗?

黑客需要发送生日攻击, 目的是通过两边大量的数据碰撞猜到查询者发起的查询的 Transaction ID, 因为只有响应包的 Transaction ID 和请求包的一样才会被查询者接受. 但是 GFW 完全控制了中国网络的对外出口网关, 他还需要猜你的 Transaction ID 吗? 他只要直接嗅探就行了, 就好像你在自己的路由上用 TcpDump 抓取所有通过路由的网路数据一样, 既然可以嗅探到你全部的 DNS 查询数据, GFW 想要伪造一个响应包也就完全没有困难了. 除了嗅探 DNS 查询数据, 其实所有的明文传输的密码 (FTP, SMTP 等) 都是可以轻松被嗅探到的。

我们前面已经说了, DNS 是基于 UDP 协议的, 而 UDP 这种不面向连接的协议是极度缺乏安全性的, 你完全不能保证当你向一个服务器查询 DNS 记录后, 返回记录给你的真的就是那台 DNS 服务器. 我们知道, 13 套根域名服务器系统没有一套是在中国的, 中国只有镜像, 而 .com, .net 这些顶级域名称服务器在中国连镜像都没有, 而 twitter, facebook, youtube 这些域名的权威名称服务器更是毫无疑问的全部在国外, 所以 DNS 服务器在递归查询这些网站域名的 A 记录的时候, 信息一定会经过 GFW 到国外。

现在 GFW 知道了你 (某台 DNS 服务器) 要向国外的某个服务器查询 twitter.com 的 A 记录, 在国外服务器返回正确的信息给你之前, GFW 就伪造一个响应包给你, 告诉你 twitter.com 的服务器在新疆某个山沟里, IP 是什么什么. 再强调一次, UDP 包是无法判断来源的真实性的, 而 GFW 伪造的包里所有信息都符合 DNS 协议规范, DNS 服务器就接受了这个错误的记录。

那么国外的服务器就不会返回正确的信息过来了吗? 会, 一定会, 因为 UDP 虽然不安全, 但基本上还是可以送达的. 但是你别忘了, 错误的信息已经抢先进入了 DNS 服务器的缓存, 后面再过来的数据 DNS 服务器已经完全当它不存在了. 谁先到谁就是对的, 这就是扯淡的 DNS 逻辑, 而国外服务器再快, 也不可

<sup>1</sup>不光是服务器, 我们自己的机器也会对 DNS 记录有一个缓存, 可以用命令 `ipconfig /flushdns` 来清空这个缓存

<sup>2</sup>一边向服务器发动大量未知域名的解析请求, 促使服务器向上级名称服务器也发出大量请求, 一边同时伪造大量响应数据给服务器, 当数目足够大的时候, 伪造响应的 Transaction ID 就可能和服务器查询的 Transaction ID 重合, 使得伪造的响应被服务器接受

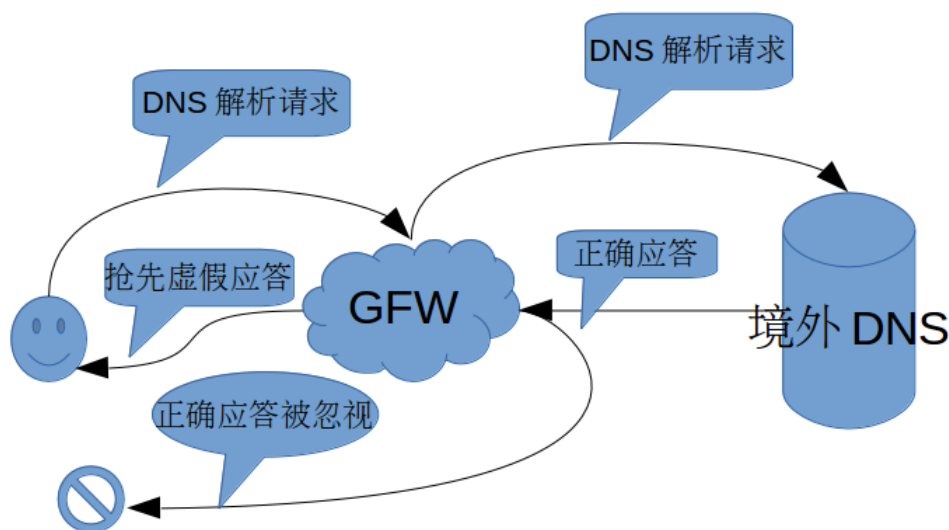


图 2.1: DNS 劫持流程

能快过守在边境的 GFW 网关, 所以你在用 8.8.8.8 配合 dig 这些工具查询域名时会发现, 也许解析 taobao.com 花了 200 毫秒, 而解析 twitter.com 只花了 30 毫秒, 那是因为这个根本就是 GFW 在国内发回给你的。

以上所描述的, GFW 抢先于境外服务器返回给你一个错误结果的行为, 就叫做 DNS 劫持。根据我的观察分析, GFW 这个劫持行为是发生在所有对国外 IP 进行敏感域名查询的时候, 不是说只有你在往 8.8.8.8 这些知名国外 DNS 查询的时候才会发生劫持, 你也不要指望说能不能找到个比较低调没被 GFW 发现的 DNS 服务器 IP 可以不被劫持, 没用的。GFW 甚至都不用判断目标 IP 是什么, 只要是通过出口网关往外面跑的, 查询敏感域名的全部劫持, 这个我可以很明确的告诉你。

说到这个什么低调的没被 GFW 发现的 DNS, 我又忍不住要说个题外话。Google 被封锁之后, 有些人知道可以通过查找未被封锁 IP 的方式访问 Google, 这本来没什么, 各有各的方法。可偏偏有些人, 自己知道了这些 IP 之后就把这些信息当成了自己的私有财产。你不肯分享给别人就算了, 还要去攻击那些分享这些 IP 的人, 说什么这会让 GFW 赶尽杀绝的。我只想这几条:

1. 没别人的分享, 你他妈哪里知道这些 IP 的? 真正自己去找到这些 IP 的人根本不会说这种话。
2. 你他妈真以为 GFW 会需要你告诉他哪些 IP 没被封? 真要想动手, 全国局域网 + 白名单模式早就可以启动了。
3. 自我阉割的人只有做太监一条路可以走, 还未必做得成。

4. Google 的 IP 从来就不是你的私有财产, 这种想法的本质, 除了自私还是自私.

好吧我骂完了, 关于 DNS 劫持的具体发生阶段和一些实例分析, 可以参考我网站上的文章:[重新理解 GFW 的 DNS 污染和劫持策略](#).

### 2.1.1 虚假 IP 劫持

那么,GFW 会返回一些什么样的错误信息给你呢? 就目前各方面统计的信息来看,GFW 返回给你的虚假信息, 其实就是篡改了域名的 A 记录部分, 把你引导错误的 IP 就行了. 准确的说, 不只是 A 记录, 对于记录 IPV6 地址的 AAAA 记录他们也是一样会劫持的, 相关信息参考本站文章:[GFW 的 DNS 劫持也会干扰 IPV6 的 AAAA 记录](#). 目前还值的庆幸的是,GFW 返回的虚假记录只用了一些特定的 IP, 我们可以通过一些手段<sup>3</sup>来过滤掉这些信息, 从而获得正确的解析结果.

目前已知的,GFW 用于 DNS 劫持污染的 IP 有 49 个, 如下表:

```
74.125.127.102
74.125.155.102
74.125.39.102
74.125.39.113
189.163.17.5
209.85.229.138
249.129.46.48
128.121.126.139
159.106.121.75
169.132.13.103
192.67.198.6
202.106.1.2
202.181.7.85
203.161.230.171
203.98.7.65
207.12.88.98
208.56.31.43
209.145.54.50
209.220.30.174
209.36.73.33
211.94.66.147
213.169.251.35
216.221.188.182
216.234.179.13
243.185.187.39
37.61.54.158
4.36.66.178
46.82.174.68
59.24.3.173
64.33.88.161
```

---

<sup>3</sup>第三章 iptables 部分会提到

```
64.33.99.47
64.66.163.251
65.104.202.252
65.160.219.113
66.45.252.237
72.14.205.104
72.14.205.99
78.16.49.15
8.7.198.45
93.46.8.89
253.157.14.165
54.76.135.1
23.89.5.60
49.2.123.56
77.4.7.92
197.4.4.12
118.5.49.6
188.5.4.96
189.163.17.5
```

关于为什么使用特定 IP 而不是随机 IP, 可能的原因有以下几点:

1. 因为劫持工作量巨大, 如果使用随机数的话, 生成随机数的计算负担会更重
2. 随机 IP 可能指向正常工作的网站, 加上这种解析请求的数量巨大, 会破坏全球网络的稳定性
3. 部分 IP 指向的是制作自由门等翻墙软件的公司, 通过这种方式实际上是对他们发起 DDOS<sup>4</sup>攻击

### 2.1.2 空包劫持

除了返回错误的信息之外, 我还观察到一次很特殊的 GFW 对境内 DNS 的空包劫持. 没错, 境内, 这也在提醒我们 GFW 不是简单的只工作在出国网关上, 而是同时也分部在全国各个骨干网络上的. 这次劫持发生在我用 114DNS 查询 instagram 的一个 CDN 节点域名 “scontent-a.cdninstagram.com” 的时候. 我们的查询包默认是要求服务器做递归查询的, 按标准服务器在返回包中也应该注明它知道了你发出了递归的请求, 可是 GFW 劫持了我向 114 的这个请求, 返回了一个不需要递归请求的响应包, 同时没有任何 A 记录, 而 Reply Code 中的错误代码是没有错误.

和不检查返回源的真实性一样,DNS 甚至都不会检查应答包中关于递归请求的部分是否发生了变化, 这样一个不需递归请求, 没有 A 记录, 没有无错误代码的结果就被 DNS 的解析器 (客户端, 我们的路由或者电脑) 接受了, 因为格式完全符合协议规范啊, 然后一个明明存在的域名就成了找不到 IP 的域名了.

---

<sup>4</sup>分布式拒绝服务攻击



关于这个问题, 具体细节参考文章[GFW 的 DNS 劫持中的空包污染问题](#). 空包劫持的问题, 因 DNS,ISP 不同有很大的区别, 且针对的域名很少, 总的来说这种行为是 GFW 的 DNS 劫持中的少数派. 而对于空包污染的过滤, 规则很容易和迭代查询的数据包匹配上, 所以会阻止本地发起的迭代查询的结果. 但我们的 OS 所发出的 DNS 请求默认都是递归请求, 基本上不用担心这个问题, 如果你在使用某些调试工具 (例如 Dig+trace 选项) 的时候发现没有数据返回, 请先禁用这个空包过滤规则<sup>5</sup>.

### 2.1.3 轻松的扩散污染

GFW 劫持搞定了所有对境外服务器发起的 DNS 解析请求的, 就从源头上保证了我们的递归 DNS 服务器只可能获得敏感域名的错误 IP. 那剩下的工作就是扩散污染了.

我们前面说过,DNS 服务器有两种工作方式<sup>1.6</sup>, 分别是递归和迭代, 迭代是一种很蛋疼的工作方式, 但现实中总还是有部分 DNS 服务器是在以迭代的方式工作的. 按理说, 这些 DNS 服务器几乎是没有什么解析能力的. 可实际上呢, 大部分情况下他们又可以对几乎所有域名的解析请求都做出响应.

那么他们自己不能做递归查询去获取域名的各种记录, 这些记录是哪里来的呢?DNS 体系里, 还有一个东西叫做 Zone Transfer. 普通的 DNS 查询, 一般只是查询域名的 A 记录<sup>1.5</sup>,DNS 服务器也只是响应一条 A 记录. 但是 DNS 中还有一种特殊的记录查询, 叫做 AXFR 记录.

当一个 DNS 服务器接收到这种记录查询的时候, 如果系统的配置没有禁止这条命令的查询, 他就会把自己服务器内所存储的全部域名的全部记录都传输给请求者, 这个过程因为数据量庞大需要进行持久的连续传输, 且安全性要求高, 一般是通过 TCP 协议传输的, 如果走 UDP 的话可以被黑客用于实现能量恐怖的 DNS 放大攻击<sup>6</sup>. 这样一个查询 AXFR 记录, 并取得另一台 DNS 服务器上所有记录的过程, 就叫做 Zone Transfer.

不过一般来说 DNS 服务器都不会支持或允许你任意查询 AXFR 记录, 否则负担会大的不可思议, 一般他们会有防火墙策略只允许特定的主机查询 AXFR 记录进行 Zone Transfer. 对于那些不支持递归的 ISP 的 DNS 服务器, 只可能是以这样的形式取得域名记录的 (别跟我说拿 U 盘拷啊).

那么好了, 支持递归解析的 DNS 服务器已经被劫持了, 而只能迭代的 DNS 服务器又只能从递归服务器那里通过 Zone Transfer 复制记录, 毫无疑问复制到的也是被污染的数据, 那么全国的 DNS 就已经被 GFW 搞定了. 你也许会说, 万一我从国外的 DNS 服务器进行 Zone Transfer 呢, 你不是说 Zone Transfer 还是走 TCP 的吗, 这一般不会被劫持了吧.

我很感激你有这么聪明, 这的确是个办法 (其实我也是临时想到的). 但是我们也说了, 绝大部分 DNS 主机不会允许你做 Zone Transfer, 这个负担太重了, 基本上你只能自己去国外架设服务器, 搜集递归数据,

<sup>5</sup>具体的规则会在后面的章节讨论

<sup>6</sup>指利用伪造 UDP 包的查询源 (被攻击者的 IP), 通过 DNS 查询数据量大的域名记录 (例如 TXT 记录), 最终让 DNS 服务器把这个庞大的响应包返回到被攻击者的过程, 而庞大的 DNS 响应一般会转以 TCP 协议传送, 这就造成了被攻击者的 TCP 端口阻塞



然后 Zone Transfer 到国内了, 所以国内确实有些私人的 DNS 是没被污染的啊. 当然我觉得他们更可能的是在服务器上写死 hosts 的方式实现的, 这种方式怎么说呢, 事实正义, 而程序不正义, 任何人为干预修改域名记录的行为我觉得本质都是劫持. 而对于各大 ISP<sup>7</sup>的 DNS 来说, 你觉得国营企业, 有这可能吗?

## 2.2 敏感词过滤

我对这种技术的研究比较少, 只能大概的介绍一下, 如果需要具体的细节, 可以在你能翻墙之后参考[这篇文章](#), 还有[这篇文章](#), 或者 wikipedia 上关于[TCP 重置攻击](#)的描述.

我们平常浏览网页所用的协议叫做 HTTP 协议, 基本上你在每个网页的开头都可以看到 “http://” 的字样. HTTP 协议本身是一个比 TCP, UDP 更高一层的 (更贴近用户) 的应用层协议, 但上层协议最终都是以封包后以更底层传输层的方式进行传输. 例如 DNS 协议是用 UDP 协议传输, HTTP 协议使用的就是 TCP 协议.

我们说 TCP 相对 UDP 来说是更为安全的, 但这只是相对的, TCP 数据依然是可以被中间人修改的. 最简单的例子, 就是你们可能用过的一些屏蔽优酷, 土豆视频广告的插件和脚本, 这些东西的原理就是拦截并修改 HTTP 数据流, 讲涉及广告的部分代码进行替换. 这说明了, TCP 协议的内容是可以被感知和识别的 (比如找到广告代码的部分), 也是可以修改的 (替换成无广告的代码). 虽然 TCP 协议不会像 UDP 那么容易让第三方伪造和修改数据, 但肯定位于你和境外服务器中间的 GFW 还是可以做到的, 没见过圣旨的情况下伪造圣旨有困难, 可是让传旨太监拿着现成的圣旨改个字还是很容易的.

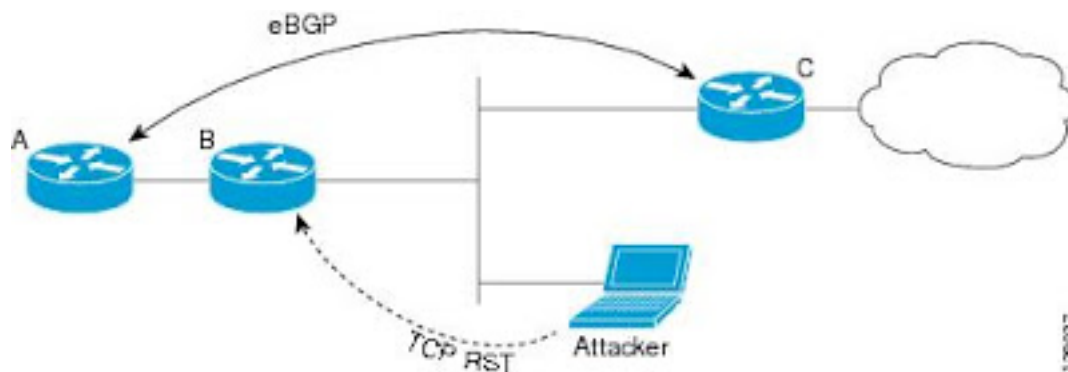


图 2.2: TCP 重置注入

HTTP 连接的建立, 有一个著名的三次握手过程: 客户端先向服务器发出连接请求, 服务器返回一个连接许可, 客户端再发起连接确认. 如果 GFW 检测到你要访问的某个地址中包含了敏感信息 (域名本身

<sup>7</sup>Internet Service Provider, 网络服务提供商, 也就是电信, 联通, 长宽, 铁通之类的

可能就是敏感信息), 他就修改或冒充服务器返回的连接许可, 告诉你服务器拒绝连接, 这样客户端就会自己放弃连接, 你们平常上网经常看到的大部分“网络连接被重置”就是这个的结果, 当然, 少数情况下可能服务器真的出问题也会拒绝连接.

你也许会问我们能不能忽略这个重置 (RST) 消息? 当然可以, 可是 GFW 比你想得更多, 他不光发送 RST 给你, 还发送 RST 给服务器, 这样服务器也会认为你已经断开连接了, 就不再发送数据给你了.

GFW 这种做法的优势很明显: 不需要对完整的 HTTP 页面数据进行检测, 只要发现 URL(网页地址) 中有敏感信息, 就直接断开了你和服务器之间的连接, 大大减轻了他自己系统的负担. 但这种做法的缺点也很明显, 检测力度太弱了. 大家最熟悉的例子莫过于 SexInSex 和草榴这些色情网站, 你直接输入他们的主站域名一般都是连接被重置, 但只要找个其他的 IP 替代域名, 一点问题都没有, 因为 GFW 对这些网站检测的敏感信息仅仅是 URL 中的域名, 在你把域名替换为不敏感的 IP 之后, 你查看的页面里面有 100 个“无码”200 个“3P”它也完全不在乎.

但不是说 GFW 对敏感信息的检测仅限于域名, 对于一些数据量很大, 内容庞杂的网站, GFW 的 URL 检测会精确到页面. 例如你在 wikipedia 上查“种猪饲养技术”, 党和国家会为多了一个纳税人感到高兴而绝对不拦你, 可是如果你要查“八九年天安门学运”, 那 90 秒内你就别想访问 wikipedia 了.

我们也经常碰到另一种情况, 就是某些网站使用的是 HTTPS 协议, 似乎在使用这些网站的时候我们从来不会碰到页面连接被重置的问题, 这是为什么呢? 这是因为 HTTPS 采用了公钥 + 私钥<sup>8</sup> 加密技术, 数据在封包成 TCP 包之前就已经被加密过, 敏感的信息被加密之后就是一堆无法解读的乱码, 这些信息对于 GFW 来说是无法解读的, 或者说解读成本是巨大的, 这就使得一直到今天 (2014 年 7 月 30 日), 我们也没有看到哪个 HTTPS 页面被 GFW 连接重置.

## 2.3 IP 阻断

这应该是我所知道的 GFW 的最后绝招了.

我们前面说了 GFW 的手段有 DNS 污染, 敏感词检测重置, 如果一个网站不用域名直接用 IP, 或者有很多方法可以让别人知道它域名的真实 IP(草榴在这方面就很出色嘛), 同时页面还都是 HTTPS 形式加密过的, GFW 就没办法了? Naive! 没办法骗你去南半球看兵马俑了, 骗你说紫禁城被拆迁了你也不相信了, 老子就把你要走的路直接给断了.

其实我觉得我应该在第一章补充一个路由概念的, 好吧, 就在这里说吧.

首先我们要知道, 网络上的任意两个 IP, 他们之间建立连接几乎都不可能是直连的, 如果任意两个 IP 都可以直连, GFW 就只能裁掉全部技术人员改建爆破大队直接破门了, 就好像如果你从广州到北京是通过

---

<sup>8</sup>在下一张讲 DNSSEC 的时候会再谈公钥 + 私钥加密体系

一条直通的封闭管道进行的, 那车匪路霸早饿死了. 无论是面向连接的 TCP, 还是不面向连接的 UDP, 数据在网络上的两个 IP 之间的传送, 都是通过一级一级的路由中继完成的, 只不过 TCP 的中继路径在连接建立之后是固定的, 而 UDP 的中继路径每次都不同.

例如你要从国内的 A 到国外的 B, 数据传送的实际路径可能是这样的:

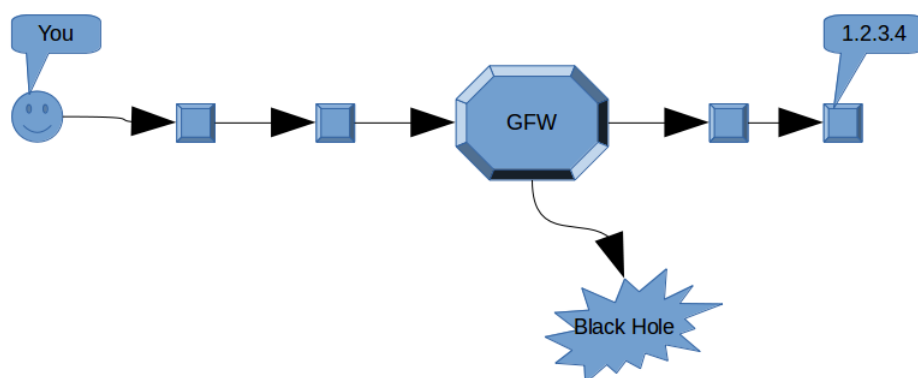


图 2.3: 静态路由丢弃

别的节点都可能会变, 但 GFW 肯定会在这条路的中间出现是不用怀疑的. 你看, GFW 这个位置不当车匪路霸都可惜了. 一个数据到达一个节点后, 下一个节点发往哪里是由一个叫路由表的东西控制的, 动态路由<sup>9</sup>表是由一套路由算法得出的, 而静态路由表则是可以人为指定的. GFW 觉得静态路由表是个好东西, 就充分发挥了它的错误利用.

假定 GFW 发现某个境外反动网站, 并且知道了他的 IP 是 “1.2.3.4”. 我们已经知道 GFW 是控制了中国的网络出口网关的, 也就是说不管你怎么访问, 不管前面后面的节点是什么, 最终都要通过 GFW 这个节点. 那么一个对 “1.2.3.4” 发起的网络请求, 在数据到达 GFW 节点之后, 根据路由表下一个节点应该发往哪里呢? GFW 猥琐地猛吸一口烟屁股, 嗯, 发往达美克星!

达美克星? 没错! 那是什么鬼地方? 不知道! 那怎么办? 爱咋办咋办! 反正这个网络请求最终的命运我是不知道, 只知道它被发送到了一个黑洞地址. 同样的, 所有发往 “1.2.3.4” 的请求全部要被发往这个根本不存在的黑洞地址. 当然你要觉得达美克星太残忍, 撒哈拉沙漠正中央也是可以考虑的选项, 总之是一个让你进得去就出不来的地址. 那么, 毫无疑问的, 国内网络通往 “1.2.3.4” 的路径就被切断了, 你知道 IP 有什么用, 部署了 HTTPS 加密又有什么用?

<sup>9</sup>虽然现在针对动态路由表的 BGP 劫持技术 GFW 也早就掌握了, 但我们先不考虑这么复杂的东西.

## 第三章 常用的辅助工具

“这个世界，没有什么比时间更宝贵的。”

再经过一些更进阶的知识准备 (你不会真以为第一章那些文科生普及文档就足够了吧), 我们很快就可以开始正题了. 注意! 本章的命令默认都是 Linux 系统命令, 因为我们最后要面对的 OpenWRT 就是一个 Linux 系统.

### 3.1 Ping 和 TCPing 命令

Ping 命令, 是一个基于网络层 (比传输层还底层) 的 ICMP(Internet Control Message Protocol) 协议的命令, 主要用于检测和目标主机之间的网络是否通畅, 以及网络的延时. Ping 命令的使用非常简单, 开始, 运行, cmd, 回车, 打开命令提示符 (我想 linux 用户应该不需要我来教你们怎么打开 terminal), “ping IP 地址”, 或者 “ping 网站域名”. Ping 命令的解读实在太人性化了, 我觉得我就没必要再解释了.

ping 命令我们一般很少加别的参数运行, windows 下默认 ping 只运行 4 次, 加上参数 “-t” 后可以让命令持续运行, 直到我们手工停止.

但是前几年, 有一种很流行的网络攻击手段<sup>1</sup>就是利用 ping 命令进行的, 所以很多主机现在都会在防火墙规则里禁止 ping 响应, 即使你可以访问网站也未必就可以 ping 通. 这种情况下我们可以利用另一个 tcping 工具, 通过对远程服务器的 TCP 端口的通畅与否进行检测, 来判断网络的畅通. tcping 在 windows 和 linux 系统中都不是内建的命令, 需要我们去下载, TCPING 的官方网站在[这里](#).

TCPing 的命令参数和使用方法和 Ping 很相似, 主要的区别是 Ping 是通过既有的 ICMP 协议进行, 而 TCPing 是通过 TCP 协议进行, 所以可以再最后加入一个端口参数, 对服务器的不同端口进行探测, 例如网页服务器默认是探测 80 端口 (不指定端口时默认就是 80), 而邮件服务器可以探测 SMTP 的 25 端口.

基本上对于一个网站来说, 80 端口是不可能关闭的, 所以我们一般会用 TCPing 来代替 Ping 对本地到服务器之间的网络通断进行探测.

---

<sup>1</sup> ping flood

## 3.2 traceroute 命令

我们在上一章最后一节2.3提到另一个路由表的概念, 我们说了路由表决定了两个 IP 之间的连接最终是通过那些节点连接起来的. 如果我们想要查看两个 IP 之间的连接实际经过了哪些节点, 就可以借助 traceroute 命令来查看. traceroute 命令的使用非常简单, “traceroute+ 目标 IP” 然后就会开始追踪出你和目标 IP 之间经过的路由节点了, 当然你也可以直接用 “traceroute+ 域名” 的方式, 前提是你执行 traceroute 命令的机器有个正常工作的 DNS 系统, 能正常解析域名.

在 windows 下, 有一个等效的命令叫做 “tracert”. 如果你有兴趣的话, 可以去[这个页面](#)看看 traceroute 的详细解释. 我们在 windows 下用 tracert 命令跟踪一下 twitter.com 试试:

```
leavi_000@Cartman-Asus /home/leavi_000
$ tracert twitter.com

通过最多 30 个跃点跟踪
到 twitter.com [199.59.148.82] 的路由:

 1  <1 毫秒    <1 毫秒    <1 毫秒 OpenWrt.lan [192.168.199.1]
 2   99 ms     98 ms     99 ms  10.10.10.1
 3  100 ms     *          99 ms
 4   99 ms    106 ms    107 ms 124.215.199.125
 5  100 ms    100 ms    117 ms otejbb205.int-gw.kddi.ne.jp [124.215.194.161]
 6  200 ms    201 ms    200 ms pajbb001.int-gw.kddi.ne.jp [203.181.100.134]
 7  374 ms     *          286 ms ix-pa4.int-gw.kddi.ne.jp [111.87.3.66]
 8  194 ms    196 ms    194 ms 199.16.159.189
 9  203 ms    200 ms    208 ms ae51.smf1-er1.twtr.com [199.16.159.29]
10  217 ms    218 ms    217 ms r-199-59-148-82.twtr.com [199.59.148.82]

跟踪完成。
```

图 3.1: 通过翻墙路由的 tracert

第三行的目标 IP 因为是我 VPN 服务器的 IP, 所以抹黑隐藏掉. 可以看到, tracert 会对每一个节点发出 3 次探测包, 并检查每一次节点响应的速度, 但这三次探测并不一定每一次都会有响应, 于是就出现了星号 “\*”. 一般来说, 如果到一个节点之后出现了 3 个星号, 就意味着到这个节点之后网路已经中断了 (大部分是被 GFW 通过静态路由丢弃了). 但也有可能是某些节点因为自身的设置不响应 tracert 的探测, 碰到这种情况继续等待一下, 如果在等待过几个节点之后探测信息又恢复了, 那说明只是个别节点不响应而已, 但网路还是通的.

而对于没有 VPN 转发的普通中国网路来说, 到 twitter 的通路一般都是断的, 结果会是这样:

```
root@OpenWrt:~# traceroute twitter.com
traceroute to twitter.com (199.59.148.82), 30 hops max, 38 byte packets
```

```

1  10.1.1.1 (10.1.1.1)  0.460 ms  0.430 ms  0.420 ms
2  161.187.184.117.in-addr.arpa (117.184.187.161)  3.610 ms  3.724 ms  *
3  153.125.181.221.in-addr.arpa (221.181.125.153)  2.505 ms  2.206 ms  4.212 ms
4  221.183.14.153 (221.183.14.153)  2.611 ms  221.176.19.49 (221.176.19.49)  3.642 ms  2.693 ms
5  221.176.16.206 (221.176.16.206)  31.164 ms  32.144 ms  33.038 ms
6  221.176.18.114 (221.176.18.114)  33.854 ms  34.504 ms  31.493 ms
7  * * *
8  * * *
9  * * *
10 * * *
11 * * *

```

可以看到从第六个几点之后就再也没有跟踪信息了, 通过一些查 IP 的网站我们可以看到, 最后一个节点“221.176.18.114”是中国移动的骨干网, 也就是说我们对 twitter 的访问在这个位置被路由丢弃了, 根本就达不到国外, 更不要说 twitter 的服务器了.

### 3.3 route 命令

既然 GFW 可以通过静态路由丢弃我们的网络请求, 那么我们自己可以用静态路由表来绕过 GFW 吗? 当然可以, 这就是 route 命令的一个重要作用了. Route 命令的 Man Page 在[这里](#), 有时间一定要多去看看这些命令的 man page, 我不可能把每一条都很详细的给你翻译过来. 我们说了 traceroute 是检查两个 IP 之间实际的路由状况, 而 route 命令就是查看当前主机上已有的路由表的命令.

直接输入 route, 我们就会看到类似如下的信息 (VPN IP 隐去):

```

root@OpenWrt:~# route
Kernel IP routing table

```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
default	10.1.1.1	0.0.0.0	UG	0	0	0	eth0.2
10.1.1.0	*	255.255.255.0	U	0	0	0	eth0.2
10.10.10.1	*	255.255.255.255	UH	0	0	0	pptp-VPN
106.XXX.XXX.XXX	10.1.1.1	255.255.255.255	UGH	0	0	0	eth0.2
106.XXX.XXX.XXX	10.1.1.1	255.255.255.255	UGH	0	0	0	eth0.2
127.0.0.1	*	255.255.255.255	UH	0	0	0	pptp-VPN
192.168.199.0	*	255.255.255.0	U	0	0	0	br-lan

分栏解释: Destination 是目标地址, default 就是默认的所有地址. Gateway 是网关我解释过了, GenMask 是掩码, 其实我也解释过了. Iface 是对应的接口名, 我也解释过了. Metric, Ref 和 Use 在很多内核中都不使用, 我们不用管. Flags 里的字母含义分别如下:

1. U, 表示这个路由活着 (Up)



2. H, 表示这货是台主机 (Host), 在网络中主机和路由是不同的
3. G, 表示要通过网关访问 (Use Gateway)

上面这个例子中,10.1.1.1 是我路由接入的默认网关, 我默认的所有网络 (default) 都是通过这个网关访问的, 这个网关对应的接口是 “eth0.2”. 而 192.168.199.0/24 是我路由后面分配出来的内网网络, 我在这个内网里互访主机是不需要经过网关的, 所以这里没有 G.

当然我们主要的目的不是查看路由表, 而是修改路由表. 我们可以看到上面的 “10.10.10.1” 是 VPN 服务器对应的网关, 我们到 VPN 服务器的网络是通畅的, VPN 服务器到 twitter 的网络也是通畅的, 那么我们就可以通过 VPN 网关来访问 twitter 了. 最简单粗暴的做法是直接把我们默认接口改成 VPN 接口, 这样我们所有的网络请求都会通过 VPN 接口进行了, 只要 GFW 不拦截我们到 VPN 的通路, 那就没什么可以阻拦我们了.

```
route add default dev pptp-VPN
```

这里的 pptp-VPN 要根据实际情况来确定, 我们在前面说过可以通过 ifconfig 命令来查看各个接口的名称, 但我们也可以通过一些脚本来自动提取这个信息, 这个可以用后面提到的 SED 配合正则表达式很轻松的实现. 如果我们不需要让所有网络都走 VPN 了, 只要删除这条路由表就行了, 命令很简单只要把 “add” 改成 “del 就行了”.

```
route del default dev pptp-VPN
```

如果让路由上的所有网络访问都走 VPN, 其实和我们直接在本机挂全局 VPN 也没什么区别, 唯一的区别就是可能方便了一些不能拨 VPN 的设备使用, 节约了一些连 VPN 的动作时间而已. 一种更精确的做法, 是指针对被 GFW 屏蔽的 IP 走 VPN, 这也是 **FreeRouter V1** 的做法. 例如我们假定 twitter.com 的 IP 是 5.5.5.5, 只需要执行如下命令即可:

```
route add -host 5.5.5.5 dev pptp-VPN
```

这个 -host 是默认选项, 不写也可以, 指的是添加一个 IP 作为目标地址. 有的时候一些网站有很多个 IP, 但一般都是处于同一个网段内, 例如我们假设 google 的 IP 范围是 6.6.6.1~6.6.6.254, 我们就用 -net 选项表示添加一个网段作为目标地址:

```
route add -net 6.6.6.0/24 dev pptp-VPN
```

具体的掩码要写多少可以根据实际情况调整, 但很多时候我们不知道确切的范围, 只好把整个 C 地址网段全部加入 VPN 路由表了. 由此可以看出, FreeRouter V1 一个很大的问题就是 IP 的搜集非常困难, 如果发生 IP 变动的話更是难上加难. 多加了关系, 顶多是多消耗一点 VPN 的流量; 如果漏加了就非常麻烦了, 对于 dropbox 这些靠客户端形式工作的东西来说, 除了抓包都没有直观的方式可以获取服务器的 IP. 这也就是为什么我会最终停止 V1 的更新, 全力投入 V2 的维护的主要原因.

## 3.4 Dig 命令

我们前面讲了如何用 traceroute 来检查网络中的路由跳跃情况, 现在来讲讲如何用 Dig 命令检查域名的解析情况.

在 windows 和 linux 系统中, 有一个很基本的域名解析工具叫 nslookup, 但相对而言 dig 的功能和返回的信息要强大得多. 在[这里](#)下载 bind, 就可以获得 dig 工具了. 你们看一下[Dig 的 Man Page](#)就会发现这货的参数多得吓人, 我们只挑一些可能常用的讲解. 首先讲一下 Dig 命令的基本使用格式:

```
dig @DNSIP Domain QueryType
```

例如我们要通过 Google DNS 来查 facebook.com 的 IPV4 地址 (A 记录) 就是这样的:

```
dig @8.8.8.8 facebook.com AAAA
```

返回的信息是这样的:

```
root@OpenWrt:~# dig @8.8.8.8 facebook.com A

; <<>> DiG 9.9.4 <<>> @8.8.8.8 facebook.com A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 702
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;facebook.com.                IN      A

;; ANSWER SECTION:
facebook.com.                723     IN      A      173.252.110.27

;; Query time: 74 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Jul 28 15:56:07 CST 2014
;; MSG SIZE rcvd: 57
```

所有以分号 “;” 开始的内容都是 dig 命令对获取到的原始返回信息的注释部分, 没有注释的部分才是我们真正要查询的内容. 这里解释部分内容:

- status:NOERROR 这是 DNS 返回的信息中 Reply Code 部分, 用于告知客户端,DNS 服务器这次进行的查询是否成功还是有错误发生



- id:702 这个就是 DNS 查询和返回信息中的 Transaction ID, 因为 UDP 协议不是按顺序到达的, 所以这个 ID 用于把查询和返回信息配对
- flags:qr rd ra 这个是 DNS 返回信息中的 Flag 部分, QR 表示这是一个返回响应包 (对应查询包), rd 表示客户端请求进行递归查询, ra 表示服务器支持递归查询
- QUERY:1 表示收到一个查询
- ANSWER: 表示回应了一个答复
- AUTHORITY:0 表示没有从域名的权威名称服务器查询数据, 可能因为 DNS 缓存中已经有这个记录了, 就不需要查询了
- ADDITIONAL:1 表示有一条额外的信息, 一般是域名的权威名称服务器的地址
- 中间就是查询的内容和回复的内容了, 应该很好理解
- 最后是整个查询过程的耗时, 以及原始信息有多少个字节

除了这种基本的查询, dig 有很多选项可以用, 常用的有:

- +short 表示不显示所有注释部分的代码, 只显示要查询的数据, 这个选项可以用在脚本里给其他命令提供域名的 IP
- +vc 表示用 TCP 方式查询, 而不是用 UDP 方式, TCP 模式查询受到 GFW 劫持的影响非常小 (不是没有, 只是范围很小)
- +dnssec 表示查询域名的 DNSSEC 签名信息, 关于 DNSSEC 会在后面讲解
- +trace 表示禁止服务器用递归方式查询, 一般来说就是让 DNS 返回根域服务器的地址, 然后客户端自己去做迭代查询, 后面的查询和 DNS 服务器不再有关系, 只取决于客户端和各级名称服务器之间的网络.

关于 dig +trace 可能是我们实际调试中用到最多的命令了, 关于这个命令的一些使用实例, 可以参考本站文章:[重新理解 GFW 的 DNS 污染和劫持策略](#).

### 3.5 正则, SED, AWK

正则表达式是一件屠龙神器, 但是要把它讲好非常困难, 我的建议是阅读这篇[正则表达式 30 分钟入门](#), 这是我目前见过最好的正则表达式入门的文章了. 我只讲一个 SED 配合正则表达式使用的例子, 让你们有一点基本的概念, 因为如果你完全不了解正则表达式, 这几个命令在你看起来可能会跟天书一样难懂.

我写过一篇文章讲的是如何从 greatfire.org 网站上抓取在中国被屏蔽率超过 70% 的网站域名, 然后生成用于 FreeRouter V2 的配置文件, 这个脚本的代码如下:

```
set -x
Output="./greatfire.conf"
if [ -f $Output ]
then
    rm $Output
fi
Threshold=70
for i in 0 1 2 3 4 5 6 7 8 9
do
    curl -s --insecure "https://zh.greatfire.org/search/alexa-top-1000-domains?page=$i" | \
    grep 'class="first"' | grep 'class="blocked"' | grep -vE "google" | \
    sed -e "s#\[^\]/]*\[\(^\[\]\)*\] \[^\%]*\%...\(^\[\]\)*\]\%.*#\1 \2#g" / \
    awk ' $2>=$Threshold' {print "ipset="/"$1"/vpn" }' \
    >>$Output
done
```

看不懂没关系, 一行行解释:

1. set -x 表示打开调试, 方便查看脚本运行信息
2. 刚开始是清理掉之前的输出文件
3. Threshold=70 是一个表示屏蔽率的变量, 我们说了是 70%
4. curl -s --insecure..... 是用 curl 下载一个网页,-s 表示静默模式工作,-insecure 是因为这是个 https 网页, 因为一些证书的问题我用这个选项绕过去
5. 最后面的 \$i 表示上面从 1 到 9 的数字, 因为要抓很多个页面
6. 另外提一下,openwrt 的 shell 是 ash, 非常精简也很简陋, 连 for 循环都只能这样一个个的写

那个网页中, 关于被屏蔽的网站部分的源码是这样的:

```
tr class="odd"><td class="first"><a href="/facebook.com">facebook.com</a></td><td月>2 2011</td><td  
class="blocked" style="background-size: 100%;">100%</td><td class="tags"><a href="/search/  
blocked" class="tag">Blocked</a>,
```

继续解释:

1. grep 'class=first' | grep 'class="blocked"' 是表示把网页众多行的源码中, 包含这两个字符串的行提取出来,
2. grep -vE "google" 是表示把包含 google 这个字符串的行剔除掉 (因为 Google 太多子域名了, 我觉得没必要全部提取)

```
sed -e "s#~[^\/*\^/]\([^\"]*\)[^\%]*%...\([^\%]*\)\%.*#\1 \2#g"/\
```

1. SED 是基于一行一行输入的流编辑工具,-e 是为了使用多个命令, 其实这里没必要.
2. 后面的 s# 一大堆东西 # 又是一大堆 #g 是 VIM 用户很熟悉的查找替换命令, 我们的目的是把域名和屏蔽率提取出来.
3. 表^ 示一个字符串的开始, 然后中间通过一次次的反斜线逐步往后定位.
4. 用括号括起来的部分, 就是一个分组, 这里的分组 1 就是域名, 分组 2 就是被屏蔽率

```
awk '$2>="'$Threshold"' {print "ipset="/"$1"/vpn" }'\
```

1. 你可以认为 awk 是一个类似 excell 的工具, 他可以把输入的数据按列进行处理
2. \$2 就是分组 2, 也就是被屏蔽率, 我们和设定的 70 去比较, 就是说如果屏蔽率大于 70% 就进行后面的操作
3. 如果被屏蔽率大于 70%, 就在 \$1, 也就是域前后分别添加一些别的字符, 最终组成我们要的配置文件

说到底我还是没怎么讲清正则表达式的理解, 其实每次写正则表达式都很头痛要调半天, 你们还是自己去上面提到的那个链接里字自己好好学学吧. 其实 FreeRouter2 项目本身并不依赖这些工具, 只是是一些辅助脚本用到了, 不过我说了要尽量让每一个细节都被你们理解, 所以至少要把用到的工具和相关的教程交给你们.

## 第四章 FreeRouter V2 的技术原理

“你在干什么，没有人可以阻止他们。”——“我就是没有人！””

如果你觉得前面的几个命令要么你早就会了，要么你用不上，那么从这节开始的内容你必须一个字一个字仔细的去看，因为这章的前三节是 FreeRouterV2 用到的关键技术所在，如果你不理解 ip 命令如何工作，不理解 iptables 是怎么工作，不理解 Dnsmasq 是怎么工作的，你根本就无法理解 FreeRouter V2 是怎么工作的。

### 4.1 IP 命令

除了前面提到的 route 命令之外，我们其实还有一个更强大的用于维护路由表的工具，就是 ip 命令，它的说明请参考[man page](#)。IP 命令的功能太过强大，我们只讲我们后面会用到的几种情况。

#### 4.1.1 table 概念

首先我们这里要先建立一个 table 概念，你可以认为 table 是一个网络访问请求（数据流）的容器。一般情况下，我们可以把符合一定特征的网络访问请求（数据流）都收集到一个 table 中，这样当我们需要对有这一类特征的网络访问请求进行处理时，就可以直接操作这个 table 了，而不需要一条一条的去处理。

在 linux 操作系统中，table 就是一个 0 到 255 的数字，当然我们也可以在 “/etc/iproute2/rt\_tables” 文件中给 table 数字对应上一个名称，然后就可以用这个名称来操作这个 table 了。例如我们的 OpenWRT 系统里的 rt\_tables 文件默认是这样的：

```
#
# reserved values
#
255 local
254 main
253 default
0  unspec
```

```
#
# local
#
#1  inr.ruhep
```

local,main,default,unspec 这些都是系统内建的 ip table, 可以看到他们分别都对应了一个数字, 你操作 table 254 和操作 table main 其实是完全一样的. 我们也可以修改这个文件, 建立一个自己的 table, 例如我们可以建立一个 ID 为 10, 名字叫 vpn 的 table. 如果我们想要在访问被封锁网站的时候通过 VPN 接口出去, 那我们就可以先对这些数据打上标签 4.2.2, 然后用 ip rule 命令把有这些标签的数据全部加入一个 table.

### 4.1.2 把数据添加到 table

假定我们现在已经把要访问全部被封锁网站的数据打上了标签 1(fwmark 1), 我们又已经建立了一个名叫 vpn 的 table, 那么命令就可以这样写:

```
ip rule add fwmark 1 priority 1984 table vpn
```

先阅读一下 man page 上对这个命令的注释:

```
ip rule [ list | add | del | flush ] SELECTOR ACTION

SELECTOR := [ from PREFIX ] [ to PREFIX ] [ tos TOS ] [ fwmark FWMARK[/MASK] ] [ dev STRING ] [ pref
NUMBER ]

ACTION := [ table TABLE_ID ] [ nat ADDRESS ] [ prohibit | reject | unreachable ] [ realms [SRCREALM/]
DSTREALM ]

TABLE_ID := [ local | main | default | NUMBER ]
```

说明如下:

1. ip rule add 表示添加一条规则.
2. fwmark 1 表示符合 fwmark(标签) 值等于 1 的数据.
3. priority 1984, 表示这条规则的优先级是 1984, 每条命令必须有独立的优先级. 优先级从 0 到 32767, 数字越小优先级越高, 但 0,32266,32267 一般都是已经分配的, 所以你要这这几个数中间选一个数作为优先级.
4. table vpn 表示把这些数据全部添加到 vpn 这个 table 里去.

如果想检查已有的 ip rule, 可以用以下的命令:

```
ip rule list
```

一般情况下你会看到如下的回显:

```
0:      from all lookup local
1984:   from all fwmark 0x1 lookup vpn
32766:  from all lookup main
32767:  from all lookup default
```

### 4.1.3 让 table 数据走 VPN 接口

现在我们已经有了一个 table 了, 如何让这个 table 走 VPN 接口呢? 先阅读一下 man page 上的说明:

```
ip route { add | del | change | append | replace | monitor } ROUTE

SELECTOR := [ root PREFIX ] [ match PREFIX ] [ exact PREFIX ] [ table TABLE_ID ] [ proto RTPROTO ] [
    type TYPE ] [ scope SCOPE ]

ROUTE := NODE_SPEC [ INFO_SPEC ]

NODE_SPEC := [ TYPE ] PREFIX [ tos TOS ] [ table TABLE_ID ] [ proto RTPROTO ] [ scope SCOPE ] [ metric
    METRIC ]

INFO_SPEC := NH OPTIONS FLAGS [ nexthop NH ] ...

NH := [ via ADDRESS ] [ dev STRING ] [ weight NUMBER ] NHFLAGS

OPTIONS := FLAGS [ mtu NUMBER ] [ advmss NUMBER ] [ rtt TIME ] [ rttvar TIME ] [ window NUMBER ] [ cwnd
    NUMBER ] [ initcwnd NUMBER ] [ ssthresh REALM ] [ realms REALM ] [ rto_min TIME ] [ initrwnd
    NUMBER ]

TYPE := [ unicast | local | broadcast | multicast | throw | unreachable | prohibit | blackhole | nat ]

TABLE_ID := [ local | main | default | all | NUMBER ]

SCOPE := [ host | link | global | NUMBER ]

FLAGS := [ equalize ]

NHFLAGS := [ onlink | pervasive ]

RTPROTO := [ kernel | boot | static | NUMBER ]
```

很复杂? 那我们看看实际用到的命令:

```
ip route add default dev pptp-VPN table vpn
```

是不是瞬间觉得很亲切了?ip route 和前面的 route 命令几乎是一样的,只不过后面多了一个 table vpn,表示添加默认接口这个动作只是用于 table vpn 里的数据的,这个灵活性就是 route 命令所没有的了. 注意! 这里的 pptp-VPN 也是要根据你系统实际 VPN 接口的名称修改的. 我们这里没有指定网关 IP(例如 “via 10.10.10.1”)之类的,因为一般一个接口连接上之后,如果网关支持 DHCP,那么在分配 IP 的时候也就会把默认网关通告给客户端,不需要我们去额外指定. 除非你的 VPN 不支持 DHCP,但这种 VPN 我还没见过.

如果你想确认某个 table 是不是走了 VPN 接口还是别的接口,可以用如下命令:

```
ip route show table vpn
```

如果确实添加默认接口成功了,你应该看到

```
default dev pptp-VPN scope link
```

其实这个添加默认接口的命令是应该早于你往这个 table 添加数据的,以免数据进来了却不知道要从哪里走,所以完整的顺序是:

1. 修改/etc/iprout2/route\_tables 建立一个 table
2. 给这个 table 指定一个默认的网络接口
3. 给指定的数据打上标签
4. 把打上了标签的数据添加到这个 table 里去

1,2,4 我们都知道怎么做了,那么如何给指定的数据打上标签呢?这就用到了 iptables 防火墙了.

## 4.2 IPTables 防火墙

netfilter 是 linux 内核的一个模块,这个模块当于 linux 系统的防火墙,它会对所有进出 linux 系统的网路数据进行管理,netfilter 防火墙模块的结构如下图所示:

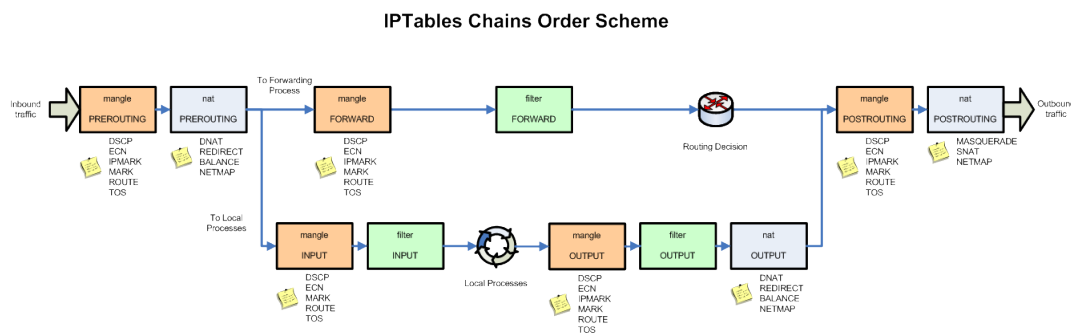


图 4.1: netfilter 防火墙结构

我们可以看到棕色框<sup>1</sup>框住的部分就是 iptables 的几个“链”，分别是 PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING，大致讲一下这几个链的工作流程。当一个数据通过网络接口进入系统之后，首先进入 PREROUTING 链进行处理。PREROUTING 链中根据对数据的不同处理，分别把处理方法写在 mangle 表和 nat 表中，mangle 表一般用于对数据添加标签方便后面的链处理，nat 表一般是进行网络地址转换和转发的动作<sup>2</sup>。

在经过两道 PREROUTING 链处理之后，数据会根据需求分别交给 INPUT 和 FORWARD 链处理。INPUT 链再经过打标签，过滤等动作后，再把数据交给上层应用的协议栈去处理。例如我们可以在 INPUT 链末端把数据交给我们写的一个应用，在应用中分析和提取数据，当应用处理过数据之后，就要把数据返还到 OUTPUT 链，OUTPUT 链再经过一次 NAT 表看是否要转址（只是要经过，不一定真的会发生转址操作），然后交给 POSTROUTING 链。

FORWARD 链一般只是经过 filter 表的过滤筛选就会把数据交给 POSTROUTING 链，从 OUTPUT 链过来的数据和从 FORWARD 链过来的数据经过 POSTROUTING 链处理后就交给了后面的输出设备了。

netfilter 是 linux 内核模块的名称，可是用户要如何操作这个模块，给防火墙添加和修改规则呢？这里用到的就是 iptables，netfilter 才是防火墙的实体，iptables 是这个实体的管理工具。因为二者的结合非常紧密，所以我们一般都习惯性的称之为 iptables 防火墙。完整的 iptables 命令说明，请参考 [iptables 的 man page](#) 和 [extention man page](#)。关于 netfilter 各个扩展模块的功能说明，可以参考 [openwrt 网站上的这个页面](#)。

#### 4.2.1 IPSET 概念

在讲 iptables 的功能之前，我们先引入一个额外的概念：ipset。

<sup>1</sup>如果不是棕色也很正常，我是色盲

<sup>2</sup>NAT, Network Address Translate, 其实是个很重要的概念，只是和我们的主题关系不大所以我没提



ipset 和前面讲的 table 有点像, 但更好理解, 它就是一个存储了 IP 值的数组, 一般以内存变量的形式驻留在系统内存中, 一重启就会消失. 要建立一个 ipset, 我们可以 ipset 的 create 命令:

```
ipset create vpn iphash -exist
```

这里表示建立一个名叫 VPN 的 iphash 类型的 ipset, iphash 类型的 IPSET 可以存储 65535 个 IP 地址, -exist 表示如果这个叫这个名字的 ipset 已经存在了, 就跳过不建立了.

如果要把一个 IP(例如 8.8.8.8) 加入一个 IPSET(例如刚刚建立的 vpn set), 那就是:

```
ipset add vpn 8.8.8.8
```

如果要删除, 把 add 改成 del 就行了

```
ipset del vpn 8.8.8.8
```

如果我们要清空整个 ipset, 就用 flush 选项:

```
ipset flush vpn
```

如果我们要测试一个 IP(还是 8.8.8.8) 在不在 IPSET(还是 vpn set) 里, 就用 TEST 选项:

```
ipset test vpn 8.8.8.8
```

## 4.2.2 iptables 的 mark 功能

我们前面说了, 要把访问被封锁网站数据加入一个 table 让它走 VPN 接口, 首先就要对数据打上标签, 那么如何用 iptables 给数据打上标签呢? 我们得先把所有被封锁网站的 IP 加入一个 IPSET, 然后我们再用 iptables 的一个名叫 set 的模块来对这个 IPSET 进行打标签的操作. 下面是 man page 上对这个模块的说明:

```
set
This module matches IP sets which can be defined by ipset(8).

[!] --match-set setname flag[,flag]...
where flags are the comma separated list of src and/or dst specifications and there can be no more
than six of them. Hence the command

iptables -A FORWARD -m set --match-set test src,dst
will match packets, for which (if the set type is ipportmap) the source address and destination
port pair can be found in the specified set. If the set type of the specified set is single
dimension (for example ipmap), then the command will match packets for which the source
address can be found in the specified set.
```

我们实际的命令是:

```
iptables -t mangle -I PREROUTING -m set --match-set vpn dst -j MARK --set-mark 1
```

解释说明:

1. `iptables -t mangle` 默认情况下 `iptables` 是对 `filter` 表操作的, 如果要指定某个表, 就要用 `-t` 选项. `-t mangle` 表示我们操作 `mangle` 表, 为什么是 `mangle` 表? 因为 `PREROUTING` 链是存放在 `mangle` 表里的, 默认的 `filter` 表里没有 `PREROUTING` 链, 那为什么是 `PREROUTING` 链呢? 往下看.
2. `-I` 表示 `insert`, 也就是插入一条规则.
3. `PREROUTING` 不管是从外部网线输入的数据, 还是从我们从电脑上发出的数据, 对于路由来说都是输入的数据, 这里的进和出是从路由的角度来看的, 原始数据进来, 处理过的数据输出, 和你是上传还是下载无关. 我们也知道 `PREROUTING` 链是整个防火墙链的最前端, 如果我们要从源头上处理数据, 就需要对这个链动手.
4. `-m set` 表示调用 `set` 模块, 所有调用模块的格式都是 “`-m 模块名`”.
5. `--match-set vpn dst`, 表示匹配一个名叫 `vpn` 的 `ipset`, `dst` 表示这是目标地址, 因为我们要访问的那些被封锁的服务器对路由来说只可能是目的地而不是源头 (`src`), `src` 是我们自己.
6. `-j MARK`, `-j` 是一个通用选项, 表示把数据交给后面的动作进行处理, 也就是 `MARK`.
7. `--set-mark 1`, 这是 `iptables` 的一个叫 `CONNMARK` 的 `extension` 提供的功能, 就是给数据打上标签了.

`iptables` 的规则插入是有顺序的, 如果在 `-I` 后面指定一个规则号, 就表示在这个规则号前面插入一条规则, 如果不指定规则号, 就表示在所有规则的最前面插入一条规则.

而 `iptables` 规则, 是从最前面的规则开始一个个尝试的, 如果一个数据和前面的某条规则匹配上了, 后面的规则就对他无效了, 所以你最后插入的规则其实是优先级最高的规则. 如果你要加入一条优先级不是那么高的规则, 可以用 `-A`, `Append`, 表示在特定或所有规则后面加入这条规则. 所以插入规则的命令格式是<sup>3</sup>:

```
iptables -t 表名 -I 链名\ 规则号\ 规则
```

在用 `iptables` 添加了规则之后, 如果想看看这些规则是否添加成功, 以及他们的顺序, 可以用这个命令:

```
iptables -t 表名 -L 链名
```

<sup>3</sup>命令中的反斜线是不需要的, 因为 `latex` 中的 `lstlisting` 原生就不支持中文, 无法正确显示空格我才加入的, 如果你知道怎么解决这个问题请告诉我, 不胜感激.

如果要删除一条规则, 就用 -D 选项, 后面可以跟规则号表示删除指定规则号, 如果不跟规则号, 则从最上面 (最优先) 的 0 号规则开始删除.

```
iptables -t 表名 -D 链名\ 规则号
```

如果你的自定义规则已经太多太乱, 你想把整个链里的规则都清空, 就用 -F, flush 选项

```
iptables -t 表名 -F 链名
```

所以现在我们知道了可以让一个 table 里的数据走 VPN 接口, 也知道了如何把打标签的数据加入一个 table, 也知道了如何把一个 IPSET 的数据都打上标签, 那么, 如何把被封锁网站的 IP 加入一个 IPSET 呢? 这里我们还是先把 IPTABLES 的其他功能讲完吧, 等到下一节我们再处理这个问题.

### 4.2.3 iptables 的 m32 模块

我们在前面 2.1 提到了 GFW 对 DNS 劫持和污染的根源是在向境外 DNS 发起解析请求时, 抢先返回虚假的 IP 信息给解析器. 根据观察分析, GFW 伪造的虚假信息格式是非常固定的, 甚至可以说是非常便于识别和拦截的. 我们只要利用 iptables 的过滤规则, 就可以很轻松的丢弃这些污染信息.

我们在前面也说了, 境外 DNS 的正确信息不是不会返回, 只是被 GFW 的虚假信息抢先占了位置导致不被解析器接受而已, 而防火墙是一个在解析器之前的关卡, 只要在这里丢弃了污染信息, 那么境外 DNS 返回的正确信息就可以作为第一个到达的信息顺利被解析器接受了.

工欲善其事, 必先利其器, 我们要想过滤 GFW 的劫持信息, 就要先了解这个数据的格式. 在网络数据抓取上, 业界公认的神器莫过于 WireShark 了, 启动 wireshark 之后选择网卡, 在过滤项中写 port 53, 表示我们只查看数据信息中包含 53 端口的信息. 为什么是 53 端口? 因为 DNS 服务器的查询端口是 53, 然后我们通过 Dig 命令, 指定使用 8.8.8.8 去查询 twitter.com 的 A 记录. 可以看到如下的查询和返回信息:

Protocol	Length	Info
DNS	82	Standard query 0x567b A twitter.com
DNS	87	Standard query response 0x567b A 59.24.3.173
DNS	98	Standard query response 0x567b A 37.61.54.158
DNS	146	Standard query response 0x567b A 199.59.149.230 A 199.59.148.10 A 199.59.149.198 A 199.59.148.82

图 4.2: 虚假信息和正确信息混杂

可以看到, GFW 凭借自己的位置优势 (离我们近) 抢先返回了两个虚假的 IP 地址给我们, 但后面 Google DNS 的正确信息也返回了, 只是没有被解析器接受, 我们要干掉的就是前面两条错误信息. 观察多次之后我们发现这种信息的格式是完全一样的, 区别仅仅是 IP 地址会在一个数组范围内变化: 可以看到 GFW 伪造的这个虚假信息非常粗糙, 一般 DNS 都会返回一些 Additional 信息, 例如告诉你这个域名的权

```

[Destination GeoIP: Unknown]
[-] User Datagram Protocol, Src Port: domain (53), Dst Port: 64282 (64282)
  Source port: domain (53)
  Destination port: 64282 (64282)
  Length: 53
  [+ Checksum: 0xdded [validation disabled]
[-] Domain Name System (response)
  [Request In: 1]
  [Time: 0.033903000 seconds]
  Transaction ID: 0x567b
  [+ Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  [+ Queries
  [-] Answers
    [-] twitter.com: type A, class IN, addr 59.24.3.173
0000  74 d0 2b d9 e5 bf 3c e5 a6 02 cb 29 08 00 45 04  t.+...<. ...)..E.
0010  00 49 03 fc 00 00 da 11 c1 25 08 08 08 08 0a 01  .I.....%......
0020  01 6e 00 35 fb 1a 00 35 dd ed 56 7b 81 80 00 01  .n.5...5 ..V{....
0030  00 01 00 00 00 00 07 74 77 69 74 74 65 72 03 63  .....t witter.c
0040  6f 6d 00 00 01 00 01 c0 0c 00 01 00 01 00 00 36  om.....6
0050  8d 00 04 3b 18 03 ad  ....

```

图 4.3: 污染信息实例

威名称服务器地址之类的.GFW 的目的就是污染, 只要把错误 IP 传达给你就完了, 所以他整个消息包在虚假 IP 之后就没了, 还真是精打细算节约流量呢.

我目前搜集和统计到的 GFW 用于 DNS 劫持的污染 IP 有 49 个, 如下表所示.

```

74.125.127.102
74.125.155.102
74.125.39.102
74.125.39.113
189.163.17.5
209.85.229.138
249.129.46.48
128.121.126.139
159.106.121.75
169.132.13.103
192.67.198.6
202.106.1.2
202.181.7.85
203.161.230.171
203.98.7.65
207.12.88.98
208.56.31.43
209.145.54.50
209.220.30.174
209.36.73.33

```

```
211.94.66.147
213.169.251.35
216.221.188.182
216.234.179.13
243.185.187.39
37.61.54.158
4.36.66.178
46.82.174.68
59.24.3.173
64.33.88.161
64.33.99.47
64.66.163.251
65.104.202.252
65.160.219.113
66.45.252.237
72.14.205.104
72.14.205.99
78.16.49.15
8.7.198.45
93.46.8.89
253.157.14.165
54.76.135.1
23.89.5.60
49.2.123.56
77.4.7.92
197.4.4.12
118.5.49.6
188.5.4.96
189.163.17.5
```

转换成 16 进制就是:

```
4A7D7F66
4A7D9B66
4A7D2766
4A7D2771
BDA31105
D155E58A
F9812E30
80797E8B
9F6A794B
A9840D67
C043C606
CA6A0102
CAB50755
CBA1E6AB
CB620741
CF0C5862
D0381F2B
D1913632
```

```
D1DC1EAE
D1244921
D35E4293
D5A9FB23
D8DDBC6
D8EAB30D
F3B9BB27
253D369E
042442B2
2E52AE44
3B1803AD
402158A1
4021632F
4042A3FB
4168CAFC
41A0DB71
422DFCED
480ECD68
480ECD63
4E10310F
0807C62D
5D2E0859
FD9D0EA5
364C8701
1759053C
31027B38
4D04075C
C504040C
76053106
BC050460
BDA31105
```

我不一定会及时更新本文档, 最新的数据我会发布到 FreeRouter V2 项目本身和[这篇文章里](#). 为什么这里要把 IP 地址转换成 16 进制呢? 因为你知道整个计算机是基于 2 进制的, 从上面的抓包你也可以看到实际的 IP 地址都是 16 进制的, 我们过滤这些地址要用的 iptables 的 u32 模块也只认识 16 进制的数字.

u32 模块是 iptables 的一个扩展包, 他允许你从一个 IP 数据包中抓取 4 个字节 (32 比特) 的数据, 然后对这个 4 个字节的数据进行数值分析, 在符合条件之后交给 iptables 进行各种处理. 而 IP 地址刚好就是 4 个字节, 那么我们只需要在 DNS 返回信息的 IP 包中提取出 IP 地址信息, 然后和上面的这个列表对比, 如果发现匹配就丢弃这个数据就可以了.

我们用到的完整命令如下:

```
iptables -t mangle -I PREROUTING -p udp --sport 53 -m u32 --u32 "0&0x0F000000=0x05000000 && 22&0
xFFFF016=0x4A7D7F66,0x4A7D9B66,0x4A7D2766,0x4A7D2771,0xBDA31105,0xD155E58A,0xF9812E30,0x80797E8B,0
x9F6A794B,0xA9840D67" -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m u32 --u32 "0&0x0F000000=0x05000000 && 22&0
```

```
xFFFF016=0xC043C606,0xCA6A0102,0xCAB50755,0xCBA1E6AB,0xCB620741,0xCF0C5862,0xD0381F2B,0xD1913632,0xD1DC1EAE,0xD1244921" -j DROP

iptables -t mangle -I PREROUTING -p udp --sport 53 -m u32 --u32 "0&0x0F000000=0x05000000 && 22&0
xXXXX016=0xD35E4293,0xD5A9FB23,0xD8DDBC6B,0xD8EAB30D,0xF3B9BB27,0x253D369E,0x042442B2,0x2E52AE44,0
x3B1803AD,0x402158A1" -j DROP

iptables -t mangle -I PREROUTING -p udp --sport 53 -m u32 --u32 "0&0x0F000000=0x05000000 && 22&0
xXXXX016=0x4021632F,0x4042A3FB,0x4168CAFC,0x41A0DB71,0x422DFCED,0x480ECD68,0x480ECD63,0x4E10310F,0
x0807C62D,0x5D2E0859" -j DROP

iptables -t mangle -I PREROUTING -p udp --sport 53 -m u32 --u32 "0&0x0F000000=0x05000000 && 22&0
xXXXX016=0xFD9D0EA5,0x364C8701,0x1759053C,0x31027B38,0x4D04075C,0xC504040C,0x76053106,0xBC050460,0
xBDA31105" -j DROP
```

命令有 5 条是因为 u32 模块一次最多允许我们匹配 10 个数据, 具体格式的分析我们只需要看一条就够了, 以第一条为例详细说明:

1. `iptables -t mangle -I PREROUTING`, 这些和上一节是一样的, 不重复了
2. `-p udp`, 表示只分析 UDP 协议的数据, 因为 DNS 查询默认都是 UDP 协议的
3. `-s port 53`, 表示源端口是 53 的数据, 因为 DNS 服务器返回的数据都是从它的 53 端口返回的
4. `-m u32 --u32`, 这里表示使用 u32 模块, 这是个标准命令格式

在开始后面的 u32 模块分析前,我们先补充介绍一下 DNS 响应包的数据组成. 首先这是一个 IP 包,就有 IP 报头,另外这是 UDP 协议的,就有 UDP 报头,后面的 UDP 数据就是 DNS 服务器的响应数据了,所以格式是这样的:

```
[IP_Header]::[UDP_Header]::[UDP_Data==DNS_Response]
```

u32 模块部分的匹配代码是这样的:

0&amp;0x0F000000=0x05000000 &amp;&amp; 22&amp;0xFFFF@16=0x4A7D7F66

- 0

第一个 0 表示从第 0 个字节开始抓 4 个字节, 计算机世界一直都是从 0 开始数的。

- &0x0F000000

“&”表示把前面抓到的 4 个字节和“&”后面的掩码进行“与”操作,这个就算是文科生也应该在计算机基础课上学过,不解释了。0x0F000000 就是这个掩码了,这表示其他位全部置 0,只保留第 0 字节的低 4 位。

- =0x05000000



“=0x05000000”表示判断前面进行完与操作的数是不是等于 0x05000000. 如果你知道 IP 包的数据格式, 就知道第 0 个字节的低 4 位表示的是 IP 报头的长度, 一般来说都是 20 字节但也有特例. 这里我们要确定这个报头没有什么特殊格式, 长度确实是 20, 为什么要判断这个长度后面会解释.

- 22&0xFFFF

这个表示从第 22 个字节开始取 4 个字节 (22~25), 和掩码 0xFFFF(0x0000FFFF) 与操作, 也就是取 24,25 字节的内容, 得到一个数 X. 你抓包分析看一下就会发现, 24,25 字节对应的数字是 UDP\_Header 里表示 UDP 包的长度, 也就是 UDP\_Header+UDP\_Data 的总长度.

- @

这个 @ 的作用非常大, 一定要理解. @ 表示从 IP 包的最开始往后跳跃 X 个字节, 具体跳跃多少呢? 就是 @ 前面的数值了. 我们说了 @ 前面的一个与操作得到了 UDP 包的长度, 也就是说这里我们要从 0 开始往后跳过 UDP 包的长度. 整个 IP 包的长度是 IP 头长度 (20)+UDP 包的长度 (X), 现在我们从 0 开始跳过了 UDP 包的长度 (X), 就位于了整个 IP 包的最末再往前倒退 20 个字节的位置.

- 16

我们前面的抓包已经看到了, 虚假的 IP 信息刚好位于整个 IP 包的最后 4 个字节, 我们现在已经处在最末 -20 个字节的位置了, 那么从这个位置开始数, 抓第 16 个字节开始的 4 个字节就行了, 也就是 16~19 字节, 为什么不是 17~20? 我们是从第 0 个字节开始算的, 别忘了这点.

后面的 -j DROP 就很好理解了, 直接丢弃. 在我们的防火墙中应用以上规则, GFW 的对境外 DNS 查询的劫持信息就被过滤掉了, 剩下的就是正确的解析信息了.

但是要记住! 只有对境外 DNS 查询的污染包是 GFW 发出的, 也只有这个情况下这个数据格式才是和我们的命令匹配的. 虽然这可以解决大部分问题了, 但有的时候我们还是需要境内 DNS+ 境外 DNS 组合的方式, 因为境内 DNS 的速度比较快, 而且他们对一些大网站的节点解析都比较适合我们.

但是我们在这些境内 DNS 面对的问题不是 GFW 的劫持, 而是他们给出的原始数据就是被污染 (被劫持后污染) 的. 这样的结果是, 他们给出的 DNS 响应包会因为服务器配置的格式不同而千差万别, 而且基本确定的是 IP 都不是在最末的 4 字节上, 因为正常的响应包末尾一般都是 Additional RRs 数据. 我们想要使用境内 DNS 获得高速的解析又不想被污染, 就要对境内 DNS 的响应数据也做过滤, 那么这种 IP 位置不固定的响应包要如何过滤呢? 这里我们就要用到更强大的 string 模块了.

#### 4.2.4 iptables 的 string 模块

通过上一节的介绍, 我们也看到了 u32 模块第一个缺点是只能针对特定位置的 4 个字节数据进行分析, 如果位置不固定就没有办法了. 如果要对位置不固定的数据进行分析, 我们就要使用 string 模块, 它可

以帮我们在一个 IP 包里搜索任意位置, 看是否有匹配的字符串. 当然 u32 的固定位置分析也不能算是完全的缺点, 只针对固定位置固定长度的数据分析意味着 u32 模块消耗的 CPU 资源非常少, 因为我们基本上只需要做一些简单的与或操作和比较计算就可以了. 而 string 模块强大的搜索功能, 其代价就是消耗了更多的 CPU 资源.

和 u32 一样, string 模块搜索 IP 时也是使用 16 进制, 因为原始数据就是这样的. 因为计算复杂度很高, 一条 string 匹配的 iptables 命令只能一次只能匹配一个 IP(字符串), 所以我们得写 49 条命令:

```
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4A7D7F66|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4A7D9B66|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4A7D2766|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4A7D2771|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|BDA31105|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D155E58A|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|F9812E30|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|80797E8B|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|9F6A794B|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|A9840D67|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|C043C606|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|CA6A0102|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|CAB50755|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|CBA1E6AB|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|CB620741|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|CF0C5862|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D0381F2B|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D1913632|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D1DC1EAE|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D1244921|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D35E4293|" --from
```

```

60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D5A9FB23|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D8DDBC6|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|D8EAB30D|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|F3B9BB27|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|253D369E|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|042442B2|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|2E52AE44|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|3B1803AD|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|402158A1|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4021632F|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4042A3FB|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4168CAFC|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|41A0DB71|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|422DFCED|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|480ECD68|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|480ECD63|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4E10310F|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|0807C62D|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|5D2E0859|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|FD9D0EA5|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|364C8701|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|1759053C|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|31027B38|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|4D04075C|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|C504040C|" --from

```

```
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|76053106|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|BC050460|" --from
60 --to 180 -j DROP
iptables -t mangle -I PREROUTING -p udp --sport 53 -m string --algo bm --hex-string "|BDA31105|" --from
60 --to 180 -j DROP
```

相对来说 string 模块的命令就好理解多了,

- -m string

表示启用 string 模块进行匹配,

- -algo bm

表示启用贝叶 (Boyer-Moore) 字符串搜索算法, 另一种算法是 kmp(Knuth-Pratt-Morris), 具体应该只是效率和不同应用的区别, 算法原理我们就不细究了.

- -hex-string

表示要匹配 HEX 格式的 IP 地址了, 注意这里前面不要有 0x 表示 16 进制了, 直接写 16 进制数据就行, 后面跟着的就是 IP 地址的 16 进制格式.

- -from 60 -to 180

这表示的是搜索的偏置范围 (offset), 从第 60 字节开始到 180 字节结束, 一般来说 DNS 返回包的长度很少有超过 180 字节, 而 A 记录 IP 值的位置因为 IP 和 UDP 协议报文格式的关系, 基本上不可能出现在 60 字节之前, 所以指定这个范围是足够的, 同时也可以大大减轻搜索算法的运算压力. 如果你担心有什么奇葩 DNS 返回的结果在这范围之外, 删除这两个偏置选项就行了, 这样会默认搜索整个字符串, 但相应的搜索消耗的 CPU 资源就更多了.

和 u32 模块的使用一样, 把这些规则保存为防火墙规则就可以过滤所有境内和境外 DNS 中的污染和劫持信息了, 要注意 string 模块的过滤其实已经包含了 u32 模块的功能, 不需要再添加 u32 模块的过滤规则了.

但是境内 DNS 的原始数据都是污染的, 你过滤掉这些污染的信息后也不会给你返回正确的信息. 而你希望的结果肯定是同时享受境内 DNS 的高速解析速度, 又可以让境外 DNS 给我们某些敏感网站的正确解析结果. 要实现这样的组合, 就要在 iptables 的过滤规则之下, 继续配合下一节提到的 Dnsmasq 了.

## 4.3 Dnsmasq 解析器

Dnsmasq 是 OpenWRT 系统默认内置的 DNS 解析器, 要注意这个解析器不等同于 nslookup 和 dig, 后面的只是调试工具, 除了返回一些查询信息外不能给任何系统和应用提供解析结果, 而 Dnsmasq 可以.

我们需要对 Dnsmasq 的性质有个基本认识, Dnsmasq 并不能算是一个 DNS 服务器软件, 他的准确定义是一个 DNS 转发器. 就是说, 它把收到的 DNS 解析请求, 转发到一些 DNS 服务器上去, 然后从 DNS 服务器接收返回的结果, 最后再返回给你? 你也许会觉得这玩意这么蛋疼, 我自己查询不好吗, 干嘛要你来转发? 其实它能做的不仅仅是转发, 转发仅仅是他实现 DNS 查询的方式, 我们知道它即不能递归查询, 又不是蛋疼的迭代, 能做的当然就是转发了.

首先它提供了一个缓存功能, 这可以大大减轻你查询的网络请求次数, 这也是为什么很多时候我们可以把他当一个 DNS 服务器来用的原因. 除此之外 Dnsmasq 提供了非常多灵活的选项让我们可以完全掌控 DNS 解析的行为, 具体的内容请参考它的 [Man Page](#), 我们在 FreeRouter V2 中要用到的是他的以下几个功能.

### 4.3.1 IPSET 功能

我记得前面我挖了个坑, 就是关于如何把被屏蔽网站的 IP 全部搜集到一个 IPSET 里的问题<sup>4.2.2</sup>, 现在我们就来解决它. Dnsmasq 从 2.66 版开始就支持 IPSET 功能了, 注意 OpenWRT 默认自带的 Dnsmasq 是精简版本并没有 IPSET 功能, 需要我们自己替换为 dnsmasq-full, 也就是完整版. 如果要确定自己的 Dnsmasq 有没有 IPSET 功能支持, 可以 SSH 登录路由后输入 “dnsmasq -version” 查看当前的版本和编译信息, 可以看到下面的信息中已经包含了 ipset, 如果显示的是 “no-ipset”, 就说明没有 IPSET 功能支持.

```
root@OpenWrt:~# dnsmasq --version
Dnsmasq version 2.71 Copyright (c) 2000-2014 Simon Kelley
Compile time options: IPv6 GNU-getopt no-DBus no-i18n no-IDN DHCP DHCPv6 no-Lua TFTP no-contrack ipset
auth DNSSEC

This software comes with ABSOLUTELY NO WARRANTY.
Dnsmasq is free software, and you are welcome to redistribute it
under the terms of the GNU General Public License, version 2 or 3.
```

这个功能的作用, 就是把指定域名的解析结果存入一个 IP, 具体哪些域名可以通过配置文件来指定. 例如我们现在不能访问 google 了<sup>4</sup>, 那么我们就把 Google.com 的 IP 都加入一个叫 vpn 的 IPSET, 然后配合我们前面学过的 iptables mark 功能, ip route 命令把对 google IP 的访问全部通过 VPN 接口进行. 我们的命令是这样的:

---

<sup>4</sup>不知道以后你们看到这段是什么感想, 反正中国在现在 (2014 年 7 月 30 日) 是真的不能访问 Google 了

```
ipset=/google.com/vpn
```

非常令人感动的一点是,dnsmasq 是支持从顶级域往下匹配的,也就是说你写了 google.com 之后,什么 mail.google.com,maps.google.com 的二级三级四级域名也全部被匹配了,后面“vpn”就是我们事先通过 ipset 命令建立好的用于存储这些 IP 的 IPSET.

你也许会想,Google 那么多 IP,Dnsmasq 能全部都查到? 查不到! 可是也完全没必要查到. 因为你能用到的 IP 就是 DNS 解析给你的 IP, 其他的 IP 如果你连解析都没去解析他, 为什么要知道呢? 就好比你要去美国出差, 但是老板只让你坐国航的飞机, 那为什么还要去查东航的机票价格呢?

这其实也就是 FreeRouter V2 里相对 V1 最优美的地方了, 只有在你需要访问某个被封锁的 IP 时它才会起作用, 而且具体什么 IP 可能被封锁是完全由你自己指定的, 因为域名是你指定的. 在 FreeRouter V1 里这一切是通过静态路由表的方式实现的, 这种方法的缺点是:

1. 要精确搜集一个网站的全部子域名和对应的全部 IP 极度困难
2. 以为精确搜集很难, 所以经常不得不把整个网段加入静态路由表, 但整个网段中有很多 IP 其实是被屏蔽的
3. 很多 IP 对应的网站普通人听都没听过, 这辈子可能也都不会上, 但是如果不加注释的话, 你根本不知道 IP 对应的到底是什么网站,
4. 因为 IP 的可读性太差, 如果哪个网站解封了, 或者根本就停掉了, 要找出网站对应的 IP 再删除都很繁琐

而用 Dnsmasq 的 IPSET 功能就完全没有这种问题了, 一个域名就基本能帮你搞定一个网站的访问了<sup>5</sup>, 虽然有可能出现部分解析的 IP 并没有被屏蔽也被走了 VPN 的情况 (因为 Google 的 IP 实在太多了), 但肯定都还是 google 的 IP, 不会影响到其他任何网站. 所以我们唯一需要维护的, 就是被封锁的网站的域名列表, 这也就是我对 FreeRouter V2 最主要的维护工作了.

其实你自己就可以去维护这个列表, 我想只要智商正常的人看到这里应该都知道这个列表该怎么写. 我只是提醒你们一下,windows 格式和 unix 格式对于换行符的处理是不同的, 这使得在 windows 下编辑的文件在 linux 系统下可能会无法被正确识别. 最稳妥的办法是 SSH 登录进路由之后用 vim 编辑, 或者在 windows 平台用 vim 以 unix 编辑;vim 可以保留文件的 unix 格式不变, 而且如果有多余的换行符, 你在 vim 下是可以看得出来的..

你们也许听说过一些什么 gfw\_list 之类的域名列表, 那个列表里有几千个被封锁的域名, 但是我看了一下觉得那个列表的质量实在是太糟糕了. 那个列表似乎是一个论坛社区的人自己在维护, 人人都可以提

<sup>5</sup>Google 和 facebook 这类大网站还有很多负责存储静态文件和其他内容的独立域名, 也要一并加入列表才行

交,但从来没人检查,很多明显是部署在国内的网站也被他们加进去了.这年头靠得住的人真没几个,所以我决定还是自己搜集比较好.

目前这个列表还不到 200 个域名,可是我几乎已经感觉不到墙的存在了,至少你们在讨论 Google 被封和解封这两个话题的时候我根本察觉不到,youtube 该看啥就看啥,twitter 的推送都很正常.Porn tube 什么的虽然也可以上但口味太重不适合我,中国人嘛,就该上草榴(敢求邀请码一律拉黑)!

### 4.3.2 为 Dnsmasq 指定 DNS

我们说了 Dnsmasq 其实是一个 DNS 转发器,只是把我们的 DNS 解析请求转发到各个公共 DNS 上,那么我们就必须为 Dnsmasq 指定好使用哪些 DNS.

在 OpenWRT 系统中,Dnsmasq 的默认 DNS 服务器来源是通过在/etc/config/dhcp 文件中配置实现的:

```
config dnsmasq
    option domainneeded '1'
    option boguspriv '1'
    option filterwin2k '0'
    option localise_queries '1'
    option rebind_protection '1'
    option rebind_localhost '1'
    option local '/lan/'
    option domain 'lan'
    option expandhosts '1'
    option nonegcache '0'
    option authoritative '1'
    option readethers '1'
    option leasefile '/tmp/dhcp.leases'
    option resolvfile '/tmp/resolv.conf.auto'
```

我们看到最后一行是 resolvfile,查一下 Dnsmasq 的 Man Page 就会看到 Dnsmasq 会通过读取这个选项中指定的文件来获取 DNS 服务器列表,如果没有通过这个选项指定 DNS 服务器的配置文件的话,dnsmasq 默认会去读取/etc/resolv.conf 文件读取 DNS 服务器列表.

我们再看看这个/etc/resolv.conf.auto 文件的内容,不同设备可能有区别,但格式基本如此:

```
# Interface WAN
nameserver 8.8.8.8
nameserver 8.8.4.4
# Interface VPN
nameserver 8.8.8.8
nameserver 8.8.4.4
```



这个数据是怎么来的呢? 原来当我们的每一个网络接口成功连接上之后 (interface up), 如果这个接口上有配置 DNS 服务器, 或者 DHCP 服务器通告了 DNS 服务器, 就会在这个文件里写入这个接口的 DNS 服务器. 因为我们自己在 WAN 和 VPN 的接口设定中都指定了 Google DNS, 结果就生成了我们上面的文件.

如果我们要指定 DNS 服务器, 而不是让 Dnsmasq 自动从各个接口获取 (因为 ISP 分配给我们的 DNS 基本都是被污染的), 最简单也是最简陋的方法就是手工在每个接口的配置界面中指定 DNS, 这样自动获取到的也是我们指定的 DNS. 但是如果网络环境发生了变化, 或者需要重新设置接口, 就又要重新设置一次. 而且这实际上就修改到了 /etc/config/network 文件, 倒不是说不能这样改, 只是每个人的网路环境是不同的, 这个文件也会因人而异, 这样让我们很难提供一个统一的解决方案.

还有一个方法就是修改 /etc/config/dhcp 文件, 把最后的 resolvfile 选项改成我们指定的文件, 然后在那个文件里写入 DNS 服务器列表, 但是和上面的原因一样, 修改 config 目录的文件会导致无法提供统一的解决方案.

最后我发现 dnsmasq 提供了一个参数叫做 “no-resolv”, 这个选项允许不从任何文件中读取 DNS 服务器列表, 改用 server 参数手工指定 DNS 服务器. 这个时候我们就只需要在 Dnsmasq.conf(Dnsmasq 的配置文件) 中写如下命令就可以了:

```
#disable resolv file
no-resolv
server=8.8.8.8
```

井号 “#” 开始的部分是注释代码, 会被 Dnsmasq 忽略掉, no-resolv 就是不读取任何任何列表文件, server=8.8.8.8 就表示用 8.8.8.8 作为一个 DNS 服务器.

如果我们想知道 Dnsmasq 到底使用了什么 DNS 服务器, 可以先重启 Dnsmasq 服务:

```
/etc/init.d/dnsmasq restart
```

然后通过 “logread” 命令查看 dnsmasq 启动过程的记录, 所有用到的 DNS 服务器信息都会显示出来.

### 4.3.3 添加多个 DNS

我们知道基本上只有境外的 DNS 才可能给我们返回某些敏感域名的正确解析结果, 国内 DNS 从源头上就被污染了. 但访问境外 DNS 的一个问题是他们的速度普遍比国内的慢, ISP 提供的 DNS 延时可能只有两三毫秒, 而 Google DNS 在国内某些地区的延时高达 400 毫秒 (忽然变 30 毫秒的那是 GFW 劫持). 而且大部分时间我们访问的网站可能还是普通的国内网站, 这些网站用境内 DNS 解析的时候往往会返回给我们一个更合理的数据.

例如淘宝的一些静态文件是存放在 CDN<sup>6</sup>节点上的, 这些节点可能用了类似 “a.tbcdn.cn” 这样的域名. 这些节点在国内外都有分布, 因为这些域名本身的名称服务器有比较智能的动态分配功能, 当你用 Google DNS 去解析的时候, 他们会以为你是一个美国客人, 就返回了一个美国节点的 IP 给你, 而用 114DNS 的时候他们觉得你是个中国客人, 就给你返回了一个中国节点的 IP, 后者明显才是我们要的结果. 如果只是分配到国外节点速度慢也就罢了, 可是淘宝的一些日本节点 IP 在国内根本就被屏蔽了的, 结果就出现了访问 taobao 的时候很多内容加载不了, 页面不完整.

所以我们希望的就是同时使用境内和境外 DNS, 第一步首先是添加这些境内境外的 DNS 服务器:

```
#Local Process
server=127.0.0.1
#Google DNS
server=8.8.8.8
#Norton DNS
server=199.85.127.20
#Comodo DNS
server=8.26.56.26
#DNS Advantage
server=156.154.70.1
#Verizon DNS
server=4.2.2.4
#NTT DNS
server=129.250.35.250
#114 DNS
server=114.114.114.114
#Ali DNS
server=223.5.5.5
#CNNIC
server=1.2.4.8
```

按照 Dnsmasq 的说法, 第一条添加 127.0.0.1 是为了让 Dnsmasq 成为路由器自身的 Dns 缓存, 实际上我测下来这个好像可有可无, 加上也没坏处. 后面我们依次添加了国内和国外公共 DNS. 但是这并不等于工作就完成了, 因为默认情况下 Dnsmasq 只会使用第一个 DNS(127.0.0.1 会被忽略), 也就是 “8.8.8.8”, 这和我们所要的结果完全不同.

我们还需要添加一个参数:

```
all-servers
```

这个参数是告诉 Dnsmasq, 在收到一个 DNS 解析请求之后, 向所有的 DNS 服务器都发起查询, 然后接受并返回最先返回的那个结果给客户端. 所以现在的工作情况是:

---

<sup>6</sup>Content Deliver Network, 一种网络加速技术

1. 对于绝大部分国内的和没被 GFW 盯上的网站, 因为他们的域名解析结果并没有被污染, 所以速度最快的国内 DNS 优先返回了解析结果给我们, 这个结果是真实的, 不会被 iptables 过滤.
2. 而对于被 GFW 污染了的域名, 境内 DNS 返回的信息全是被污染的, 但已经被我们的 iptables 防火墙挡掉了, 这个结果对 Dnsmasq 来说是不可见的.
3. 所以 Dnsmasq 只能继续等境外 Dnsmasq 返回的结果 (GFW 对境外 DNS 的劫持信息会被 iptables 挡掉), 当第一个境外 DNS 返回的正确结果到达后, 就返回给客户端.

#### 4.3.4 屏蔽运营商的 DNS 广告

所以, 我们现在在速度和正确的结果上达到了最佳的平衡. 但是还有个残留的小问题: 这就是如果你在添加境内 DNS 的时候使用了 ISP 的 DNS 的话, 你会发现当你不小心输错域名, 访问了一个不存在的域名的时候, 会被指向到一个运营商提供的网页那里. 这个网页可能是什么电信导航首页, 也可能是他们的合作伙伴的导航首页之类的, 如果你觉得这都没什么问题那这节可以不看.

这种行为本质上其实也是一种 DNS 劫持, 对于不存在的域名, 标准的 DNS 服务器不应该返回任何 A 记录地址, 而且应该在返回包的 Reply Code 中置 03(status: NXDOMAIN), 表示这是一个不存在的域名. 但是 OpenDNS 和国内的运营商的 DNS 就会利用这个机会, 对不存在的域名返回一个他们制定的服务器 IP, 从而引导所有对不存在域名的访问都访问他们的广告商页面, 从而赚取广告收益.

如果你很讨厌这种行为, 那你可以选择不用他们的 DNS, 但他们的 DNS 往往还都是最快的 (毕竟他是你的运营商), 所以如果要既用他们的 DNS 又避免这些广告, 就可以用 Dnsmasq 的一个 bogus-nxdomain 的选项. 这个选项的诞生原因和我们面对的情况几乎是一样的:

```
-B, --bogus-nxdomain=<ipaddr>  
    Transform replies which contain the IP address given into "No such domain" replies. This is  
    intended to counteract a devious move made by Verisign in September 2003 when they started  
    returning the address of an advertising web page in response to queries for unregistered names  
    , instead of the correct NXDOMAIN response. This option tells dnsmasq to fake the correct  
    response when it sees this behaviour. As at Sept 2003 the IP address being returned by  
    Verisign is 64.94.110.11
```

当我们搜集到运营商把不存在域名指向了哪些 IP 之后, 我们用这个选项加入 dnsmasq 的配置中, 如果 DNS 服务器返回的 IP 是这些 IP, Dnsmasq 就认为这些是虚假的信息, 直接忽略它然后告诉客户端这个域名不存在. 关于这些 IP 地址, [这里](#)有一个比较全面的搜集列表, 摘录如下:

```
#Some DNS server will return certain IP for non-exist domain for their advertisement, block these IP  
here.  
#OpenDNS  
bogus-nxdomain=67.215.65.132  
bogus-nxdomain=67.215.77.132
```

```
bogus-nxdomain=208.69.34.132
bogus-nxdomain=208.69.32.132

#DNSPaï
bogus-nxdomain=123.125.81.12
bogus-nxdomain=101.226.10.8

#Nanfang Unicom (nfdnserror1.wo.com.cn to nfdnserror17.wo.com.cn)
bogus-nxdomain=220.250.64.18
bogus-nxdomain=220.250.64.19
bogus-nxdomain=220.250.64.20
bogus-nxdomain=220.250.64.21
bogus-nxdomain=220.250.64.22
bogus-nxdomain=220.250.64.23
bogus-nxdomain=220.250.64.24
bogus-nxdomain=220.250.64.25
bogus-nxdomain=220.250.64.26
bogus-nxdomain=220.250.64.27
bogus-nxdomain=220.250.64.28
bogus-nxdomain=220.250.64.29
bogus-nxdomain=220.250.64.30
bogus-nxdomain=220.250.64.225
bogus-nxdomain=220.250.64.226
bogus-nxdomain=220.250.64.227
bogus-nxdomain=220.250.64.228

#Shandong Unicom (sddnserror1.wo.com.cn to sddnserror9.wo.com.cn)
bogus-nxdomain=123.129.254.11
bogus-nxdomain=123.129.254.12
bogus-nxdomain=123.129.254.13
bogus-nxdomain=123.129.254.14
bogus-nxdomain=123.129.254.15
bogus-nxdomain=123.129.254.16
bogus-nxdomain=123.129.254.17
bogus-nxdomain=123.129.254.18
bogus-nxdomain=123.129.254.19

#Wuhan Telecom
bogus-nxdomain=58.53.211.46
bogus-nxdomain=58.53.211.47

#Nanjing Telecom
bogus-nxdomain=202.102.110.203
bogus-nxdomain=202.102.110.205

#Shanghai Telecom
bogus-nxdomain=180.168.41.175

#Beijing Unicom (bjdnserror1.wo.com.cn to bjdnserror5.wo.com.cn)
bogus-nxdomain=202.106.199.34
```

```
bogus-nxdomain=202.106.199.35
bogus-nxdomain=202.106.199.36
bogus-nxdomain=202.106.199.37
bogus-nxdomain=202.106.199.38
```

```
#Chengdu Telecom
```

```
bogus-nxdomain=61.139.8.101
bogus-nxdomain=61.139.8.102
bogus-nxdomain=61.139.8.103
bogus-nxdomain=61.139.8.104
```

```
#Hangzhou Telecom
```

```
bogus-nxdomain=60.191.124.236
```

如果使用了 ISP 的 DNS, 把这个文件也加入我们的 Dnsmasq 配置项之后, 我们的 DNS 节方案基本就是完美的了. 我前面说了不做任何涉及商业利益的选项, 所以这个文件只会是个选项而不会部署到 FreeRouter V2 中, 在你使用了 ISP 的 DNS 之后可以自己选择是否使用, 如果你不用 ISP 的 DNS, 可以忽略.

#### 4.3.5 限定范围的解析

目前为止我们的 DNS 解决方案都是基于用 iptables 过滤来阻挡 GFW 的污染信息, 虽然这些规则其实已经有效的工作了好几年 (GFW 的更新真的很慢), 但是我们不能保证哪天会不会有新的污染 IP 被投入使用, 这样我们的防火墙规则可能就过滤不了了.

其实我们从一开始就提到了, GFW 可以对 DNS 进行劫持的根源是 UDP 协议本身的不安全, 如果我们可以保证路由器到 DNS 服务器之间的网络是安全的, 那么 GFW 的 DNS 劫持也就没办法进行了. 例如我们现在可以把所有到 Google DNS 的访问都通过 route 命令添加到 VPN 接口进行:

```
route add -host 8.8.8.8 dev pptp-VPN
```

这个时候我们和 8.8.8.8 的通信就是通过加密过的 PPTP 协议进行的了, 目前来说 GFW 还是不能分析和劫持 PPTP 协议的<sup>7</sup>, 这样我们就可以获得正确的 DNS 解析了.

那国内 DNS 怎么办? 不用吧, 速度太慢. 用吧, 走 VPN 也没用啊, 因为服务器上的原始数据就是被污染的. Dnsmasq 的 server 选项还有一个特殊的应用方法, 就是让指定的域名用指定的 DNS 解析, 这样我们可以默认用走 VPN 的 Google DNS 解析, 但对于那些没有被污染又经常访问的域名就指定让国内 DNS 来解析, 例如:

<sup>7</sup> 虽然有研究证明 PPTP 的加密方式强度不够, 如果运用强大的运算力, 花上几天也可以解密 PPTP 协议, 但是我真的不能想象 GFW 要疯到什么地步才会去对所有 PPTP 协议解密, 如果是针对性的解密, 我想大部分人都还没这个待遇

```
##local.conf generated date:Sun Jul 13 20:53:40 CST 2014##  
##All .CN Domain##  
server=/cn/114.114.114.114  
##Alexa Top500 In China##  
server=/baidu.com/114.114.114.114  
server=/qq.com/114.114.114.114  
server=/taobao.com/114.114.114.114  
server=/weibo.com/114.114.114.114  
server=/hao123.com/114.114.114.114
```

我们说了 Dnsmasq 是可以从顶级域开始往下匹配的,server=/cn/114.114.114.114 表示所有的 CN 域名都交给 114 来解析,因为 CN 域名的特殊性(这是中国的国别域名,控制权完全在党国手里),基本上不可能有什么反动网站会用这个后缀的域名,其实稍微有点远见的商业网站也不该用 cn 域名. 在国外注册个 .com 域名,这就是你的私有财产,只要不违反相关的规定谁能动你? 国内虽说有法律,但是,你懂的.

这种限定范围解析的方法,是在 GFW 做重大更新导致现有规则失效的情况下的备选项,但在解析速度和效果上肯定不是最佳的,所以这个限定国内域名的列表我只会作为一个参考选项在 FreeRouter V2 里提供,但并不会导入,如果需要你可以自己决定是否使用.

## 4.4 DNSSEC 的原理和实用性

以下两节原文在[这里](#)和[这里](#),DNSSEC 是未来的 DNS 系统的一个趋势有必要了解一下,但不了解并不影响我们对 FreeRouter V2 项目工作原理的理解. 另外讲解 DNSSEC 原理的时候讲解了前年提到的公钥 + 私钥非对称加密体系,这是网络安全中非常重要的一个概念,也是密码学出现以来最伟大的创造,非常有必要了解一下它的基本原理. 我只整理一下格式稍微合并一下内容.

### 4.4.1 简介

**DNSSEC**是为了解决传统 DNS 系统中的各种不安全性,由**IETF**制定的一套配合现有 DNS 系统的安全扩展系统,目标在于解决各种 DNS 缓存投毒/生日攻击/DNS 劫持等问题,从源头上保证 DNS 数据的正确性和完整性. 这里我大致讲一下它的工作原理,并分析一下这个系统目前的实用性.

### 4.4.2 传统 DNS 系统的弱点

DNS 系统的通讯采用的是 UDP 协议,这种不面向连接的协议适用于快速大量的传输小数据,这也是 DNS 系统采用 UDP 的原因之一. 当然 DNS 也可以使用 TCP 协议,不过那是默认用于 NameServer(名称服务器,以下简称 NS) 之间的 Zone Transfer 的,普通的 DNS 解析器并不会使用 TCP 协议向 NS 发起请

求. 作为整个互联网最重要的基础协议之一,DNS 其实是一个非常脆弱的系统, 它面对的主要可能问题有 DNS 缓存投毒/DNS 劫持/生日攻击/DNS 数据泄露等, 前两者对普通用户的危害是非常恐怖的, 它完全可以把一个银行域名指向黑客的服务器.

这些弱点的根本原因是 UDP 协议是不面向连接的协议, 没人可以保证端对端的传输是安全的, 而 DNS 协议本身在建立初期又实在太纯洁了, 根本没有考虑各种后来可能出现的攻击,DNS 协议本身对数据的验证是非常脆弱的. 基本的验证只有一个 Transaction ID, 这种东西用生日攻击就可以猜测出来, 成功率根本不用你操心; 如果是在局域网, 直接在网关监听甚至可以捕获 Transaction ID, 捕获后直接用于伪造 DNS 解析数据返回给客户端就行了, 在你用 Google DNS 查询 twitter.com 的时候,GFW 就是这么干的.

在讲解 DNSSEC 的原理之前, 我们必须先科普一下:

#### 4.4.3 公钥/私钥加密的基本原理

公钥私钥加密体系是基于这样两个前提:

- 你产生一对公钥/私钥之后, 如果你仅仅持有公钥或者私钥中的一个密钥, 无论你配合明文, 密文还是别的数据, 都不可能推导出另一个密钥.
- 无论公钥还是私钥都可以用于加密数据, 但加密后的数据要想解密, 就必须由另一半完成.

所以公钥/私钥体系的应用主要有两种:

- 加密数据

例如现在 A 要给 B 发送数据, 他就先用 B 的公钥加密消息 (因为公钥是公开的, 人人都可以有的), 然后发送给 B, 即使是广播发送也无所谓, 因为只有 B 才持有 B 的私钥, 所以只有 B 才能解密这个消息.

- 验证签名

例如我要告诉你我是总统, 我就用总统的私钥加密一个签名然后给你看, 因为你们都有总统的公钥, 所以都可以解密这个签名看看是不是我. 其实我用任何私钥加密数据, 你们都可以用总统的公钥解密, 但是只有用总统的私钥加密的数据, 解密后的签名才是有意义的, 否则可能只是一堆乱码, 你们一看: 我靠, 神经病啊, 打死!

#### 4.4.4 DNSSEC 中公钥私钥的应用

在传统的 DNS 系统中, 各种 A 记录,CNAME 记录,MX 记录, 统称为 RR(Resource Record ), 这些 RR 一旦 DNS 服务器发送给解析器, 解析器就无条件接受, 也不管数据对不对 (可能被缓存投毒了), 甚至



到底是不是服务器发回来的 (可能被 DNS 劫持了), 其实也不是解析器想这样无条件接受, 而是解析器除了接受也没办法验证啊.

为了解决这个问题,DNSSEC 增加了 RRSIG(Resource Record Signature,RR 签名), DNSKEY(Domain' s Public Key, 域名公钥),DS(DiaoSiDelegation Signer , 上级授权签名) 这三种记录, 同时从根域名到顶级域名服务器都各自增加了一套公钥/私钥. 我们用一个例子来解释 DNSSEC 如何利用这几个记录.

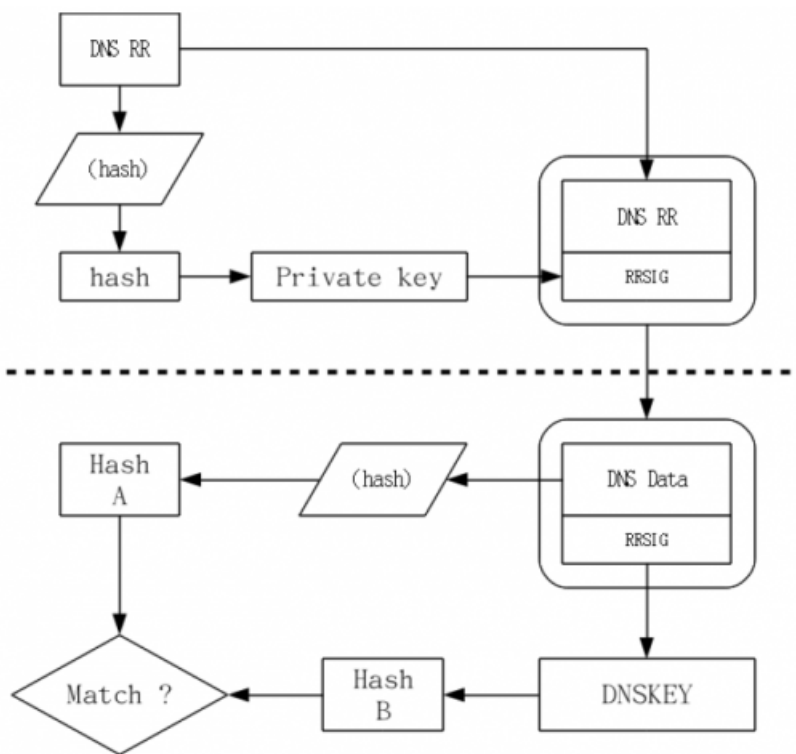


图 4.4: DNSSEC 中 RRSIG 的作用

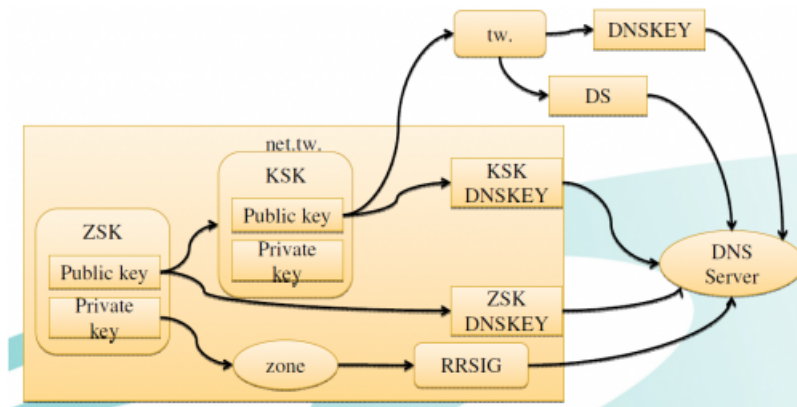


图 4.5: DNSSEC 中 DS 和 DNSKEY 记录的作用

1. 我们 (一台 DNS 服务器) 现在要查询 paypal.com 的 DNS 记录, paypal.com 的名称服务器 (以下简称 PP\_NS) 返回了对应的 A 记录 IP 数据 (RR, Resource Record, A 记录, MX 记录这些记录都统称为 RR) 和一长串字符 (RRSIG), 还有对应的 DNSKEY 和 DS 记录, 那我怎么知道这个 IP 是真的 A 记录, 而不是被修改了的呢?
2. 先看看 RRSIG 是什么产生的, PP\_NS 对 RR 做哈希, 然后用自己的私钥对这个哈希进行加密, 就产生了 RRSIG.
3. 而查询到的 DNSKEY, 就是 PP\_NS 的公钥, 我们用这个公钥对这个 RRSIG 解密就得到了 RR 的哈希值, 然后我们自己再对 RR 做一次哈希, 通过判断两个哈希是否一样就可以知道数据是否正确了.
4. 你也许会想, 我 tm 怎么知道这些 DNSKEY 真的就是 PP\_NS 发回来的, 说不定连你整套公钥/密钥都是黑客伪造的呢? 这里就用到了 DS.
5. DS 的产生是这样的, PP\_NS 把自己的公钥交给给上级的 NS, 也就是 .com 顶级域名的名称服务器 (以下简称 COM\_NS) 那里, 用 COM\_NS 的私钥对 PP\_NS 的公钥加密, 就得到了 DS.
6. 如果你不相信 PP\_NS 的公钥, 你可以去 COM\_NS 那里查 COM\_NS 的公钥, 然后对 PP\_NS 的 DS 记录解密, 如果解密结果和 DNSKEY 对上了, 就说明 DNSKEY 是对的. COM\_NS 这个级别的数据你总该相信了吧.
7. 如果连 COM\_NS 的公钥都不相信, 没关系, COM\_NS 自己也有 DS 记录, 这个 DS 是由跟服务器 (ROOT\_NS) 用自己的私钥加密产生的, 你再用 ROOT\_NS 的公钥, 做一样的解密验证就可以了.
8. 如果你不相信 ROOT\_NS 的公钥, 那你他妈别玩了! 这个公钥是你自己告诉 DNS 服务器的, 根本不用你去查询.

ROOT\_NS 的公钥, 就是一般的解析器和 DNS 服务器里设定的 Trust Anchor, 因为这个数据最终是

你自己设定的, 所以整个验证链最后的安全阀其实在你手里, 但你有责任自己去维护这个 Trust Anchor 的正确性. 如果 ROOT\_NS 的公钥变动, 就必须跟着修改, 否则顶层一级的解密校验就会失败. 这个 Trust Anchor 虽然是你自己输入, 但并不是你随意产生的, 这个实际上已经由 IANA 公布, 可以在[这里](#)找到最新的 Root Trust Anchor.

以上就是 DNSSEC 的大致原理, 简单来说就是从已知的根域开始, 一级级对下级签名, 然后查询者可以通过这个签名链的逆过程验证每一个数据是否真实.

#### 4.4.5 DNSSEC 的最大问题, 是域名签名部署的缺失

一个非常严重的问题是, 整个数据验证过程, 必须是在域名做了签名, 并且向上级 NS 提交生成了 DS 的情况下才可能实现 (没有向上级域提交并通过签名的 Key 是无法验证的, 自然没有意义). 那么现在又多少域名做了这样的签名呢?**千分之三**! 没错, 一个害得老子都不知道怎么打千分号的数据. 你常用的大网站, 什么 google.com, twitter.com, youtube.com, facebook.com, 嗯..... 统统没有 DNSSEC 签名!

造成这个问题的根本原因不是这些网络巨头不知道域名安全的重要性, Google 的 Public DNS 甚至是全球最早支持 DNSSEC 的公共 DNS 服务器之一. 他们的问题在于网站规模过于庞大, 一个域名对应的 IP 数目多得难以想象, 而且为了实现网络的最佳优化, 这些 IP 记录经常会变动, 这就使得 RRSIG 几乎不可能在他们上面实现, 一旦记录出错造成用户无法访问, 每一秒的损失都难以估量.

目前来说, 部署 DNSSEC 的网站主要是 Paypal 这种金融网站和一些政府网站, 因为这些网站的特殊性要求他们在安全性上不能有任何缺失, 而且相对来说他们不会面对 Google, facebook 这些网站的 IP 过多的问题. 至于在顶级域名与以下的各个域名 TLD 的支持, 情况倒还算理想, ICANN 在 2010 年的时候忽然急速推进 DNSSEC 的部署, 使得整个体系的上层都还算健全.

如果一个域名没有做 DNSSEC 签名, 那么 DNSSEC 还能保护它吗?

#### 4.4.6 对未签名的域名, DNSSEC 几乎不能提供任何保护

我们说了, DNSSEC 只是 DNS 的一个扩展, 添加了一套可用于数据验证的域名记录, 并没有对 DNS 协议的数据传输方式和报文格式做任何改进修改. 就好像加入 DNS 原来没有 CNAME 记录, 现在添加了 CNAME 记录的支持, 但是你并没有去用它, 那这个记录对你有用吗? 当然没有.

对于没有签名的域名, 不管你用什么样的解析器, 使用什么样的 DNS 服务器, 你锁查询到的 DNS 记录之前都没有发生任何变化. 只是在你向一个支持 DNSSEC 的 DNS 服务器查询的时候, 他的返回包里会多一个 DO 的 Flag, 表示他知道你的解析器支持 DNSSEC 了, 可是域名本身并没有提供 DNSSEC 数据给你啊, 我也没法给你造出来.

#### 4.4.7 对签名的域名, 中间人攻击依然有可能进行

除了 RRSIG,DS,DNSKEY 这三条记录之外, 支持 DNSSEC 的解析器和 DNS 服务器通信中, 还会增加两个标签位:

- AD(authentic data).DNS 服务器默认会对 RRSIG 的有效性进行检查, 检查确认签名有效, 就会把 AD 位置 1, 下发给解析器.
- CD(Check Disable). 默认情况下, 如果 DNS 服务器返回的 AD 位置 1 了, 客户端就会接受服务器的检查结果, 自己就不再对结果做验证了. 如果客户端不信任服务器的验证数据, 就在请求包中把 CD 位置 1, 告诉服务器老子不用你检查, 签名有没有用我自己来看. 这种情况下服务器返回的数据包中,AD 位会置 0 表示它也不知道这签名是不是有效, 因为你都告诉它不用去检查了嘛, 那他干嘛还犯贱. 虽然这个有点蛋疼, 但是在客户端做签名验证, 才是最安全的做法, 为什么呢?

一个普遍的问题就是, 现在的各种解析器/客户端中, 很多都没有完善自己的签名检查部分, 从上面的原理中你也看到了, 这个签名检查其实是个很蛋疼的过程, 几乎相当于一次递归解析的过程了, 耗时耗力. 所以要么是因为自己没有验证签名的能力, 要么是觉得这个验证过程太消耗时间和资源, 当前的各种解析器/客户端默认都不会发送这个自作多情的 CD(Check Disable). 对于 Dnsmasq 来说, 如果你想让 Dnsmasq 自己验证签名发送 CD 位, 就要启用 “-dnssec-debug” 选项,ManPage 中的建议直接就是 “should not be set in production.”

那么好了, 既然客户端不检查签名, 只要看到 AD 位置 1 就相信, 黑客就太高兴了. 他随便伪造个符合 DNSSEC 标准的返回包 (反正猜解 Transaction ID 什么的早就不是问题了), 修改数据, 把 AD 位置 1, 搞定! 前面这么多服务器的 DNSSEC 部署的努力, 在最后一站全部化作泡影.

对于这个问题, 可能的办法是通过 TSIG(Transaction Signature) 来实现数据传输中的数据完整性验证, 但 TSIG 的问题是解析器和 Recursive DNS 服务器之间也要部署类似 https 加密那样的一套共享密钥, 这个部署工作会是个噩梦. 关于这个方法, 可以参考[相应的论文](#). 当然, 目前来说更实际的做法, 就是 OpenDNS 提出的[DnsCurve](#), 也就是你们可能用过的[DnsCrypt](#), 这是一种加密客户端到 DNS 服务器双向通讯数据的做法, 具体的实现原理请参考[Dnscrypt 官网](#).

另外, 就实用性而言, 中国人还要面对一个很严重的问题:

#### 4.4.8 国内支持 DNSSEC 的 DNS 服务器极度缺乏

你所知道的几乎全部国内的公共 DNS 都是不支持 DNSSEC 的, 当然, 把国内换成国外, 就刚好反过来了. 但使用国外 DNS 的问题除了速度慢, 就是我们上面提到了, 对那些未签名的域名来说, 传输的最后一站中的域名劫持根本无法避免, 在客户端没有验证签名能力或者默认不做本地签名验证的情况下 (这还是现在的默认情况), 就算是签名的域名也躲不掉被劫持的厄运.

总结下来,DNSSEC 的安全性,数据完整性保障,建立在一个由域名/从根域到底层名称服务器/DNS 服务器/解析器组成的完整的安全链上,这个链上的任何一个环节出问题,都会让 DNSSEC 的安全性荡然无存.DNSSEC 作为现有 DNS 系统的增补,有效部署难度极大.要求所有域名都有 DNSSEC 签名这一项就无法通过任何策略来推进,除非 DNSSEC 签名全部免费 (Godaddy 一年的签名要 36 美元),并作为强制标准要求所有域名注册商和 NS 服务器都支持,否则 DNSSEC 始终只能是服务于极小部分域名.

## 4.5 给 Dnsmasq 启用 DNSSEC

### 4.5.1 域名

这是一切开始的前提.域名本身必须有 DNSSEC 签名,对于没有 DNSSEC 签名的域名来说,DNSSEC 几乎不能提供任何额外的保护作用.但目前做了 DNSSEC 签名的域名极其稀少,我们要访问的绝大部分被污染的域名都没有这个签名.这部分是在文章写完一段时间后添加的,现在来看,在域名本身不做 DNSSEC 签名的情况下,指望 DNSSEC 解决任何问题都是妄想.

### 4.5.2 客户端

必须有对应的 DNSSEC 支持工具,OpenWRT 上默认的 `dnsmasq` 是没有的,需要我们替换为 `dnsmasq-full` 才可以,这个可以通过使用 OpenWRT 的 Image Generator 的 PACKAGES 参数来实现,移除默认的 `dnsmasq`,添加 `dnsmasq-full`.

```
PACKAGES="-dnsmasq dnsmasq-full"
```

### 4.5.3 服务器

必须是一个支持 DNSSEC 扩展的 DNS 服务器,这种服务器目前在国内非常少,但国外非常多,我知道 Google DNS 是支持的,而 OpenDNS 是明确不支持的,这个还一度给我添加了一些麻烦.

### 4.5.4 Dnsmasq 的配置:

在 `/etc/dnsmasq.conf` 中,或者如果你在 `dnsmasq.conf` 中通过 `conf-dir` 参数指定了一个配置目录的话,就在那个目录下添加一个 `*.conf` 文件,写入以下参数:

```
dnssec
```



这个参数是基本的启用 DNSSEC 的基本参数, 表示要求查询域名记录的 DNSSEC 扩展信息.

```
trust-anchor=.,19036,8,2,49AAC11D7B6F6446702E54A1607371607A1A41855200FD2CE1CDDE32F24E8FB5
```

这个参数是配置默认的 trust-anchor, 是使用 dnssec 必须加入的参数, 具体说明参考[dnsmasq man page](#), 配置内容可以从[IANA 网站](#)获得. 这个相当于数据加密解密中的公钥 (public-key), 理想情况 (所有子域名都部署了 DNSSEC) 下, 我们只需要添加根域 (root zone) 的 public-key 就可以了, 现在常用的各个顶级和国别域名的部署情况已经基本达到理想状况了.

```
dnssec-check-unsigned
```

这个参数需要非常注意! 因为 DNSSEC 默认只会对有 DNSSEC 签名的返回结果去做验证, 如果一个返回结果是未签名的,Dnsmasq 默认会相信这个数据是真实有效的, 并不会去管这个域名是不是真的没有签名, 还是别人伪造了数据包. 如果启用下面这个选项,Dnsmasq 会对未签名的返回数据做检查, 确定域名是真的没有签名, 而不是一个有签名的域名被人伪造了无签名的数据来欺骗你.

要检查域名是真的没签名还是假的没签名, 就要求 DNS 服务器必须支持 DNSSEC, 否则一切工作都无法进行, 域名就无法解析. 对于 114 这类不支持 DNSSEC 的 DNS 来说, 它返回的所有数据都是没有签名的, 而 Dnsmasq 又无法通过 114 DNS 去验证无签名的真实性, 那么所有的域名解析都会失效.

```
dnssec-no-timecheck
```

DNSSEC 对于服务器返回的签名数据是有一个信赖时间窗口的, 过了这个时间窗数据就不再被信赖. 但对于一些没有 RTC 的机器来说, 这就产生了一个蛋生鸡鸡生蛋的问题, 因为他们需要通过 NTP 服务器来获取时间, 可是获取 NTP 服务器的地址又必须先解析 NTP 服务器的域名, 可是现在机器没有正确的时间, 导致信赖时间窗口无效, 又解析不了正确的域名地址, 这就麻烦大了. 下面这个参数对于 OpenWRT 这样的路由来说都应该默认启用, 启用之后, 在系统获得正确的时间之前, 解析结果不受信赖时间窗口的限制, 等系统获得正确的系统时间之后, 这个信赖时间窗口机制才重新开启. 配置完成之后:

```
/etc/init.d/dnsmasq restart
```

重启 Dnsmasq 服务即可生效.

我刚说 OpenDNS 给我造成了一点小麻烦, 就是我在 OpenWRT 中, 同时设置了 GoogleDNS 和 OpenDNS 作为默认的 DNS, 没有设定严格按顺序查询, 也许是 OpenDNS 的速度稍微快一点 (我这里 Google DNS 延迟 400 多毫秒) 结果 Dnsmasq 默认用了 OpenDNS, 而 OpenDNS 又不支持 DNSSEC, 结果就是我上面对 “dnssec-check-unsigned” 的解释,DNS 解析服务完全失效了.

在注释掉 “dnssec-check-unsigned” 这个参数后可以解析数据了, 但你用 DNSSEC 配合一个不支持 DNSSEC 的服务器解析数据, 怎么可能避免污染呢, 这让我一度以为是 Dnsmasq 的 DNSSEC 没有工作.

在把 OpenDNS 从 WAN 和 VPN 接口的设置中都删除之后, 确保了系统只用 Google DNS, 现在 DNSSEC 才开始工作了.

#### 4.5.5 DNSSEC 的效果:

首先各种有效性测试你必须在路由后面的设备上进行, 你直接在路由上用 dig 或者 nslookup 查询的话, 这个结果是不完全依赖 Dnsmasq 的. Dig 也好, nslookup 也好, 这些都不等同于你系统在使用中的 resolver(dnsmasq), 虽然他们可以帮你检查 resolver 的工作情况, 但这二者本质上是不同的东西, 这个基本概念必须清楚

因为 Dnsmasq 作为 resolver, 他有自己的 dnssec 验证机制, 对于不合格的数据会丢弃, 但 dig 和 nslookup 只是个程序, 他只处理并显示收到的 udp 或 tcp 包, 它们并不知道去对 dnssec 的数据做验证和丢弃.

**被 GFW 劫持后的 DNS, 会从一个没有配置 DNSSEC 的服务器上优先返回虚假的数据, 这就出现了一个极大的矛盾:**

1. 如果我们不启用“ dnssec-check-unsigned” 这个选项, 那么 Dnsmasq 就会接受 GFW 劫持后的数据, 防护彻底丧失.
2. 如果我们启用“ dnssec-check-unsigned” 这个选项, GFW 伪造的返回包是不包含 DNSSEC 支持信息的, Dnsmasq 会认为这是一个不支持 DNSSEC 的服务器返回的结果, 从而丢弃这个数据. 但是 114 之类的国内 DNS 也都是没有 DNSSEC 的, 他们的解析结果也同样会被丢弃, 那么我们就无法把国内的域名交给这些国内 DNS 解析来解决 CDN 和解析速度问题.

因为我们常用的, 被污染的域名几乎都没有做 DNSSEC 签名, 所以 DNSSEC 几乎不能给我们提供任何帮助. 现在能起作用, **也仅仅是因为 GFW 伪造的数据包是一个不支持 DNSSEC 的 DNS 服务器的格式**, 在开启 dnssec-check-unsigned 这个选项的时候, Dnsmasq 会像丢弃 114DNS 的解析结果一样丢弃了 **GFW 的伪造包**. 如果哪天 GFW 更新, 同时发送无 DNSSEC 和有 DNSSEC 的伪造包, 那 DNSSEC 就完全帮不上忙了, 除非那些域名本身有 DNSSEC 签名.



## 第五章 FreeRouter 实战

“如果没有爱, 一个故事怎么会有幸福的结局?”

### 5.1 FreeRouter V2 的工作流程

我们前面已经把 FreeRouter V2 所用到的技术全部讲完了, 在开始实际部署前先总结一下整个工作流程:

1. 客户端发起访问 Twitter.com 的请求, 同时发出 DNS 解析请求, 请求送达路由器
2. 路由器上的 Dnsmasq 向所有 DNS 服务器发起解析请求
3. GFW 抢先返回虚假解析结果, 被 iptables 防火墙拦截
4. 真正的 DNS 服务器返回的数据随后到达, 被 Dnsmasq 接受
5. Dnsmasq 发现 twitter.com 在配置的 IPSET 列表里, 将这个解析的 IP 加入 IPSET
6. iptables 防火墙把目标地址属于这个 IPSET 的数据全部打上标签
7. ip rule 规则把打上标签的数据全部加入一个 table
8. ip route 把这个 table 里的数据全部转发到 VPN 接口
9. 对 twitter.com 的访问数据全部通过 VPN 接口进行, 绕过 GFW 的 IP 阻断

如果文字不好理解, 看图吧:

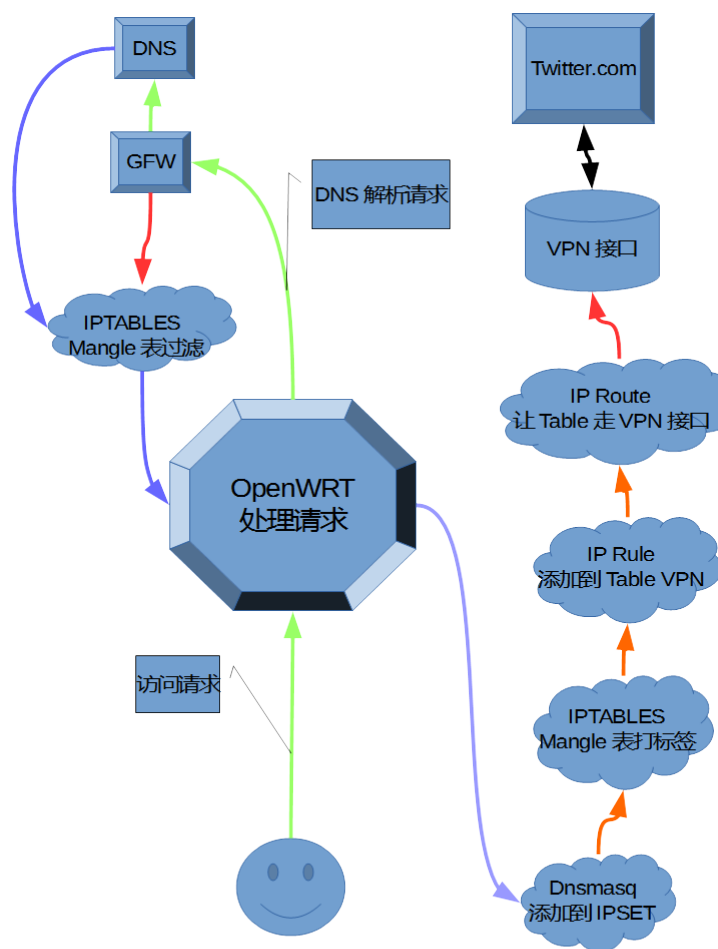


图 5.1: FreeRouter V2 工作流程

## 5.2 OpenWRT 基础

### 5.2.1 OpenWRT 是什么?

OpenWRT 的官方网站在这里:<https://openwrt.org/>.

首先你要知道 OpenWRT 是什么,OpenWRT 是一个基于 Linux 内核的第三方路由器操作系统,和一般的桌面 Linux 系统不同,路由器的操作系统是实时系统<sup>1</sup>,但什么是实时系统其实和你们关系不大,不用管它了.你要知道的是,OpenWRT 能够成为一个路由器的系统,除了内核之外主要依靠的是大量的 Linux 网络工具和内核模块,是这些工具和模块提供了 OpenWRT 的网络管理能力,要使用特定的功能就必须在 OpenWRT 中又对应的软件包 (Package) 支持.

所谓的第三方,指的是一般的路由器在出厂前基本都不会预装这个系统,如果你要使用,就要像手机刷机一样自己把 OpenWRT 固件刷进你的路由.如果你缺乏 OpenWRT 的基础知识,请去阅读 OpenWRT 的[官方 Wiki](#),我不会去帮你解答和项目无关的 OpenWRT 的问题.

### 5.2.2 我的路由能刷 OpenWRT 吗?

OpenWRT 提供另一个支持 OpenWRT 的[硬件 Wiki 列表](#),注意 ipad 上的 safari 可能打不开这个页面,换 chrome 或者用桌面版的其他浏览器.当然这个 wiki 的更新有时候并不及时,你可以直接去他们的下载站点,根据自己路由器的 CPU 型号,去 trunk 目录里,选择对应的目录查看是否有自己的路由器对应的固件,地址是:

<http://downloads.openwrt.org/snapshots/trunk/>

最新的代码和最新添加的硬件支持都在 trunk 目录里,但 trunk 目录也意味着这是开发中的代码,可能伴随着一些未知的 bug 和不稳定性.

一般来说,OpenWRT 支持最好最全面的是 Atheros 的 AR71xx 系列的 CPU,其次是 BroadCom 的一些老的型号 CPU.你也许可以再国内的一些论坛找到一些没被 OpenWRT 官方支持的路由器的 OpenWRT 固件,虽然他们提供了固件,但 OpenWRT 官方未必提供了对应芯片的 Package 支持,要知道每种芯片对应的 Package 是要用各自芯片的 SDK 重新编译才能用的.所以,除非你动手能力超强能自己搞定所有的 package 编译,我从来都不会建议你买这些没官方支持的路由.

---

<sup>1</sup>Real Time OS

### 5.2.3 我的路由需要多大 ROM?

实现一个有管理界面 (luci), 能安装完 FreeRouter V2 全部功能脚本的路由, 需要你的路由器至少有 4M 的 Flash 空间, 一般而言这个都是可以满足的. 如果你希望充分发挥路由的功能, 例如添加文件共享, USB 共享, 脱机下载这些额外的功能, 建议至少有 8MB 的空间, 这些东西和本项目无关, 不要来烦我. 如果你有兴趣研究 GFW, 或者学习 linux 的网络管理, 那么你可能需要安装一些 linux 的调试跟踪工具在路由上, 这样也至少建议有 8MB 的空间.

### 5.2.4 我应该下载选哪个版本的固件?

这里的版本有两个概念, 一个是真正的版本, 你可以看到在他们的下载[目录首页](#), 有 backfire, attitude\_adjustment, 这些目录, 这些都是 OpenWRT 的重大版本更新, 类似 windows xp, windows 7 这样的概念, 当前最新的正式发布的版本是 attitude\_adjustment, 正在开发中的 barrier\_breaker 已经发布了[RC2 版本](#). 目前 barrier\_breaker RC2 已经很稳定了, 当然 trunk 目录中的最新代码可能会因为导入新功能而产生之前没有的 bug, 并不是说最新就一定最稳定, RC 版的稳定性一般来说比 trunk 版要高.

另外一个版本概念, 就是固件的文件系统格式和升级方法. 一些老的路由上可能只有 jffs 文件系统的支持, 而新的一般都支持 squashfs. 你不需要知道这两者的具体内容, 只需要知道 squashfs 允许你对整个路由文件系统进行操作, 而 jffs 文件系统只允许你对一个叫 jffs 的文件夹的内容进行修改, 这是非常不方便的, 本书的讨论默认基于 squashfs 文件系统进行.

而在升级方式上, 如果你路由现在的固件还是出厂的固件, 就应该选择文件名后面带 factory 的, 表示从原厂固件开始升级, 升级之后就是 OpenWRT 系统了. 如果你已经刷成 OpenWRT 系统了, 只需要下载 sysupgrade 版本就行了.

### 5.2.5 如何刷固件?

在前面提到的那个[wiki 页面](#)中, 点击进入每个支持 OpenWRT 的路由的 wiki 页面后, 一般都会有很详细的刷机过程指导, 大部分情况下你只需要下载 factory 版的固件, 然后从原厂固件的固件升级页面直接升级就可以了. 具体操作, 请自行阅读相关的刷机说明.

### 5.2.6 为什么刷了固件后打不开路由管理界面?

第一次刷完官方的 OpenWRT 固件后, 路由器的地址一般都是 192.168.1.1, 如果不确定, 可以用前面教你们的 traceroute 命令查看第一个路由节点是什么. OpenWRT 官方的固件默认都不提供 luci 管理界面,

只提供 23 端口的 telnet 进行远程管理. 这个端口默认没有密码, 用 telnet 登录上之后你可以自己安装 luci 管理界面.

### 5.2.7 如何安装各种 package?

在 OpenWRT 下安装各种软件和在 ubuntu 下安装软件几乎是一样的, 只是命令有些不同, 以安装 luci web 管理界面 package 为例, telnet 登录到路由之后, 执行如下命令:

```
opkg update
```

这个过程是从 openwrt 官方更新源列表, 对于 trunk 版的固件, 经常会因为 trunk 目录里的代码更新导致版本需求和你现有的固件不匹配, 所以你最好能学会后面自己打包固件的方法.

```
opkg install luci
```

这是安装 luci 管理界面, 如果一个 package 有什么其他依赖的 package, 也会一并被安装, 不用你额外操心.

```
opkg install luci-ssl
```

如果你需要的话, 还可以安装 luci-ssl, 提供 ssl 加密的管理界面登录.

```
opkg install luci-i18n-chinese
```

如果你需要中文界面, 就安装对应的中文支持.

luci 管理界面本身的运行需要一个 http 服务器组件叫做 uhttpd, 在你安装 luci 的时候默认就会安装这个组件, 但它不会自己启动, 所以你要手工启动它:

```
/etc/init.d/uhttpd start
```

为了让以后每次重启路由都自动启动 uhttpd 服务, 需要使能这个服务:

```
/etc/init.d/uhttpd enable
```

更完整的使用 OPKG 命令安装 package 的指导, 请参考[OpenWRT 官方页面](#)介绍.

## 5.3 FreeRouter V2 需要的组件

FreeRouter V2 使用了一些普通人可能用不到的组件, 依次介绍必要的组件:

- luci, 可选项目:luci-ssl,luci-i18n-chinese,luci-app-upnp luci-app-qos luci-app-samba

这是路由的管理界面, 前面已经提到了, 可以方便我们添加管理各种接口, 虽然不是必须的, 但有这个确实方便很多. 后面几个可选项目, 分别是管理页面的 ssl 支持, 中文支持,upnp 管理,qos 管理, 和 samba 共享管理.

- ipset

用于创建 ipset

- kmod-ipt-ipopt

iptables 对应的内核模块, 用于实现让 iptables 给数据添加标签 (fwmark)

- ip

用于 ip rule 添加和 ip route 路由表管理

- iptables-mod-filter 和 kmod-ipt-filter

iptables 的 string 模块和对应的内核支持模块

- iptables-mod-u32 和 kmod-ipt-u32

iptables 的 u32 模块和对应的内核支持模块

- ppp-mod-pptp

用于连接 PPTP VPN 的接口模块

- dnsmasq-full

这个需要特别注意!OpenWRT 官方固件自带的 Dnsmasq 是 lite 版本, 不带 IPSET 功能, 也没有 DNSSEC 支持 (虽然这个没什么太大用处), 必须替换成 dnsmasq-full 版本才可以使用.dnsmasq-full 是官方提供的版本, 不需要自己重新编译.

以下都是可选项目, 适合喜欢折腾的人, 普通用户不需要这些东西:

- bash 和 bash-completion

OpenWRT 自带的 shell 是 Ash, 非常简陋, 很多命令和功能都不能使用, 限制也很多, 你可以自己安装 bash. 如果需要把默认的 shell 替换成 bash, 需要修改/etc/passwd 文件, 把里面的 “/bin/ash” 改成 “/bin/bash”, 重启后就可以生效了.

- vim-full 和 vim-runtime

vim 是这个世界上最好的编辑器, 没有之一, emacs 逆党速速退散! SlickEdit 用户们, 跟我一起去抗议 SE 一年 60 刀的维护费吧, 太黑了.

- bind-dig 和 bind-libs

安装我们前面说到的 Dig 命令工具

- curl

我的某些脚本会用到 curl, 这个东西比较大, 大概有 1MB.

## 5.4 部署 FreeRouterV2 的文件

### 5.4.1 系统的基本设置

刷好固件后首先从 luci 界面登录 (假定你已经安装好了所有必需的 package), 默认是没有密码的:

OpenWrt

**未设置密码!**  
尚未设置密码。请为root用户设置密码以保护主机并开启SSH。  
[跳转到密码配置页...](#)

**需要授权**  
请输入用户名和密码。

用户名

密码

Powered by LuCI Trunk (svn-r10459) OpenWrt Barrier Breaker r41679

图 5.2: luci 登录界面



登录后按提示跳转到密码设定页面, 设定路由器管理密码, 点击“保存应用”。这个密码也是你用 ssh 管理路由, 和用 winscp 部署文件时用的密码, 用户名默认是 root.



图 5.3: 密码设定页面

第一次设定密码后稍微等待几分钟, 因为后台生成 SSH Key 的时间非常长, 然后就可以用你习惯的 Shell 工具登录了, Cygwin 就是很好的工具, 文件管理在 Windows 下推荐用免费的 WinSCP.

## 5.4.2 VPN 接口设定

依次点击管理页面上方的“网络”-“接口”, 点击左下方的“添加新接口”, 接口名称随便填, 协议选择 PPTP.



图 5.4: 开始建立 VPN 接口

按照你自己的 VPN 服务器信息和用户名密码填写:

**接口 - VPNX**  
配置网络接口信息。

一般设置

基本设置 高级设置 防火墙设置

状态 pptp-VPNx 接收: 0.00 B (0 类) 发送: 0.00 B (0 类)

协议 PPTP

VPN服务器 1.1.1.1

PAP/CHAP用户名 user

PAP/CHAP密码 password

图 5.5: 填写服务器 IP, 用户名和密码

在高级设置中, 取消勾选 “使用默认网关”:

一般设置

基本设置 高级设置 防火墙设置

开机自动运行 ☒

Use builtin IPv6-management ☒

使用默认网关 ☐ 留空则不配置默认路由

使用端局通告的DNS服务器 ☒ 留空则忽略所通告的DNS服务器地址

LCP响应故障阈值 0

图 5.6: 不要选使用默认网关

防火墙设置中, 把 VPN 接口设置在和 WAN 接口同一个区域, 否则 VPN 接口会被防火墙限制在内网:



图 5.7: 配置 VPN 接口的防火墙区域

全部设置完之后点击保存应用, 如果正常连接上的话, 应该可以看到这个接口有数据, 否则数据都是 0. 通过 SSH 登录后, 如果 VPN 连接成功, 使用 `ifconfig` 命令应该可以看到我们自己刚才设定的 VPN 接口, 名称应该是 `pptp-` 我们设定的接口名字”。

### 5.4.3 部署文件

使用 WinSCP, 用 SSH 的用户名和密码登录进路由, 把 FreeRouter V2 项目中 OpenWRT 目录下的文件, 上传到路由器的 ROM 中. OpenWRT 目录下的文件已经是按照路由文件系统的目录结构组织好的, 一定要保持这个结构不变, 每一个文件的位置都不能变动。

这些文件的说明如下:

- `/etc/dnsmasq.conf`

这是 dnsmasq 的配置文件, 里面的内容只有一行, 就是把 dnsmasq 的配置目录设定为 `/etc/dnsmasq.d`, 表示让 Dnsmasq 从 `/etc/dnsmasq.d` 这个目录去读取配置文件. 注意这个目录下的所有文件都会被 Dnsmasq 启动的时候读取作为配置项, 所以不要在这个目录下放任何不是配置文件的东西, 这会让 Dnsmasq 启动失败。

- `/etc/dnsmasq.d/gfw.conf`

这个文件里包含了被封锁的域名的列表, 以及把他们的 IP 导入一个名叫 `vpn` 的 IPSET 的选项. 如果你发现哪个网站不能访问, 可以用同样的格式添加进去, 重启 Dnsmasq 并清空你本地的 DNS 缓存就可以了。

- `/etc/dnsmasq.d/option.conf`

这个文件包含了 Dnsmasq 的配置选项, 包括禁止读取 `resolve` 文件, 使用全部 DNS 服务器, 以及启用 DNSSEC 等。

- /etc/dnsmasq.d/server.conf

这个文件包含了我们要用到的全部 DNS 服务器, 默认包含了一些知名的国外公共 DNS 和国内公共 DNS.

- /etc/iproute2/rt\_tables

这个文件里, 我们设定了一个名字叫 vpn 的 route table, 把他的 ID 设置为 10.IPSET 的名称和 table 的名称并不需要一样,table 名称和相关的 ip 命令对应,IPSET 名称和相关的 iptables 命令及 dnsmasq 选项对应.

- /etc/firewall.user

这个文件包含了我们所有的防火墙过滤规则,IPSET 的建立命令也是写在这个文件里 (会自动判断 IPSET 是否已经建立过而跳过), 每次防火墙启动或者重启的时候这个文件里的命令都会被执行. 这里的命令执行时会往/tmp/vpn.log 这个 log 文件写入日志, 便于我们调试.

- /etc/ppp/ip-up.d/vpnup.sh

这个脚本命令会在 VPN 连接成功的时候自动被调用, 注意目录名称绝对不能改, 必须是 ip-up.d, 如果你的路由上没有这个目录 (一般刚开始都没有), 可以直接把整个目录复制过去, 或者手工建立. 这个脚本里包含了 ip rule 规则,ip route 规则, 以及在最后重启一下 Dnsmasq 服务. 这个脚本也会在/tmp/vpn.log 文件里写入日志信息.

- /etc/ppp/ip-down.d/vpndown.sh

这个脚本的命令在 VPN 断开时会自动执行, 主要是把 vpnup 中建立的 ip rule 规则删除, 等 vpn 重新连上后 vpnup.sh 脚本会重新建立这些规则. 其他的说明和 vpnup.sh 一样.

建议不要用覆盖整个文件夹的方式, 因为你获得的代码在各个操作系统之间传送,/etc 目录的属性已经不确定,总之你最终要保证/etc 目录的属性是 755, 这非常重要, 因为属性如果不是 755, 会有很多服务在启动时没有读取/etc 目录的权限继而导致启动失败. 你可以用 winscp 直接修改目录属性为 755, 也可以用 SSH 中的 chmod 修改这个目录的属性.

另外,/etc/ppp/ip-up.d 和/etc/ppp/ip-down.d 目录下的两个 \*.sh 脚本, 必须确保他们有可执行权限, 可以通过命令下面的命令来修改:

```
chmod +x \etc\ppp\ip-up.d\vpnup.sh
chmod +x \etc\ppp\ip-down.d\vpndown.sh
```

#### 5.4.4 调试和检查

包含必须组件的固件刷好后, 接口设定完, 文件部署完, 重启路由应该就可以开始工作了. 我在脚本中写入了一些基本的调试信息, 这些信息会写入/tmp/vpn.log 文件中, 如果全部正常启动的话, 查看该文件应该看到类似如下的信息:

```
root@OpenWrt:cat /tmp/vpn.log
Firewall start @20:53:52@2014-07-29
Create IPSET @20:53:52@2014-07-29
Add PREROUTING Rules @20:53:52@2014-07-29
Add DNS Purify Rules @20:53:52@2014-07-29
VPN UP @20:58:10@2014-07-29
Add VPN device for table vpn @20:58:10@2014-07-29
Add IPs marked by firewall to table vpn @20:58:10@2014-07-29
```

从 Firewall start 到 Add DNS Purify Rules 都是 firewall.user 里的命令, 后面的命令是 vpnup.sh 的命令, 如果哪个文件命令对应的 log 没有出现, 请检查文件的部署是否正确, 以及文件是否有可执行权限.

更细致的检查命令我在前面的章节都已经说过了, 这里不再重复. 基本上只要文件位置正确, 文件权限属性正确, 不会有什么问题.

需要强调的一点是:你在客户端机器上 (你的 PC, 手机, 平板) 绝对不能设置任何指定的 DNS,DNS 必须设置成自动获取, 因为手工指定 DNS 会使得 DNS 解析过程绕过 Dnsmasq, 导致域名 IP 无法被搜集到. 而且你手工指定 DNS 也不可能获得比自动获取 DNS 更好的解析效果, 如果非要手工指定, 就指定成路由器的 IP 好了, 这和自动分配的结果是一样的, 但不会让解析数据被绕过.

### 5.5 自己打包固件

如果你经常折腾路由, 可能一不小心就把配置文件弄乱了, 如果一不小心弄得太乱了, 可能就只有恢复出厂设置了, 想想一切又要从冰冷漆黑的 telnet 界面重新开始就心寒啊. 如果我们可以把 FreeRouter V2 所需的 Package 和配置文件直接打包到路由器固件中, 那么如果出了问题, 只要直接恢复出厂设置就可以了; 什么设置接口, 安装 package, 部署文件都不用操心了, 简直就是路由器的一键还原.

你要庆幸我们用的是 OpenWRT, 对 OpenWRT 来说自己打包一个系统实在是太容易了, 因为官方就给我们提供了现成的工具:Image Generator<sup>2</sup>.RC 版本的 Image Generator 一般都不提供完整的 Package 文件, 所以我们不考虑, 你可以选择稳定的 attitude\_adjustment 的版本的 Image Generator, 或者最新的开发版本 Image Generator, 以 AR71xx 芯片的为例, 他们的下载地址分别是: [Atitude Adjust 版本](#), [Barrier Breaker 开发版本](#).

---

<sup>2</sup>也叫 Image Builder

忘了说一下,Image Generator 是一个 Linux 下的工具,Windows 用户想用的话先装虚拟机安装个 ubuntu 什么的.OpenWRT 官方有这个工具的[详细使用方法](#),我只举个实例介绍一下,如果不清楚请自己去官方 wiki 查看.

下载完之后,先解压这个压缩包:

```
tar -xjvf OpenWrt-ImageBuilder-ar71xx_generic-for-linux-x86_64.tar.bz2
```

进入刚刚解压生成的目录:

```
cd OpenWrt-ImageBuilder-ar71xx_generic-for-linux-x86_64
```

就在这个位置,不要动了,我们先查看一下这个 Image Builder 支持那些型号路由固件的生成,输入命令:

```
make info
```

我们会看到如下的一些信息:

```
WNDR3700:
  NETGEAR WNDR3700/WNDR3800/WNDRMAC
  Packages: kmod-usb-core kmod-usb-ohci kmod-usb2 kmod-ledtrig-usbdev kmod-leds-wndr3700-usb
WNR2000V3:
  NETGEAR WNR2000V3
  Packages: kmod-usb-core kmod-usb-ohci kmod-usb2 kmod-ledtrig-usbdev
WNR612V2:
  NETGEAR WNR612V2 / On Networks N150
  Packages: kmod-usb-core kmod-usb-ohci kmod-usb2 kmod-ledtrig-usbdev
```

因为很多固件虽然型号不同,但内部大同小异是一个系列的,所以在 Image Builder 里被列入了一个 PROFILE,例如 WNDR3700 和 WNDR3800 就是统一用了 WNDR3700 的 PROFILE 名称,但最终生成固件的时候还是会生成不同型号名称的固件.下面的 package 是指除了 OpenWRT 系统必须的 package 之外,这个 PROFILE 默认还打包了什么 Package,例如 WNDR3700 系列的就加入了“kmod-leds-wndr3700-usb”这个独有的 USB 指示灯模块.

如果我们要生成某个 PROFILE 下所有型号路由的固件(以 WNDR3700 系列为例),只需要执行如下命令:

```
make image PROFILE='WNDR3700'
```

生成的固件会在 ./bin/ar71xx 目录下,但是这样生成的固件和官方提供的固件几乎是一样的,我们可以通过 PACKAGE 选项添加我们要的 package,删除不要的 package.例如我们需要 dnsmasq-full 版,而不需要默认的精简版 dnsmasq,同时需要把其他 FreeRouter 必须的 package 都打包进去,就可以这样写:

```
make image PROFILE="WNDR3700" PACKAGES="luci luci-ssl luci-i18n-chinese -dnsmasq dnsmasq-full ip
ipset iptables-mod-ipopt iptables-mod-filter kmod-ipt-filter iptables-mod-u32 kmod-ipt-u32 ppp-
mod-pptp"
```

如果我们还想把配置文件也一起打包进去呢? 加上 FILES 参数就行了,

```
make image PROFILE="WNDR3700" PACKAGES="luci luci-ssl luci-i18n-chinese -dnsmasq dnsmasq-full ip
ipset iptables-mod-ipopt iptables-mod-filter kmod-ipt-filter iptables-mod-u32 kmod-ipt-u32 ppp-
mod-pptp" FILES="Files/"
```

注意这个 Files 目录现在是放在了 Image Builder 的根目录下的, 你放在别的路径就要修改对应的参数. 现在 Files 目录就相当于 OpenWRT 的根目录, 你要打包的配置文件, 必须按照在 OpenWRT 的 Flash 里出现的位置放置在 Files 目录里. 一般而言, 除了 FreeRouter V2 里涉及的文件之外, 你的配置信息都保存在 /etc/config 目录下, 把这个目录从 OpenWRT 上下载下来, 放进 /Files/etc/ 目录下就行了 (注意目录结构一致).

#### 刷写新的固件

上传兼容的sysupgrade固件以刷新当前系统。

保留配置: ☐

固件文件:  openwrt-ar71xx-generic-wndr3800-squashfs-sysupgrade.bin

图 5.8: 升级时不要保留配置

注意在更新固件的时候, 如果我们勾选了保留配置, 那么有些文件是会被新固件里的文件覆盖的, 例如你现有的用户名和密码就不会被覆盖, 还有一些 config 目录下的配置也不会被覆盖. 如果这给你带来了麻烦, 例如某些 config 目录下的东西没有被更新, 那么为了确保所有文件都被替换就不要勾选这项. 当然如果你只是更新其他部分的内容 (例如被封锁域名的列表), 可以保留配置, 因为如果不保留配置的话, 我们的密码会被清空的, 到时候又要重新设置密码.



## 关于未来

我的工作本身并不是软件工程师, 虽然我可能会写一些单片机的代码, 但我的主要工作还是画电路板. 我交代这个背景不是要为手册中可能的缺失和错误作辩解, 我只是想嘲笑一些拿着程序员的名片却毫无学习精神的人, 是的, 我就是在嘲笑他们. 在这个世界上, 做成一件事重要的不是你的背景, 史蒂芬 • 周很早就说过一切的关键只有两个字: 用心.

写这个手册的目的是给我自己建一道防火墙, 挡掉几乎所有关于 FreeRouter V2 项目的一些令人反感的疑问. 最主要的问题是可能有人担心这个项目会不会留有后门, 我想在一切技术手段和原理都解释清楚后应该不会有人再提这样贬低自己智商的问题了. 另外 github 上目前有限的几个 issue 大部分都让我看得很头痛, 到底是中国人普遍英语水平太烂还是他们都有一套自己的字典呢? 因为我觉得很多人似乎根本就搞不懂 issue 这个词的意思, 我尊重你的智商所以我不会替你翻译. 什么时候你们才能像一个硬件工程师一样对 issue 这个词产生敬畏呢?

如果在 github 提交的 issue 里看到以下问题, 除非我爱你, 我会直接 close 并 lock 掉, 绝对不回复.

- 和项目本身毫无关系的内容的. Issue 不是留言板, 你如果需要通过 issue 才能联系到我, 那你应该知道我不爱你.
- 要现成固件和其他现成成果的. 原理已经全部说明了, 连打包固件的方法都教给你了, 你能学会尊重自己的能力吗? 要知道我不欠你钱.
- 询问所有我在本手册中已经解释过的问题的. 既然你无视我的劳动成果, 那我为什么要尊重你?
- 言语缺乏基本教养的人. 我只是替你父母感到难过, 我一直都觉得说脏话只是丢自己的人而已.

欢迎就以下问题提出响应:

- 代码和规则中的错误和可以改进的地方, 注意我不是要你给我提需求
- 发现 GFW 的 DNS 劫持有了新的 IP 和格式, 以及其他新出现的 GFW 的手段

感谢每一个看完本手册的人!