## NAME

dbus-daemon - Message bus daemon

## SYNOPSIS

**dbus-daemon**

**dbus-daemon** [--version] [--session] [--system] [--config-file=*FILE*]
[--print-address [=*DESCRIPTOR*]] [--print-pid [=*DESCRIPTOR*]] [--fork]
[--nosyslog] [--syslog] [--syslog-only] [--ready-event-handle=value]

## DESCRIPTION

**dbus-daemon** is the D-Bus message bus daemon. See **http://www.freedesktop.org/software/dbus/** for more information about the big picture. D-Bus is first a library that provides one-to-one communication between any two applications; **dbus-daemon** is an application that uses this library to implement a message bus daemon. Multiple programs connect to the message bus daemon and can exchange messages with one another.

There are two standard message bus instances: the systemwide message bus (installed on many systems as the "messagebus" init service) and the per-user-login-session message bus (started each time a user logs in). **dbus-daemon** is used for both of these instances, but with a different configuration file.

The --session option is equivalent to "--config-file=/usr/local/share/dbus-1/session.conf" and the --system option is equivalent to "--config-file=/usr/local/share/dbus-1/system.conf". By creating additional configuration files and using the --config-file option, additional special-purpose message bus daemons could be created.

The systemwide daemon is normally launched by an init script, standardly called simply "messagebus".

The systemwide daemon is largely used for broadcasting system events, such as changes to the printer queue, or adding/removing devices.

The per-session daemon is used for various interprocess communication among desktop applications (however, it is not tied to X or the GUI in any way).

SIGHUP will cause the D-Bus daemon to PARTIALLY reload its configuration file and to flush its user/group information caches. Some configuration changes would require kicking all apps off the bus; so they will only take effect if you restart the daemon. Policy changes should take effect with SIGHUP.

## OPTIONS

The following options are supported:

**--config-file=FILE**
Use the given configuration file.

**--fork**
Force the message bus to fork and become a daemon, even if the configuration file does not specify that it should. In most contexts the configuration file already gets this right, though. This option is not supported on Windows.

**--nofork**
Force the message bus not to fork and become a daemon, even if the configuration file specifies that it should. On Windows, the dbus-daemon never forks, so this option is allowed but does nothing.

**--print-address[=DESCRIPTOR]**
Print the address of the message bus to standard output, or to the given file descriptor. This is used by programs that launch the message bus.

**--print-pid[=DESCRIPTOR]**
Print the process ID of the message bus to standard output, or to the given file descriptor. This is used by programs that launch the message bus.

**--session**
Use the standard configuration file for the per-login-session message bus.

**--system**
Use the standard configuration file for the systemwide message bus.

**--version**
Print the version of the daemon.

**--introspect**
Print the introspection information for all D-Bus internal interfaces.

**--address[=ADDRESS]**
Set the address to listen on. This option overrides the address configured in the configuration file via the <listen> directive. See the documentation of that directive for more details.

**--systemd-activation**

Enable systemd-style service activation. Only useful in conjunction with the systemd system and session manager on Linux.

**--nopidfile**

Don't write a PID file even if one is configured in the configuration files.

**--syslog**

Force the message bus to use the system log for messages, in addition to writing to standard error, even if the configuration file does not specify that it should. On Unix, this uses the syslog; on Windows, this uses OutputDebugString().

**--syslog-only**

Force the message bus to use the system log for messages, and *not* duplicate them to standard error. On Unix, this uses the syslog; on Windows, this uses OutputDebugString().

**--nosyslog**

Force the message bus to use only standard error for messages, even if the configuration file specifies that it should use the system log.

**--ready-event-handle=value**

With this option, the dbus daemon raises an event when it is ready to process connections. The *handle* must be the Windows handle for an event object, in the format printed by the **printf** format string %p. The parent process must create this event object (for example with the **CreateEvent** function) in a nonsignaled state, then configure it to be inherited by the dbus-daemon process. The dbus-daemon will signal the event as if via **SetEvent** when it is ready to receive connections from clients. The parent process can wait for this to occur by using functions such as **WaitForSingleObject**. This option is only supported under Windows. On Unix platforms, a similar result can be achieved by waiting for the address and/or process ID to be printed to the inherited file descriptors used for **--print-address** and/or **--print-pid**.

## CONFIGURATION FILE

A message bus daemon has a configuration file that specializes it for a particular application. For example, one configuration file might set up the message bus to be a systemwide message bus, while another might set it up to be a per-user-login-session bus.

The configuration file also establishes resource limits, security parameters, and so forth.

The configuration file is not part of any interoperability specification and its backward compatibility is not guaranteed; this document is documentation, not specification.

The standard systemwide and per-session message bus setups are configured in the files "/usr/local/share/dbus-1/system.conf" and "/usr/local/share/dbus-1/session.conf". These files normally <include> a system-local.conf or session-local.conf in /usr/local/etc/dbus-1; you can put local overrides in those files to avoid modifying the primary configuration files.

The standard system bus normally reads additional XML files from /usr/local/share/dbus-1/system.d. Third-party packages should install the default policies necessary for correct operation into that directory, which has been supported since dbus 1.10 (released in 2015).

The standard system bus normally also reads XML files from /usr/local/etc/dbus-1/system.d, which should be used by system administrators if they wish to override default policies.

Third-party packages would historically install XML files into /usr/local/etc/dbus-1/system.d, but this practice is now considered to be deprecated: that directory should be treated as reserved for the system administrator.

The configuration file is an XML document. It must have the following doctype declaration:

```
<!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-Bus Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
```

The following elements may be present in the configuration file.

Root element.

The well-known type of the message bus. Currently known values are "system" and "session"; if other values are set, they should be either added to the D-Bus specification, or namespaced. The last <type> element "wins" (previous values are ignored). This element only controls which message bus specific environment variables are set in activated clients. Most of the policy that distinguishes a session bus from the system bus is controlled from the other elements in the configuration file.

If the well-known type of the message bus is "session", then the DBUS_STARTER_BUS_TYPE environment variable will be set to "session" and the DBUS_SESSION_BUS_ADDRESS environment variable will be set to the address of the session bus. Likewise, if the type of the message bus is

"system", then the DBUS_STARTER_BUS_TYPE environment variable will be set to "system" and the DBUS_SYSTEM_BUS_ADDRESS environment variable will be set to the address of the system bus (which is normally well known anyway).

Example: <type>session</type>

- 

Include a file <include>filename.conf</include> at this point. If the filename is relative, it is located relative to the configuration file doing the including.

<include> has an optional attribute "ignore_missing=(yes|no)" which defaults to "no" if not provided. This attribute controls whether it’s a fatal error for the included file to be absent.

- 

Include all files in <includedir>foo.d</includedir> at this point. Files in the directory are included in undefined order. Only files ending in ".conf" are included.

This is intended to allow extension of the system bus by particular packages. For example, if CUPS wants to be able to send out notification of printer queue changes, it could install a file to /usr/local/share/dbus-1/system.d that allowed all apps to receive this message and allowed the printer daemon user to send it.

- 

The user account the daemon should run as, as either a username or a UID. If the daemon cannot change to this UID on startup, it will exit. If this element is not present, the daemon will not change or care about its UID.

The last <user> entry in the file "wins", the others are ignored.

The user is changed after the bus has completed initialization. So sockets etc. will be created before changing user, but no data will be read from clients before changing user. This means that sockets and PID files can be created in a location that requires root privileges for writing.

- 

If present, the bus daemon becomes a real daemon (forks into the background, etc.). This is generally used rather than the --fork command line option.

If present, the bus daemon keeps its original umask when forking. This may be useful to avoid affecting the behavior of child processes.

If present, the bus daemon will log to syslog. The --syslog, --syslog-only and --nosyslog command-line options take precedence over this setting.

If present, the bus daemon will write its pid to the specified file. The --nopidfile command-line option takes precedence over this setting.

If present, connections that authenticated using the ANONYMOUS mechanism will be authorized to connect. This option has no practical effect unless the ANONYMOUS mechanism has also been enabled using the <*auth*> element, described below.

Using this directive in the configuration of the well-known system bus or the well-known session bus will make that bus insecure and should never be done. Similarly, on custom bus types, using this directive will usually make the custom bus insecure, unless its configuration has been specifically designed to prevent anonymous users from causing damage or escalating privileges.

Add an address that the bus should listen on. The address is in the standard D-Bus format that contains a transport name plus possible parameters/options.

On platforms other than Windows, unix-based transports (unix, systemd, launchd) are the default for both the well-known system bus and the well-known session bus, and are strongly recommended.

On Windows, unix-based transports are not available, so TCP-based transports must be used. Similar to remote X11, the tcp and nonce-tcp transports have no integrity or confidentiality protection, so they should normally only be used across the local loopback interface, for example using an address like tcp:host=127.0.0.1 or nonce-tcp:host=localhost. In particular, configuring the well-known system bus or the well-known session bus to listen on a non-loopback TCP address is insecure.

Developers are sometimes tempted to use remote TCP as a debugging tool. However, if this functionality is left enabled in finished products, the result will be dangerously insecure. Instead of using remote TCP, developers should **relay connections via Secure Shell or a similar protocol**[1].

Remote TCP connections were historically sometimes used to share a single session bus between login sessions of the same user on different machines within a trusted local area network, in conjunction with unencrypted remote X11, a NFS-shared home directory and NIS (YP) authentication. This is insecure against an attacker on the same LAN and should be considered strongly deprecated; more specifically, it is insecure in the same ways and for the same reasons as unencrypted remote X11 and NFSv2/NFSv3. The D-Bus maintainers recommend using a separate session bus per (user, machine) pair, only accessible from within that machine.

Example: <listen>unix:path=/tmp/foo</listen>

Example: <listen>tcp:host=localhost,port=1234</listen>

If there are multiple <listen> elements, then the bus listens on multiple addresses. The bus will pass its address to started services or other interested parties with the last address given in <listen> first. That is, apps will try to connect to the last <listen> address first.

tcp sockets can accept IPv4 addresses, IPv6 addresses or hostnames. If a hostname resolves to multiple addresses, the server will bind to all of them. The family=ipv4 or family=ipv6 options can be used to force it to bind to a subset of addresses

Example: <listen>tcp:host=localhost,port=0,family=ipv4</listen>

A special case is using a port number of zero (or omitting the port), which means to choose an available port selected by the operating system. The port number chosen can be obtained with the --print-address command line parameter and will be present in other cases where the server reports its own address, such as when DBUS_SESSION_BUS_ADDRESS is set.

Example: <listen>tcp:host=localhost,port=0</listen>

tcp/nonce-tcp addresses also allow a bind=hostname option, used in a listenable address to configure the interface on which the server will listen: either the hostname is the IP address of one of the local machine’s interfaces (most commonly 127.0.0.1), a DNS name that resolves to one of those IP addresses, ’0.0.0.0’ to listen on all IPv4 interfaces simultaneously, or ’::’ to listen on all IPv4 and IPv6 interfaces simultaneously (if supported by the OS). If not specified, the default is the same value as "host".

Example: <listen>tcp:host=localhost,bind=0.0.0.0,port=0</listen>

Lists permitted authorization mechanisms. If this element doesn’t exist, then all known mechanisms are allowed. If there are multiple <auth> elements, all the listed mechanisms are allowed. The order in which mechanisms are listed is not meaningful.

On non-Windows operating systems, allowing only the EXTERNAL authentication mechanism is strongly recommended. This is the default for the well-known system bus and for the well-known session bus.

Example: <auth>EXTERNAL</auth>

Example: <auth>DBUS_COOKIE_SHA1</auth>

Adds a directory to search for .service files, which tell the dbus-daemon how to start a program to provide a particular well-known bus name. See the D-Bus Specification for more details about the contents of .service files.

If a particular service is found in more than one <servicedir>, the first directory listed in the configuration file takes precedence. If two service files providing the same well-known bus name are found in the same directory, it is arbitrary which one will be chosen (this can only happen if at least one of the service files does not have the recommended name, which is its well-known bus name followed by ".service").

<standard_session_servicedirs/> requests a standard set of session service directories. Its effect is similar to specifying a series of <servicedir/> elements for each of the data directories, in the order given here. It is not exactly equivalent, because there is currently no way to disable directory monitoring or enforce strict service file naming for a <servicedir/>.

As with <servicedir/> elements, if a particular service is found in more than one service directory, the first directory takes precedence. If two service files providing the same well-known bus name are found in the same directory, it is arbitrary which one will be chosen (this can only happen if at least one of the service files does not have the recommended name, which is its well-known bus name followed by ".service").

On Unix, the standard session service directories are:

- if XDG_RUNTIME_DIR is set (see the XDG Base Directory Specification for details of XDG_RUNTIME_DIR): this location is suitable for transient services created at runtime by systemd generators (see **systemd.generator**(7)), session managers or other session infrastructure. It is an extension provided by the reference implementation of dbus-daemon, and is not standardized in the D-Bus Specification.

  Unlike the other standard session service directories, this directory enforces strict naming for the service files: the filename must be exactly the well-known bus name of the service, followed by ".service".

  Also unlike the other standard session service directories, this directory is never monitored with **inotify**(7) or similar APIs. Programs that create service files in this directory while a dbus-daemon is running are expected to call the dbus-daemon's ReloadConfig() method after they have made changes.

- where XDG_DATA_HOME defaults to ~/.local/share (see the XDG Base Directory Specification): this location is specified by the D-Bus Specification, and is suitable for per-user, locally-installed software.

- for each directory in XDG_DATA_DIRS, where XDG_DATA_DIRS defaults to /usr/local/share:/usr/share (see the XDG Base Directory Specification): these locations are specified by the D-Bus Specification. The defaults are suitable for software installed locally by a system administrator (/usr/local/share) or for software installed from operating system packages (/usr/share). Per-user or system-wide configuration that sets the XDG_DATA_DIRS environment variable can extend this search path to cover installations in other locations, for example ~/.local/share/flatpak/exports/share/ and /var/lib/flatpak/exports/share/ when **flatpak**(1) is used.

- for the *${datadir}* that was specified when dbus was compiled, typically /usr/share: this location is an extension provided by the reference dbus-daemon implementation, and is suitable for software stacks installed alongside dbus-daemon.

The "XDG Base Directory Specification" can be found at **http://freedesktop.org/wiki/Standards/basedir-spec** if it hasn't moved, otherwise try your favorite search engine.

On Windows, the standard session service directories are:

- if %CommonProgramFiles% is set: this location is suitable for system-wide installed software packages

- share/dbus-1/services directory found in the same directory hierarchy (prefix) as the dbus-daemon: this location is suitable for software stacks installed alongside dbus-daemon

The <standard_session_servicedirs/> option is only relevant to the per-user-session bus daemon defined in /usr/local/etc/dbus-1/session.conf. Putting it in any other configuration file would probably be nonsense.

-

<standard_system_servicedirs/> specifies the standard system-wide activation directories that should be searched for service files. As with session services, the first directory listed has highest precedence.

On Unix, the standard system service directories are:

- this location is specified by the D-Bus Specification, and is suitable for software installed locally by the system administrator

- this location is specified by the D-Bus Specification, and is suitable for software installed by operating system packages

- for the *${datadir}* that was specified when dbus was compiled, typically /usr/share: this location is an extension provided by the reference dbus-daemon implementation, and is suitable for software stacks installed alongside dbus-daemon

- this location is specified by the D-Bus Specification, and was intended for software installed by operating system packages and used during early boot (but it should be considered deprecated, because the reference dbus-daemon is not designed to be available during early boot)

On Windows, there is no standard system bus, so there are no standard system bus directories either.

The <standard_system_servicedirs/> option is only relevant to the per-system bus daemon defined in /usr/local/share/dbus-1/system.conf. Putting it in any other configuration file would probably be

nonsense.

- <servicehelper/> specifies the setuid helper that is used to launch system daemons with an alternate user. Typically this should be the dbus-daemon-launch-helper executable in located in libexec.

  The <servicehelper/> option is only relevant to the per-system bus daemon defined in /usr/local/share/dbus-1/system.conf. Putting it in any other configuration file would probably be nonsense.

- <limit> establishes a resource limit. For example:

  ```
  <limit name="max_message_size">64</limit>
  <limit name="max_completed_connections">512</limit>
  ```

  The name attribute is mandatory. Available limit names are:

  ```
  "max_incoming_bytes"        : total size in bytes of messages
                                incoming from a single connection
  "max_incoming_unix_fds"     : total number of unix fds of messages
                                incoming from a single connection
  "max_outgoing_bytes"        : total size in bytes of messages
                                queued up for a single connection
  "max_outgoing_unix_fds"     : total number of unix fds of messages
                                queued up for a single connection
  "max_message_size"          : max size of a single message in
                                bytes
  "max_message_unix_fds"      : max unix fds of a single message
  "service_start_timeout"     : milliseconds (thousandths) until
                                a started service has to connect
  "auth_timeout"              : milliseconds (thousandths) a
                                connection is given to
                                authenticate
  "pending_fd_timeout"        : milliseconds (thousandths) a
                                fd is given to be transmitted to
                                dbus-daemon before disconnecting the
                                connection
  ```

"max_completed_connections" : max number of authenticated connections
"max_incomplete_connections" : max number of unauthenticated connections
"max_connections_per_user" : max number of completed connections from the same user (only enforced on Unix OSs)
"max_pending_service_starts" : max number of service launches in progress at the same time
"max_names_per_connection" : max number of names a single connection can own
"max_match_rules_per_connection": max number of match rules for a single connection
"max_replies_per_connection" : max number of pending method replies per connection (number of calls-in-progress)
"reply_timeout" : milliseconds (thousandths) until a method call times out

The max incoming/outgoing queue sizes allow a new message to be queued if one byte remains below the max. So you can in fact exceed the max by max_message_size.

max_completed_connections divided by max_connections_per_user is the number of users that can work together to denial-of-service all other users by using up all connections on the systemwide bus.

Limits are normally only of interest on the systemwide bus, not the user session buses.

The <policy> element defines a security policy to be applied to a particular set of connections to the bus. A policy is made up of <allow> and <deny> elements. Policies are normally used with the systemwide bus; they are analogous to a firewall in that they allow expected traffic and prevent unexpected traffic.

Currently, the system bus has a default-deny policy for sending method calls and owning bus names, and a default-allow policy for receiving messages, sending signals, and sending a single success or error reply for each method call that does not have the NO_REPLY flag. Sending more than the expected number of replies is not allowed.

In general, it is best to keep system services as small, targeted programs which run in their own process and provide a single bus name. Then, all that is needed is an <allow> rule for the "own" permission to let the process claim the bus name, and a "send_destination" rule to allow traffic from some or all uids

to your service.

The <policy> element has one of four attributes:

context="(default|mandatory)"
at_console="(true|false)"
user="username or userid"
group="group name or gid"

Policies are applied to a connection as follows:

- all context="default" policies are applied
- all group="connection's user's group" policies are applied in undefined order
- all user="connection's auth user" policies are applied in undefined order
- all at_console="true" policies are applied
- all at_console="false" policies are applied
- all context="mandatory" policies are applied

Policies applied later will override those applied earlier, when the policies overlap. Multiple policies with the same user/group/context are applied in the order they appear in the config file.

*<deny>*
*<allow>*

A <deny> element appears below a <policy> element and prohibits some action. The <allow> element makes an exception to previous <deny> statements, and works just like <deny> but with the inverse meaning.

The possible attributes of these elements are:

send_interface="interface_name" | "*"
send_member="method_or_signal_name" | "*"
send_error="error_name" | "*"
send_broadcast="true" | "false"
send_destination="name" | "*"
send_destination_prefix="name"
send_type="method_call" | "method_return" | "signal" | "error" | "*"
send_path="/path/name" | "*"

receive_interface="interface_name" | "*"
receive_member="method_or_signal_name" | "*"
receive_error="error_name" | "*"
receive_sender="name" | "*"
receive_type="method_call" | "method_return" | "signal" | "error" | "*"
receive_path="/path/name" | "*"

send_requested_reply="true" | "false"
receive_requested_reply="true" | "false"

eavesdrop="true" | "false"

own="name" | "*"
own_prefix="name"
user="username" | "*"
group="groupname" | "*"

Examples:

<deny send_destination="org.freedesktop.Service" send_interface="org.freedesktop.System" send_member="Re
<deny send_destination="org.freedesktop.System"/>
<deny receive_sender="org.freedesktop.System"/>
<deny user="john"/>
<deny group="enemies"/>

The <deny> element’s attributes determine whether the deny "matches" a particular action. If it matches, the action is denied (unless later rules in the config file allow it).

Rules with one or more of the send_* family of attributes are checked in order when a connection attempts to send a message. The last rule that matches the message determines whether it may be sent. The well-known session bus normally allows sending any message. The well-known system bus normally allows sending any signal, selected method calls to the **dbus-daemon**, and exactly one reply to each previously-sent method call (either success or an error). Either of these can be overridden by configuration; on the system bus, services that will receive method calls must install configuration that allows them to do so, usually via rules of the form <policy context="default"><allow send_destination="..."/><policy>.

Rules with one or more of the receive_* family of attributes, or with the eavesdrop attribute and no others, are checked for each recipient of a message (there might be more than one recipient if the message is a broadcast or a connection is eavesdropping). The last rule that matches the message

determines whether it may be received. The well-known session bus normally allows receiving any message, including eavesdropping. The well-known system bus normally allows receiving any message that was not eavesdropped (any unicast message addressed to the recipient, and any broadcast message).

The eavesdrop, min_fds and max_fds attributes are modifiers that can be applied to either send_* or receive_* rules, and are documented below.

send_destination and receive_sender rules mean that messages may not be sent to or received from the *owner* of the given name, not that they may not be sent *to that name*. That is, if a connection owns services A, B, C, and sending to A is denied, sending to B or C will not work either. As a special case, send_destination="*" matches any message (whether it has a destination specified or not), and receive_sender="*" similarly matches any message.

A send_destination_prefix rule opens or closes the whole namespace for sending. It means that messages may or may not be sent to the *owner* of any name matching the prefix, regardless of whether it is the primary or the queued owner. In other words, for <allow send_destination_prefix="a.b"/> rule and names "a.b", "a.b.c", and "a.b.c.d" present on the bus, it works the same as if three separate rules: <allow send_destination="a.b"/>, <allow send_destination="a.b.c"/>, and <allow send_destination="a.b.c.d"/> had been defined. The rules for matching names are the same as in own_prefix (see below): a prefix of "a.b" matches names "a.b" or "a.b.c" or "a.b.c.d", but not "a.bc" or "a.c". The send_destination_prefix attribute cannot be combined with the send_destination attribute in the same rule.

Rules with send_broadcast="true" match signal messages with no destination (broadcasts). Rules with send_broadcast="false" are the inverse: they match any unicast destination (unicast signals, together with all method calls, replies and errors) but do not match messages with no destination (broadcasts). This is not the same as send_destination="*", which matches any sent message, regardless of whether it has a destination or not.

The other send_* and receive_* attributes are purely textual/by-value matches against the given field in the message header, except that for the attributes where it is allowed, * matches any message (whether it has the relevant header field or not). For example, send_interface="*" matches any sent message, even if it does not contain an interface header field. More complex glob matching such as foo.bar.* is not allowed.

"Eavesdropping" occurs when an application receives a message that was explicitly addressed to a name the application does not own, or is a reply to such a message. Eavesdropping thus only applies to messages that are addressed to services and replies to such messages (i.e. it does not apply to signals).

For <allow>, eavesdrop="true" indicates that the rule matches even when eavesdropping. eavesdrop="false" is the default and means that the rule only allows messages to go to their specified recipient. For <deny>, eavesdrop="true" indicates that the rule matches only when eavesdropping. eavesdrop="false" is the default for <deny> also, but here it means that the rule applies always, even when not eavesdropping. The eavesdrop attribute can only be combined with send and receive rules (with send_* and receive_* attributes).

The [send|receive]_requested_reply attribute works similarly to the eavesdrop attribute. It controls whether the <deny> or <allow> matches a reply that is expected (corresponds to a previous method call message). This attribute only makes sense for reply messages (errors and method returns), and is ignored for other message types.

For <allow>, [send|receive]_requested_reply="true" is the default and indicates that only requested replies are allowed by the rule. [send|receive]_requested_reply="false" means that the rule allows any reply even if unexpected.

For <deny>, [send|receive]_requested_reply="false" is the default but indicates that the rule matches only when the reply was not requested. [send|receive]_requested_reply="true" indicates that the rule applies always, regardless of pending reply state.

The min_fds and max_fds attributes modify either send_* or receive_* rules. A rule with the min_fds attribute only matches messages if they have at least that many Unix file descriptors attached. Conversely, a rule with the max_fds attribute only matches messages if they have no more than that many file descriptors attached. In practice, rules with these attributes will most commonly take the form <allow send_destination="..." max_fds="0"/>, <deny send_destination="..." min_fds="1"/> or <deny receive_sender="*" min_fds="1"/>.

Rules with the user or group attribute are checked when a new connection to the message bus is established, and control whether the connection can continue. Each of these attributes cannot be combined with any other attribute. As a special case, both user="*" and group="*" match any connection. If there are no rules of this form, the default is to allow connections from the same user ID that owns the **dbus-daemon** process. The well-known session bus normally uses that default behaviour, while the well-known system bus normally allows any connection.

Rules with the own or own_prefix attribute are checked when a connection attempts to own a well-known bus names. As a special case, own="*" matches any well-known bus name. The well-known session bus normally allows any connection to own any name, while the well-known system bus normally does not allow any connection to own any name, except where allowed by further configuration. System services that will own a name must install configuration that allows them to do so, usually via rules of the form <policy user="some-system-user"><allow own="..."/></policy>.

<allow own_prefix="a.b"/> allows you to own the name "a.b" or any name whose first dot-separated elements are "a.b": in particular, you can own "a.b.c" or "a.b.c.d", but not "a.bc" or "a.c". This is useful when services like Telepathy and ReserveDevice define a meaning for subtrees of well-known names, such as org.freedesktop.Telepathy.ConnectionManager.(anything) and org.freedesktop.ReserveDevice1.(anything).

It does not make sense to deny a user or group inside a <policy> for a user or group; user/group denials can only be inside context="default" or context="mandatory" policies.

A single <deny> rule may specify combinations of attributes such as send_destination and send_interface and send_type. In this case, the denial applies only if both attributes match the message being denied. e.g. <deny send_interface="foo.bar" send_destination="foo.blah"/> would deny messages with the given interface AND the given bus name. To get an OR effect you specify multiple <deny> rules.

You can’t include both send_ and receive_ attributes on the same rule, since "whether the message can be sent" and "whether it can be received" are evaluated separately.

Be careful with send_interface/receive_interface, because the interface field in messages is optional. In particular, do NOT specify <deny send_interface="org.foo.Bar"/>! This will cause no-interface messages to be blocked for all services, which is almost certainly not what you intended. Always use rules of the form: <deny send_interface="org.foo.Bar" send_destination="org.foo.Service"/>

- 

The <selinux> element contains settings related to Security Enhanced Linux. More details below.

- 

An <associate> element appears below an <selinux> element and creates a mapping. Right now only one kind of association is possible:

```
<associate own="org.freedesktop.Foobar" context="foo_t"/>
```

This means that if a connection asks to own the name "org.freedesktop.Foobar" then the source context will be the context of the connection and the target context will be "foo_t" - see the short discussion of SELinux below.

Note, the context here is the target context when requesting a name, NOT the context of the connection owning the name.

There’s currently no way to set a default for owning any name, if we add this syntax it will look like:

<associate own="*" context="foo_t"/>

If you find a reason this is useful, let the developers know. Right now the default will be the security context of the bus itself.

If two <associate> elements specify the same name, the element appearing later in the configuration file will be used.

The <apparmor> element is used to configure AppArmor mediation on the bus. It can contain one attribute that specifies the mediation mode:

<apparmor mode="(enabled|disabled|required)"/>

The default mode is "enabled". In "enabled" mode, AppArmor mediation will be performed if AppArmor support is available in the kernel. If it is not available, dbus-daemon will start but AppArmor mediation will not occur. In "disabled" mode, AppArmor mediation is disabled. In "required" mode, AppArmor mediation will be enabled if AppArmor support is available, otherwise dbus-daemon will refuse to start.

The AppArmor mediation mode of the bus cannot be changed after the bus starts. Modifying the mode in the configuration file and sending a SIGHUP signal to the daemon has no effect on the mediation mode.

## INTEGRATING SESSION SERVICES

Integration files are not mandatory for session services: any program with access to the session bus can request a well-known name and provide D-Bus interfaces.

Many D-Bus session services support service activation, a mechanism in which the **dbus-daemon** can launch the service on-demand, either by running the session service itself or by communicating with **systemd --user**. This is set up by creating a service file in the directory *${datadir}*/dbus-1/services, for example:

```
[D-BUS Service]
Name=com.example.SessionService1
Exec=/usr/bin/example-session-service
# Optional
```

SystemdService=*example-session-service*

See the **D-Bus Specification**[2] for details of the contents and interpretation of service files.

If there is a service file for *com.example.SessionService1*, it should be named *com.example.SessionService1*.service, although for compatibility with legacy services this is not mandatory.

Session services that declare the optional SystemdService must also provide a systemd user service unit file whose name or Alias matches the SystemdService (see **systemd.unit**(5), **systemd.service**(5) for further details on systemd service units), for example:

```
[Unit]
Description=Example session service

[Service]
Type=dbus
BusName=com.example.SessionService1
ExecStart=/usr/bin/example-session-service
```

## INTEGRATING SYSTEM SERVICES

The standard system bus does not allow method calls or owning well-known bus names by default, so a useful D-Bus system service will normally need to configure a default security policy that allows it to work. D-Bus system services should install a default policy file in *${datadir}*/dbus-1/service.d, containing the policy rules necessary to make that system service functional. A best-practice policy file will often look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE busconfig PUBLIC
 "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
  <policy user="_example">
    <allow own="com.example.Example1"/>
  </policy>

  <policy context="default">
    <allow send_destination="com.example.Example1"/>
  </policy>
```

```
</busconfig>
```

where *_example* is the username of the system uid that will run the system service daemon process, and *com.example.Example1* is its well-known bus name.

The policy file for *com.example.Example1* should normally be named *com.example.Example1*.conf.

Some existing system services rely on more complex <policy> rules to control the messages that the service can receive. However, the **dbus-daemon**'s policy language is not well-suited to finer-grained policies: any policy has to be expressed in terms of D-Bus interfaces and method names, not in terms of higher-level domain-specific concepts like removable or built-in devices. It is recommended that new services should normally accept method call messages from all callers, then apply a sysadmin-controllable policy to decide whether to obey the requests contained in those method call messages, for example by consulting **polkit**[3].

Like session services, many D-Bus system services support service activation, a mechanism in which the **dbus-daemon** can launch the service on-demand, either by running the system service itself or by communicating with **systemd**. This is set up by creating a service file in the directory *${datadir}*/dbus-1/system-services, for example:

```
[D-BUS Service]
Name=com.example.Example1
Exec=/usr/sbin/example-service
User=_example
# Optional
SystemdService=dbus-com.example.Example1.service
```

See the **D-Bus Specification**[2] for details of the contents and interpretation of service files.

If there is a service file for *com.example.Example1*, it must be named *com.example.Example1*.service.

System services that declare the optional SystemdService must also provide a systemd service unit file whose name or Alias matches the SystemdService (see **systemd.unit**(5), **systemd.service**(5) for further details on systemd service units), for example:

```
[Unit]
Description=Example service

[Service]
Type=dbus
```

BusName=*com.example.Example1*
ExecStart=*/usr/sbin/example-service*

[Install]
WantedBy=multi-user.target
Alias=dbus-*com.example.Example1*.service

**SELINUX**

See **http://www.nsa.gov/selinux/** for full details on SELinux. Some useful excerpts:

Every subject (process) and object (e.g. file, socket, IPC object, etc) in the system is assigned a collection of security attributes, known as a security context. A security context contains all of the security attributes associated with a particular subject or object that are relevant to the security policy.

In order to better encapsulate security contexts and to provide greater efficiency, the policy enforcement code of SELinux typically handles security identifiers (SIDs) rather than security contexts. A SID is an integer that is mapped by the security server to a security context at runtime.

When a security decision is required, the policy enforcement code passes a pair of SIDs (typically the SID of a subject and the SID of an object, but sometimes a pair of subject SIDs or a pair of object SIDs), and an object security class to the security server. The object security class indicates the kind of object, e.g. a process, a regular file, a directory, a TCP socket, etc.

Access decisions specify whether or not a permission is granted for a given pair of SIDs and class. Each object class has a set of associated permissions defined to control operations on objects with that class.

D-Bus performs SELinux security checks in two places.

First, any time a message is routed from one connection to another connection, the bus daemon will check permissions with the security context of the first connection as source, security context of the second connection as target, object class "dbus" and requested permission "send_msg".

If a security context is not available for a connection (impossible when using UNIX domain sockets), then the target context used is the context of the bus daemon itself. There is currently no way to change this default, because we’re assuming that only UNIX domain sockets will be used to connect to the systemwide bus. If this changes, we’ll probably add a way to set the default connection context.

Second, any time a connection asks to own a name, the bus daemon will check permissions with the

security context of the connection as source, the security context specified for the name in the config file as target, object class "dbus" and requested permission "acquire_svc".

The security context for a bus name is specified with the <associate> element described earlier in this document. If a name has no security context associated in the configuration file, the security context of the bus daemon itself will be used.

**APPARMOR**

The AppArmor confinement context is stored when applications connect to the bus. The confinement context consists of a label and a confinement mode. When a security decision is required, the daemon uses the confinement context to query the AppArmor policy to determine if the action should be allowed or denied and if the action should be audited.

The daemon performs AppArmor security checks in three places.

First, any time a message is routed from one connection to another connection, the bus daemon will check permissions with the label of the first connection as source, label and/or connection name of the second connection as target, along with the bus name, the path name, the interface name, and the member name. Reply messages, such as method_return and error messages, are implicitly allowed if they are in response to a message that has already been allowed.

Second, any time a connection asks to own a name, the bus daemon will check permissions with the label of the connection as source, the requested name as target, along with the bus name.

Third, any time a connection attempts to eavesdrop, the bus daemon will check permissions with the label of the connection as the source, along with the bus name.

AppArmor rules for bus mediation are not stored in the bus configuration files. They are stored in the application’s AppArmor profile. Please see *apparmor.d(5)* for more details.

**DEBUGGING**

If you’re trying to figure out where your messages are going or why you aren’t getting messages, there are several things you can try.

Remember that the system bus is heavily locked down and if you haven’t installed a security policy file to allow your message through, it won’t work. For the session bus, this is not a concern.

The simplest way to figure out what’s happening on the bus is to run the *dbus-monitor* program, which comes with the D-Bus package. You can also send test messages with *dbus-send*. These programs have their own man pages.

If you want to know what the daemon itself is doing, you might consider running a separate copy of the daemon to test against. This will allow you to put the daemon under a debugger, or run it with verbose output, without messing up your real session and system daemons.

To run a separate test copy of the daemon, for example you might open a terminal and type:

```
DBUS_VERBOSE=1 dbus-daemon --session --print-address
```

The test daemon address will be printed when the daemon starts. You will need to copy-and-paste this address and use it as the value of the DBUS_SESSION_BUS_ADDRESS environment variable when you launch the applications you want to test. This will cause those applications to connect to your test bus instead of the DBUS_SESSION_BUS_ADDRESS of your real session bus.

DBUS_VERBOSE=1 will have NO EFFECT unless your copy of D-Bus was compiled with verbose mode enabled. This is not recommended in production builds due to performance impact. You may need to rebuild D-Bus if your copy was not built with debugging in mind. (DBUS_VERBOSE also affects the D-Bus library and thus applications using D-Bus; it may be useful to see verbose output on both the client side and from the daemon.)

If you want to get fancy, you can create a custom bus configuration for your test bus (see the session.conf and system.conf files that define the two default configurations for example). This would allow you to specify a different directory for .service files, for example.

## AUTHOR

See **http://www.freedesktop.org/software/dbus/doc/AUTHORS**

## BUGS

Please send bug reports to the D-Bus mailing list or bug tracker, see **http://www.freedesktop.org/software/dbus/**

## NOTES

1. relay connections via Secure Shell or a similar protocol
   https://lists.freedesktop.org/archives/dbus/2018-April/017447.html

2. D-Bus Specification
   https://dbus.freedesktop.org/doc/dbus-specification.html

3. polkit
   https://www.freedesktop.org/wiki/Software/polkit/