fastd Documentation

Release 22+

Matthias Schiffer

Aug 25, 2023

User manual

fastd is a very small VPN daemon which tunnels IP packets and Ethernet frames over UDP. It supports various modern encryption and authentication schemes and can be used in many different network topologies (1:1, 1:n, meshed).

fastd runs on Linux, FreeBSD, OpenBSD and macOS. Android support exists in the code, but is currently unmaintained. Binary packages are provided by many major Linux distributions.

CHAPTER 1

Command line options

Command line options and config files are parsed in order they are specified, so config files specified before other options are overwritten by the other options, config files specified later will overwrite options specified before.

--help, -h Shows this help text

--version, -v Shows the fastd version

--daemon, -d Runs fastd in the background

--pid-file <filename> Writes fastd's PID to the specified file.

--status-socket <socket> Configures a socket to get fastd's status.

–log-level error|warn|info|verbose|debug|debug2 Sets the stderr log level; default is info, if no alternative log destination ist configured. If logging to syslog or files is enabled, the default is not to log to stderr.

–syslog-level error|warn|info|verbose|debug|debug2 Sets the log level for syslog output; default is not to use syslog.

--syslog-ident <ident> Sets the syslog identification; default is 'fastd'.

--config, -c <filename> Loads a config file. - can be specified to read a config file from stdin.

--config-peer <filename> Loads a config file for a single peer. The filename will be used as the peer name.

--config-peer-dir <dir> Loads all files from a directory as peer configs. On SIGHUP fastd will reload peer directories.

–mode, -m tap|multitap|tun Sets the mode of the interface; default is TAP mode.

--interface, -i <name> Sets the name of the TUN/TAP interface to use. If not specified, default names specified by the system will be used.

--mtu, -M <mtu> Sets the MTU; must be at least 576. You should read MTU configuration, the default 1500 is suboptimal in most setups.

--bind, -b <address:port> Sets the bind address. Address can be an IPv4 address or an IPv6 address, or the keyword any. IPv6 addresses must be put in square brackets.

Default is to bind to a random port, for IPv4 and IPv6. You can specify one IPv4 and one IPv6 bind address, or both at once as any. It is currently not possible to specify an IPv6 link-local address on the command line.

--protocol, -p <protocol> Sets the handshake protocol. Currently the only protocol available is ec25519-fhmqvc, which provides a secure authentication of peers based on public/secret keys.

--method <method> Sets the encryption/authentication method. See the page *Encryption & authentication methods* for more information about the supported methods. More than one method can be specified; the earlier you specify a method the higher is the preference for a method, so methods speficied later will only be used if a peer doesn't support the first methods.

--forward Enables forwarding of packets between clients; read the paragraph about this option before use!

--on-pre-up <command> Sets a shell command to execute before interface creation. See the detailed documentation below for an overview of the available environment variables.

--on-up <command> Sets a shell command to execute after interface creation. See the detailed documentation below for an overview of the available environment variables.

--on-down <command> Sets a shell command to execute before interface destruction. See the detailed documentation below for an overview of the available environment variables.

--on-post-down <command> Sets a shell command to execute after interface destruction. See the detailed documentation below for an overview of the available environment variables.

--on-connect <command> Sets a shell command to execute when a handshake is sent to establish a new connection.

--on-establish <command> Sets a shell command to execute when a new connection is established. See the detailed documentation below for an overview of the available environment variables.

--on-disestablish <command> Sets a shell command to execute when a connection is lost. See the detailed documentation below for an overview of the available environment variables.

--on-verify <command> Sets a shell command to execute to check a connection attempt by an unknown peer. See the detailed documentation below for more information and an overview of the available environment variables.

--verify-config Checks the configuration and exits.

--generate-key Generates a new keypair.

--show-key Shows the public key corresponding to the configured secret.

--machine-readable Suppresses output of explaining text in the –show-key and –generate-key commands.

CHAPTER 2

Configuration file format

2.1 Main configuration

Example config:

```
# Log warnings and errors to stderr
log level warn;

# Log everything to syslog
log to syslog level debug;

# Set the interface name
interface "mesh-vpn";

# Support salsa2012+umac and null methods, prefer salsa2012+umac
method "salsa2012+umac";
method "null";

# Bind to a fixed port, IPv4 only
bind 0.0.0.0:10000;

# Secret key generated by `fastd --generate-key`
secret "78dfb05fe0aa586fb017de566b0d21398ac64032fcf1c765855f4d538cc5a357";

# Set the interface MTU for TAP mode with xsalsa20/aes128 over IPv4 with a base MTU␣
↪of 1492 (PPPoE)
# (see MTU selection documentation)
mtu 1426;

# Include peers from the directory 'peers'
include peers from "peers";
```

```
bind <IPv4 address>[:<port>] [ interface "<interface>" ] [ default [ ipv4 ] ];
```

```
bind <IPv6 address>[:<port>] [ interface "<interface>" ] [ default [ ipv6 ] ];
bind any[:<port>] [ interface "<interface>" ] [ default [ ipv4|ipv6 ] ];
bind <IPv4 address> [port <port>] [ interface "<interface>" ] [ default [ ipv4
] ];
bind <IPv6 address> [port <port>] [ interface "<interface>" ] [ default [ ipv6
] ];
bind any [port <port>] [ interface "<interface>" ] [ default [ ipv4|ipv6 ] ];
```

Sets the bind address, port and possibly interface. May be specified multiple times. The keyword any makes fastd bind to the unspecified address for both IPv4 and IPv6.

IPv6 address must be put in square brackets. It is possible to specify an IPv6 link-local address with an interface in the usual notation (e.g. [fe80::1%eth0]).

The default option makes it the default address for outgoing connections for IPv4, IPv6 or both.

When an address with port 0 is configured, a random port will be selected, which will not change as long as fastd is running.

When the port is omitted completely, a new socket with a random port will be created for each outgoing connection. This has the side effect that the options for packet marks and interface-specific binds (except IPv6 link-local addresses) will only work with the `CAP_NET_ADMIN` capability. If fastd is built with capability support, it will automatically retain these capabilities; otherwise, fastd must run as root.

Configuring no bind address at all is equivalent to the setting `bind any`, meaning fastd will use a random port for each outgoing connection both for IPv4 and IPv6.

```
cipher "<cipher>" use "<implementation>";
```

Chooses a specific impelemenation for a cipher. Normally, the default setting is already the best choice. Note that specific implementations may be unavailable on some platforms or disabled during compilation. The available ciphers and implementations are:

- `aes128-ctr`: AES128 in counter mode
  - `openssl`: Use implementation from OpenSSL's libcrypto
- `null`: No encryption (for authenticated-only methods using composed_gmac)
  - `memcpy`: Simple memcpy-based implementation
- `salsa20`: The Salsa20 stream cipher
  - `xmm`: Optimized implementation for x86/amd64 CPUs with SSE2 support
  - `nacl`: Use implementation from NaCl or libsodium
- `salsa2012`: The Salsa20/12 stream cipher
  - `xmm`: Optimized implementation for x86/amd64 CPUs with SSE2 support
  - `nacl`: Use implementation from NaCl or libsodium

```
drop capabilities yes|no|early|force;
```

By default, fastd switches to the configured user and/or drops its POSIX capabilities after the on-up command has been run. When drop capabilities is set to *early*, the on-up command is run after the

privileges have been dropped, when set to *no*, the POSIX capabilities aren't dropped at all (but the user is switched after the on-up command has been run nevertheless).

fastd automatically detects which capabilities are required for normal operation and retains these capabilities. This can be overridden using the *force* value (this may make sense if persistent TUN/TAP interfaces are used which may be used without special privileges by fastd.)

```
forward yes|no;
```

Enables or disabled forwarding packets between peers. Care must be taken not to create forwarding loops.

```
group "<group>";
```

Sets the group to run fastd as.

```
hide ip addresses yes|no;
```

Hides IP addresses in log output.

```
hide mac addresses yes|no;
```

Hides MAC addresses in log output.

```
include "<file>";
```

Includes another configuration file. Relative paths are interpreted relatively to the including file.

```
include peer "<file>" [ as "<name>" ];
```

Includes a peer configuration (and optionally gives the peer a name).

```
include peers from "<dir>";
```

Includes each file in a directory as a peer configuration. These peers are reloaded when fastd receives a SIGHUP signal.

```
interface "<name>";
```

Sets the name of the TUN/TAP interface to use; it will be set by the OS when no name is configured explicitly.

In TUN/multi-TAP mode, either peer-specific interface names need to be configured, or one (but not both) of the following patterns must be used to set a unique interface name for each peer:

- `%n`: The peer's name
- `%k`: The first 16 hex digits of the peer's public key

```
log level fatal|error|warn|info|verbose|debug|debug2;
```

Sets the default log level, meaning syslog if there is currently a level set for syslog, and stderr otherwise.

```
log to stderr level fatal|error|warn|info|verbose|debug|debug2;
```

Sets the stderr log level. By default no log messages are printed on stderr, unless no other log destination is configured, which causes fastd to log to stderr with level info.

```
log to syslog [ as "<ident>" ] [ level
fatal|error|warn|info|verbose|debug|debug2 ];
```

Sets the syslog log level. By default syslog isn't used.

```
mac "<MAC>" use "<implementation>";
```

Chooses a specific impelemenation for a message authentication code. Normally, the default setting is already the best choice. Note that specific implementations may be unavailable on some platforms or disabled during compilation. The available MACs and implementations are:

- `ghash`: The MAC used by the GCM and GMAC methods
  - `pclmulqdq`: An optimized implementation for modern x86/amd64 CPUs supporting the PCLMULQDQ instruction
  - `builtin`: A generic implementation
- `uhash`: The MAC used by the UMAC methods
  - `builtin`: A generic implementation

```
method "<method>";
```

Sets the encryption/authentication method. See the page *Encryption & authentication methods* for more information about the supported methods. When multiple method statements are given, the first one has the highest preference.

```
mode tap|multitap|tun;
```

Sets the mode of the interface; the default is TAP mode.

In TAP mode, a single interface will be created for all peers, in multi-TAP and TUN mode, each peers gets its own interface.

```
mtu <MTU>;
```

Sets the MTU; must be at least 576. You should read the page *MTU configuration* as the default 1500 is suboptimal in most setups.

```
offload l2tp yes|no;
```

Use the L2TP kernel implementation for the "null@l2tp" method. Enabling offloading allows for significantly higher throughput, as data packets don't need to be copied between kernel and userspace.

L2TP offloading is only avaiable when the following conditions are met:

- A Linux kernel with L2TP Ethernet Pseudowire support is required
- L2TP offloading must be enabled in the fastd build (default on Linux)
- `mode` must be set to `multitap`
- `persist interface` must be set to `no`

Using the multi-TAP mode can be inconvenient with high numbers of peers, as it will create a separate network interface for each peer. As peers in TAP and multi-TAP modes can communicate with each other, using TAP mode without offloading on a powerful server, but multi-TAP on weak devices that only establish a single connection anyways is an option that combines convenience with high performance.

Using L2TP offloading requires fastd to set the *SO_REUSEADDR* flag flag on its sockets. This allows other local users to open a socket with the same address/port combination, making it possible to monitor and inject traffic on such unencrypted connections without special privileges. For this reason, using a privileged bind port (below 1024) is recommended when using the offload feature on hosts shared with other users.

```
on pre-up [ sync | async ] "<command>";
on up [ sync | async ] "<command>";
on down [ sync | async ] "<command>";
on post-down [ sync | async ] "<command>";
on connect [ sync | async ] "<command>";
on establish [ sync | async ] "<command>";
on disestablish [ sync | async ] "<command>";
```

Configures a shell command that is run after the interface is created, before the interface is destroyed, when a handshake is sent to make a new connection, when a new peer connection has been established, or after a peer connection has been lost. fastd will block until the command has finished, to long-running processes should be started in the background.

pre-up, up, down and post-down commands are executed synchronously by default, meaning fastd will block until the commands have finished, while the other commands are executed asynchronously by default. This can be changed using the keywords sync and async.

All commands except pre-up and post-down may be overriden per peer group.

The following environment variables are set by fastd for all commands:

- `FASTD_PID`: fastd's PID
- `INTERFACE`: the interface name

- INTERFACE_MTU: the configured MTU
- LOCAL_KEY: the local public key

For on connect, on establish and on disestablish the following variables are set in addition:

- LOCAL_ADDRESS: the local IP address
- LOCAL_PORT: the local UDP port
- PEER_ADDRESS: the peer's IP address
- PEER_PORT: the peer's UDP port
- PEER_NAME: the peer's name in the local configuration
- PEER_KEY: the peer's public key

```
on verify [ sync | async ] "<command>";
```

Configures a shell command that is run on connection attempts by unknown peers. The same environment variables as in the on establish command are supplied. When the commands returns 0, the connection is accepted, otherwise the handshake is ignored. By default, fastd ignores connections from unknown peers.

Verify commands are executed asynchronously by default. This can be changed using the keywords sync and async.

The on-verify command my be put into a peer group to define which peer group unknown peers are added to. This may be used to apply a peer limit only to unknown peers.

```
packet mark <mark>;
```

Defines a packet mark to set on fastd's packets, which can be used in an ip rule.

Marks can be specified in decimal, hexadecimal (with a leading 0x), and octal (with a leading 0).

```
peer "<name>" { peer configuration }
```

An inline peer configuration.

```
peer group "<name>" { configuration }
```

Configures a peer group.

```
peer limit <limit>;
```

Sets the maximum number of connections for the current peer group.

```
persist interface yes|no;
```

If set to *no*, fastd will create peer-specific interfaces only as long as there's an active session with the peer. Does not have an effect in TUN mode.

By default, interfaces are persistent.

```
pmtu yes|no|auto;
```

Does nothing; the `pmtu` option is only supported for compatiblity with older versions of fastd.

```
protocol "<protocol>";
```

Sets the handshake protocol; at the moment only ec25519-fhmqvc is supported.

```
secret "<secret>";
```

Sets the secret key.

```
status socket "<socket>";
```

Configures a UNIX socket which can be used to retrieve the current state of fastd. An example script to get the status can be found at `doc/examples/status.pl` in the fastd repository.

```
user "<user>";
```

Sets the user to run fastd as.

2.2 Peer configuration

Example config:

```
key "f05c6f62337d291e34f50897d89b02ae43a6a2476e2969d1c8e8104fd11c1873";
remote 192.0.2.1:10000;
remote [2001:db8::1]:10000;
remote ipv4 "fastd.example.com" port 10000;
```

```
include "<file>";
```

Includes another configuration file.

```
interface "<name>";
```

Sets the name of the peer-specific TUN/TAP interface to use.

Does have no effect in TAP mode.

```
key "<key>";
```

Sets the peer's public key.

```
mtu <MTU>;
```

Sets the MTU for a peer-specific interface; must be at least 576.

Does have no effect in TAP mode.

```
remote <IPv4 address>:<port>;
remote <IPv6 address>:<port>;
remote [ ipv4|ipv6 ] "<hostname>":<port>;
remote <IPv4 address> port <port>;
remote <IPv6 address> port <port>;
remote [ ipv4|ipv6 ] "<hostname>" port <port>;
```

Sets the IP address or host name to connect to. If a peer doesn't have a remote address configured, incoming connections are accepted, but no own connection attempts will be made.

The ipv4 or ipv6 options can be used to force fastd to resolve the host name for the specified protocol version only.

Starting with fastd v9, multiple remotes may be given for a single peer. If this is the case, they will be tried one after another. Starting with fastd v11, all addresses a given hostname resolves to are taken into account, not only the first one. This can be use to specify alternative hostname, addresses and/or ports for the same host; all remotes must still refer to the same peer as the public key must be unique.

```
float yes|no;
```

Set to `yes` to allow incoming connections from any IP address/port for this peer. By default, incoming connections are only accepted from the addresses/ports configured using the `remote` option when such an option exists. Peers without `remote` are always floating.

CHAPTER 3

Encryption & authentication methods

fastd supports various combinations of ciphers and authentication schemes using different method providers. All ciphers, message authentication codes (MACs) and method providers can be disabled during compilation to reduce the binary size.

See Benchmarks for an overview of the performance of the different methods.

3.1 Recommended methods

The method `salsa2012+umac` is recommended for authenticated encyption. `null+salsa2012+umac` is the recommended method for authenticated-only operation.

Salsa20/12 is a stream cipher with very high speed and a very comfortable security margin. It has been chosed for the software profile in the eSTREAM project in 2008.

UMAC is an extremely fast message authentication code which is provably secure and optimized for software implementations.

3.2 List of methods

3.2.1 Encrypted methods

Method	Method provider	Cipher	MAC	Notes
`aes128-gcm`	generic-gmac	aes128-ctr	ghash	[2]
`salsa20+gmac`	generic-gmac	salsa20	ghash	
`salsa2012+gmac`	generic-gmac	salsa2012	ghash	
`aes128-ctr+umac`	generic-umac	aes128-ctr	uhash	[2]
`salsa20+umac`	generic-umac	salsa20	uhash	
`salsa2012+umac`	generic-umac	salsa2012	uhash	
`aes128-ctr+poly1305`	generic-poly1305	aes128-ctr	none[1]	[2],[3]
`salsa20+poly1305`	generic-poly1305	salsa20	none[1]	[3]
`salsa2012+poly1305`	generic-poly1305	salsa2012	none[1]	[3]

This list is not exhaustive. It is possible to combine different ciphers for data and authentication tag encryption using the *composed-gmac* and *composed-umac* method providers; these methods aren't listed here as this is not very useful.

3.2.2 Authenticated-only methods

Method	Method provider	Cipher	MAC	Notes
`null+aes128-gmac`	composed-gmac	aes128-ctr	ghash	[2],[4]
`null+salsa20+gmac`	composed-gmac	salsa20	ghash	[4]
`null+salsa2012+gmac`	composed-gmac	salsa2012	ghash	[4]
`null+aes128-ctr+umac`	composed-umac	aes128-ctr	uhash	[2],[4]
`null+salsa20+umac`	composed-umac	salsa20	uhash	[4]
`null+salsa2012+umac`	composed-umac	salsa2012	uhash	[4]

3.2.3 Methods without security

Method	Method provider	Cipher	MAC	Notes
`null@l2tp`	null-l2tp	none	none	[5]
`null`	null	none	none	[5]

[2] AES is very slow without OpenSSL support. OpenSSL's AES implementation may be suspect to cache timing side channels when no hardware support like AES-NI is available.

[1] The MAC is integrated in the method provider.

[3] Poly1305 is very slow on embedded systems.

[4] The cipher is used to encrypt the authentication tag only, the actual data is transmitted unencrypted.

[5] Only authentication of peers' IP addresses, but no encryption or authentication of any data is provided.

CHAPTER 4

MTU configuration

The default MTU of fastd is 1500. This allows briding the fastd interface in TAP mode with other interface with the same MTU, but will usually cause fastd's UDP packets to be fragmented. Fragmentation can lower the performance or even cause connectivity problems when broken routers filter ICMP packets, so if possible the MTU should be chosen small enough so that IP fragmentation can be avoided. Unlike OpenVPN, fastd doesn't support fragmentation itself, but relies on the IP stack to fragment packets when necessary.

4.1 Guidelines

- The basic overhead of a fastd packet in TUN mode over IPv4 is 28 bytes plus method-specific overhead
- Method "null" uses 1 additional header byte, "null@l2tp" 8 bytes, and all other methods 24 bytes
- TAP mode needs 14 bytes more than TUN mode
- Tunneling over IPv6 needs 20 bytes more than IPv4

4.2 Examples

Your base MTU is 1500 and you want to use TUN mode over IPv4 with any crypto method: Choose 1500 - 28 - 24 = 1448 bytes.

Your base MTU is 1492 (like most German DSL lines) and you want to use TAP mode over IPv4 with any crypto method: Choose 1492 - 28 - 24 - 14 = 1426 bytes.

Conservative choice when you want to transfer IPv6 inside the tunnel: Choose 1280 Bytes (not relevant when you use batman-adv inside the tunnel as batman-adv will take care of the inner fragmentation.)

Conservative choice when you don't know anything (but assume the base MTU is at least 1280 so IPv6 can be supported) and w Choose 1280 - 28 - 24 - 14 - 20 = 1194 bytes.

CHAPTER 5

fastd v22

The main improvement of fastd v22 is the L2TP kernel offloading support, which brings fastd's throughput for unsecured connections on par with other L2TP solutions like Tunneldigger, while maintaining most of fastd's flexibility. It is even possible to use fast L2TP connections for some peers and secure encryption for others in a single fastd instance.

5.1 New features

- Added new method "null@l2tp"

  Like the old "null" method, "null@l2tp" doesn't provide any security. In TAP mode, it uses a packet format compatible with L2TPv3 Ethernet Pseudowires (RFC3931 and RFC4719) for payload data.

  Using "null@lt2p" for new unsecured deployments and migrating existing "null" setups is recommended for a number of reasons:

  - "null" uses a 1-byte packet header, which can make data transfer between kernel and userspace slightly slower on platforms that care about alignment
  - The L2TP-compatible data format facilitates debugging, as packet sniffers like Wireshark can decode the payload
  - L2TP can be offloaded to the Linux kernel, significantly increasing throughput

  See *offload configuration* for information on the setup and limitations of the L2TP offload feature.

- Added support for NetBSD (tested on NetBSD 9.2)

5.2 Bugfixes

- Fix build for MacOS

  This issue was introduced during the move to the Meson build system in fastd v20.

- Fix TUN mode crash on FreeBSD/OpenBSD

  This issue is a regression introduced in fastd v20. The buffer management optimization caused an assertion failure in many configurations upon reading packets from the TUN interface.

- Fix version number format

  When not building from Git, fastd v21 would format its own version number as "21" rather than "v21", deviating from previous releases. This is fixed with v22.

5.3 Other changes

- A new handshake format has been introduced, prepending an L2TPv3 Control Message header to the actual fastd handshake. This improves certain interactions between fastd and the L2TP kernel module used for offloading.

  To maintain compatibility with older fastd versions, both handshake formats are accepted. For the initial handshake packet, an old and a new format packet are sent at the same time.

  Sessions established using the old handshake format are marked with "compat mode" in the log.

CHAPTER 6

fastd v21

This is a critical bugfix release. All users of fastd v20 must update.

6.1 Bugfixes

The new buffer management of fastd v20 revealed that received packets with an invalid type code were handled incorrectly, leaking the packet buffer. This lead to an assertion failure as soon as the buffer pool was empty, crashing fastd.

Older versions of fastd are affected as well, but display a different behaviour: instead of crashing, the buffer leaks will manifest as a regular memory leak. This can still be used for Denial of Service attacks, so a patch for older versions will be provided, for the case that users can't or do not want to update to a newer version yet.

CHAPTER 7

fastd v20

7.1 New features

- The OpenWrt init script has been migrated to *USE_PROCD*. This allows automatic restarts of fastd in the case of crashes.
  - The UCI options `up` and `down` have been removed, as their implementation was incompatible with procd-managed services. Use the options `on_up` and `on_down` instead.
- Stale status sockets are deleted automatically now, so a restart after a crash does not fail. To avoid deleting active status sockets, a lock file is created next to the socket file.

7.2 Bugfixes

- The *forward* feature was implemented incorrectly for configurations that use different encryption methods for different peers. As *forward* is generally disabled when running mesh routing protocols over fastd, affected configurations are very uncommon.
  - Packets forwarded from peers using the *null* method were not aligned correctly, which could lead to inferior performance or crashes on non-x86 platforms.
  - Certain combinations of encryption methods led to crashes due to insuffucient buffer space, triggering an assertion failure.

7.3 Other changes

- The CMake-based build system of fastd has been replaced with the more modern Meson. Updated build instructions can be found in the *Building fastd* section.
- Memory management of fastd's packet buffers has been optimized, increasing throughput by 5~10% for many encryption methods, especially on low-end hardware.

- The fastd build requires at least Bison 2.6 now.

CHAPTER 8

fastd v19

This is mostly a maintenance release with few new features.

8.1 New features

- Add support for OpenSSL 1.1+
- Allow binding to a fixed random port
  - By specifying port 0 in a *bind* directive, fastd will bind to a random port that is stable over the whole runtime of the fastd instance. The existing behaviour to use a new random port for each connection is preserved (by not specifying a port at all).

8.2 Removed features

- The *secure handshakes* option is deprecated and has no effect with fastd v19; the old (pre-v11) insecure handshake scheme is not supported anymore
- The deprecated *xsalsa20-poly1305* method has been removed; *salsa20+poly1305* and various faster methods exist since fastd v11
- As libsodium removed the *aes128-ctr* cipher, fastd doesn't support it anymore either (for both libsodium and NaCl). For AES support, fastd must be built with OpenSSL.

8.3 Bugfixes

- Fix build with custom CMAKE_MODULE_PATH (as often used by embedded build environments like buildroot
- Fix build on MacOS 10.12+

- Fix fast reconnect when changing networks on recent Linux kernels
- Fix segfault in *tun*/*multitap* mode with *persist interface no*
- Fix segfault in resolver with musl libc 1.1.20+
- Fix segfault when failing to create an interface on FreeBSD
- Do not print local address as a v4-mapped IPv6 address in log messages and script environments for sockets bound to *any*
- Fix OpenWrt initscript with multiple instances
- Fix OpenWrt initscript with multiple interfaces (*tun*/*multitap* mode)
- Fix *tap*/*multitap* modes on OpenBSD 5.9+
  - Note: This breaks support for older OpenBSD versions

8.4 Other changes

- Allocation functions were hardened against a number of theoretical integer overflow issues
- The alternative handshake format introduced in fastd v17 was removed again. The benefit of making endianess of the fastd packet formats more consistent does not outweigh the downsides of creating an incompatible fastd protocol version with a future release.

CHAPTER 9

fastd v18

9.1 New features

9.1.1 Multi-interface modes

A single fastd instance can now manage multiple TUN/TAP interfaces. This allows to use multiple peers and peer directories in TUN mode, creating one interface for each peer. *on-up* and *on-down* scripts are run once for each interface.

By default, all interfaces are created on startup or peer reload; the option *interface persist* can be used to change this behaviour.

In addition to the multi-peer TUN mode, it is also possible to make fastd create one interface per peer in TAP mode now. This is enabled by the setting *mode multitap* (the option for multi-interface TUN mode is just *mode tun*, as there is no TUN mode which handles multiple peers on a single interface.)

Multi-TAP mode is compatible with TAP mode, i.e. the peer may be configured in normal TAP mode (and may use a fastd version without multi-TAP support.

If explicit interface names are configured, these names must now be set for each peer, which may either be done explicitly, or using *name patterns*.

Peer-specific interfaces may also be configured with peer-specific MTUs.

9.1.2 Interface cleanup on FreeBSD/OpenBSD

FreeBSD and OpenBSD do not automatically destroy TUN/TAP interfaces. fastd will now destroy the interfaces it creates on these systems on exit.

9.1.3 Improved capability management

fastd will now automatically retain all POSIX capabilities it needs, so all options should now work without full root privileges.

The option *drop capabilities force* may be used to drop CAP_NET_ADMIN even when fastd would normally retain it.

9.1.4 More powerful peer groups

All *on-** options may now be overridden per peer group.

In particular, the *on-verify* option may be moved into a peer group to determine the peer group of unconfigured peers. This allows to set a peer limit for unconfigured peers without globally limiting the peer count.

9.2 Bugfixes

- When linked with NaCl instead of libsodium, fastd would use SSE for salsa20/salsa2012 on x86 even after determining that SSE is not availble. This led to crashes or transmission failures on CPUs like the Geode.
- Fix crash on x86-64 systems when built with certain combinations of GCC version and stack-protector compiler flags (observed on Fedora)
- fastd did reject configurations which contain neither static peers nor peer directories, but a *on-verify* option
- The status socket is now removed correctly if fastd exits with an error message
- fastd did exit with regular exit code 0 instead of re-raising the termination signal after cleanup
- Fix in-tree compile on non-Linux systems

9.3 Other changes

- fastd now requires at least libuecc v6 (v7 recommended)
- Some error conditions that can't be recovered from will now cause fastd to exit instead of just logging an error message. This allows service managers like systemd/procd to restart fastd, so proper operation can be restored.

CHAPTER 10

fastd v17

10.1 New features

- Per-peer-group method specification

  It is now possible to override the supported crypto methods per peer group.

- Connection reset via SIGUSR2

  Sending a SIGUSR2 to the fastd process will reset all connections.

- Support for Android 4.1+

  Contributed by Rick Lei. See `doc/README-Android.md`.

- Faster handshake

  fastd's handshake should now take significantly less time (about 30-50%, not regarding the network latency). Due tue this change fastd depends on libuecc v5 (which is released together with fastd v17) now.

10.2 Bugfixes

- Removed broken `pmtu` option

  The `pmtu` option was changed into a no-op (and fastd's behaviour was changed to what was `pmtu no` before) as fastd didn't handle a potentially discovered smaller path MTU correctly. It will probably return in a future version of fastd.

- Improve handling of incoming packets from many peers after restarting fastd

  fastd will generate only one handshake per peer every 15 seconds now instead of one handshake per incoming packet.

- Added a missing security check during handshake

  While I don't think this issue allowed an attacker to impersonate a legitimate peer or perform a man-in-the-middle attack, fastd did accept some weird keys (the identity point) as valid keys, which shouldn't be possible.

- Fixed handling of severely reordered packets

  While fastd is supposed to handle reordered packets up to 64 sequence numbers, a bug would cause it to drop all older packets after a packet with a sequence number more than 64 packets in the future was received.

  The "verification failed" message has been downgraded from the "verbose" to the "debug2" level as it will cause a lot of log spam when there is extreme reordering.

- x86 uClibc workaround

  A workaround has been added for systems without or with broken `epoll_pwait` libc wrappers. One libc with such a broken wrapper is the uClibc version used in OpenWrt on x86, which made fastd fail on OpenWrt x86 systems.

- Only send packets from configured bind addresses

  When a configuration file contains only an IPv4 bind address and fastd tried to connect to an IPv6 remote address, it would use a random source port instead of falling back to IPv4 (and vice-versa).

  The behaviour without any bind addresses in the configuration hasn't been changed.

10.3 Other changes

- Better debug messages

  The sender's public key will now be printed with more messages regarding handshake issues.

- New handshake format

  Some parts of the handshake had been submitted as little endian for historical reasons. As the normal network byte order is big endian, support for a new handshake format using big endian has been added.

  fastd will continue to send its handshake the old format for the next versions to maintain compatiblity, but it does also understand the new format and will thus also work with future fastd versions which use the new handshake.

- MTU mismatch is fatal

  fastd will now refuse to perform a handshake instead of just printing a warning when its configured MTU doesn't match the peer's one. Such a configuration is always broken and will lead to issues with big packets.

CHAPTER 11

fastd v16

11.1 Bugfixes

- Fix segmentation fault after peers with static IP addresses have been loaded
- Fix segmentation fault when status sockets are used with unnamed peers (e.g. peers authenticated by a on-verify handler)

11.2 Other changes

- The JSON output of the status sockets has changed

  To fix using the status socket with peers without names or with duplicate names, the peers' public keys are now use as the keys in the JSON object.

CHAPTER 12

fastd v15

12.1 New features

- New message authentication code UMAC

  The new message authentication code UMAC provides very high security with much higher performance than the old GMAC methods. "salsa2012+umac" and "null+salsa2012+umac" are the new recommended methods for authenticated encryption and authenticated-only operation.

- Status socket

  A unix socket can be configured with the new *status socket* option. fastd will dump its current state as JSON on every connection on this socket; this status output is much more detailed than the old SIGUSR1 output. SIGUSR1 is ignored now.

  To compile fastd with status socket support, libjson-c is required. An example script to get the status can be found at `doc/examples/status.pl`.

- MacOS X support

  fastd should now also run on recent versions of MacOS X. The inofficial TUN/TAP driver is required for this.

- New Sphinx-based documentation
- Fix warnings with CMake 3.0
- OpenWrt: allow setting on-connect, on-verify, on-establish… hooks via UCI
- OpenWrt: allow specifying bind interfaces in UCI

12.2 Bugfixes

- Signal handling improvements

  This should fix an issue where asynchronous handler scripts would be left as zombie processes occasionally.

- Config check fixes in TUN mode

  For some configuration mistakes, fastd would segfault instead of printing an error message.

12.3 Other changes

- Nicer error messages for common configuration mismatches like having no common methods
- When no port is given in a *bind* directive, a new random port will be chosen now for every new connection attempt (like it was already done when no bind address was configured at all)

  This allows setting additional bind options like interface binds without setting a static port.
- The peer hashtable is now grown dynamically to reduce memory usage for small numbers of peers and improve performance for huge numbers of peers
- Major refactoring: the internal peer and peer config structs have been merged
- Internally, int64 timestamps in milliseconds are now used always instead of struct timespec

  Milliseconds resolution and int64 range is completely sufficient, and many parts of the code have become simpler due to this change.

CHAPTER 13

ec25519

13.1 Twisted Edwards curves

In general, a twisted Edwards curve is a mathematical group on the points satisfying an equation of the form

$$ax^2 + y^2 = 1 + dx^2y^2$$

For purposes of cryptography the curve is defined on a finite field.

The corresponding group law is

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}\right)$$

For further information on twisted Edwards curves see [BBJ+08].

13.1.1 Extended coordinate representation

Representing a curve point as an coordinate pair (x, y) is rather inconvenient for calculations on points as reciprocation is a very expensive operation. [HWCD08] specifies an alternative representation: the *extended coordinate* representation, which stores a point as a tuple of four coodinates X, Y, Z and T, satisfying the following equations:

$$x = \frac{X}{Z} \qquad y = \frac{Y}{Z} \qquad x \cdot y = \frac{T}{Z}$$

By storing the denominator of the fractions as Z, consequent group operations can be performed without having to compute reciprocals until a canonical representation is needed again. The additional value T is used to speed up some operations.

The extended coordinate representation of twisted Edwards curves allows very efficient *strongly unified addition*; the term *strongly unified addition* denotes that the implementation of the addition operation can be used to double a point as well, so the special case of adding a point to itself doesn't have to be implemented specifically.

As the data of the Explicit-Formulas Database [EFD] suggests, the extended coordinate representation of twisted Edwards curves allows strongly unified addition with the least number of operations of all similar curve types and representations in the database (i. e. 9 multiplications), which is the principal reason a twisted Edwards curve has been chosen for fastd's handshake.

13.1.2 Point compression

As the points of an elliptic curve satisfy a curve equation, it is possible to transform the coordinates of a point into a more compact representation for transmission or storage. The twisted Edwards curve equation can be transformed to:

$$y^2 = \frac{1 - ax^2}{1 - dx^2}$$

As one can easily see, there are at most two possible y values for each value of x (this rule also holds when the elliptic curve is defined over a finite field), thus one bit is enough to distinguish between the two values.

For the curve used by fastd this means: as it is defined over a field with the cardinality $2^{255} - 19$, 255 bit are necessary to store a coordinate. Point compression allows to conveniently pack the 255 bit x coordinate with the least significant bit of the y coordinate into a 256 bit representation.

Even though this optimization is quite obvious, it was protected by US patent 6,141,420 ([VMA00]), which chould have complicated the operation of fastd when subject to the US patent law. Fortunately, the patent has expired on 29 July 2014.

13.2 The curve used by ec25519

The curve used by ec25519 is based on Curve25519 (see [Ber06]).

Curve25519 uses a Montgomery curve in a reduced representation, which allows very fast scalar multiplication, but makes it impossible to perform simple additions on curve points. Therefore an equivalent twisted Edwards curve is used for fastd.

Curve25519 is defined by the following equation:

$$v^2 = u^3 + 486662u^2 + u$$

over the prime field F_p for the prime $p = 2^{255} - 19$.

[BBJ+08] states that for all Montgomery curves

$$Bv^2 = u^3 + Au^2 + u$$

with $A \in F_p \setminus \{-2, 2\}$ and $B \in F_p \setminus \{0\}$ there is a birationally equivalent twisted Edwards curve

$$ax^2 + y^2 = 1 + dx^2y^2 \text{ with } a = \frac{A+2}{B} \text{ and } d = \frac{A-2}{B},$$

thus leading to the following curve equation:

$$486664x^2 + y^2 = 1 + 486660x^2y^2$$

13.2.1 Generator point

Curve25519 uses a point with

$$u = 9$$

as its generator; the v coordinate is not specified as it is not needed by the algorithm.

The two possible v coordinates are:

$v1 =$ `0x20ae19a1b8a086b4e01edd2c7748d14c923d4d7e6d7c61b229e9c5a27eced3d9`

$v2 =$ `0x5f51e65e475f794b1fe122d388b72eb36dc2b28192839e4dd6163a5d81312c14`

Out of (u, v_1) and (u, v_2), the point (u, v_1) has been arbitrarily chosen to be used in fastd; using the equivalence between Montgomery and twisted Edwards curves given by [BBJ+08]

$$x = \frac{u}{v}$$
$$y = \frac{u - 1}{u + 1}$$

this leads to the coordinates

$$x = \texttt{0x547c4350219f5e19dd26a3d6668b74346a8eb726eb2396e1228cfa397ffe6bd4}$$
$$y = \texttt{0x6666666666666666666666666666666666666666666666666666666666666658}$$

which specify the generator point G that is used by fastd's `ec25519-fhmqvc`. Like (u, v_1) on the Montgomery curve, the point $G = (x, y)$ on the twisted Edwards curve has the order

$$|G| = 2^{252} + 27742317777372353535851937790883648493$$

13.3 Implementation

The elliptic curve operations used by fastd have been implemented as a reusable library, *libuecc*, which is developed together with fastd. Large portions of the implementation, especially arithmetic modulo $2^{255} - 19$, haven been taken from the original Curve25519 implementation, which has been released in to the public domain by its author D. J. Bernstein.

Like in the Curve25519 implementation, great care has been taken to ensure that there are no data-dependent branches or array accesses, thus making *libuecc* resistant to timing attacks.

13.4 Bibliography

CHAPTER 14

FHMQV-C

FHMQV (Fully Hashed Menezes-Qu-Vanstone) is an extended, implicitly authenticated Diffie-Hellman key exchange which has been specified in [SEB09], correcting issues found in the earlier MQV ([LMQ+98]) and Hashed MQV ([Kra05]) algorithms. It should be noted that proof of security provided by [SEB09] was recently found to be faulty in [LSW+14]; nevertheless it is very unlikely that this has an impact on the security of the algorithm in practise.

The modified algorithm FHMQV-C specified in the same document also provides *Perfect Forward Secrecy* (PFS), which isn't the case for the simple FHMQV algorithm.

Like all MQV protocols, Alice and Bob have two key pairs each in FHMQV-C:

- A long term key pair (called a and $\hat{A}$ for Alice, b and $\hat{B}$ for Bob)

  Alice and Bob must know each other's long term public keys in advance, as they are used to authenticate themselves against each other.

- A handshake key pair (called x and X for Alice, y and Y for Bob) generated randomly for each handshake

The algorithm further makes use of some arbitrary cryptographic hash and MAC functions:

- $\bar{H}$: A cryptographic hash function with an output half the length of the secret keys
- *MAC*: A message authentication code (keyed hash) function
- KDF_1: A key derivation function with an output that can be used as key for the *MAC* function
- KDF_2: A key derivation function with an output with the desired length of the shared session key

The following description of the protocol has been directly taken from [SEB09] with only minor formal changes. Upper case letter denote group elements here, lower case letter scalars; $\mathcal{G}^*$ is the subgroup generated by G and q is the cardinality of $\mathcal{G}^*$.

14.1 Protocol specification

I. The initiator Alice does the following:

   a) Choose $x \in [1, q-1]$ and compute $X = xG$.

b) Send $(\hat{A}, \hat{B}, X)$ to Bob.

II. At the receipt of $(\hat{A}, \hat{B}, X)$ Bob does the following:

a) Verify that $X \in \mathcal{G}^*$.

b) Choose $y \in [1, q-1]$, compute $Y = yG$.

c) Compute $d = \bar{H}(X, Y, \hat{A}, \hat{B})$ and $e = \bar{H}(Y, X, \hat{A}, \hat{B})$.

d) Compute $s_B = y + eb \mod q$, $\sigma_B = s_B(X + dA)$.

e) Compute $K_1 = \mathit{KDF}_1(\sigma_B, \hat{A}, \hat{B}, X, Y)$ and $t_B = \mathit{MAC}_{K_1}(\hat{B}, Y)$.

f) Send $(\hat{B}, \hat{A}, Y, t_B)$ to Alice.

III. At the receipt of $(\hat{B}, \hat{A}, Y, t_B)$ Alice does the following:

a) Verify that $Y \in \mathcal{G}^*$.

b) Compute $d = \bar{H}(X, Y, \hat{A}, \hat{B})$ and $e = \bar{H}(Y, X, \hat{A}, \hat{B})$.

c) Compute $s_A = x + da \mod q$, $\sigma_A = s_A(Y + eB)$.

d) Compute $K_1 = \mathit{KDF}_1(\sigma_A, \hat{A}, \hat{B}, X, Y)$.

e) Verify that $t_B = \mathit{MAC}_{K_1}(\hat{B}, Y)$.

f) Compute $t_A = \mathit{MAC}_{K_1}(\hat{A}, X)$.

g) Send t_A to Bob.

h) Compute $K_2 = \mathit{KDF}_2(\sigma_A, \hat{A}, \hat{B}, X, Y)$.

IV. At the receipt of t_A, Bob does the following:

a) Verify that $t_A = \mathit{MAC}_{K_1}(\hat{A}, X)$.

b) Compute $K_2 = \mathit{KDF}_2(\sigma_A, \hat{A}, \hat{B}, X, Y)$.

V. The shared session key is K_2.

The third message allows Bob to ensure that he is actually communicating with Alice before the handshake is completed and thus prevents the attack on PFS described in [Kra05] that affects all 2-message key exchange protocols.

14.2 Usage in fastd

fastd performs the FHMQV-C key exchange on the group specified in *ec25519*.

FHMQV-C makes use of several cryptographic hash and key derivation functions that are not given in the specification. fastd uses the following definitions for these functions:

$$
\begin{aligned}
d|e &= \text{SHA256}(Y|X|\hat{B}|\hat{A}) \\
K_1 = \mathit{KDF}_1(\sigma, \hat{A}, \hat{B}, X, Y) &= \text{HKDF-SHA256}(\texttt{0x00}^{32}, \sigma, \hat{A}|\hat{B}|X|Y, 32) \\
K_2 = \mathit{KDF}_2(\sigma, \hat{A}, \hat{B}, X, Y) &= \text{HKDF-SHA256}(K_1, \sigma, \hat{A}|\hat{B}|X|Y|\mathit{method}, *)
\end{aligned}
$$

where $V|W$ designates the concatenation of the binary strings V and W and

$$\text{HKDF}(\mathit{salt}, \mathit{IKM}, \mathit{info}, L) = \text{HKDF-Expand}(\text{HKDF-Extract}(\mathit{salt}, \mathit{IKM}), \mathit{info}, L)$$

See [FIPS180] (SHA256), [RFC2104] (HMAC) and [RFC5869] (HKDF) for the specifications of these algorithms.

As one can see, the calculation of d and e deviates from the FHMQV-C specification, which uses a hash function $\bar{H}$ with half-width (127 bit in the case of `ec25519-fhmqvc`) output, defining d and e as

$$d = \bar{H}(X|Y|\hat{A}|\hat{B})$$
$$e = \bar{H}(Y|X|\hat{A}|\hat{B})$$

fastd uses a single 256 bit hash SHA256$(Y|X|\hat{B}|\hat{A})$ instead and cuts it into two 128 bit pieces which are used as d and e. This optimization allows reusing the SHA256 implementation that is already used for KDF_1 and KDF_2 and saves one hash calculation.

Furthermore, starting with fastd v11 a *TLV authentication tag* protecting the whole handshake packet is used instead of the values t_A and t_B, which verify the public keys only. To generate this tag, HMAC-SHA256$(K_1, \cdot)$ is applied to a pseudo TLV record list, which is the same as the TLV record list sent in the actual handshake packet, with the exception of the *TLV authentication tag* value, which is replaced by zeros. This ensures that no part of the handshake after the initial packet has been manipulated, preventing downgrade attacks.

For the exact sequence of handshake packets see *Handshake protocol*.

14.3 Bibliography

CHAPTER 15

Ciphers

Generally, all ciphers used by fastd are stream ciphers.

This means that the cipher outputs a cipher stream indistinguishable from a random byte stream which can be used to encrypt packets of any length without a need for padding the packet site to a multiple of a block size by just XORing the cipher stream with the packet.

15.1 AES128-CTR

The Advanced Encryption Standard is a widely used, highly regarded block cipher specified in [FIPS197].

In counter mode a nonce of up to 12 bytes in concatenated with a 4 byte counter; this value is encrypted with the block cipher to compute 16 bytes of the cipher stream.

AES128 has been chosen in contrast to the stronger variants AES192 and AES256 as hardware acceleration for AES128 is more widely available on embedded hardware. Using this acceleration hardware from userspace through the alg_if interface of the Linux kernel is very complex though, so support for it has been removed from fastd again (but may still be used through OpenSSL.

One issue with the AES algorithm is that it is very hard to implement in a way that is safe against cache timing attacks (see [Ber05a] for details). Because of that fastd can make use of two different AES implementations: a very secure, but also very slow implementation from the NaCl library, and the implementations from OpenSSL (which can either use hardware acceleration like AES-NI, or a fast, but potentially insecure software implementation).

15.2 Salsa20(/12)

Salsa20 (see [Ber07]) is a state-of-the-art stream cipher which is very fast and very secure. In contrast to AES, it is easily implementable without any timing side channels.

Salsa20/12 is a variant of Salsa20 which uses only 12 instead of 20 rounds to improve performance. The Salsa20/12 has been chosen for the software profile on the eSTREAM portfolio in 2011 as it has a very high throughput while providing a very comfortable security margin.

The even more reduced variant Salsa20/8 has also been evaluated for fastd, but the performance gain has been to small to warrant the significantly reduced security.

15.3 Bibliography

CHAPTER 16

Message Authentication Codes

16.1 GHASH / Galois/Counter Mode (GCM) / GMAC

The Galois/Counter Mode is a very well-known mode of operation for block ciphers which was specified in [MV04]. GMAC is a authentication-only variant of the algorithm.

While the original specification only considers block ciphers, GCM can also be specified in terms of the Counter mode (CTR) of the block cipher. The counter mode transforms a block cipher into a stream cipher. This allows it to replace the block cipher by any stream cipher while preserving all security guarantees; thereforce fastd allows to use GMAC with any supported stream cipher.

One particular issue with GCM/GMAC is that it is hard to implement in software. Usually it is implemented using lookup table, which might exhibit cache timing side channels. This issue doesn't affect modern x86 CPUs providing the PCLMUL instruction, as PCLMUL allows performing carry-less multiplications without a lookup table.

16.2 UHASH / UMAC

The UMAC message authentication code defined in [RFC4418] is a strongly universal hash function, which is formed by defining a universal hash function UHASH and XORing it with a pad generated by a block cipher like AES.

In fastd, the pad can be generated by any supported stream cipher, and the key derivation function specified in the RFC has been replaced by HKDF.

The UHASH function is optimized for efficient implementation in software on 32bit CPUs. Therefore UMAC is much more performant than GMAC, especially on embedded systems, and doesn't exhibit any timing side channels.

16.3 Bibliography

CHAPTER 17

Method providers

See *Encryption & authentication methods* for details about the method configuration and recommendations.

17.1 generic-gmac

The *generic-gmac* provider combines the GHASH message authetication code with any stream cipher, which is used both to encrypt the data and the authentication tag.

After the last encrypted data block, a block containing the length of the data (in bits, big endian) is passed to the GHASH function as defined by the GCM specification.

The method names normally have the form "<cipher>+gmac", and "aes128-gcm" for the AES128 cipher.

17.2 composed-gmac

The *composed-gmac* provider combines the GHASH message authetication code with two stream ciphers, where the first one is used to encrypt the data and the second one for the authentication tag. As only the authentication tag must be encrypted, "null" can be used as the first cipher for authenticated-only methods.

After the last encrypted data block, a block with the first 8 bytes containing the length of the data (in bits, big endian) and the other 8 bytes set to zero is passed to the GHASH function. This differs from the size block used by the *generic-gmac* for historical reasons.

The method names normally have the form "<cipher>+<cipher>+gmac", and "<cipher>+aes128-gmac" for the AES128 cipher.

17.3 generic-umac

The *generic-umac* provider combines the UHASH message authetication code with any stream cipher, which is used both to encrypt the data and the authentication tag.

The method names have the form “<cipher>+umac”.

17.4 composed-umac

The *composed-umac* provider combines the UHASH message authetication code with two stream ciphers, where the first one is used to encrypt the data and the second one for the authentication tag. As only the authentication tag must be encrypted, “null” can be used as the first cipher for authenticated-only methods.

The method names have the form “<cipher>+<cipher>+umac”.

17.5 generic-poly1305

The *generic-umac* provider combines the Poly1305 message authentication code with any stream cipher, which is used both to encrypt the data and the authentication tag. This method was added to replace the original *xsalsa20-poly1305* method, but may be removed as well in the long term as UMAC is generally more performant and makes the same security guarantees.

The method names have the form “<cipher>+poly1305”.

17.6 null-l2tp

The “null@l2tp” method doesn’t procide any encryption or authentication. Unlike the older “null” method, this method uses the L2TPv3 (RFC3931) protocol for its data packets. This allows to use the L2TPv3 Ethernet Pseudowire implementation of the Linux kernel to offload data forwarding to kernel space in Multi-TAP mode.

17.7 null

The “null” method doesn’t provide any encryption or authentication.

17.8 cipher-test

The *cipher-test* method can be used to run a cipher without any authentication. This isn’t secure and should be used for tests and benchmarks only.

The method names have the form “<cipher>+cipher-test”.

CHAPTER 18

Building fastd

18.1 Dependencies

- libuecc (>= v6; >= v7 recommended; developed together with fastd)
- libsodium or NaCl (for most crypto methods)
- bison (>= 2.6)
- pkg-config

Optional:

- libcap (if `capabilities` is enabled; Linux only; can be disabled if you don't need POSIX capability support)
- libmnl (for L2TP offload support; Linux only)
- libjson-c (if `status_socket` is enabled)
- libssl (if `cipher_aes128-ctr` is enabled)

18.2 Building

Starting with v20, fastd uses the Meson build system.

```
# Get fastd (or use the release tarballs)
git clone https://github.com/NeoRaider/fastd.git

# Set up a build dir
meson setup fastd fastd-build -Dbuildtype=release
cd fastd-build

# Build fastd, binary can be found in the src subdir of the build dir
ninja
```

(continues on next page)

(continued from previous page)

```
# Install in the system
ninja install
```

18.3 Build settings

The build can be configured using the command `meson configure`; running it without any additional arguments will show all available variables. Settings can be passed to `meson setup` or `meson configure` using `-DVARIABLE=VALUE`.

- By default, fastd will build against libsodium. If you want to use NaCl instead, add `-Duse_nacl=true`
- If you have a recent enough toolchain (GCC 4.8 or higher recommended), you can enable link-time optimization by adding `-Db_lto=true`
- Instead of using an installed version of libmnl, it is possible to build it as part of fastd itself by setting `-Dlibmnl_builtin=true`. This is recommended for constrained targets only and not for regular Linux distributions.

CHAPTER 19

Protocol specification

19.1 Basic protocol design

fastd uses UDP as the transport protocol for its packets. UDP has been chosen instead of raw IP packets (as they are used by IPIP and 6in4 tunnels or IPsec) to simplify the deployment of multiple fastd instances on the same host using different UDP ports and allow passing through common NAT routers without explicit configuration.

The first byte of the UDP payload is used to discern the different packet types used by fastd. Since fastd v22, the following packet types are used:

- `0x00` Data packet (v22+)
- `0x01` Handshake packet (pre-v22)
- `0x02` Data packet (pre-v22)
- `0xC8` L2TP control message header (v22+)

fastd v22 still supports the pre-v22 packet types, so communication between old and new versions is possible.

19.2 L2TP control message headers

Since fastd v22, all handshake packets may be prefixed with an L2TP control message header to make sure these packets are not considered data packets by the L2TP kernel code even with potential future extensions of the L2TP protocol. The basic format of this header is the following (as specified in RFC3931):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T|L|x|x|S|x|x|x|x|x|x|x|  Ver  |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Control Connection ID                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(continues on next page)

(continued from previous page)

```
|                Ns               |                Nr               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

When sending a packet with an L2TP header, the following rules apply:

- Only the *T*, *L*, and *S* flags are set (first byte is `0xC8`)
- *Ver* is set to 3 (second byte is `0x03`)
- *Length* is set to 12 (only the header itself is counted)
- *Control Connection ID*, *Ns* and *Nr* are unused, they are set to 0

When receiving packets, only the first two bytes are verified. Packets with unexpected values in these bytes are discarded.

When replying to a handshake packet, fastd will insert the L2TP header when the peer has signaled that it supports such packets using the *L2TP_SUPPORT* flag. Initial handshakes will be sent twice at the same time, once with and once without an L2TP header, so both new and old versions of fastd can be supported.

fastd v22 and newer ignore handshake packets without L2TP header when the *L2TP_SUPPORT* flag is set, so only one of the two handshake packets with identical content will be handled.

In addition to handshakes, data packets may be prefixed with such L2TP control message headers as well, but this is rarely useful, as it reduces the usable MTU of a tunnel. The "null@l2tp" method makes use of this for keepalive packets, so they are passed up to the fastd userspace when using the L2TP kernel offload feature.

19.3 Handshake format

The first 4 bytes (after the L2TP header if it exists, of the packet otherwise) form the handshake header:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x01     |      0x00     |           TLV Length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The rest of the handshake packet consists of TLV records, the total length of which is given by the *TLV Length* header field (in big-endian byte order).

Each of the following TLV records starts with a 2-byte type field, followed by a 2-byte length field and the arbitrary-length value. There is no special alignment defined for the TLV records. All integers that are part of the TLV format (in particular, the type and length fields) are encoded in little-endian byte order.

19.3.1 TLV record types

Record ID	Value description	Format	Values
`0x0000`	Handshake type	1-byte unsigned integer	{1, 2, 3}
`0x0001`	Reply code	1-byte unsigned integer	{0 (success), 1 (mandatory record missing), 2 (unacceptable value)}
`0x0002`	Error detail	1/2-byte unsigned integer	Record type which caused an error
`0x0003`	Flags	variable-length bit field	L2TP_SUPPORT=0x01 (sender supports L2TP control message headers)
`0x0004`	Mode	1-byte unsigned integer	{0 (TAP mode), 1 (TUN mode)}
`0x0005`	Protocol name	variable-length string	"ec25519-fhmqvc"
`0x0006`	Sender key	32-byte public key	
`0x0007`	Recipient key	32-byte public key	
`0x0008`	Sender handshake key	32-byte public key	
`0x0009`	Recipient handshake key	32-byte public key	
`0x000a`	Authentication tag (obsolete)	32-byte opaque value	Not used anymore
`0x000b`	MTU	2-byte unsigned integer	
`0x000c`	Method name	variable-length string	
`0x000d`	Version name	variable-length string	
`0x000e`	Method list	zero-separated string list	
`0x000f`	TLV authentication tag	32-byte opaque value	

19.3.2 Handshake protocol

The following specification describes the current handshake as it is performed by fastd versions since v11.

The handshake protocol consists of three packets. See also: *ec25519*, *FHMQV-C*

The following fields are sent in all three packets as different fastd versions expect them in different parts of the handshake:

- Mode (TUN/TAP)
- MTU
- fastd version (e.g. `v15`)
- Protocol name (`ec25519-fhmqvc`)

Handshake request

The first packet of a handshake contains the following additional fields:

- Handshake type (0x01)
- FHMQV-C values:
  - Sender key $\hat{A}$
  - Recipient key $\hat{B}$
  - Sender handshake key X

The recipient key may be omitted if the recipient identity is unknown because the handshake was triggered by an unexpected data packet.

Handshake reply

The second packet of a handshake contains the following additional fields:

- Handshake type (0x02)
- Reply code (0x00)
- Method list (list of all supported methods)
- FHMQV-C values:
  - Sender key $\hat{B}$
  - Recipient key $\hat{A}$
  - Sender handshake key Y
  - Recipient handshake key X
  - TLV authentication tag MAC_B

Handshake finish

The second packet of a handshake contains the following additional fields:

- Handshake type (0x03)
- Reply code (0x00)
- Method (the chosen encryption/authentication scheme)
- FHMQV-C values:
  - Sender key $\hat{A}$
  - Recipient key $\hat{B}$
  - Sender handshake key X
  - Recipient handshake key Y
  - TLV authentication tag MAC_A

Handshake error

When an unacceptable handshake is received, fastd will respond with an error packet. The error packet contains the following fields:

- Handshake type (the type of the packet that is answered plus 1)
- Reply code (0x01 when a record is missing from the handshake, 0x02 when a value is unacceptable)
- Error detail (the record type ID which caused the error)

19.4 Payload packets

The payload packet structure is defined by the methods; at the moment most methods use the same format, starting with a 24 byte header, followed by the actual payload:

- Byte 1: Packet type (`0x00` when both sides of a connection are fastd v22 or newer, `0x02` otherwise)
- Byte 2: Flags (method-specific; unused, always `0x00`)
- Bytes 3-8: Packet sequence number/nonce (big endian; incremented by 2 for each packet; one side of a connection uses the even sequence numbers and the other side the odd ones)
- Bytes 9-24: Authentication tag (method-specific)

The "null" method uses only a 1-byte header: The packet type is directly followed by the payload data.

The "null@l2tp" method uses an 8-byte header, which is the same as the L2TPv3 Session Header over UDP, with no Cookie and no L2-Specific Sublayer (as specified in RFC3931). fastd always uses the L2TP Session ID 1.

Bibliography

[BBJ+08] D. J. Bernstein, P. Birkner, M. Joye, T. Lange and C. Peters, "Twisted Edwards curves", in Progress in Cryptology—AFRICACRYPT 2008, Springer, 2008, pp. 389–405.

[Ber06] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records", in Public Key Cryptography-PKC 2006, Springer, 2006, pp. 207–228.

[EFD] D. J. Bernstein and T. Lange, "Explicit-Formulas Database—Genus-1 curves over large-characteristic fields". [Online] http://hyperelliptic.org/EFD/g1p/index.html

[HWCD08] H. Hisil, K. K.-H. Wong, G. Carter and E. Dawson, "Twisted Edwards curves revisited", in Advances in Cryptology—ASIACRYPT 2008, Springer, 2008, pp. 326–343.

[VMA00] S. A. Vanstone, R. C. Mullin and G. B. Agnew, "Elliptic curve encryption systems", US Patent 6,141,420, 2000.

[FIPS180] National Institute of Standards and Technology, "Secure hash standards (SHS)", Federal Information Processing Standard 180-4, 2012. [Online] http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[Kra05] H. Krawczyk, "HMQV: a high-performance secure Diffie-Hellman protocol", Cryptology ePrint Archive, Report 2005/176, http://eprint.iacr.org/, 2005.

[LMQ+98] L. Law, A. Menezes, M. Qu, J. Solinas and S. Vanstone, "An efficient protocol for authenticated key agreement", Designs, Codes and Cryptography, vol. 28, pp. 361–377, 1998.

[LSW+14] S. Liu, K. Sakurai, J. Weng, F. Zhang, and Y. Zhao, "Security Model and Analysis of FHMQV, Revisited", in Information Security and Cryptology, pp. 255–269, Springer, 2014.

[RFC2104] H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104 (Informational), Updated by RFC 6151, Internet Engineering Task Force, 1997. [Online] http://www.ietf.org/rfc/rfc2104.txt

[RFC5869] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC5869 (Informational), Internet Engineering Task Force, 2010. [Online] http://www.ietf.org/rfc/rfc5869.txt

[SEB09] A. P. Sarr, P. Elbaz–Vincent and J. Bajard, "A secure and efficient authenticated Diffie–Hellman protocol", Cryptology ePrint Archive, Report 2009/408, http://eprint.iacr.org/, 2009.

[Ber05a] D. J. Bernstein, "Cache-timing attacks on AES", 2005. [Online] http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[Ber07] D. J. Bernstein, "The Salsa20 family of stream ciphers", 2007. [Online] http://cr.yp.to/snuffle/salsafamily-20071225.pdf

[FIPS197] National Institute of Standards and Technology, "ADVANCED ENCRYPTION STANDARD (AES)", Federal Information Processing Standard 197, 2001. [Online] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[MV04] D. McGrew and J. Viega, "The Galois/counter mode of operation (GCM)", Submission to NIST Modes of Operation Process, 2004.

[RFC4418] T. Krovetz, "UMAC: Message Authentication Code using Universal Hashing", RFC4418 (Informational), Internet Engineering Task Force, 2006. [Online] http://www.ietf.org/rfc/rfc4418.txt