INTERNET ARCHIVE WaybackMachine
http://www.ct2wj.com/2016/02/28/shadowsocks-android-source-code-analysis/ Go
43 captures
6 Mar 2016 - 20 Aug 2018
DEC JAN FEB
2016 10 2018 2019
About this capture

shadowsocks-android源码解析

Posted on 2016-02-28 | In network |

0x00 简介

shadowsocks-android(ss-android)是一款android上的科学上网软件，特点是无需root权限就能实现全局代理。作为一个shadowsocks重度使用者，本来想学习一下，但发现网上相关文章和资料还比较少，因此有了本篇文章，旨在分享自己对ss-android源码的一些理解。本文分析的源码基于ss-android的2.9.11版本。

0x01 ss-android进程概述

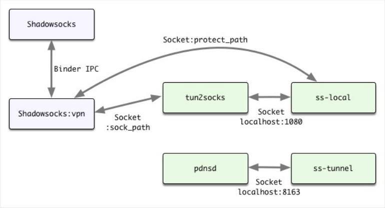
本文先从整体上阐述ss-android进程之间的关系，然后从每个进程角度出发，结合源码分析ss-android科学上网的实现原理。ss-android启动时默认工作在vpn模式，创建2个java进程**shadowsocks**，**shadowsocks:vpn**以及4个native进程**ss-local**、**pdnsd**、**ss-tunel**、**tun2socks**：

```
u0_a124   14935 1857  1058332 56560 SyS_epoll_ 00000000 S com.github.shadowsocks:vpn
u0_a124   16336 1857  1235568 88380 SyS_epoll_ 00000000 S com.github.shadowsocks
```

```
u0_a124   20419 14935 3376   1136  SyS_epoll_ 00000000 S /data/data/com.github.shadowsocks/ss-local
u0_a124   20421 14935 7688   804   do_sigtime 00000000 S /data/data/com.github.shadowsocks/pdnsd
u0_a124   20425 14935 3112   940   SyS_epoll_ 00000000 S /data/data/com.github.shadowsocks/ss-tunnel
u0_a124   20431 14935 3812   1420  SyS_epoll_ 00000000 S /data/data/com.github.shadowsocks/tun2socks
```

ss-android进程启动流程以及进程间通信关系图：

```
shadowsocks => shadowsocks:vpn => ss-local => pdnsd => ss-tunnel => tun2socks
```



0x02 shadowsocks与shadowsocks:vpn

shadowsocks进程主要负责UI展示，用户可以设置shadowsocks服务器地址端口，选择加密算法，工作模式（vpn或nat），是否采用udp转发等。shadowsocks在`onCreate`时会去`attachService`，默认bind的是`ShadowsocksVpnService`。

ServiceBoundContext.scala

```
def attachService(callback: IShadowsocksServiceCallback.Stub = null) {
```

```
  this.callback = callback
  if (bgService == null) {
    val s =
      if (ShadowsocksApplication.isVpnEnabled) classOf[ShadowsocksVpnService] else classOf[ShadowsocksNa

    val intent = new Intent(this, s)
    intent.setAction(Action.SERVICE)

    connection = new ShadowsocksServiceConnection()
    bindService(intent, connection, Context.BIND_AUTO_CREATE)
    startService(intent)
  }
}
```

`ShadowsocksVpnService` 工作在shadowsocks:vpn进程，shadowsocks:vpn是后台进程，除了启动native进程，流量统计等功能也是shadowsocks:vpn和shadowsocks协作完成的。shadowsocks通过 `bindService` 和shadowsocks:vpn建立Binder IPC，注册跨进程callback将流量变化展示到shadowsocks。

当用户点击浮窗开启vpn时，shadowsocks执行 `prepareStartService`：

Shadowsocks.scala

```
def prepareStartService() {
  ThrowableFuture {
    if (ShadowsocksApplication.isVpnEnabled) {
      val intent = VpnService.prepare(this)
      if (intent != null) {
        startActivityForResult(intent, Shadowsocks.REQUEST_CONNECT)
      } else {
        handler.post(() => onActivityResult(Shadowsocks.REQUEST_CONNECT, Activity.RESULT_OK, null))
      }
    } else {
      serviceStart()
    }
  }
}
```

`VpnService.prepare` 去检查系统当前是否存在一个vpn连接。如果不存在，会返回一个让用户确认开启vpn界面的intent，这时 `onActivityResult` 传入 `Activity.RESULT_OK`。在返回 `Activity.RESULT_OK` 的情况下，shadowsocks会执行 `bgService.start`，准备让shadowsocks:vpn启动native进程。

Shadowsocks.scala

```
override def onActivityResult(requestCode: Int, resultCode: Int, data: Intent) = resultCode match {
  case Activity.RESULT_OK =>
    serviceStart()
  case _ =>
    cancelStart()
}
```

```
 def serviceStart() {
   bgService.start(ConfigUtils.load(ShadowsocksApplication.settings))
...
 }
```

`bgServive.start` 会在shadowsocks:vpn中调起 `startRunner`， `ShadowsocksVpnThread.start` 会另起线程创建socket（socket文件是/data/data/com.github.shadowsocks/protect_path），以unix domain socket的方式和native进程ss-local进行IPC，目的是把远程shadowsocks服务端地址"保护"起来，原因后面会详细分析。

接着做dns解析， `Utils.resolve` 得到远程shadowsocks代理服务器的ip地址。

ShadowsocksVpnService.scala

```
 override def startRunner(config: Config) {
   super.startRunner(config)
   vpnThread = new ShadowsocksVpnThread(this)
   vpnThread.start()
...
   ThrowableFuture {
...
     if (config != null) {
...
       // Resolve the server address
       var resolved: Boolean = false
       if (!Utils.isNumeric(config.proxy)) {
         Utils.resolve(config.proxy, enableIPv6 = true) match {
           case Some(addr) =>
             config.proxy = addr
             resolved = true
           case None => resolved = false
         }
       } else {
```

```
         resolved = true
       }

       if (resolved && handleConnection) {
         changeState(State.CONNECTED)
       } else { ... }
     }
   }
 }
```

服务器地址解析成功后，最终调用 `handleConnection` 去启动ss-local、tun2socks、pdnsd和ss-tunnel。

```
/** Called when the activity is first created. */
def handleConnection: Boolean = {
  startShadowsocksDaemon()
  if (!config.isUdpDns) {
    startDnsDaemon()
    startDnsTunnel()
  }

  val fd = startVpn()

  if (fd != -1) {
    var tries = 1
    while (tries < 5) {
      Thread.sleep(1000 * tries)
      if (System.sendfd(fd) != -1) {
        return true
      }
      tries += 1
    }
  }
  false
}
```

0x03 ss-local

ss-local是本地socks代理服务进程，可以看作一个通道，通道一端连着墙外的shawdowsocks服务器，另一端接着手机的本地客户端进程。

```
本地客户端进程 <=> (本地服务端:1080) ss-local (本地socks客户端) <==| GFW |==> 远程socks服务端
```

那么ss-local是如何让客户端进程把数据发送到远程socks服务端呢?

ss-local文件执行后，`main` 函数会在 `create_and_bind` 创建和返回本地服务端socket fd，监听本地进程在localhost:1080上的连接请求。`ev_io_init` 用来在fd上注册监听事件以及对应的回调函数，当listenfd上 `EV_READ` 发生时，`accept_cb` 被回调。内核在某个fd上准备好数据后，这个fd就是**可读**的，`EV_READ` 就会在新的轮询中被触发；同样，如果send buffer还有空间，就是**可写**的，`EV_WRITE` 会被触发。

local.c

```
int main(int argc, char **argv)
{       ...
    // Setup socket
    int listenfd;
    listenfd = create_and_bind(local_addr, local_port);
    if (listenfd < 0) {
        FATAL("bind() error");
    }
    if (listen(listenfd, SOMAXCONN) == -1) {
        FATAL("listen() error");
    }
    setnonblocking(listenfd);

    listen_ctx.fd = listenfd;

    ev_io_init(&listen_ctx.io, accept_cb, listenfd, EV_READ);
    ev_io_start(loop, &listen_ctx.io);
    ...
}
```

`create_and_bind`

```
for (rp = result; rp != NULL; rp = rp->ai_next) {
    listen_sock = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
```

listenfd收到数据时，说明有本地客户端进程连接进来了，`accept_cb` 被回调，为返回的serverfd注册I/O事件回调函数 `server_recv_cb` 和 `server_send_cb` 。当本地客户端进程有数据到达时，`server_recv_cb` 被回调，`recv` 接收socket的数据到 `buf->array` （此时的buf是 `server->buf` ）。

```
void accept_cb(EV_P_ ev_io *w, int revents)
{
    listen_ctx_t *listener = (listen_ctx_t *)w;
    int serverfd           = accept(listener->fd, NULL, NULL);
    ...
    server_t *server = new_server(serverfd, listener->method);
    server->listener = listener;
}
```

```
static server_t *new_server(int fd, int method)
{        ...
    server->fd                  = fd;
         ...
    ev_io_init(&server->recv_ctx->io, server_recv_cb, fd, EV_READ);
    ev_io_init(&server->send_ctx->io, server_send_cb, fd, EV_WRITE);

    return server;
}
```

```
static void server_recv_cb(EV_P_ ev_io *w, int revents)
{       ...
    if (remote == NULL) {
        buf = server->buf;
    } else {
        buf = remote->buf;
    }
    ...
    r = recv(server->fd, buf->array, BUF_SIZE, 0);
    ...
```

`server_recv_cb` 函数比较长，当 `server->stage` 到达阶段 `5` 时， `create_remote` 会根据前面解析好的shadowsocks服务器ip和端口来创建socket，得到remotefd。下一次循环ss-local作为本地socks客户端，会尝试与远程shadowsocks服务器建立连接。

`server_recv_cb`

```
...
    while (1) {
    ...
                server->stage = 5;
...
                if ((acl && (request->atyp == 1 || request->atyp == 4) && acl_match_ip(host))) {
                    struct sockaddr_storage storage;
                    memset(&storage, 0, sizeof(struct sockaddr_storage));
                    if (get_sockaddr(host, port, &storage, 0) != -1) {
                        remote         = create_remote(server->listener, (struct sockaddr *)&storage);
                        remote->direct = 1;
                    }
                } else {
                    remote = create_remote(server->listener, NULL);
                }
```

```
  if (server->stage == 5) {
  ...
      if (!remote->send_ctx->connected) {
...
          remote->buf->idx = 0;

          if (!fast_open || remote->direct) {
              // connecting, wait until connected
              connect(remote->fd, (struct sockaddr *)&(remote->addr), remote->addr_len);
```

这里的buf还是 `server->buf` ，并且 `buf->len > 0` 。远程socks服务端建立连接成功后，把 `server->buf->array` 数据直接 `memcpy` 到 `remote->buf->array` 。为一下步数据加密做准备。

```
    if (remote == NULL) {
        buf = server->buf;
    } else {
        buf = remote->buf;
    }
...
                if (!remote->direct) {
                    if (auth) {
                        abuf->array[0] |= ONETIMEAUTH_FLAG;
                        ss_onetimeauth(abuf, server->e_ctx->evp.iv, BUF_SIZE);
                    }

                    brealloc(remote->buf, buf->len + abuf->len, BUF_SIZE);
                    memcpy(remote->buf->array, abuf->array, abuf->len);
                    remote->buf->len = buf->len + abuf->len;

                    if (buf->len > 0) {
                        if (auth) {
                            ss_gen_hash(buf, &remote->counter, server->e_ctx, BUF_SIZE);
                        }
```

```
                        memcpy(remote->buf->array + abuf->len, buf->array, buf->len);
                    }
                } else {
                    if (buf->len > 0) {
                        memcpy(remote->buf->array, buf->array, buf->len);
                        remote->buf->len = buf->len;
                    }
                }
```

加密初始化时，ss-local-vpn.conf中password作为 `salt` 参与加密过程，method在 `supported_ciphers` 中找到method对应的下标m，决定对称加密算法(推荐使用 `aes-256-cfb`)。确定算法后，远程socks服务端采用相同的算法进行解密。这样ss-local和远程shadowsocks服务进程就可以理解双方传递的数据了。注意，**这里并不需要和服务端进行密钥交换，加密的设计初衷只是为了混淆传数据，增加GFW检查流量特征的计算量。**

```
int enc_init(const char *pass, const char *method)
{
    int m = TABLE;
    if (method != NULL) {
        for (m = TABLE; m < CIPHER_NUM; m++)
            if (strcmp(method, supported_ciphers[m]) == 0) {
                break;
            }
...
        enc_key_init(m, pass);
    }
    return m;
}
```

`bytes_to_key`

```
int bytes_to_key(const cipher_kt_t *cipher, const digest_type_t *md,
                 const uint8_t *pass, uint8_t *key, uint8_t *iv)
{ ...
#if defined(USE_CRYPTO_OPENSSL)
    return EVP_BytesToKey(cipher, md, NULL, pass, datal, 1, key, iv);
```

`enc_key_init`

```
void enc_key_init(int method, const char *pass)
{...
        cipher = (cipher_kt_t *)get_cipher_type(method)
...
    enc_key_len = bytes_to_key(cipher, md, (const uint8_t *)pass, enc_key, iv);
```

`EVP_BytesToKey`

```
int EVP_BytesToKey(const EVP_CIPHER *type, const EVP_MD *md,
             const unsigned char *salt, const unsigned char *data, int datal,
             int count, unsigned char *key, unsigned char *iv)
{ ...
                if (salt != NULL)
                        if (!EVP_DigestUpdate(&c,salt,PKCS5_SALT_LEN))
                                goto err;
```

接着，对数据加密。

```
    while (1) {
        // local socks5 server
        if (server->stage == 5) {
...

            if (!remote->direct && remote->send_ctx->connected && auth) {
                ss_gen_hash(remote->buf, &remote->counter, server->e_ctx, BUF_SIZE);
            }

            // insert shadowsocks header
            if (!remote->direct) {
#ifdef ANDROID
                tx += remote->buf->len;
#endif
                int err = ss_encrypt(remote->buf, server->e_ctx, BUF_SIZE);

...
            }
```

如果内核空间remotefd的send buffer还没满，`remote_send_cb` 被回调，若 `remote->buf->len` 长度大于0，说明有数据正在等待发送，ss-local会通过 `send` 直接发送给远程socks服务端。

```
static void remote_send_cb(EV_P_ ev_io *w, int revents)
{       ...
    remote_t *remote              = remote_send_ctx->remote;
    ...
```

```
    if (remote->buf->len == 0) {
        // close and free
        close_and_free_remote(EV_A_ remote);
        close_and_free_server(EV_A_ server);
        return;
    } else {
        // has data to send
        ssize_t s = send(remote->fd, remote->buf->array + remote->buf->idx,
                         remote->buf->len, 0);
            ...
}
```

分析到这儿，ss-local发送数据给远程shadowsocks服务端的基本流程清楚了，但还有个问题，**本地连上ss-local是哪个进程?**

0x04 tun2socks

由于ss-local连上了shadowsocks服务器，需要科学上网的本地进程都得想办法连上ss-local。要让进程都连上ss-local要做额外配置工作，最好是能有一个**全局代理**把流量都转给ss-local。

```
本地进程 <=> 全局代理 <=> ss-local
```

Android 4.0推出了 `VpnSerivce`，目的是让开发者在没有root权限的情况下也能实现全局代理。`VpnService` 会创建虚拟网卡设备：**tun设备**（对应的文件叫**tun设备文件**）。vpn进程shadowsocks:vpn拿到tunfd后就拥有了读写虚拟网卡ip包的能力，但是采用socks5协议的ss-local工作在L4，没法直接处理L3的ip包，所以**tun2socks**成了让ss-android成为全局socks代理的关键。tun2socks在用户态实现了一个轻量级的TCP/IP栈（lwip），来实现L3的包转成L4的数据流。

```
本地进程 => tun设备（虚拟网卡）=> tun2socks => ss-local => 物理网卡 => ...
```

有了tun2socks的L3转L4的能力，还剩下2个关键问题需要解决：

> 问题1：**怎么让tun2socks直接读取tun设备的ip包?**
> 问题2：**怎么让所有的包都经过tun设备?**

问题1的解决办法是让tun2socks跨进程拿到tunfd。tunfd是在shadowsocks:vpn进程 `startVpn` 时返回的。tun2socks是native进程，而shadowsocks:vpn是java进程，为了在不同的进程中共享tunfd，shadowsocks:vpn作为客户端进程和tun2socks通过unix domain socket方式进行IPC，对应的socket文件是/data/data/com.github.shadowsocks/sock_path。通过jni方法 `System.sendfd` 传递tunfd。

ShadowsocksVpnService.scala

```
  def handleConnection: Boolean = {
    startShadowsocksDaemon()
...
    val fd = startVpn()

    if (fd != -1) {
      var tries = 1
      while (tries < 5) {
        Thread.sleep(1000 * tries)
        if (System.sendfd(fd) != -1) {
          return true
        }
        tries += 1
      }
    }
    false
```

`System.sendfd` 对应native方法 `Java_com_github_shadowsocks_system_sendfd`。shadowsocks:vpn拿到tunfd后，利用 `ancil_send_fd` 传给tun2socks进程。这样tun2socks就具备直接读写虚拟网卡ip包的能力了。

System.cpp

```
jint Java_com_github_shadowsocks_system_sendfd(JNIEnv *env, jobject thiz, jint tun_fd) {
    int fd;
    struct sockaddr_un addr;

    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        return (jint)-1;
    }

    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, "/data/data/com.github.shadowsocks/sock_path", sizeof(addr.sun_path)-1);

    if (connect(fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        close(fd);
```

```
        return (jint)-1;
    }

    if (ancil_send_fd(fd, tun_fd)) {
        close(fd);
        return (jint)-1;
    }

    close(fd);
    return 0;
}
```

tun2socks能直接读写ip包还只是开始，问题2**怎么让所有的包都经过tun设备**，我们看一下tunfd的创建过程。

ShadowsocksVpnService.scala

```
  val VPN_MTU = 1500
  val PRIVATE_VLAN = "26.26.26.%s"

  def startVpn(): Int = {
    val builder = new Builder()
    builder
      .setSession(config.profileName)
      .setMtu(VPN_MTU)
      .addAddress(PRIVATE_VLAN.formatLocal(Locale.ENGLISH, "1"), 24)
      .addDnsServer("8.8.8.8")
...
    if (config.route == Route.ALL || config.route == Route.BYPASS_CHN) {
      builder.addRoute("0.0.0.0", 0)
    } else { ... }

    builder.addRoute("8.8.0.0", 16)
...
      conn = builder.establish()
...
    val fd = conn.getFd
```

首先，在用户允许Vpn的情况下，shadowsocks:vpn会尝试用 `builder.establish` 创建tun设备。 `conn.getFd` 返回的fd就是tun设备fd，前面我们知道，有了tunfd可以读写ip数据包了。

ps：vpn连接在整个系统中只能存在一个，新的vpn连接会去覆盖老的。tunfd被close了或者vpn进程挂了，网络就会自动恢复。

接着， `builder.addAddress` 添加 `26.26.26.1/24` 到tun设备，作为tun设备ip地址。 `addDnsServer` 把 `8.8.8.8` （google dns服务器地址）作为vpn连接的dns解析服务器地址。

```
root@android:/dev # ifconfig tun0
tun0: ip 26.26.26.1 mask 255.255.255.0 flags [up point-to-point running]
```

如果Route设置工作在 `ALL` 或 `BYPASS_CHN` 模式，路由 `0.0.0.0:0` 被添加到tun设备，不管包的目标ip是多少都保证经过tun设备，也就是说， `ALL` 或 `BYPASS_CHN` 模式下**所有本地流量**都会从tun设备出去。此外，默认 `8.8.0.0:16` 也添加到tun设备路由，保证至少到google dns服务器做dns解析的包经过tun设备。

ip路由信息：

```
root@android:/ # ip r
0.0.0.0/1 dev tun0  scope link
default via 10.0.3.2 dev eth1
default via 10.0.3.2 dev eth1  metric 205
8.8.0.0/16 dev tun0  scope link
10.0.3.0/24 dev eth1  proto kernel  scope link  src 10.0.3.15  metric 205
10.0.3.2 dev eth1  scope link
26.26.26.0/24 dev tun0  proto kernel  scope link  src 26.26.26.1
128.0.0.0/1 dev tun0  scope link
192.168.58.0/24 dev eth0  proto kernel  scope link  src 192.168.58.101
```

但是， `ALL` 或 `BYPASS_CHN` 模式下也会引发一个**隐藏的坑**：如果所有的流量都走tun设备（L3），tun2socks又从tunfd读出L3的包拆成L4的数据流转给ss-local，从ss-local出来准备发给远程shadowsocks服务器的流量（L4）又必然会到达tun设备(L3)，这样就直接**死循环**了。

```
tun设备(L3) => tun2socks(L3 -> L4) => ss-local(L4) => tun设备(L3) => ...
```

解决办法就是把远程shadowsocks服务器remotefd“保护”起来，让发到shadowsocks服务器流量不再走tun设备，而是直接通过物理网卡。在ss-local的 `server_recv_cb` 把remotefd传给 `protect_socket` ：

local.c

```
static void server_recv_cb(EV_P_ ev_io *w, int revents)
{       ...
```

```
                if (vpn) {
                    if (protect_socket(remote->fd) == -1) {
```

和前面跨进程传递tunfd类似，不过这里IPC双方换成了ss-local和shadowsocks:vpn，ss-local通过unix domain socket方式和shadowsocks:vpn进程进行本地IPC（0x02提到过），利用 `ancil_send_fd` 传递remotefd给shadowsocks:vpn进程。

android.c

```
int protect_socket(int fd)
{       ...
    if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        return -1;
    }

    const char path[] = "/data/data/com.github.shadowsocks/protect_path";
    ...
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, path, sizeof(addr.sun_path) - 1);

    if (connect(sock, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
        ...
    if (ancil_send_fd(sock, fd)) {
        ...
}
```

shadowsocks:vpn通过 `socket.getAncillaryFileDescriptors` 先拿到 `FileDescriptor`，再通过 `getInt.invoke` 反射读出remotefd，用 `vpnService.protect(fd)` 保护起来，从而避免死循环。

ShadowsocksVpnThread.scala

```
val PATH = "/data/data/com.github.shadowsocks/protect_path"

override def run() {
...
  while (isRunning) {
      val socket = serverSocket.accept()

      pool.execute(() => {
        try {
        ...
          val fds = socket.getAncillaryFileDescriptors

          if (fds.nonEmpty) {
            val fd = getInt.invoke(fds(0)).asInstanceOf[Int]
            val ret = vpnService.protect(fd)

            // Trick to close file decriptor
            System.jniclose(fd)
```

0x05 pdnsd与ss-tunnel

ss-android还剩下2个native进程**pdnsd**和**ss-tunnel**没有介绍，pdnsd是一个带永久缓存的dns代理服务器，而ss-tunnel主要用来实现本地端口转发，为什么前面VpnService设置了DNSServer还需要pdnsd+ss-tunnel呢，pdnsd+ss-tunnel可以不启动吗?

答案是：在设置udp转发的情况下可以。开启udp转发，做dns查询的udp包和其他udp包没有区别，都是通过ss-local转发给shadowsock服务器。我们重点看**没有设置udp转发的情况。**

shadowsocks:vpn启动tun2socks进程的时候，如果 `config.isUdpDns` 为false，会把 `--dnsgw` 也就是dns网关地址 `26.26.26.1:8153` 作为tun2socks的启动参数。

ShadowsocksVpnService.scala

```
def startVpn(): Int = {
...
  if (config.isUdpDns)
    cmd += " --enable-udprelay"
  else
    cmd += " --dnsgw %s:8153".formatLocal(Locale.ENGLISH, PRIVATE_VLAN.formatLocal(Locale.ENGLISH, "1"))
```

tun2socks的 `process_device_dns_packet` 一旦发现去做dns解析的udp包，就会把udp包的ip和端口强行改成dnsgw的ip和端口 `26.26.26.1:8153`，而 `8153` 端口是pdnsd进程监听的端口。

tun2socks.c

```
int process_device_dns_packet (uint8_t *data, int data_len)
```

```
{ ...
    // do nothing if we don't have dnsgw
    if (!options.dnsgw) {
        goto fail;
    }
        ...
            // to port 53 is considered a DNS packet
            to_dns = udp_header.dest_port == hton16(53);

            // from port 8153 is considered a DNS packet
            from_dns = udp_header.source_port == dnsgw.ipv4.port;
        ...
            // modify DNS packet
            if (to_dns) {
            ...
                // build IP header
                ipv4_header.destination_address = dnsgw.ipv4.ip;
                ipv4_header.source_address = netif_ipaddr.ipv4;

                // build UDP header
                udp_header.dest_port = dnsgw.ipv4.port;
            } else if (from_dns) {
            ...
                Connection * con = find_connection(udp_header.dest_port);
                if (con != NULL)
                {
                    // build IP header
                    ipv4_header.source_address = con->remote_addr.ipv4.ip;
                    ipv4_header.destination_address = con->local_addr.ipv4.ip;

                    // build UDP header
                    udp_header.source_port = con->remote_addr.ipv4.port;

                    remove_connection(con);

                }
```

JLLK
tomchen
GitHub

ss-tunnel会在本地起服务进程，然后同样转发dns查询的udp包，通过远程shadowsocks服务器给靠谱的dns服务器（例如`8.8.8.8:53`）。

JLLK
tomchen
GitHub

系统默认的dns查询就是走udp的，我们知道ss-android成为全局代理后，tun2socks已经把所有做dns查询的udp包都转给了dnsgw（pdnsd），如果pdnsd+ss-tunnel不启动，那么显然所有的dns查询都会失败。

既然这么麻烦，那直接直接走udp转发不就好了吗，**为什么还要用pdnsd+ss-tunnel呢?**

这是因为GFW会在境内骨干路由器上做入侵检测，一旦发现是查询google.com这种不和谐域名，就会伪造一个错误的ip地址给查询者。android系统dns查询默认走udp，但udp是不可靠的，会把最先返给自己的ip当作是目标ip，这个伪造的ip当然是无法正常访问的，这就是所谓的**dns污染**。

```
客户端进程 <=> 路由器dns <=> 运营商dns <=> | GFW | <=> 根dns
```

而pdnsd主要用来提供一个无污染的dns，它不仅可以转发dns查询给上游dns服务器，修改TTL，提供缓存功能，尤为重要的是，pdnsd可以修改dns查询方式为tcp，**从而避免dns污染。**

JLLK
tomchen
GitHub

0x06 结语

ss-android最核心的功能几乎全是由c代码实现，依赖开源库也比较多，涉及到unix，内核，网络和安全等知识，加上ss-android的java层代码基本上是用scala写的，分析ss-android上层代码需要一定的scala功底，了解函数式编程等，这些都造成了ss-android学习的门槛。

本篇文章希望对正在学习ss-android的朋友有所帮助，由于时间有限，分析有误的地方也恳请指出。

最后附上本人的github：https://github.com/JLLK。

版权声明

JLLK
tomchen
GitHub

CT2WJ by CHEN TAO is licensed under a Creative Commons BY-NC-ND 4.0 International License.

由陈涛创作并维护的CT2WJ博客采用创作共用保留署名-非商业-禁止演绎4.0国际许可证。

本文首发于CT2WJ博客（http://ct2wj.com），版权所有，侵权必究。

本文永久链接：http://ct2wj.com/2016/02/28/shadowsocks-android-source-code-analysis/

#Android #c #pdnsd #scala #shadowsocks #tun2socks #vpn

< 一种按模块拆分dex和适机加载的方法

JLLK
tomchen
GitHub