GitHub - securesocketfunneling/ssf: Secure Socket Funneling - Network tool and toolkit - TCP and UDP port forwarding, SOCKS proxy, remote shell, standalone and cross platform

By olachere

Archived: 2026-04-05 14:06:01 UTC

Secure Socket Funneling (SSF) is a network tool and toolkit.

It provides simple and efficient ways to forward data from multiple sockets (TCP or UDP) through a single secure TLS tunnel to a remote computer.

SSF is cross platform (Windows, Linux, OSX) and comes as standalone executables.

Features:

- Local and remote TCP port forwarding
- Local and remote UDP port forwarding
- Local and remote SOCKS server
- Local and remote shell through sockets
- File copy
- Native relay protocol
- TLS connection with the strongest cipher-suites

Download prebuilt binaries

Documentation

Build on Windows

Build on Unix/Linux

Cross compiling SSF (e.g. Raspberry Pi)

How to use

Command line

Client

Usage: `ssf[.exe] [options] server_address`

Options:

- `-v verbose_level` : Verbosity: critical|error|warning|info|debug|trace (default: info)
- `-q` : Quiet mode. Do not print logs
- `-p port` : Remote port (default: 8011)
- `-c config_file_path` : Specify configuration file. If not set, 'config.json' is loaded from the current working directory
- `-m attempts` : Max unsuccessful connection attempts before stopping (default: 1)
- `-t delay` : Time to wait before attempting to reconnect in seconds (default: 60)
- `-n` : Do not try to reconnect client if connection is interrupted
- `-g` : Allow gateway ports. Allow client to bind local sockets for a service to a specific address rather than "localhost"
- `-S` : Display microservices status (on/off)

Services options:

- `-D [[bind_address]:]port` : Run a SOCKS proxy on the server accessible on `[[bind_address]:]port` on the local side
- `-F [[bind_address]:]port` : Run a SOCKS proxy on the local host accessible from the server on `[[bind_address]:]port`
- `-X [[bind_address]:]port` : Forward server shell I/O to the specified port on the local side. Each connection creates a new shell process
- `-Y [[bind_address]:]port` : Forward local shell I/O to the specified port on the server
- `-L [[bind_address]:]port:host:hostport` : Forward TCP connections to `[[bind_address]:]port` on the local host to `host:hostport` on the server
- `-R [[bind_address]:]port:host:hostport` : Forward TCP connections to `[[bind_address]:]port` on the server to `host:hostport` on the local side
- `-U [[bind_address]:]port:host:hostport` : Forward local UDP traffic on `[[bind_address]:]port` to `host:hostport` on the server
- `-V [[bind_address]:]port:host:hostport` : Forward UDP traffic on `[[bind_address]:]port` on the server to `host:hostport` on the local side

Server

Usage: `ssfd[.exe] [options]`

Options:

- `-v verbose_level` : Verbosity: critical|error|warning|info|debug|trace (default: info)
- `-q` : Quiet mode. Do not print logs
- `-c config_file_path` : Specify configuration file. If not set, 'config.json' is loaded from the current working directory
- `-p port` : Local port (default: 8011)
- `-R` : The server will only relay connections
- `-l host` : Set server bind address
- `-g` : Allow gateway ports. Allow client to bind local sockets for a service to a specific address rather than "localhost"
- `-S` : Display microservices status (on/off)

Copy

The copy feature must be enabled on both client and server configuration file:

```
{
  "ssf": {
    "services": {
      "copy": { "enable": true }
    }
  }
}
```

Usage: `ssfcp[.exe] [options] [host@]/absolute/path/file [[host@]/absolute/path/file]`

Options:

- `-v verbose_level` : Verbosity: critical|error|warning|info|debug|trace (default: info)
- `-q` : Quiet mode. Do not print logs
- `-c config_file_path` : Specify configuration file. If not set, 'config.json' is loaded from the current working directory
- `-p port` : Remote port (default: 8011)
- `-t` : Use stdin as input
- `--resume` : Attempt to resume file transfer if the destination file exists

- `--check-integrity` : Check file integrity at the end of the transfer
- `-r` : Copy files recursively
- `--max-transfers arg` : Max transfers in parallel (default: 1)

Examples

Client

The client will run a SOCKS proxy on port 9000 and transfer connection requests to the server **192.168.0.1:8000**

```
ssf -D 9000 -c config.json -p 8000 192.168.0.1
```

Server

The server will be bound to port **8011** on all the network interfaces

The server will be bound to **192.168.0.1:9000**

```
ssfd -p 9000 -l 192.168.0.1
```

Copy local file to remote filesystem

```
ssfcp [-c config_file] [-p port] path/to/file host@absolute/path/directory_destination
```

```
ssfcp [-c config_file] [-p port] path/to/file* host@absolute/path/directory_destination
```

```
ssfcp [-c config_file] [-p port] -r path/to/dir host@absolute/path/directory_destination
```

Pipe file from standard input to remote filesystem

```
data_in_stdin | ssfcp [-c config_file] [-p port] -t host@path/to/destination/file_destination
```

Copy remote files to local filesystem :

```
ssfcp [-c config_file] [-p port] remote_host@path/to/file absolute/path/directory_destination
```

```
ssfcp [-c config_file] [-p port] remote_host@path/to/file* absolute/path/directory_destination
```

```
ssfcp [-c config_file] [-p port] -r remote_host@path/to/dir absolute/path/directory_destination
```

Configuration file

```
{
  "ssf": {
    "arguments": "",
    "circuit": [],
    "http_proxy": {
      "host": "",
      "port": "",
      "user_agent": "",
      "credentials": {
        "username": "",
        "password": "",
        "domain": "",
        "reuse_ntlm": true,
        "reuse_nego": true
      }
    },
    "socks_proxy": {
      "version": 5,
      "host": "",
      "port": "1080"
    },
    "tls" : {
      "ca_cert_path": "./certs/trusted/ca.crt",
      "cert_path": "./certs/certificate.crt",
      "key_path": "./certs/private.key",
      "key_password": "",
      "dh_path": "./certs/dh4096.pem",
      "cipher_alg": "DHE-RSA-AES256-GCM-SHA384"
    },
    "services": {
      "datagram_forwarder": { "enable": true },
      "datagram_listener": {
        "enable": true,
        "gateway_ports": false
      },
      "stream_forwarder": { "enable": true },
      "stream_listener": {
        "enable": true,
        "gateway_ports": false
      },
      "copy": { "enable": false },
```

```
      "shell": {
        "enable": false,
        "path": "/bin/bash|C:\\windows\\system32\\cmd.exe",
        "args": ""
      },
      "socks": { "enable": true }
    }
  }
}
```

Arguments

Configuration key	Description
arguments	use configuration arguments instead of given CLI arguments (except `-c`)

The `arguments` key lets the user customize the command line arguments in the configuration file. This feature is a convenient way to save different client connection profiles.

Given the following configuration file `conf.json`:

```
{
  "ssf": {
    "arguments": "10.0.0.1 -p 443 -D 9000 -L 11000:localhost:12000 -v debug"
  }
}
```

SSF will extract the given arguments and use them as a replacement of the initial arguments (except `-c`).

For example, `ssf -c conf.json` will be equivalent to `ssf 10.0.0.1 -p 443 -D 9000 -L 11000:localhost:12000 -v debug`:

- connect to `10.0.0.1:443` (`10.0.0.1 -p 443`)
- start the SOCKS service (`-D 9000`)
- start the TCP port forwarding service (`-L 11000:localhost:12000`)
- set verbosity level to debug (`-v debug`)

Circuit

Configuration key	Description
circuit	relay chain servers used to establish the connection to the remote server

The circuit is a JSON array containing the bounce servers and ports which will be used to establish the connection. They are listed as follow:

```
{
  "ssf": {
    "circuit": [
      {"host": "SERVER1", "port":"PORT1"},
      {"host": "SERVER2", "port":"PORT2"},
      {"host": "SERVER3", "port":"PORT3"}
    ]
  }
}
```

This configuration will create the following connection chain:

```
CLIENT -> SERVER1:PORT1 -> SERVER2:PORT2 -> SERVER3:PORT3 -> TARGET
```

Proxy

SSF supports connection through:

- HTTP proxy by using the `CONNECT` HTTP method
- SOCKS proxy (v4 or v5)

HTTP proxy

Configuration key	Description
http_proxy.host	HTTP proxy host
http_proxy.port	HTTP proxy port
http_proxy.user_agent	User-Agent header value in HTTP CONNECT request
http_proxy.credentials.username	proxy username credentials (all platform: Basic or Digest, Windows: NTLM and Negotiate if reuse = false)
http_proxy.credentials.password	proxy password credentials (all platform: Basic or Digest, Windows: NTLM and Negotiate if reuse = false)
http_proxy.credentials.domain	user domain (NTLM and Negotiate auth on Windows only)
http_proxy.credentials.reuse_ntlm	reuse current computer user credentials to authenticate with proxy NTLM auth (SSO)
http_proxy.credentials.reuse_kerb	reuse current computer user credentials (Kerberos ticket) to authenticate with proxy Negotiate auth (SSO)

Supported authentication schemes:

- Basic
- Digest
- NTLM (Windows only)
- Negotiate with Kerberos (reuse computer user credentials)

SOCKS proxy

Configuration key	Description
socks_proxy.version	SOCKS version (4 or 5)
socks_proxy.host	SOCKS proxy host
socks_proxy.port	SOCKS proxy port

No authentication scheme supported.

TLS

Using external files

Configuration key	Description
tls.ca_cert_path	relative or absolute filepath to the CA certificate file
tls.cert_path	relative or absolute filepath to the instance certificate file
tls.key_path	relative or absolute filepath to the private key file
tls.key_password	key password
tls.dh_path	relative or absolute filepath to the Diffie-Hellman file (server only)
tls.cipher_alg	cipher algorithm

With default options, the following files and folders should be in the working directory of the client or the server:

- `./certs/dh4096.pem`
- `./certs/certificate.crt`
- `./certs/private.key`
- `./certs/trusted/ca.crt`

Where:

- **dh4096.pem** contains the Diffie-Hellman parameters (generate DH parameters)
- **certificate.crt** and **private.key** are the certificate and the private key of the SSF server or client (generate certificate)
- **ca.crt** is the concatenated list of certificates trusted by the SSF server or client (generate CA)

If you want those files at different paths, it is possible to customize them thanks to the TLS path keys:

```
{
  "ssf": {
    "tls" : {
      "ca_cert_path": "./certs/trusted/ca.crt",
      "cert_path": "./certs/certificate.crt",
      "key_path": "./certs/private.key",
      "key_password": "",
      "dh_path": "./certs/dh4096.pem",
      "cipher_alg": "DHE-RSA-AES256-GCM-SHA384"
    }
  }
}
```

Using configuration file only

Configuration key	Description
tls.ca_cert_buffer	CA certificate file content in PEM format (:warning: `\n` between data and PEM header/footer)
tls.cert_buffer	instance certificate file content in PEM format (:warning: `\n` between data and PEM header/footer)
tls.key_buffer	private key file content in PEM format (:warning: `\n` between data and PEM header/footer)
tls.key_password	key password
tls.dh_buffer	Diffie-Hellman parameters file content in PEM format (:warning: `\n` between data and PEM header/footer, server only)
tls.cipher_alg	cipher algorithm

You can integrate the TLS parameters directly into the configuration file by using the `tls.ca_cert_buffer`, `tls.cert_buffer`, `tls.key_buffer` and `tls.dh_buffer` keys.

```
{
  "ssf": {
    "tls" : {
      "ca_cert_buffer":"-----BEGIN CERTIFICATE-----\n...\n-----END CERTIFICATE-----",
      "cert_buffer":"-----BEGIN CERTIFICATE-----\n...\n-----END CERTIFICATE-----",
      "key_buffer":"-----BEGIN RSA PRIVATE KEY-----\n...\n-----END RSA PRIVATE KEY-----",
      "key_password": "",
```

```
      "dh_buffer":"-----BEGIN DH PARAMETERS-----\n...\n-----END DH PARAMETERS-----",
      "cipher_alg": "DHE-RSA-AES256-GCM-SHA384"
    }
  }
}
```

Certificates, private keys and DH parameters must be in PEM format. ⚠️ `\n` between data and PEM header/footer are mandatory.

Microservices

Configuration key	Description
services.*.enable	enable/disable microservice
services.*.gateway_ports	enable/disable gateway ports
services.shell.path	binary path used for shell creation
services.shell.args	binary arguments used for shell creation

SSF's features are built using microservices (TCP forwarding, remote SOCKS, ...)

There are 7 microservices:

- stream_forwarder
- stream_listener
- datagram_forwarder
- datagram_listener
- copy
- socks
- shell

Each feature is the combination of at least one client side microservice and one server side microservice.

This table sums up how each feature is assembled:

ssf feature	microservice client side	microservice server side
`-L` : TCP forwarding	stream_listener	stream_forwarder
`-R` : remote TCP forwarding	stream_forwarder	stream_listener
`-U` : UDP forwarding	datagram_listener	datagram_forwarder
`-V` : remote UDP forwarding	datagram_forwarder	datagram_listener
`-D` : SOCKS	stream_listener	socks

ssf feature	microservice client side	microservice server side
`-F` : remote SOCKS	socks	stream_listener
`-X` : shell	stream_listener	shell
`-Y` : remote shell	shell	stream_listener

This architecture makes it easier to build remote features: they use the same microservices but on the opposite side.

`ssf` and `ssfd` come with pre-enabled microservices. Here is the default microservices configuration:

```
{
  "ssf": {
    "services": {
      "datagram_forwarder": { "enable": true },
      "datagram_listener": { "enable": true },
      "stream_forwarder": { "enable": true },
      "stream_listener": { "enable": true },
      "socks": { "enable": true },
      "copy": { "enable": false },
      "shell": { "enable": false }
    }
  }
}
```

To enable or disable a microservice, set the `enable` key to `true` or `false`.

Trying to use a feature requiring a disabled microservice will result in an error message.

How to generate certificates for TLS connections

Manually

Generating Diffie-Hellman parameters

```
openssl dhparam 4096 -outform PEM -out dh4096.pem
```

Generating a self-signed Certification Authority (CA)

First of all, create a file named *extfile.txt* containing the following lines:

```
 [ v3_req_p ]
basicConstraints = CA:FALSE
```

```
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
```

Then, generate a self-signed certificate (the CA) *ca.crt* and its private key *ca.key*:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout ca.key -out ca.crt -days 3650
```

Generating a private key and a certificate (signed with the CA)

Generate a private key *private.key* and a certificate signing request *certificate.csr*:

```
openssl req -newkey rsa:4096 -nodes -keyout private.key -out certificate.csr
```

Generate the certificate (*certificate.pem*) by signing the CSR with the CA (*ca.crt*, *ca.key*):

```
openssl x509 -extfile extfile.txt -extensions v3_req_p -req -sha1 -days 3650 -CA ca.crt -CAkey ca.ke
```
