shadowsocks 源码分析：整体结构

2017-03-25 | research

对待科学上网，要拿出科学严谨的态度。在群众广泛使用的工具中，shadowsocks 历经多年屹立不倒，其中的原因值得深入探究。本系列文章从源码级别解读 shadowsocks，揭开科学上网工具的内幕。

文中提及的 shadowsocks 是 @clowwindy 使用 Python 编写的原始实现，版本号为 2.9.1，可从 GitHub 官方页面 下载。

shadowsocks 工程文件分布如下（略去测试文件）：

```
├── CHANGES
├── CONTRIBUTING.md
├── debian
│   ├── changelog
│   ├── compat
│   ├── config.json
│   ├── control
│   ├── copyright
│   ├── docs
│   ├── init.d
│   ├── install
│   ├── rules
│   ├── shadowsocks.default
│   ├── shadowsocks.manpages
│   ├── source
│   │   └── format
│   ├── sslocal.1
│   └── ssserver.1
├── Dockerfile
├── LICENSE
```

```
├── MANIFEST.in
├── README.md
├── README.rst
├── setup.py
├── shadowsocks
│   ├── asyncdns.py
│   ├── common.py
│   ├── crypto
│   │   ├── __init__.py
│   │   ├── openssl.py
│   │   ├── rc4_md5.py
│   │   ├── sodium.py
│   │   ├── table.py
│   │   └── util.py
│   ├── daemon.py
│   ├── encrypt.py
│   ├── eventloop.py
│   ├── __init__.py
│   ├── local.py
│   ├── lru_cache.py
│   ├── manager.py
│   ├── server.py
│   ├── shell.py
│   ├── tcprelay.py
│   └── udprelay.py
├── tests
└── utils
    ├── autoban.py
    ├── fail2ban
    │   └── shadowsocks.conf
    └── README.md
```

可见，其工程核心代码均位于 `shadowsocks` 目录下。其它文件和目录则提供了打包、测试等功能。

像 shadowsocks 这种代码量不足一万行的小型工程，通读源码仅需几个小时，阅览顺序也没有特别的讲究。但是，当遇到代码量十万行甚至百万行的大型工程时，阅读全部代码是一件不可能完成的任务，用正确的顺序浏览就变得尤为重要。一般来说，打开一个陌生的工程，最好先定位其 `main` 函数。抓住了程序的入口，就抓住了逻辑的根节点，此后再分别运用广度优先搜索、深度优先搜索、回溯搜索等多种手段，逐步描绘出工程的大体轮廓。

shadowsocks 在 `local.py` 和 `server.py` 两处分别定义了 `main` 函数。因为这两个 `main` 函数属于不同的模块，这是完全合法的。在 `shell.print_help` 中，它们被分别赋予了 `sslocal` 和 `ssserver` 这

两个名字，我们也可以称之为客户端和服务端。对比查看 `local.py` 和 `server.py`，服务端为了支持多组端口和多进程，写了不少额外的代码。如果把这两部分剥离掉，剩下的主干结构与客户端差异甚微：

```
# shadowsocks/local.py

    daemon.daemon_exec(config)
    ...
    dns_resolver = asyncdns.DNSResolver()
    tcp_server = tcprelay.TCPRelay(config, dns_resolver, ...)
    udp_server = udprelay.UDPRelay(config, dns_resolver, ...)
    loop = eventloop.EventLoop()
    dns_resolver.add_to_loop(loop)
    tcp_server.add_to_loop(loop)
    udp_server.add_to_loop(loop)
    ...
    loop.run()
```

在 `main` 函数中，shadowsocks 解析配置文件或命令行参数后，首先注册了系统守护进程，以便于在后台运行。之后，它创建了 DNS 解析器、TCP 中继器、UDP 中继器三个组件，并用事件循环将这三者统一起来。事件循环开始后，这些组件会在外部事件的触发下实现预定的行为，直至程序遭遇内部错误或因捕获外部信号而退出。

客户端与服务端的区别在于传给 `TCPRelay` 和 `UDPRelay` 的第三个参数 `is_local` 的值不同。也就是说，它们的行为差异要进入上述两个类中才能看出来。从代码体量出发，先分析 `UDPRelay` 显然是个更明智的选择。在构造函数里，我们看到了这样的代码：

```
# shadowsocks/udprelay.py

class UDPRelay(object):
    def __init__(self, config, dns_resolver, is_local, stat_callback=None):
        self._config = config
        if is_local:
            self._listen_addr = config['local_address']
            self._listen_port = config['local_port']
            self._remote_addr = config['server']
            self._remote_port = config['server_port']
        else:
            self._listen_addr = config['server']
            self._listen_port = config['server_port']
```

```
14              self._remote_addr = None
15              self._remote_port = None
```

不难看出，shadowsocks 的整体结构可以用下图表示出来：

```
user request <----> sslocal
                        |
                        |
                        |
destination  <----> ssserver
```

客户端 `sslocal` 监听 `local_address:local_port` 端口。当用户请求事件触发时，客户端试图从该端口读取数据，加密后发送至 `server:server_port`。服务端 `ssserver` 收到客户端的请求后，解密数据内容，解析目标 IP 地址，把用户的请求转发至目标服务器。当 `ssserver` 收到目标服务器的回应后，再把这些信息加密并送回客户端，由客户端最终响应用户的请求。

值得注意的是，虽然客户端和服务端都有 DNS 解析器，但他们承担的责任有很大的差异。客户端的 DNS 用于解析服务端的 IP 地址，如果在配置文件中使用 IP 代替域名，则客户端的解析器是一个可以移除的组件。服务端的 DNS 将用户请求中的域名翻译成目标服务器的 IP 地址，是必要的组件。考虑到服务端的部署地点，通常不会受到 DNS 污染的影响，因此可以安全地使用系统默认的 DNS 服务器。

shadowsocks # proxy

‹ LoveLive！学园偶像祭无氪党攻略

git cherry pick, reorder commit & delete commit ›