文末有福利 | HTTPS 劫持漫谈：代理劫持与透明劫持

原创 V1ll4n CTstack 安全社区 2020-11-12 19:08

本文共计4719字，阅读时间预计10分钟

HTTPS 劫持其实是一个老生常谈的话题，从上古时代的 HTTPS 降级（maybe sslstrip），到现在很稳定的安全工具 `burpsuite`，`xray` 等，以及一些常见的网络调试工具 fiddler、charles、surge，很多都直接或者间接利用了 HTTPS 的 MITM 攻击。

在很多朋友尝试 HTTPS 中间人的时候，其实都有过这样的想法：“我挂在路由器上，让我所有的的 HTTPS 流量全都变成明文！”。但是往往受限于性能问题以及部署问题，导致了很多人“只是想想”。

与此同时，不管是 HTTP 还是 `HTTPS 的劫持`（点题了qwq），除了解决安全工程问题，其实还有更多别的用途：比如你连入了一个 WIFI，需要 Web 端认证，这个时候你的路由器本质就是在“劫持”你到认证页面，当你认证成功，路由器防火墙策略把你的流量放行，并停止劫持。

除了网络调试与安全测试之外，其实还有更多别的用途，比如劫持网络，把某个系统的升级软件、脚本替换成你自己的恶意软件下载地址来 Getshell。

毕竟劫持不仅可以劫持用户端，服务端返回数据也可以被劫持：操纵用户手脚还是蒙住用户的眼睛都可以。

核心原理：不安全的CA导致信任链崩坏

我们上一节提到的所有的劫持应用等，其实都有一个基础前提，就是“不安全的 CA 被信任了”；

如果没有进行这一部分基础知识的积累，很多人不会把 CA 的安全性、重要性想的太高，毕竟你直接信任了一个 CA，也并不会导致说马上被 getshell 掉，对吗？甚至很多安全从业人员对 CA 的态度也很暧昧；

之前在长亭科技工作的时候，进行很多校招或者社招面试，我都会比较喜欢问面试者 HTTPS 劫持的核心原理，答上来的人很少：大多数人要不就是上来给我背书，要不就是只知道怎么下载 `Burp`

`suite` 的证书，怎么添加信任。但是问具体中间人细节实现的时候，大多数人都傻眼了。

简单来说就是：不安全的 CA 可以给任何网站进行签名，TLS 服务端解密需要服务端私钥和服务段证书，然而这个不安全的 CA 可以提供用户暂时信任的服务端私钥和证书，这就好比你信任了一个信用极差的人进入你的家，这个信用极差的人可以在你的家里乱翻乱拿无恶不作。

详细的内容限于篇幅我们不多说了，毕竟这也不是本文重点。

不能进行 HTTPS 劫持的场景

当然，为了防止你的 HTTPS 服务被劫持，你可以使用以下的操作进行加固，来防止劫持。

服务端证书锁定：SSL/TLS pinning

这种场景一般用于移动端 APP 的防劫持，Twitter，Google 系 APP 很多都在使用这项技术，核心原理是把服务端证书或其他凭证内置在客户端，在客户端访问服务端的时候，会验证服务端证书有没有被替换。

> https://medium.com/@zhangqichuan/explain-ssl-pinning-with-simple-codes-eaee95b70507

由于 SSL/TLS pining 强制验证服务端返回的证书中是否包含目标证书，如果我们可以弄到原本服务端证书，给他返回去是不是也可以绕过？理论上当然可以，但是这样，没有私钥的你也没有办法解密 TLS 流量了。

银弹？证书锁定带来的尴尬问题

如果仅仅只是塞一个证书锁定在客户端，那么客户端过期了的话，怎么办呢？那就只能强制用户更新客户端，这种 breaking 一般是不被允许的。

如果塞一个公钥到客户端，如果证书、公钥被吊销，或者私钥被泄露的话，仍然是需要用户强制更新客户端。

但是我相信任何处于业务快速膨胀期或者研发水平跟不上的 APP 都不会这么操作的，一般有很硬后台，很高安全性要求，有成熟研发体系和 PKI 体系的公司可能才会有这种需求或能力做这件事情。

最高安全等级：x509 双向认证

诸如 WPA2 企业级认证 EAP-TLS 的认证方式，很多系统都将 TLS 的双向认证作为最高标准、最高安全等级的认证方式。

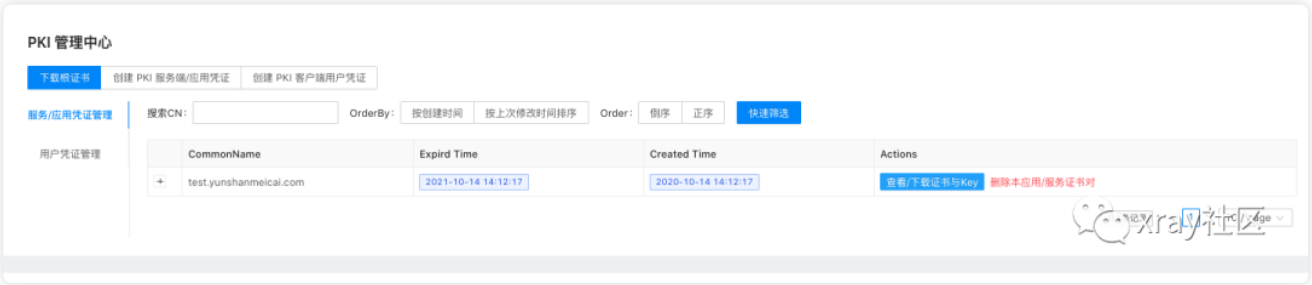
同样的，对于 HTTP 服务，同样也可以签署开启了 X509 双向认证的 TLS 连接。

> https://zh.wikipedia.org/wiki/%E5%85%AC%E9%96%8B%E9%87%91%E9%91%B0%E5%9F%BA%E7%A4%8E%E5%BB%BA%E8%A8%AD

日常生活中，很多商业产品都是通过 PKI 体系的证书来控制产品的授权。

高要求：PKI 体系构建和研发过程

当然，如果公司已经具备如下的系统 （PKI 的简单实现 - 截取 Palm SIEM 的部分功能）。



并且研发支持自动化申请签发证书，用户更新证书，APP 更新证书等高成本高规范的流程，PKI 确实可以带了很高的安全保障。

客户端缺陷：SNI 不明确

虽然大多数情况下，HTTPS 可以理解为 TLS + HTTP，但是仍然有一些细节需要注意。

SNI 是什么呢？简单来说，一个 HTTPS 服务器可能存在多个域名（同学们可以回忆一下 Apache 虚拟主机的配置这项功能），当进行 TLS 握手的时候，不同的网站对应的 TLS 服务端配置不一样，如果不携带想要访问哪个域名，那么服务器或中间件确实不知道应该转发到具体的哪个虚拟站点上，直接导致了 TLS 的混乱。

为了解决这个问题，ClientHello 中的 SNI 在 TLSv1.2 开始支持，我们可以设置 `Server Name` 这个字段，设置为网站域名。

同样的，MITM 也依赖 ServerName，因为 ServerName 可能直接决定了中间人要连接的目标是啥，如果没有 SNI，中间人则没有办法寻找到连接目标，同样的也没有办法对网站进行临时签名了。

疑问：HTTP 的 Host 不是有域名信息吗？为什么不能用？

这个问题答案很简单 HTTPS 中，TLS 是 HTTP 的前置条件，域名拿不到，没有办法进行 TLS 签名，所以只能依赖 ClientHello 中的 ServerName 了。

第二通道：HTTP 之上的加密

做过微信公众号后台或者尝试劫持微信的朋友都可能很容易理解这个问题，HTTP 传输的内容也是经过加密的，可以很轻松让中间人劫持变得没有意义。

小总结

当然防止 HTTPS 劫持的方案不只有这么多，其实会有很多种细节处理方案都可以导致 HTTPS 中间人劫持失效。

讲解中间人劫持的攻防其实是为了大家更深入了解学习，而并不只是总停留在表面操作。

常见方法

`mitmproxy` 是一个好项目，能够很好地帮助大家理解 MITM 原理

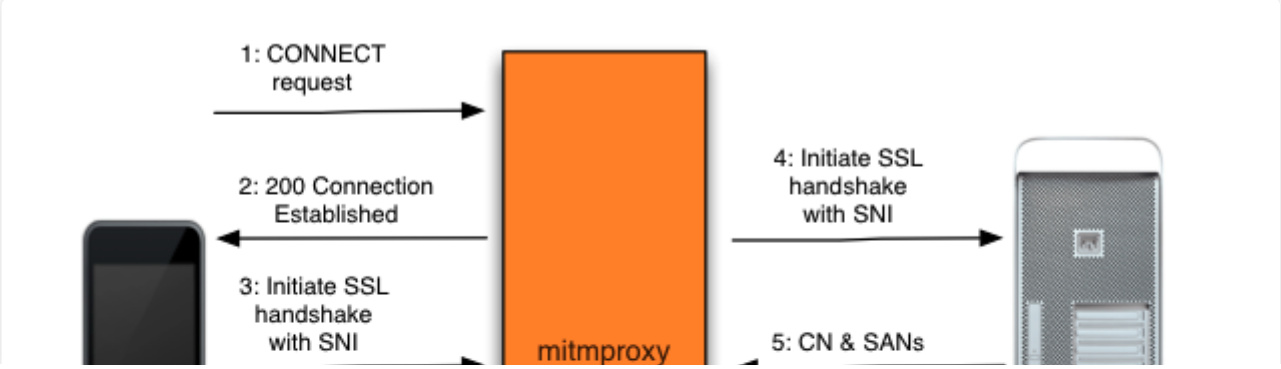
> https://docs.mitmproxy.org/stable/concepts-howmitmproxyworks/

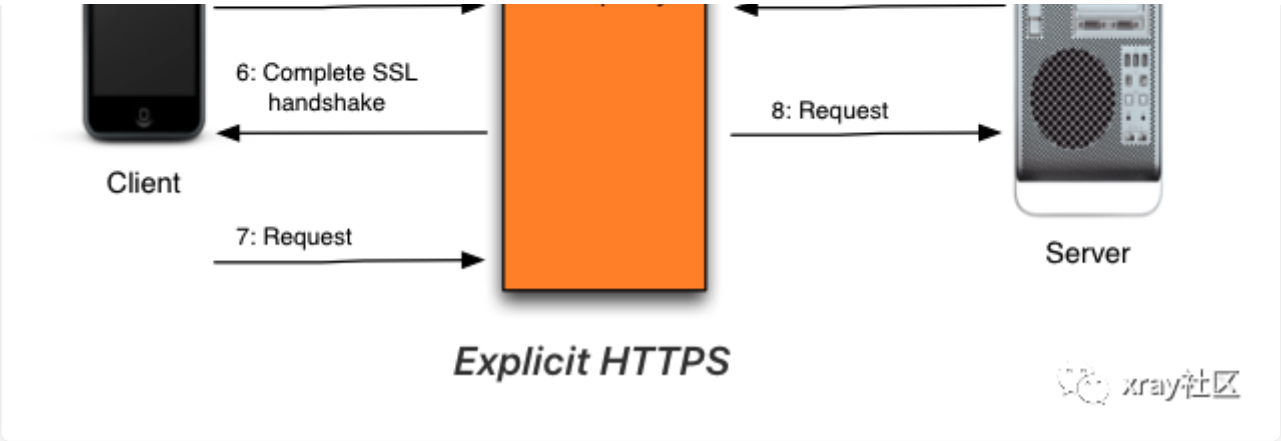
但是缺陷也比较明显，`mitmproxy` 基于 Python，在高负载的情况下并不是特别理想，再加上源码保护等需求，一般 mitmproxy 并不是太适合在工程中或者商业产品中使用；但是大家用他来写个 Demo 或者 Poc 或者论文，基本是没啥问题的。

设置代理以劫持 HTTPS

使用 Charles，Fiddler，Burpsuite，xray 等的同学，可能对这再熟悉不过了，启动代理服务器，浏览器配置一个代理就可以抓包进行调试、漏洞测试了。

引用 `mitmproxy 原理文档`（入口：https://docs.mitmproxy.org/stable/concepts-howmitmproxyworks/）中的图来简单说明我们最常见的 HTTPS 劫持的方案实现：





1.先通过一个简单的 HTTP Connect 请求来连接代理服务器；

2.代理服务器返回 200，表明 Connect 管道建立完毕；

3.Client 开始进行 TLS 连接，中间人通过 SNI 来获知需要连接的目标是谁；

4.中间人连接真正的服务器；

5.中间人开始根据 SNI 和 CA 自动签发假的服务端证书，并返回给用户，并进行握手；

6.握手成功，客户端开始发送 HTTP 请求；

7.中间人开始做用户到真正服务器的流量互相转发（可劫持）。

遵循协议参考，基础协议

> HTTP Proxy.
>
> https://www.ietf.org/rfc/rfc2068.txt
>
> Subject Key Identifier support for end entity certificate.
>
> https://www.ietf.org/rfc/rfc3280.txt (section 4.2.1.2)

核心实现代码与关键实现

整体流程实现

`golang martian` 的核心代码入口：
https://github.com/google/martian/blob/5f8cb6346337bcd48a84656159acb6e724c16fa9/proxy.go#L288

```go
func (p *Proxy) handleConnectRequest(ctx *Context, req *http.Request, session *Session, brw *bufio.ReadWriter, conn n
        if err := p.reqmod.ModifyRequest(req); err != nil {
                log.Errorf("martian: error modifying CONNECT request: %v", err)
                proxyutil.Warning(req.Header, err)
        }
        if session.Hijacked() {
                log.Debugf("martian: connection hijacked by request modifier")
                return nil
        }

        if p.mitm != nil {
                log.Debugf("martian: attempting MITM for connection: %s / %s", req.Host, req.URL.String())

                res := proxyutil.NewResponse(200, nil, req)

                if err := p.resmod.ModifyResponse(res); err != nil {
                        log.Errorf("martian: error modifying CONNECT response: %v", err)
                        proxyutil.Warning(res.Header, err)
                }
                if session.Hijacked() {
                        log.Infof("martian: connection hijacked by response modifier")
                        return nil
                }

                if err := res.Write(brw); err != nil {
                        log.Errorf("martian: got error while writing response back to client: %v", err)
```

返回 Connect 方法的 200 响应，表明构建成功


```go
                b := make([]byte, 1)
                if _, err := brw.Read(b); err != nil {
                        log.Errorf("martian: error peeking message through CONNECT tunnel to determine type: %v", err)
                }

                // Drain all of the rest of the buffered data.
                buf := make([]byte, brw.Reader.Buffered())
                brw.Read(buf)

                // 22 is the TLS handshake.
                // https://tools.ietf.org/html/rfc5246#section-6.2.1
                if b[0] == 22 {
                        // Prepend the previously read data to be read again by
                        // http.ReadRequest.
                        tlsconn := tls.Server(&peekedConn{conn, io.MultiReader(bytes.NewReader(b), bytes.NewReader(buf), conn)}, p.mitm.TLSForHost(req.Host))

                        if err := tlsconn.Handshake(); err != nil {
                                p.mitm.HandshakeErrorCallback(req, err)
                                return err
                        }

                        var nconn net.Conn
                        nconn = tlsconn
                        // If the original connection is a traffic shaped connection, wrap the tls
                        // connection inside a traffic shaped connection too.
                        if ptsconn, ok := conn.(*trafficshape.Conn); ok {
                                nconn = ptsconn.Listener.GetTrafficShapedConn(tlsconn)
                        }
                        brw.Writer.Reset(nconn)
                        brw.Reader.Reset(nconn)
                        return p.handle(ctx, nconn, brw)
                }
```

读取一个字节是为了区分 HTTPS 连接

升级 TCP 为 TLS

大家最关心的自动签发证书功能在哪呢?

在 Golang 中，为服务器生成 tls.Config 就可以升级 TLS

`martian` 生成服务端 TLS 的代码入口：
https://github.com/google/martian/blob/b99070ab9f10410a3f8d25fae01dd611a6cc79ff/mitm/mitm.go#L183

```go
// TLS returns a *tls.Config that will generate certificates on-the-fly using
// the SNI extension in the TLS ClientHello.
func (c *Config) TLS() *tls.Config {
        return &tls.Config{
                InsecureSkipVerify: c.skipVerify,
                GetCertificate: func(clientHello *tls.ClientHelloInfo) (*tls.Certificate, error) {
                        if clientHello.ServerName == "" {
                                return nil, errors.New("mitm: SNI not provided, failed to build certificate")
                        }

                        return c.cert(clientHello.ServerName)
                },
                NextProtos: []string{"http/1.1"},
        }
}
```

SNI

`martian` 自动签发证书代码入口：
https://github.com/google/martian/blob/b99070ab9f10410a3f8d25fae01dd611a6cc79ff/mitm/mitm.go#L214

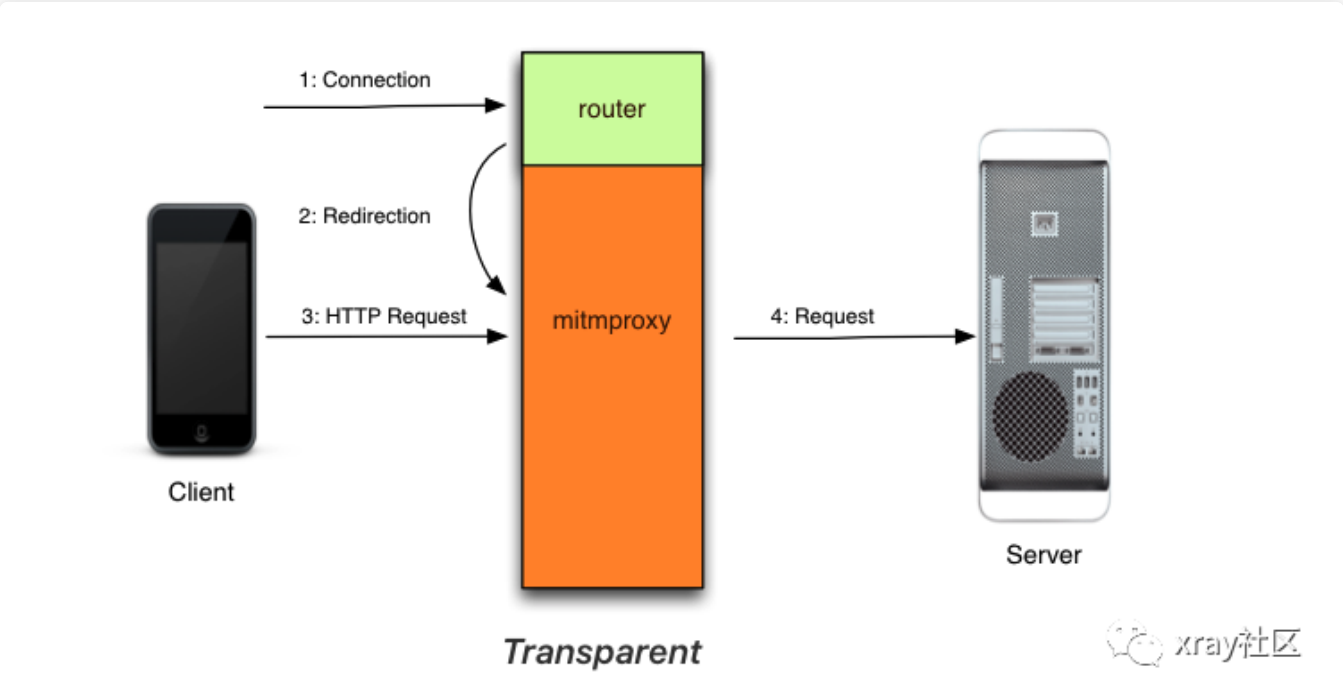
总结

针对这种最常见应用最广的劫持方案，我们到此介绍就结束了，如果这部分代码大家有所熟悉，那么 martian 这个库用起来会更好用，当然这件事情本来就不是特别困难；同样的，你可以根据 martian 开发一下，甚至写一个你自己的 xray 都是可以的。

透明代理劫持 HTTPS

其实我写这篇文章，主要是打算介绍这种劫持方案，何谓透明劫持？区别于依赖 HTTP Proxy 协议/功能的代理劫持，自然就是不需要设置代理，由路由器（当然也不止路由器），把流量设置到目标服务器上，目标服务器来进行 HTTPS 劫持。

优势就是不需要客户端配置，当然劣势也明显，需要路由设置（传输层劫持）。



1.客户端连接服务器；

2.路由器重定向客户端连接到中间人；

3.中间人通过 SNI 获知需要连接哪个具体的目标网站；

4.替换证书... 成功劫持 （同上，不做过多叙述了）；

很简单，我们可以在路由器上通过 iptables 来重定向客户端连接。

如何实现?

其实关键点和上述放的代码基本类似，为了防止滥用我可能并不会直接放出可用代码，只在关键点做伪代码或代码节选，供大家参考和突破难点。

当前实现的版本比 mitmproxy 性能和可扩展性高很多，由于 Go 本身的性能相对还行，体验还是相对良好。

支持多协议 HTTPS/HTTP

```go
func (m *MITMServer) handleHTTPS(ctx context.Context, conn net.Conn, origin string) error {
    pc := NewPeekableNetConn(conn)
    raw, err := pc.Peek( i: 1)
    if err != nil {
        return utils.Errorf( origin: "peek [%s] failed: %s", conn.RemoteAddr(), err)
    }

    log.Infof( format: "peek one char[%#v] for %v", raw, conn.RemoteAddr().String())

    var isHttps = utils.NewAtomicBool()
    var httpConn net.Conn
    switch raw[0] {
    case 0x16: // https
        log.Infof( format: "serving https for: %s", conn.RemoteAddr().String())
        tconn := tls.Server(pc, m.mitmConfig.TLS())
        err := tconn.Handshake()
        if err != nil {
            return utils.Errorf( origin: "tls handshake failed: %s", err)
        }
        log.Infof( format: "conn: %s handshake finished", conn.RemoteAddr().String())
        httpConn = tconn
        isHttps.Set()
    default: // http
        log.Infof( format: "start to serve http for %s", conn.RemoteAddr().String())
        httpConn = pc
        isHttps.UnSet()
    }

    //log.Infof("parse req http finished: %v", spew.Sdump(req))
    if httpConn == nil {
        return nil
    }
```


关键点

```go
var remoteConn net.Conn
if !isHttps.IsSet() {
    log.Infof( format: "tcp connect to %s", target)
```

```
        remoteConn, err = dialer.Dial( network: "tcp", target)
        if err != nil {
            return utils.Errorf( origin: "remote tcp://%v failed to dial: %s", utils.HostPort(host, port), err)
        }
    } else {
        log.Infof( format: "tcp+tls connect to %s", target)
        remoteConn, err = tls.DialWithDialer(dialer, network: "tcp", target, &tls.Config{
            InsecureSkipVerify: true,
            ServerName:         originHost,
        })
        if err != nil {
            return utils.Errorf( origin: "remote tcp+tls://%v failed: %s", target, err)
        }
    }
    defer remoteConn.Close()

    // 以下是转发模式的，不做劫持
    if m.transparentHijackMode == nil || !m.transparentHijackMode.IsSet() {...} else {
        // 接下来是如何进行网络交互?
        // 透明模式，劫持开启之后回调才会生效

        // 劫持第一个 request
        var reqBytes = readerBuffer.Bytes()
        if m.transparentHijackRequest != nil {
            reqBytes = m.transparentHijackRequest(isHttps.IsSet(), readerBuffer.Bytes())
        }

        _, err = remoteConn.Write(reqBytes)
        if err != nil {
            return utils.Errorf( origin: "write first http.Request raw []byte failed: %s", err)
        }

        var req = req
        var rspRaw bytes.Buffer
        rsp, err := http.ReadResponse(bufio.NewReader(io.TeeReader(remoteConn, &rspRaw)), req)
        if err != nil {
            return utils.Errorf( origin: "read response for req[%v]->%v failed: %s", req.URL.String(), remoteConn
        }
        if rsp.Body != nil {
            rspBody, _ := ioutil.ReadAll(rsp.Body)
            if len(rspBody) > 0 {
                log.Infof( format: "rsp body found length: %v", len(rspBody))
            }
        }
        log.Info( v...: "first req and rsp recv finished!")
```

非 TLS 直接就可以劫持，很容易

针对 TLS 连接，尊重协议需要设置 SNI （ServerName）

使用 TeeReader 读出原始报文，不要修改

我为什么不直接抄 Martian 的代码?

如果大家有使用过 Martian 这个库的话，可能会对这些代码很熟悉

```
    group.AddRequestModifier(header.NewHopByHopModifier())

    // fix err frame
    group.AddRequestModifier(header.NewBadFramingModifier())

    // mirror request
    group.AddRequestModifier(NewRequestModifier(func(req *http.Request) error {...}))

    if m.allowForwarded {
        group.AddRequestModifier(header.NewForwardedModifier())
    }
```

```
    if m.via != "" {
        vm := header.NewViaModifier(m.via)
        group.AddRequestModifier(vm)
        group.AddResponseModifier(vm)
    }

    // 证书-下载证书
```


我们在 martian 中需要通过 AddRequestModifier 对请求进行 Hook

实际在使用的过程中，写入网络的对象，多是经过 Golang http 标准库进行处理的，在某些时候，可能并不能保证和原本传输的内容一摸一样。需要 Fix/Patch 之类的操作。

- 镜像场景

在关键的地方，我们使用 TeeReader 读出镜像流量直接对可用的 Bytes 进行转发，能最大程度保证不破坏内容同时也保证性能，由于镜像数据其实不关劫持流程什么关系，我们可以随意启动别的 Goroutine 来执行。

- 劫持场景

如果劫持方式被设置了，我们本身的需求在于修改数据包的内容，那么我个人倾向于直接交由一个完整的 Bytes 给上游，返回一个修改后的 Bytes，而不做过多处理，虽然这一个步骤是同步的，上游可以封装他自己的任何对象，最后序列化的结果也是上游负责，最大程度保证尊重开发者意愿。

透明劫持的其他玩法（CA被信任）

不管是代理劫持 HTTPS 也好还是透明劫持 HTTPS，至少我们可以用它做很多事情，当然这些事情不一定是涉及到安全领域。

既然是中间人，我们就既可以扮演客户端也可以扮演服务器。

高效可用路由器解密 HTTPS 的方案

前段时间我们做了一件很有趣的事，在路由器上配置了一个 hostapd 然后开启了 https 透明劫持，通过 iptables 来控制想要劫持的具体网站（当然也可以劫持全部 https、http），确实爽了一把，后来我想，大概抓出来的流量我就存下来，抽空挨着测试，也并不需要 xray 这种实时进行安全测试。

当然除了进行安全测试之外，想要分析流量内容，分析敏感数据是否外传，审计流量也是完全可以做到的。我相信很多人或者公司，或多或少对自己或者自己员工都有一点好奇心对吧?

当然，想要实现这些东西不仅仅是劫持就完了，就算我把劫持框架开源了，仍然需要大量的基础设施需要完善。

IoT 设备连接 Wifi 劫持导致的 getshell

很早些年间，在进汽车安全测试的时候，车机也算是一种 IoT 设备吧，连接上 Wifi 之后，选择车机的自动升级功能或者下载 APP 功能的时候，我们可以轻松劫持掉下载链接，把下载链接替换成 msf 生成的安卓马儿，直接 Getshell。

当然，我们可以如法炮制很多种 Getshell 的方案，说着转眼就盯上了智能咖啡机.....

其实这种情形非常非常非常非常常见，大部分 IoT 厂商对中间人劫持是敏感的，但是他们选择的大多数方案是使用自签名证书，并且不让你能设置代理，如果自己有 PKI 体系并且开启了 X509 双向认证，那可能风险会大大降低。但是往往他们并不会这么做，虽然不让客户端设置代理，但是路由器可以设置代理啊，这就导致了透明代理成了非常有效的劫持手段。

而且由于上述原因最尴尬的情况很可能发生：大多数时候 IoT 设备都不会验证证书是否合理

所以 IoT 设备其实蛮尴尬的。

配合 DNS 劫持或者 ARP 劫持，内网直接透明劫HTTPS

这个当然也是老树开新花，很久很久以前，内网劫持 HTTPS 采用的方案是 ARP 劫持 + DNS 劫持 + HTTPS 降级。但是都 2020 年了，我们自然有了各种对抗，HSTS 和服务器的规范化配置，导致 HTTPS 降级越来越少，ssltrip 慢慢退出了历史舞台，但是好在我们也似乎不需要再使用 ssltrip，透明劫持可以很容易做到。

当然使用这种手段一般是我们无法控制路由器了。

小总结

其实玩法太多了，受限于文章篇幅和主题，我也不打算过多讲细节了

1.免代理低成本的被动扫描；

2.某些 APP 监控与舆情审计。

.....

安全启示

不要信任不认识的 CA，因为毕竟很多人对“不安全的 CA”认识也一直停留在嘴上和书本上。

对于个人

很多人有一些疑问，CA 信任这么重要，我信任了公司的根证书，那是不是意味着，我的流量只要想劫持，是一定可以劫持的?

这个答案是肯定的，认真看完本文之后你会发现这其实很危险，至少技术上是完全可行的，只是我的代码并没有放出来而已，同样大家都可以自己写一个这样的服务器出来。

对于开发者、公司、厂商

1.有能力做 PKI 体系一定要去实现；

2.防劫持不仅防的是数据泄露问题，也会一定程度上防止被 getshell 或者 rce；

3.安全非小事，开发运维需谨慎；

4.技术无罪，请用在正途。

文章已于2020/11/12修改