Shadowsocks: A secure SOCKS5 proxy

S.D.T

January 4, 2019

1 Overview

Shadowsocks is a secure split proxy loosely based on SOCKS5.

```
client <---> ss-local <--[encrypted]--> ss-remote <---> target
```

The Shadowsocks local component (`ss-local`) acts like a traditional SOCKS5 server and provides proxy service to clients. It encrypts and forwards data streams and packets from the client to the Shadowsocks remote component (`ss-remote`), which decrypts and forwards to the target. Replies from target are similarly encrypted and relayed by `ss-remote` back to `ss-local`, which decrypts and eventually returns to the original client.

1.1 Addressing

Addresses used in Shadowsocks follow the SOCKS5 address format:

```
    [1-byte type][variable-length host][2-byte port]
```

The following address types are defined:

- `0x01`: host is a 4-byte IPv4 address.
- `0x03`: host is a variable length string, starting with a 1-byte length, followed by up to 255-byte domain name.
- `0x04`: host is a 16-byte IPv6 address.

The port number is a 2-byte big-endian unsigned integer.

1.2 TCP

`ss-local` initiates a TCP connection to `ss-remote` by sending an encrypted data stream starting with the target address followed by payload data. The exact encryption scheme differs depending on the cipher used.

```
[target address][payload]
```

`ss-remote` receives the encrypted data stream, decrypts and parses the leading target address. It then establishes a new TCP connection to the target and forwards payload data to it. `ss-remote` receives reply from the target, encrypts and forwards it back to the `ss-local`, until `ss-local` disconnects.

1.3 UDP

`ss-local` sends an encrypted data packet containing the target address and payload to `ss-remote`.

```
[target address][payload]
```

Upon receiving the encrypted packet, `ss-remote` decrypts and parses the target address. It then sends a new data packet containing only the payload to the target. `ss-remote` receives data packets back from target and prepends the target address to the payload in each packet, then sends encrypted copies back to `ss-local`.

```
[target address][payload]
```

Essentially, `ss-remote` is performing Network Address Translation for `ss-local`.

2 Stream Cipher

Stream ciphers provide only confidentiality. Data integrity and authenticity is not guaranteed. Users should use AEAD ciphers whenever possible.

The following stream ciphers provide reasonable confidentiality.

Name	Key Size	IV Length
aes-128-ctr	16	16
aes-192-ctr	24	16
aes-256-ctr	32	16

Name	Key Size	IV Length
aes-128-cfb	16	16
aes-192-cfb	24	16
aes-256-cfb	32	16
camellia-128-cfb	16	16
camellia-192-cfb	24	16
camellia-256-cfb	32	16
chacha20-ietf	32	12

2.1 Stream Encryption/Decryption

`Stream_encrypt` is a function that takes a secret key, an initialization vector, a message, and produces a ciphertext with the same length as the message.

```
Stream_encrypt(key, IV, message) => ciphertext
```

`Stream_decrypt` is a function that takes a secret key, an initializaiton vector, a ciphertext, and produces the original message.

```
Stream_decrypt(key, IV, ciphertext) => message
```

The key can be input directly from user or generated from a password. The key derivation is following `EVP_BytesToKey(3)` in OpenSSL. The detailed spec can be found here.

2.2 TCP

A stream cipher encrypted TCP stream starts with a randomly generated initializaiton vector, followed by encrypted payload data.

```
[IV][encrypted payload]
```

2.3 UDP

A stream cipher encrypted UDP packet has the following structure

```
[IV][encrypted payload]
```

Each UDP packet is encrypted/decrypted independently with a randomly generated initialization vector.

3 AEAD Ciphers

AEAD stands for Authenticated Encryption with Associated Data. AEAD ciphers simultaneously provide confidentiality, integrity, and authenticity. They have excellent performance and power efficiency on modern hardware. Users should use AEAD ciphers whenever possible.

The following AEAD ciphers are recommended. Compliant Shadowsocks implementations must support `chacha20-ietf-poly1305`. Implementations for devices with hardware AES acceleration should also implement `aes-128-gcm`, `aes-192-gcm`, and `aes-256-gcm`.

Name	Key Size	Salt Size	Nonce Size	Tag Size
chacha20-ietf-poly1305	32	32	12	16
aes-256-gcm	32	32	12	16
aes-192-gcm	24	24	12	16
aes-128-gcm	16	16	12	16

The way Shadowsocks using AEAD ciphers is specified in SIP004 and amended in SIP007.

3.1 Key Derivation

The master key can be input directly from user or generated from a password. The key derivation is still following `EVP_BytesToKey(3)` in OpenSSL like stream ciphers.

HKDF_SHA1 is a function that takes a secret key, a non-secret salt, an info string, and produces a subkey that is cryptographically strong even if the input secret key is weak.

```
HKDF_SHA1(key, salt, info) => subkey
```

The info string binds the generated subkey to a specific application context. In our case, it must be the string “ss-subkey” without quotes.

We derive a per-session subkey from a pre-shared master key using `HKDF_SHA1`. Salt must be unique through the entire life of the pre-shared master key.

3.2 Authenticated Encryption/Decryption

`AE_encrypt` is a function that takes a secret key, a non-secret nonce, a message, and produces ciphertext and authentication tag. Nonce must be unique for a given key in each invocation.

```
    AE_encrypt(key, nonce, message) => (ciphertext, tag)
```

`AE_decrypt` is a function that takes a secret key, non-secret nonce, ciphertext, authentication tag, and produces original message. If any of the input is tampered with, decryption will fail.

```
    AE_decrypt(key, nonce, ciphertext, tag) => message
```

3.3 TCP

An AEAD encrypted TCP stream starts with a randomly generated salt to derive the per-session subkey, followed by any number of encrypted chunks. Each chunk has the following structure:

```
[encrypted payload length][length tag][encrypted payload][payload tag]
```

Payload length is a 2-byte big-endian unsigned integer capped at 0x3FFF. The higher two bits are reserved and must be set to zero. Payload is therefore limited to 16*1024 - 1 bytes.

The first AEAD encrypt/decrypt operation uses a counting nonce starting from 0. After each encrypt/decrypt operation, the nonce is incremented by one as if it were an unsigned little-endian integer. Note that each TCP chunk involves two AEAD encrypt/decrypt operation: one for the payload length, and one for the payload. Therefore each chunk increases the nonce twice.

3.4 UDP

An AEAD encrypted UDP packet has the following structure

```
    [salt][encrypted payload][tag]
```

The salt is used to derive the per-session subkey and must be generated randomly to ensure uniqueness. Each UDP packet is encrypted/decrypted independently, using the derived subkey and a nonce with all zero bytes.

4 Transport plugin

4.1 Architecture Overview

The plugin of shadowsocks is very similar to the Pluggable Transport plugins from Tor project. Dislike the SOCKS5 proxy design in PT, every SIP003 plugin works as a tunnel (or called local port forwarding). This design aims to avoid per-connection arguments in PT, leading to much easier implementation.

```
+-----------+                    +--------------------------+
| SS Client +-- Local Loopback --+  Plugin Client (Tunnel)  +--+
+-----------+                    +--------------------------+  |
                                                               |
          Public Internet (Obfuscated/Transformed traffic) ==> |
                                                               |
+-----------+                    +--------------------------+  |
| SS Server +-- Local Loopback --+  Plugin Server (Tunnel)  +--+
+-----------+                    +--------------------------+
```

4.2 Life cycle of a plugin

Very similar to PT, the plugin client/server is started as child process of shadowsocks client/server.

If any error happens, the child process of plugin should exit with a error code. Then, the parent process of shadowsocks stops as well (SIGCHLD).

When a shadowsocks client/server is stopped by user, the child process of plugin will also be terminated.

4.3 Passing arguments to a plugin

A plugin accepts arguments through environment variables.

a. Four **MUST-HAVE** environment variables are `SS_REMOTE_HOST`, `SS_REMOTE_PORT`, `SS_LOCAL_HOST` and `SS_LOCAL_PORT`. `SS_REMOTE_HOST` and `SS_REMOTE_PORT` are the hostname and port of the remote plugin service. `SS_LOCAL_HOST` and `SS_LOCAL_PORT` are the hostname and port of the local shadowsocks or plugin service.

b. One **OPTIONAL** environment variable is `SS_PLUGIN_OPTIONS`. If a plugin requires additional arguments, like path to a config file, these arguments can be passed as extra options in a formatted string. An example is 'obfs=http;obfs-host=www.baidu.com', where semicolons, equal signs and backslashes **MUST** be escaped with a backslash.

4.4 Compatibility with PT

For all the plugins from Tor projects, there are two possible ways to support them. 1) We can fork these plugins and modify them to support SIP003, e.g. obfs4-tunnel. 2) Implement a adapter of PT as SIP003 plugin.

4.5 Licenses of plugins

As all plugin services should run in a separate process, they can pick any license they like. There is no GPL restrictions for any plugin providers.

4.6 Restrictions

a. Plugin over plugin is NOT supported. Only one plugin can be enabled when a shadowsocks service is started. If you really need this feature, implement a plugin-over-plugin transport as a SIP003 plugin.
b. Only TCP traffic is forwarded. For now, there is no plan to support UDP traffic forwarding.

4.7 Example projects

- A SIP003 plugin for traffic obfuscating: simple-obfs.
- A shadowsocks implementation based on SIP003: shadowsocks/shadowsocks-libev.

5 URI Scheme

Shadowsocks supports a standard URI scheme, following RFC3986:

```
SS-URI = "ss://" userinfo "@" hostname ":" port ["/"] ["?"plugin] ["#" tag]
userinfo = websafe-base64-encode-utf8(method ":" password)
```

The last `/` should be appended if plugin is present, but is optional if only tag is present. For example:

```
    ss://YmYtY2ZiOnRlc3Q@192.168.100.1:8888/
        ?plugin=url-encoded-plugin-argument-value
        &unsupported-arguments=should-be-ignored
        #Dummy+profile+name.
```

This kind of URIs can be parsed by standard libraries provided by most languages.

For plugin argument, we use the similar format as `TOR_PT_SERVER_TRANSPORT_OPTIONS`, which have the format like

```
    simple-obfs;obfs=http;obfs-host=www.baidu.com
```

where colons, semicolons, equal signs and backslashes MUST be escaped with a backslash.

Examples:

```
ss://YWVzLTEyOC1nY206dGVzdA==@192.168.100.1:8888#Example1

ss://cmM0LW1kNTpwYXNzd2Q=@192.168.100.1:8888/
    ?plugin=obfs-local%3Bobfs%3Dhttp#Example2
```

6 Official implementations

6.1 Servers

- shadowsocks: The original Python implementation.
- shadowsocks-libev: Lightweight C implementation for embedded devices and low end boxes. Very small footprint (several megabytes) for thousands of connections.
- shadowsocks-go: Go implementation with multi-port, multi-password, user management and traffic statistics support for commercial deployments.
- go-shadowsocks2: Another Go implementation focusing on core features and code reusability.

6.1.1 Feature comparison

	ss	ss-libev	ss-go	go-ss2
TCP Fast Open	Y	Y	N	N
Multiuser	Y	Y	Y	N
Management API	Y	Y	N	N
Redirect mode	N	Y	N	Y
Tunnel mode	Y	Y	N	Y
UDP Relay	Y	Y	Y	Y
AEAD ciphers	Y	Y	N	Y
Plugin	N	Y	N	N

6.2 Clients

- shadowsocks-android: Android client.
- shadowsocks-windows: Windows client.
- shadowsocksX-NG: MacOS client.

- shadowsocks-qt5: Cross-platform client for Windows/MacOS/Linux.

6.2.1 Feature comparison

	ss-win	ssx-ng	ss-qt5	ss-android
System Proxy	Y	Y	N	Y
CHNRoutes	Y	Y	N	Y
PAC Configuration	Y	Y	N	N
Profile Switching	Y	Y	Y	Y
QR Code Scan	Y	Y	Y	Y
QR Code Generation	Y	Y	Y	Y
AEAD ciphers	Y	Y	N	Y
Plugin	Y	Y	N	Y