NAME

systemd.exec - Execution environment configuration

SYNOPSIS

service.service, *socket*.socket, *mount*.mount, *swap*.swap

DESCRIPTION

Unit configuration files for services, sockets, mount points, and swap devices share a subset of configuration options which define the execution environment of spawned processes.

This man page lists the configuration options shared by these four unit types. See **systemd.unit(5)** for the common options of all unit configuration files, and **systemd.service(5)**, **systemd.socket(5)**, **systemd.swap(5)**, and **systemd.mount(5)** for more information on the specific unit configuration files. The execution specific configuration options are configured in the [Service], [Socket], [Mount], or [Swap] sections, depending on the unit type.

In addition, options which control resources through Linux Control Groups (cgroups) are listed in **systemd.resource-control(5)**. Those options complement options listed here.

AUTOMATIC DEPENDENCIES

A few execution parameters result in additional, automatic dependencies to be added.

Units with *WorkingDirectory=* or *RootDirectory=* set automatically gain dependencies of type *Requires=* and *After=* on all mount units required to access the specified paths. This is equivalent to having them listed explicitly in *RequiresMountsFor=*.

Similar, units with *PrivateTmp=* enabled automatically get mount unit dependencies for all mounts required to access /tmp and /var/tmp.

Units whose standard output or error output is connected to **journal**, **syslog** or **kmsg** (or their combinations with console output, see below) automatically acquire dependencies of type *After=* on systemd-journald.socket.

OPTIONS

WorkingDirectory=

Takes a directory path relative to the service's root directory specified by *RootDirectory=*, or the special value "~". Sets the working directory for executed processes. If set to "~", the home directory of the user specified in *User=* is used. If not set, defaults to the root directory when systemd is running as a system instance and the respective user's home directory if run as user. If the setting is prefixed with the "-" character, a missing working directory is not considered fatal. If *RootDirectory=* is not set, then *WorkingDirectory=* is relative to the root of the system running the service manager. Note that setting this parameter might result in additional dependencies to be added to the unit (see above).

RootDirectory=

Takes a directory path relative to the host's root directory (i.e. the root of the system running the service manager). Sets the root directory for executed processes, with the **chroot(2)** system call. If this is used, it must be ensured that the process binary and all its auxiliary files are available in the **chroot()** jail. Note that setting this parameter might result in additional dependencies to be added to the unit (see above).

The *PrivateUsers=* setting is particularly useful in conjunction with *RootDirectory=*. For details, see below.

User=, *Group=*

Set the UNIX user or group that the processes are executed as, respectively. Takes a single user or group name, or numeric ID as argument. For system services (services run by the system service manager, i.e. managed by PID 1) and for user services of the root user (services managed by root's instance of **systemd --user**), the default is "root", but *User=* may be used to specify a different user. For user services of any other user, switching user identity is not permitted, hence the only valid setting is the same user the user's service manager is running as. If no group is set, the default group of the user is used. This setting does not affect commands whose command line is prefixed with "+".

DynamicUser=

Takes a boolean parameter. If set, a UNIX user and group pair is allocated dynamically when the unit is started, and released as soon as it is stopped. The user and group will not be added to /etc/passwd or /etc/group, but are managed transiently during runtime. The **nss-systemd(8)** glibc NSS module provides integration of these dynamic users/groups into the system's user and group databases. The user and group name to use may be configured via *User=* and *Group=* (see above). If these options are not used and dynamic user/group allocation is enabled for a unit, the name of the dynamic user/group is implicitly derived from the unit name. If the unit name without the type suffix qualifies as valid user name it is used directly, otherwise a name incorporating a hash of it is used. If a statically allocated user or group of the configured name already exists, it is used and no dynamic user/group is allocated. Dynamic users/groups are allocated from the UID/GID range 61184...65519. It is recommended to avoid this range for regular system or login users. At any point in time each UID/GID from this range is only assigned to zero or one dynamically allocated users/groups in use. However, UID/GIDs are recycled after a unit is terminated. Care should be taken that any processes running as part of a unit for which dynamic users/groups are enabled do not leave files or directories owned by these users/groups around, as a different unit might get the same UID/GID assigned later on, and thus gain access to these files or directories. If *DynamicUser=* is enabled, *RemoveIPC=*, *PrivateTmp=* are implied. This ensures that the lifetime of IPC objects and temporary files created by the executed processes is bound to the runtime of the service, and hence the lifetime of the dynamic user/group. Since /tmp and /var/tmp are usually the only world-writable directories on a system this ensures that a unit making use of dynamic user/group allocation cannot leave files around after unit termination. Moreover *ProtectSystem=strict* and *ProtectHome=read-only* are implied, thus prohibiting the service to write to arbitrary file system locations. In order to allow the service to write to certain directories, they have to be whitelisted using *ReadWritePaths=*, but care must be taken so that UID/GID recycling doesn't create security issues involving files created by the service. Use *RuntimeDirectory=* (see below) in order to assign a writable runtime directory to a service, owned by the dynamic user/group and removed automatically when the unit is terminated. Defaults to off.

SupplementaryGroups=

Sets the supplementary Unix groups the processes are executed as. This takes a space-separated list of group names or IDs. This option may be specified more than once, in which case all listed groups are set as supplementary groups. When the empty string is assigned, the list of supplementary groups is reset, and all assignments prior to this one will have no effect. In any way, this option does not override, but extends the list of supplementary groups configured in the system group database for the user. This does not affect commands prefixed with "+".

RemoveIPC=

Takes a boolean parameter. If set, all System V and POSIX IPC objects owned by the user and group the processes of this unit are run as are removed when the unit is stopped. This setting only has an effect if at least one of *User=*, *Group=* and *DynamicUser=* are used. It has no effect on IPC objects owned by the root user. Specifically, this removes System V semaphores, as well as System V and POSIX shared memory segments and message queues. If multiple units use the same user or group the IPC objects are removed when the last of these units is stopped. This setting is implied if *DynamicUser=* is set.

Nice=

Sets the default nice level (scheduling priority) for executed processes. Takes an integer between -20 (highest priority) and 19 (lowest priority). See **setpriority(2)** for details.

OOMScoreAdjust=

Sets the adjustment level for the Out-Of-Memory killer for executed processes. Takes an integer between -1000 (to disable OOM killing for this process) and 1000 (to make killing of this process under memory pressure very likely). See **proc.txt**[1] for details.

IOSchedulingClass=

Sets the I/O scheduling class for executed processes. Takes an integer between 0 and 3 or one of the strings **none**, **realtime**, **best-effort** or **idle**. See **ioprio_set(2)** for details.

IOSchedulingPriority=
Sets the I/O scheduling priority for executed processes. Takes an integer between 0 (highest priority) and 7 (lowest priority). The available priorities depend on the selected I/O scheduling class (see above). See **ioprio_set(2)** for details.

CPUSchedulingPolicy=
Sets the CPU scheduling policy for executed processes. Takes one of **other**, **batch**, **idle**, **fifo** or **rr**. See **sched_setscheduler(2)** for details.

CPUSchedulingPriority=
Sets the CPU scheduling priority for executed processes. The available priority range depends on the selected CPU scheduling policy (see above). For real-time scheduling policies an integer between 1 (lowest priority) and 99 (highest priority) can be used. See **sched_setscheduler(2)** for details.

CPUSchedulingResetOnFork=
Takes a boolean argument. If true, elevated CPU scheduling priorities and policies will be reset when the executed processes fork, and can hence not leak into child processes. See **sched_setscheduler(2)** for details. Defaults to false.

CPUAffinity=
Controls the CPU affinity of the executed processes. Takes a list of CPU indices or ranges separated by either whitespace or commas. CPU ranges are specified by the lower and upper CPU indices separated by a dash. This option may be specified more than once, in which case the specified CPU affinity masks are merged. If the empty string is assigned, the mask is reset, all assignments prior to this will have no effect. See **sched_setaffinity(2)** for details.

UMask=
Controls the file mode creation mask. Takes an access mode in octal notation. See **umask(2)** for details. Defaults to 0022.

Environment=
Sets environment variables for executed processes. Takes a space-separated list of variable assignments. This option may be specified more than once, in which case all listed variables will be set. If the same variable is set twice, the later setting will override the earlier setting. If the empty string is assigned to this option, the list of environment variables is reset, all prior assignments have no effect. Variable expansion is not performed inside the strings, however, specifier expansion is possible. The $ character has no special meaning. If you need to assign a value containing spaces to a variable, use double quotes (") for the assignment.

Example:

Environment="VAR1=word1 word2" VAR2=word3 "VAR3=$word 5 6"

gives three variables "VAR1", "VAR2", "VAR3" with the values "word1 word2", "word3", "$word 5 6".

See **environ(7)** for details about environment variables.

EnvironmentFile=
Similar to *Environment=* but reads the environment variables from a text file. The text file should contain new-line-separated variable assignments. Empty lines, lines without an "=" separator, or lines starting with ; or # will be ignored, which may be used for commenting. A line ending with a backslash will be concatenated with the following one, allowing multiline variable definitions. The parser strips leading and trailing whitespace from the values of assignments, unless you use double quotes (").

The argument passed should be an absolute filename or wildcard expression, optionally prefixed with "-", which indicates that if the file does not exist, it will not be read and no error or warning message is logged. This option may be specified more than once in which case all specified files are read. If the empty string is assigned to this option, the list of file to read is reset, all prior assignments have no effect.

The files listed with this directive will be read shortly before the process is executed (more specifically, after all processes from a previous unit state terminated. This means you can generate these files in one unit state, and read it with this option in the next).

Settings from these files override settings made with *Environment=*. If the same variable is set twice from these files, the files will be read in the order they are specified and the later setting will override the earlier setting.

PassEnvironment=

Pass environment variables from the systemd system manager to executed processes. Takes a space-separated list of variable names. This option may be specified more than once, in which case all listed variables will be set. If the empty string is assigned to this option, the list of environment variables is reset, all prior assignments have no effect. Variables that are not set in the system manager will not be passed and will be silently ignored.

Variables passed from this setting are overridden by those passed from *Environment=* or *EnvironmentFile=*.

Example:

PassEnvironment=VAR1 VAR2 VAR3

passes three variables "VAR1", "VAR2", "VAR3" with the values set for those variables in PID1.

See **environ(7)** for details about environment variables.

StandardInput=

Controls where file descriptor 0 (STDIN) of the executed processes is connected to. Takes one of **null**, **tty**, **tty-force**, **tty-fail**, **socket** or **fd**.

If **null** is selected, standard input will be connected to /dev/null, i.e. all read attempts by the process will result in immediate EOF.

If **tty** is selected, standard input is connected to a TTY (as configured by *TTYPath=*, see below) and the executed process becomes the controlling process of the terminal. If the terminal is already being controlled by another process, the executed process waits until the current controlling process releases the terminal.

tty-force is similar to **tty**, but the executed process is forcefully and immediately made the controlling process of the terminal, potentially removing previous controlling processes from the terminal.

tty-fail is similar to **tty** but if the terminal already has a controlling process start-up of the executed process fails.

The **socket** option is only valid in socket-activated services, and only when the socket configuration file (see **systemd.socket(5)** for details) specifies a single socket only. If this option is set, standard input will be connected to the socket the service was activated from, which is primarily useful for compatibility with daemons designed for use with the traditional **inetd(8)** daemon.

The **fd** option connects the input stream to a single file descriptor provided by a socket unit. A custom named file descriptor can be specified as part of this option, after a ":" (e.g. "fd:*foobar*"). If no name is specified, "stdin" is assumed (i.e. "fd" is equivalent to "fd:stdin"). At least one socket unit defining such name must be explicitly provided via the *Sockets=* option, and file descriptor name may differ from the name of its containing socket unit. If multiple matches are found, the first one will be used. See *FileDescriptorName=* in **systemd.socket(5)** for more details about named descriptors and ordering.

This setting defaults to **null**.

StandardOutput=

Controls where file descriptor 1 (STDOUT) of the executed processes is connected to. Takes one of **inherit**, **null**, **tty**, **journal**, **syslog**, **kmsg**, **journal+console**, **syslog+console**, **kmsg+console**, **socket** or **fd**.

inherit duplicates the file descriptor of standard input for standard output.

null connects standard output to /dev/null, i.e. everything written to it will be lost.

tty connects standard output to a tty (as configured via *TTYPath=*, see below). If the TTY is used for output only, the executed process will not become the controlling process of the terminal, and will not fail or wait for other processes to release the terminal.

journal connects standard output with the journal which is accessible via **journalctl(1)**. Note that everything that is written to syslog or kmsg (see below) is implicitly stored in the journal as well, the specific two options listed below are hence supersets of this one.

syslog connects standard output to the **syslog(3)** system syslog service, in addition to the journal. Note that the journal daemon is usually configured to forward everything it receives to syslog anyway, in which case this option is no different from **journal**.

kmsg connects standard output with the kernel log buffer which is accessible via **dmesg(1)**, in addition to the journal. The journal daemon might be configured to send all logs to kmsg anyway, in which case this option is no different from **journal**.

journal+console, **syslog+console** and **kmsg+console** work in a similar way as the three options above but copy the output to the system console as well.

socket connects standard output to a socket acquired via socket activation. The semantics are similar to the same option of *StandardInput=*.

The **fd** option connects the output stream to a single file descriptor provided by a socket unit. A custom named file descriptor can be specified as part of this option, after a ":" (e.g. "fd:*foobar*"). If no name is specified, "stdout" is assumed (i.e. "fd" is equivalent to "fd:stdout"). At least one socket unit defining such name must be explicitly provided via the *Sockets=* option, and file descriptor name may differ from the name of its containing socket unit. If multiple matches are found, the first one will be used. See *FileDescriptorName=* in **systemd.socket(5)** for more details about named descriptors and ordering.

If the standard output (or error output, see below) of a unit is connected to the journal, syslog or the kernel log buffer, the unit will implicitly gain a dependency of type *After=* on systemd-journald.socket (also see the automatic dependencies section above).

This setting defaults to the value set with **DefaultStandardOutput=** in **systemd-system.conf(5)**, which defaults to **journal**. Note that setting this parameter might result in additional dependencies to be added to the unit (see above).

StandardError=

Controls where file descriptor 2 (STDERR) of the executed processes is connected to. The available options are identical to those of *StandardOutput=*, with some exceptions: if set to **inherit** the file descriptor used for standard output is duplicated for standard error, while **fd** operates on the error stream and will look by default for a descriptor named "stderr".

This setting defaults to the value set with **DefaultStandardError=** in **systemd-system.conf(5)**, which defaults to **inherit**. Note that setting this parameter might result in additional dependencies to be added to the unit (see above).

TTYPath=

Sets the terminal device node to use if standard input, output, or error are connected to a TTY (see above). Defaults to /dev/console.

TTYReset=

Reset the terminal device specified with *TTYPath=* before and after execution. Defaults to "no".

TTYVHangup=

Disconnect all clients which have opened the terminal device specified with *TTYPath=* before and after execution. Defaults to "no".

TTYVTDisallocate=
: If the terminal device specified with *TTYPath=* is a virtual console terminal, try to deallocate the TTY before and after execution. This ensures that the screen and scrollback buffer is cleared. Defaults to "no".

SyslogIdentifier=
: Sets the process name to prefix log lines sent to the logging system or the kernel log buffer with. If not set, defaults to the process name of the executed process. This option is only useful when *StandardOutput=* or *StandardError=* are set to **syslog**, **journal** or **kmsg** (or to the same settings in combination with **+console**).

SyslogFacility=
: Sets the syslog facility to use when logging to syslog. One of **kern**, **user**, **mail**, **daemon**, **auth**, **syslog**, **lpr**, **news**, **uucp**, **cron**, **authpriv**, **ftp**, **local0**, **local1**, **local2**, **local3**, **local4**, **local5**, **local6** or **local7**. See **syslog(3)** for details. This option is only useful when *StandardOutput=* or *StandardError=* are set to **syslog**. Defaults to **daemon**.

SyslogLevel=
: The default syslog level to use when logging to syslog or the kernel log buffer. One of **emerg**, **alert**, **crit**, **err**, **warning**, **notice**, **info**, **debug**. See **syslog(3)** for details. This option is only useful when *StandardOutput=* or *StandardError=* are set to **syslog** or **kmsg**. Note that individual lines output by the daemon might be prefixed with a different log level which can be used to override the default log level specified here. The interpretation of these prefixes may be disabled with *SyslogLevelPrefix=*, see below. For details, see **sd-daemon(3)**. Defaults to **info**.

SyslogLevelPrefix=
: Takes a boolean argument. If true and *StandardOutput=* or *StandardError=* are set to **syslog**, **kmsg** or **journal**, log lines written by the executed process that are prefixed with a log level will be passed on to syslog with this log level set but the prefix removed. If set to false, the interpretation of these prefixes is disabled and the logged lines are passed on as-is. For details about this prefixing see **sd-daemon(3)**. Defaults to true.

TimerSlackNSec=
: Sets the timer slack in nanoseconds for the executed processes. The timer slack controls the accuracy of wake-ups triggered by timers. See **prctl(2)** for more information. Note that in contrast to most other time span definitions this parameter takes an integer value in nano-seconds if no unit is specified. The usual time units are understood too.

LimitCPU=, *LimitFSIZE=*, *LimitDATA=*, *LimitSTACK=*, *LimitCORE=*, *LimitRSS=*, *LimitNOFILE=*, *LimitAS=*, *LimitNPROC=*, *LimitMEMLOCK=*, *LimitLOCKS=*, *LimitSIGPENDING=*, *LimitMSGQUEUE=*, *LimitNICE=*, *LimitRTPRIO=*, *LimitRTTIME=*
: Set soft and hard limits on various resources for executed processes. See **setrlimit(2)** for details on the resource limit concept. Resource limits may be specified in two formats: either as single value to set a specific soft and hard limit to the same value, or as colon-separated pair **soft:hard** to set both limits individually (e.g. "LimitAS=4G:16G"). Use the string *infinity* to configure no limit on a specific resource. The multiplicative suffixes K, M, G, T, P and E (to the base 1024) may be used for resource limits measured in bytes (e.g. LimitAS=16G). For the limits referring to time values, the usual time units ms, s, min, h and so on may be used (see **systemd.time(7)** for details). Note that if no time unit is specified for *LimitCPU=* the default unit of seconds is implied, while for *LimitRTTIME=* the default unit of microseconds is implied. Also, note that the effective granularity of the limits might influence their enforcement. For example, time limits specified for *LimitCPU=* will be rounded up implicitly to multiples of 1s. For *LimitNICE=* the value may be specified in two syntaxes: if prefixed with "+" or "-", the value is understood as regular Linux nice value in the range -20..19. If not prefixed like this the value is understood as raw resource limit parameter in the range 0..40 (with 0 being equivalent to 1).

  Note that most process resource limits configured with these options are per-process, and processes may fork in order to acquire a new set of resources that are accounted independently of the original process, and may thus escape limits set. Also note that *LimitRSS=* is not implemented on Linux, and setting it has no effect. Often it is advisable to prefer the resource controls listed in **systemd.resource-**

control(5) over these per-process limits, as they apply to services as a whole, may be altered dynamically at runtime, and are generally more expressive. For example, *MemoryLimit=* is a more powerful (and working) replacement for *LimitRSS=*.

For system units these resource limits may be chosen freely. For user units however (i.e. units run by a per-user instance of **systemd(1)**), these limits are bound by (possibly more restrictive) per-user limits enforced by the OS.

Resource limits not configured explicitly for a unit default to the value configured in the various *DefaultLimitCPU=*, *DefaultLimitFSIZE=*, ... options available in **systemd-system.conf(5)**, and – if not configured there – the kernel or per-user defaults, as defined by the OS (the latter only for user services, see above).

Table 1. Resource limit directives, their equivalent ulimit shell commands and the unit used

Directive	**ulimit** equivalent	**Unit**
LimitCPU=	ulimit -t	Seconds
LimitFSIZE=	ulimit -f	Bytes
LimitDATA=	ulimit -d	Bytes
LimitSTACK=	ulimit -s	Bytes
LimitCORE=	ulimit -c	Bytes
LimitRSS=	ulimit -m	Bytes
LimitNOFILE=	ulimit -n	Number of File Descriptors
LimitAS=	ulimit -v	Bytes
LimitNPROC=	ulimit -u	Number of Processes
LimitMEMLOCK=	ulimit -l	Bytes
LimitLOCKS=	ulimit -x	Number of Locks
LimitSIGPENDING=	ulimit -i	Number of Queued Signals
LimitMSGQUEUE=	ulimit -q	Bytes
LimitNICE=	ulimit -e	Nice Level
LimitRTPRIO=	ulimit -r	Realtime Priority
LimitRTTIME=	No equivalent	Microseconds

PAMName=

Sets the PAM service name to set up a session as. If set, the executed process will be registered as a PAM session under the specified service name. This is only useful in conjunction with the *User=* setting. If not set, no PAM session will be opened for the executed processes. See **pam(8)** for details.

CapabilityBoundingSet=

Controls which capabilities to include in the capability bounding set for the executed process. See **capabilities(7)** for details. Takes a whitespace-separated list of capability names, e.g. **CAP_SYS_ADMIN**, **CAP_DAC_OVERRIDE**, **CAP_SYS_PTRACE**. Capabilities listed will be included in the bounding set, all others are removed. If the list of capabilities is prefixed with "~", all but the listed capabilities will be included, the effect of the assignment inverted. Note that this option also affects the respective capabilities in the effective, permitted and inheritable capability sets. If this option is not used, the capability bounding set is not modified on process execution, hence no limits on the capabilities of the process are enforced. This option may appear more than once, in which case the bounding sets are merged. If the empty string is assigned to this option, the bounding set is reset to the empty capability set, and all prior settings have no effect. If set to "~" (without any further argument), the bounding set is reset to the full set of available capabilities, also undoing any previous settings. This does not affect commands prefixed with "+".

AmbientCapabilities=

Controls which capabilities to include in the ambient capability set for the executed process. Takes a whitespace-separated list of capability names, e.g. **CAP_SYS_ADMIN**, **CAP_DAC_OVERRIDE**,

CAP_SYS_PTRACE. This option may appear more than once in which case the ambient capability sets are merged. If the list of capabilities is prefixed with "~", all but the listed capabilities will be included, the effect of the assignment inverted. If the empty string is assigned to this option, the ambient capability set is reset to the empty capability set, and all prior settings have no effect. If set to "~" (without any further argument), the ambient capability set is reset to the full set of available capabilities, also undoing any previous settings. Note that adding capabilities to ambient capability set adds them to the process's inherited capability set.

Ambient capability sets are useful if you want to execute a process as a non-privileged user but still want to give it some capabilities. Note that in this case option **keep-caps** is automatically added to *SecureBits=* to retain the capabilities over the user change. *AmbientCapabilities=* does not affect commands prefixed with "+".

SecureBits=

Controls the secure bits set for the executed process. Takes a space-separated combination of options from the following list: **keep-caps**, **keep-caps-locked**, **no-setuid-fixup**, **no-setuid-fixup-locked**, **noroot**, and **noroot-locked**. This option may appear more than once, in which case the secure bits are ORed. If the empty string is assigned to this option, the bits are reset to 0. This does not affect commands prefixed with "+". See **capabilities(7)** for details.

ReadWritePaths=, *ReadOnlyPaths=*, *InaccessiblePaths=*

Sets up a new file system namespace for executed processes. These options may be used to limit access a process might have to the file system hierarchy. Each setting takes a space-separated list of paths relative to the host's root directory (i.e. the system running the service manager). Note that if paths contain symlinks, they are resolved relative to the root directory set with *RootDirectory=*.

Paths listed in *ReadWritePaths=* are accessible from within the namespace with the same access modes as from outside of it. Paths listed in *ReadOnlyPaths=* are accessible for reading only, writing will be refused even if the usual file access controls would permit this. Nest *ReadWritePaths=* inside of *ReadOnlyPaths=* in order to provide writable subdirectories within read-only directories. Use *ReadWritePaths=* in order to whitelist specific paths for write access if *ProtectSystem=strict* is used. Paths listed in *InaccessiblePaths=* will be made inaccessible for processes inside the namespace (along with everything below them in the file system hierarchy).

Note that restricting access with these options does not extend to submounts of a directory that are created later on. Non-directory paths may be specified as well. These options may be specified more than once, in which case all paths listed will have limited access from within the namespace. If the empty string is assigned to this option, the specific list is reset, and all prior assignments have no effect.

Paths in *ReadWritePaths=*, *ReadOnlyPaths=* and *InaccessiblePaths=* may be prefixed with "-", in which case they will be ignored when they do not exist. Note that using this setting will disconnect propagation of mounts from the service to the host (propagation in the opposite direction continues to work). This means that this setting may not be used for services which shall be able to install mount points in the main mount namespace. Note that the effect of these settings may be undone by privileged processes. In order to set up an effective sandboxed environment for a unit it is thus recommended to combine these settings with either *CapabilityBoundingSet=~CAP_SYS_ADMIN* or *SystemCallFilter=~@mount*.

PrivateTmp=

Takes a boolean argument. If true, sets up a new file system namespace for the executed processes and mounts private /tmp and /var/tmp directories inside it that is not shared by processes outside of the namespace. This is useful to secure access to temporary files of the process, but makes sharing between processes via /tmp or /var/tmp impossible. If this is enabled, all temporary files created by a service in these directories will be removed after the service is stopped. Defaults to false. It is possible to run two or more units within the same private /tmp and /var/tmp namespace by using the *JoinsNamespaceOf=* directive, see **systemd.unit(5)** for details. This setting is implied if *DynamicUser=* is set. For this setting the same restrictions regarding mount propagation and

privileges apply as for *ReadOnlyPaths=* and related calls, see above.

PrivateDevices=
: Takes a boolean argument. If true, sets up a new /dev namespace for the executed processes and only adds API pseudo devices such as /dev/null, /dev/zero or /dev/random (as well as the pseudo TTY subsystem) to it, but no physical devices such as /dev/sda, system memory /dev/mem, system ports /dev/port and others. This is useful to securely turn off physical device access by the executed process. Defaults to false. Enabling this option will install a system call filter to block low-level I/O system calls that are grouped in the *@raw-io* set, will also remove **CAP_MKNOD** and **CAP_SYS_RAWIO** from the capability bounding set for the unit (see above), and set *DevicePolicy=closed* (see **systemd.resource-control(5)** for details). Note that using this setting will disconnect propagation of mounts from the service to the host (propagation in the opposite direction continues to work). This means that this setting may not be used for services which shall be able to install mount points in the main mount namespace. The /dev namespace will be mounted read-only and 'noexec'. The latter may break old programs which try to set up executable memory by using **mmap(2)** of /dev/zero instead of using **MAP_ANON**. This setting is implied if *DynamicUser=* is set. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above.

PrivateNetwork=
: Takes a boolean argument. If true, sets up a new network namespace for the executed processes and configures only the loopback network device "lo" inside it. No other network devices will be available to the executed process. This is useful to securely turn off network access by the executed process. Defaults to false. It is possible to run two or more units within the same private network namespace by using the *JoinsNamespaceOf=* directive, see **systemd.unit(5)** for details. Note that this option will disconnect all socket families from the host, this includes AF_NETLINK and AF_UNIX. The latter has the effect that AF_UNIX sockets in the abstract socket namespace will become unavailable to the processes (however, those located in the file system will continue to be accessible).

PrivateUsers=
: Takes a boolean argument. If true, sets up a new user namespace for the executed processes and configures a minimal user and group mapping, that maps the "root" user and group as well as the unit's own user and group to themselves and everything else to the "nobody" user and group. This is useful to securely detach the user and group databases used by the unit from the rest of the system, and thus to create an effective sandbox environment. All files, directories, processes, IPC objects and other resources owned by users/groups not equaling "root" or the unit's own will stay visible from within the unit but appear owned by the "nobody" user and group. If this mode is enabled, all unit processes are run without privileges in the host user namespace (regardless if the unit's own user/group is "root" or not). Specifically this means that the process will have zero process capabilities on the host's user namespace, but full capabilities within the service's user namespace. Settings such as *CapabilityBoundingSet=* will affect only the latter, and there's no way to acquire additional capabilities in the host's user namespace. Defaults to off.

  This setting is particularly useful in conjunction with *RootDirectory=*, as the need to synchronize the user and group databases in the root directory and on the host is reduced, as the only users and groups who need to be matched are "root", "nobody" and the unit's own user and group.

ProtectSystem=
: Takes a boolean argument or the special values "full" or "strict". If true, mounts the /usr and /boot directories read-only for processes invoked by this unit. If set to "full", the /etc directory is mounted read-only, too. If set to "strict" the entire file system hierarchy is mounted read-only, except for the API file system subtrees /dev, /proc and /sys (protect these directories using *PrivateDevices=*, *ProtectKernelTunables=*, *ProtectControlGroups=*). This setting ensures that any modification of the vendor-supplied operating system (and optionally its configuration, and local mounts) is prohibited for the service. It is recommended to enable this setting for all long-running services, unless they are involved with system updates or need to modify the operating system in other ways. If this option is used, *ReadWritePaths=* may be used to exclude specific directories from being made read-only. This

setting is implied if *DynamicUser=* is set. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above. Defaults to off.

ProtectHome=

Takes a boolean argument or "read-only". If true, the directories /home, /root and /run/user are made inaccessible and empty for processes invoked by this unit. If set to "read-only", the three directories are made read-only instead. It is recommended to enable this setting for all long-running services (in particular network-facing ones), to ensure they cannot get access to private user data, unless the services actually require access to the user's private data. This setting is implied if *DynamicUser=* is set. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above.

ProtectKernelTunables=

Takes a boolean argument. If true, kernel variables accessible through /proc/sys, /sys, /proc/sysrq-trigger, /proc/latency_stats, /proc/acpi, /proc/timer_stats, /proc/fs and /proc/irq will be made read-only to all processes of the unit. Usually, tunable kernel variables should only be written at boot-time, with the **sysctl.d(5)** mechanism. Almost no services need to write to these at runtime; it is hence recommended to turn this on for most services. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above. Defaults to off. Note that this option does not prevent kernel tuning through IPC interfaces and external programs. However *InaccessiblePaths=* can be used to make some IPC file system objects inaccessible.

ProtectControlGroups=

Takes a boolean argument. If true, the Linux Control Groups (**cgroups**(7)) hierarchies accessible through /sys/fs/cgroup will be made read-only to all processes of the unit. Except for container managers no services should require write access to the control groups hierarchies; it is hence recommended to turn this on for most services. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above. Defaults to off.

MountFlags=

Takes a mount propagation flag: **shared**, **slave** or **private**, which control whether mounts in the file system namespace set up for this unit's processes will receive or propagate mounts or unmounts. See **mount(2)** for details. Defaults to **shared**. Use **shared** to ensure that mounts and unmounts are propagated from the host to the container and vice versa. Use **slave** to run processes so that none of their mounts and unmounts will propagate to the host. Use **private** to also ensure that no mounts and unmounts from the host will propagate into the unit processes' namespace. Note that **slave** means that file systems mounted on the host might stay mounted continuously in the unit's namespace, and thus keep the device busy. Note that the file system namespace related options (*PrivateTmp=*, *PrivateDevices=*, *ProtectSystem=*, *ProtectHome=*, *ProtectKernelTunables=*, *ProtectControlGroups=*, *ReadOnlyPaths=*, *InaccessiblePaths=*, *ReadWritePaths=*) require that mount and unmount propagation from the unit's file system namespace is disabled, and hence downgrade **shared** to **slave**.

UtmpIdentifier=

Takes a four character identifier string for an **utmp(5)** and wtmp entry for this service. This should only be set for services such as **getty** implementations (such as **agetty(8)**) where utmp/wtmp entries must be created and cleared before and after execution, or for services that shall be executed as if they were run by a **getty** process (see below). If the configured string is longer than four characters, it is truncated and the terminal four characters are used. This setting interprets %I style string replacements. This setting is unset by default, i.e. no utmp/wtmp entries are created or cleaned up for this service.

UtmpMode=

Takes one of "init", "login" or "user". If *UtmpIdentifier=* is set, controls which type of **utmp(5)**/wtmp entries for this service are generated. This setting has no effect unless *UtmpIdentifier=* is set too. If "init" is set, only an **INIT_PROCESS** entry is generated and the invoked process must implement a **getty**-compatible utmp/wtmp logic. If "login" is set, first an **INIT_PROCESS** entry, followed by a **LOGIN_PROCESS** entry is generated. In this case, the invoked process must implement a **login(1)**-compatible utmp/wtmp logic. If "user" is set, first an **INIT_PROCESS** entry, then a

LOGIN_PROCESS entry and finally a **USER_PROCESS** entry is generated. In this case, the invoked process may be any process that is suitable to be run as session leader. Defaults to "init".

SELinuxContext=
Set the SELinux security context of the executed process. If set, this will override the automated domain transition. However, the policy still needs to authorize the transition. This directive is ignored if SELinux is disabled. If prefixed by "-", all errors will be ignored. This does not affect commands prefixed with "+". See **setexeccon(3)** for details.

AppArmorProfile=
Takes a profile name as argument. The process executed by the unit will switch to this profile when started. Profiles must already be loaded in the kernel, or the unit will fail. This result in a non operation if AppArmor is not enabled. If prefixed by "-", all errors will be ignored. This does not affect commands prefixed with "+".

SmackProcessLabel=
Takes a **SMACK64** security label as argument. The process executed by the unit will be started under this label and SMACK will decide whether the process is allowed to run or not, based on it. The process will continue to run under the label specified here unless the executable has its own **SMACK64EXEC** label, in which case the process will transition to run under that label. When not specified, the label that systemd is running under is used. This directive is ignored if SMACK is disabled.

The value may be prefixed by "-", in which case all errors will be ignored. An empty value may be specified to unset previous assignments. This does not affect commands prefixed with "+".

IgnoreSIGPIPE=
Takes a boolean argument. If true, causes **SIGPIPE** to be ignored in the executed process. Defaults to true because **SIGPIPE** generally is useful only in shell pipelines.

NoNewPrivileges=
Takes a boolean argument. If true, ensures that the service process and all its children can never gain new privileges through **execve()** (e.g. via setuid or setgid bits, or filesystem capabilities). This is the simplest and most effective way to ensure that a process and its children can never elevate privileges again. Defaults to false, but in the user manager instance certain settings force *NoNewPrivileges=yes*, ignoring the value of this setting. This is the case when *SystemCallFilter=*, *SystemCallArchitectures=*, *RestrictAddressFamilies=*, *RestrictNamespaces=*, *PrivateDevices=*, *ProtectKernelTunables=*, *ProtectKernelModules=*, *MemoryDenyWriteExecute=*, or *RestrictRealtime=* are specified.

SystemCallFilter=
Takes a space-separated list of system call names. If this setting is used, all system calls executed by the unit processes except for the listed ones will result in immediate process termination with the **SIGSYS** signal (whitelisting). If the first character of the list is "~", the effect is inverted: only the listed system calls will result in immediate process termination (blacklisting). If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=nobody*), *NoNewPrivileges=yes* is implied. This feature makes use of the Secure Computing Mode 2 interfaces of the kernel ('seccomp filtering') and is useful for enforcing a minimal sandboxing environment. Note that the **execve**, **exit**, **exit_group**, **getrlimit**, **rt_sigreturn**, **sigreturn** system calls and the system calls for querying time and sleeping are implicitly whitelisted and do not need to be listed explicitly. This option may be specified more than once, in which case the filter masks are merged. If the empty string is assigned, the filter is reset, all prior assignments will have no effect. This does not affect commands prefixed with "+".

Note that strict system call filters may impact execution and error handling code paths of the service invocation. Specifically, access to the **execve** system call is required for the execution of the service binary — if it is blocked service invocation will necessarily fail. Also, if execution of the service binary fails for some reason (for example: missing service executable), the error handling logic might require access to an additional set of system calls in order to process and log this failure correctly. It might be necessary to temporarily disable system call filters in order to simplify debugging of such

failures.

If you specify both types of this option (i.e. whitelisting and blacklisting), the first encountered will take precedence and will dictate the default action (termination or approval of a system call). Then the next occurrences of this option will add or delete the listed system calls from the set of the filtered system calls, depending of its type and the default action. (For example, if you have started with a whitelisting of **read** and **write**, and right after it add a blacklisting of **write**, then **write** will be removed from the set.)

As the number of possible system calls is large, predefined sets of system calls are provided. A set starts with "@" character, followed by name of the set.

Table 2. Currently predefined system call sets

Set	Description
@basic-io	System calls for basic I/O: reading, writing, seeking, file descriptor duplication and closing (**read**(2), **write(2)**, and related calls)
@clock	System calls for changing the system clock (**adjtimex**(2), **settimeofday(2)**, and related calls)
@cpu-emulation	System calls for CPU emulation functionality (**vm86**(2) and related calls)
@debug	Debugging, performance monitoring and tracing functionality (**ptrace**(2), **perf_event_open(2)** and related calls)
@io-event	Event loop system calls (**poll**(2), **select(2)**, **epoll(7)**, **eventfd(2)** and related calls)
@ipc	Pipes, SysV IPC, POSIX Message Queues and other IPC (**mq_overview**(7), **svipc(7)**)
@keyring	Kernel keyring access (**keyctl**(2) and related calls)
@module	Kernel module control (**init_module**(2), **delete_module(2)** and related calls)
@mount	File system mounting and unmounting (**mount**(2), **chroot(2)**, and related calls)
@network-io	Socket I/O (including local AF_UNIX): **socket(7)**, **unix(7)**
@obsolete	Unusual, obsolete or unimplemented (**create_module**(2), **gtty(2)**, ...)
@privileged	All system calls which need super-user capabilities (**capabilities**(7))
@process	Process control, execution, namespaces (**clone**(2), **kill(2)**, **namespaces(7)**, ...
@raw-io	Raw I/O port access (**ioperm**(2), **iopl(2)**, **pciconfig_read()**, ...)
@resources	System calls for changing resource limits, memory and scheduling parameters (**setrlimit**(2), **setpriority(2)**, ...)

Note that as new system calls are added to the kernel, additional system calls might be added to the groups above, so the contents of the sets may change between systemd versions.

It is recommended to combine the file system namespacing related options with *SystemCallFilter=~@mount*, in order to prohibit the unit's processes to undo the mappings. Specifically these are the options *PrivateTmp=*, *PrivateDevices=*, *ProtectSystem=*, *ProtectHome=*, *ProtectKernelTunables=*, *ProtectControlGroups=*, *ReadOnlyPaths=*, *InaccessiblePaths=* and *ReadWritePaths=*.

SystemCallErrorNumber=

Takes an "errno" error number name to return when the system call filter configured with *SystemCallFilter=* is triggered, instead of terminating the process immediately. Takes an error name such as **EPERM**, **EACCES** or **EUCLEAN**. When this setting is not used, or when the empty string is assigned, the process will be terminated immediately when the filter is triggered.

SystemCallArchitectures=

Takes a space-separated list of architecture identifiers to include in the system call filter. The known architecture identifiers are the same as for *ConditionArchitecture=* described in **systemd.unit(5)**, as well as **x32**, **mips64-n32**, **mips64-le-n32**, and the special identifier **native**. Only system calls of the specified architectures will be permitted to processes of this unit. This is an effective way to disable compatibility with non-native architectures for processes, for example to prohibit execution of 32-bit x86 binaries on 64-bit x86-64 systems. The special **native** identifier implicitly maps to the native architecture of the system (or more strictly: to the architecture the system manager is compiled for). If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=nobody*), *NoNewPrivileges=yes* is implied. Note that setting this option to a non-empty list implies that **native** is included too. By default, this option is set to the empty list, i.e. no architecture system call filtering is applied.

RestrictAddressFamilies=

Restricts the set of socket address families accessible to the processes of this unit. Takes a space-separated list of address family names to whitelist, such as **AF_UNIX**, **AF_INET** or **AF_INET6**. When prefixed with ~ the listed address families will be applied as blacklist, otherwise as whitelist. Note that this restricts access to the **socket(2)** system call only. Sockets passed into the process by other means (for example, by using socket activation with socket units, see **systemd.socket(5)**) are unaffected. Also, sockets created with **socketpair()** (which creates connected AF_UNIX sockets only) are unaffected. Note that this option has no effect on 32-bit x86, s390, s390x, mips, mips-le, ppc, ppc-le, pcc64, ppc64-le and is ignored (but works correctly on other architectures, including x86-64). If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=nobody*), *NoNewPrivileges=yes* is implied. By default, no restrictions apply, all address families are accessible to processes. If assigned the empty string, any previous address familiy restriction changes are undone. This setting does not affect commands prefixed with "+".

Use this option to limit exposure of processes to remote access, in particular via exotic and sensitive network protocols, such as **AF_PACKET**. Note that in most cases, the local **AF_UNIX** address family should be included in the configured whitelist as it is frequently used for local communication, including for **syslog(2)** logging.

RestrictNamespaces=

Restricts access to Linux namespace functionality for the processes of this unit. For details about Linux namespaces, see **namespaces(7)**. Either takes a boolean argument, or a space-separated list of namespace type identifiers. If false (the default), no restrictions on namespace creation and switching are made. If true, access to any kind of namespacing is prohibited. Otherwise, a space-separated list of namespace type identifiers must be specified, consisting of any combination of: **cgroup**, **ipc**, **net**, **mnt**, **pid**, **user** and **uts**. Any namespace type listed is made accessible to the unit's processes, access to namespace types not listed is prohibited (whitelisting). By prepending the list with a single tilda character ("~") the effect may be inverted: only the listed namespace types will be made inaccessible, all unlisted ones are permitted (blacklisting). If the empty string is assigned, the default namespace restrictions are applied, which is equivalent to false. Internally, this setting limits access to the **unshare(2)**, **clone(2)** and **setns(2)** system calls, taking the specified flags parameters into account. Note that — if this option is used — in addition to restricting creation and switching of the specified types of namespaces (or all of them, if true) access to the **setns()** system call with a zero flags parameter is prohibited. This setting is only supported on x86, x86-64, s390 and s390x, and enforces no restrictions on other architectures. If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied.

ProtectKernelModules=

Takes a boolean argument. If true, explicit module loading will be denied. This allows to turn off module load and unload operations on modular kernels. It is recommended to turn this on for most services that do not need special file systems or extra kernel modules to work. Default to off. Enabling this option removes **CAP_SYS_MODULE** from the capability bounding set for the unit, and installs a system call filter to block module system calls, also /usr/lib/modules is made inaccessible. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above. Note that limited automatic module loading due to user configuration or kernel mapping tables might still happen as side effect of requested user operations, both privileged and unprivileged. To disable module auto-load feature please see **sysctl.d(5)** **kernel.modules_disabled** mechanism and /proc/sys/kernel/modules_disabled documentation.

Personality=

Controls which kernel architecture **uname(2)** shall report, when invoked by unit processes. Takes one of the architecture identifiers **x86**, **x86-64**, **ppc**, **ppc-le**, **ppc64**, **ppc64-le**, **s390** or **s390x**. Which personality architectures are supported depends on the system architecture. Usually the 64bit versions of the various system architectures support their immediate 32bit personality architecture counterpart, but no others. For example, **x86-64** systems support the **x86-64** and **x86** personalities but no others. The personality feature is useful when running 32-bit services on a 64-bit host system. If not specified, the personality is left unmodified and thus reflects the personality of the host system's kernel.

RuntimeDirectory=, *RuntimeDirectoryMode=*

Takes a list of directory names. If set, one or more directories by the specified names will be created below /run (for system services) or below *$XDG_RUNTIME_DIR* (for user services) when the unit is started, and removed when the unit is stopped. The directories will have the access mode specified in *RuntimeDirectoryMode=*, and will be owned by the user and group specified in *User=* and *Group=*. Use this to manage one or more runtime directories of the unit and bind their lifetime to the daemon runtime. The specified directory names must be relative, and may not include a "/", i.e. must refer to simple directories to create or remove. This is particularly useful for unprivileged daemons that cannot create runtime directories in /run due to lack of privileges, and to make sure the runtime directory is cleaned up automatically after use. For runtime directories that require more complex or different configuration or lifetime guarantees, please consider using **tmpfiles.d(5)**.

MemoryDenyWriteExecute=

Takes a boolean argument. If set, attempts to create memory mappings that are writable and executable at the same time, or to change existing memory mappings to become executable, or mapping shared memory segments as executable are prohibited. Specifically, a system call filter is added that rejects **mmap(2)** system calls with both **PROT_EXEC** and **PROT_WRITE** set, **mprotect(2)** system calls with **PROT_EXEC** set and **shmat(2)** system calls with **SHM_EXEC** set. Note that this option is incompatible with programs that generate program code dynamically at runtime, such as JIT execution engines, or programs compiled making use of the code "trampoline" feature of various C compilers. This option improves service security, as it makes harder for software exploits to change running code dynamically. Note that this feature is fully available on x86-64, and partially on x86. Specifically, the **shmat()** protection is not available on x86. If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied.

RestrictRealtime=

Takes a boolean argument. If set, any attempts to enable realtime scheduling in a process of the unit are refused. This restricts access to realtime task scheduling policies such as **SCHED_FIFO**, **SCHED_RR** or **SCHED_DEADLINE**. See **sched(7)** for details about these scheduling policies. Realtime scheduling policies may be used to monopolize CPU time for longer periods of time, and may hence be used to lock up or otherwise trigger Denial-of-Service situations on the system. It is hence recommended to restrict access to realtime scheduling to the few programs that actually require them. Defaults to off.

ENVIRONMENT VARIABLES IN SPAWNED PROCESSES

Processes started by the system are executed in a clean environment in which select variables listed below are set. System processes started by systemd do not inherit variables from PID 1, but processes started by

user systemd instances inherit all environment variables from the user systemd instance.

$PATH
: Colon-separated list of directories to use when launching executables. Systemd uses a fixed value of /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin.

$LANG
: Locale. Can be set in **locale.conf(5)** or on the kernel command line (see **systemd(1)** and **kernel-command-line(7)**).

$USER, *$LOGNAME*, *$HOME*, *$SHELL*
: User name (twice), home directory, and the login shell. The variables are set for the units that have *User=* set, which includes user **systemd** instances. See **passwd(5)**.

$INVOCATION_ID
: Contains a randomized, unique 128bit ID identifying each runtime cycle of the unit, formatted as 32 character hexadecimal string. A new ID is assigned each time the unit changes from an inactive state into an activating or active state, and may be used to identify this specific runtime cycle, in particular in data stored offline, such as the journal. The same ID is passed to all processes run as part of the unit.

$XDG_RUNTIME_DIR
: The directory for volatile state. Set for the user **systemd** instance, and also in user sessions. See **pam_systemd(8)**.

$XDG_SESSION_ID, *$XDG_SEAT*, *$XDG_VTNR*
: The identifier of the session, the seat name, and virtual terminal of the session. Set by **pam_systemd(8)** for login sessions. *$XDG_SEAT* and *$XDG_VTNR* will only be set when attached to a seat and a tty.

$MAINPID
: The PID of the unit's main process if it is known. This is only set for control processes as invoked by *ExecReload=* and similar.

$MANAGERPID
: The PID of the user **systemd** instance, set for processes spawned by it.

$LISTEN_FDS, *$LISTEN_PID*, *$LISTEN_FDNAMES*
: Information about file descriptors passed to a service for socket activation. See **sd_listen_fds(3)**.

$NOTIFY_SOCKET
: The socket **sd_notify()** talks to. See **sd_notify(3)**.

$WATCHDOG_PID, *$WATCHDOG_USEC*
: Information about watchdog keep-alive notifications. See **sd_watchdog_enabled(3)**.

$TERM
: Terminal type, set only for units connected to a terminal (*StandardInput=tty*, *StandardOutput=tty*, or *StandardError=tty*). See **termcap(5)**.

$JOURNAL_STREAM
: If the standard output or standard error output of the executed processes are connected to the journal (for example, by setting *StandardError=journal*) *$JOURNAL_STREAM* contains the device and inode numbers of the connection file descriptor, formatted in decimal, separated by a colon (":"). This permits invoked processes to safely detect whether their standard output or standard error output are connected to the journal. The device and inode numbers of the file descriptors should be compared with the values set in the environment variable to determine whether the process output is still connected to the journal. Note that it is generally not sufficient to only check whether *$JOURNAL_STREAM* is set at all as services might invoke external processes replacing their standard output or standard error output, without unsetting the environment variable.

  This environment variable is primarily useful to allow services to optionally upgrade their used log protocol to the native journal protocol (using **sd_journal_print(3)** and other functions) if their standard output or standard error output is connected to the journal anyway, thus enabling delivery of

structured metadata along with logged messages.

$SERVICE_RESULT
: Only defined for the service unit type, this environment variable is passed to all *ExecStop=* and *ExecStopPost=* processes, and encodes the service "result". Currently, the following values are defined: "timeout" (in case of an operation timeout), "exit-code" (if a service process exited with a non-zero exit code; see *$EXIT_CODE* below for the actual exit code returned), "signal" (if a service process was terminated abnormally by a signal; see *$EXIT_CODE* below for the actual signal used for the termination), "core-dump" (if a service process terminated abnormally and dumped core), "watchdog" (if the watchdog keep-alive ping was enabled for the service but it missed the deadline), or "resources" (a catch-all condition in case a system operation failed).

  This environment variable is useful to monitor failure or successful termination of a service. Even though this variable is available in both *ExecStop=* and *ExecStopPost=*, it is usually a better choice to place monitoring tools in the latter, as the former is only invoked for services that managed to start up correctly, and the latter covers both services that failed during their start-up and those which failed during their runtime.

$EXIT_CODE, *$EXIT_STATUS*
: Only defined for the service unit type, these environment variables are passed to all *ExecStop=*, *ExecStopPost=* processes and contain exit status/code information of the main process of the service. For the precise definition of the exit code and status, see **wait(2)**. *$EXIT_CODE* is one of "exited", "killed", "dumped". *$EXIT_STATUS* contains the numeric exit code formatted as string if *$EXIT_CODE* is "exited", and the signal name in all other cases. Note that these environment variables are only set if the service manager succeeded to start and identify the main process of the service.

Table 3. Summary of possible service result variable values

$SERVICE_RESULT	*$EXIT_STATUS*	*$EXIT_CODE*
"timeout"	"killed"	"TERM", "KILL"
	"exited"	"0", "1", "2", "3", ..., "255"
"exit-code"	"exited"	"0", "1", "2", "3", ..., "255"
"signal"	"killed"	"HUP", "INT", "KILL", ...
"core-dump"	"dumped"	"ABRT", "SEGV", "QUIT", ...
"watchdog"	"dumped"	"ABRT"
	"killed"	"TERM", "KILL"
	"exited"	"0", "1", "2", "3", ..., "255"
"resources"	any of the above	any of the above
Note: the process may be also terminated by a signal not sent by systemd. In particular the process may send an arbitrary signal to itself in a handler for any of the non-maskable signals. Nevertheless, in the "timeout" and "watchdog" rows above only the signals that systemd sends have been included.		

Additional variables may be configured by the following means: for processes spawned in specific units, use the *Environment=*, *EnvironmentFile=* and *PassEnvironment=* options above; to specify variables globally, use *DefaultEnvironment=* (see **systemd-system.conf(5)**) or the kernel option *systemd.setenv=* (see **systemd(1)**). Additional variables may also be set through PAM, cf. **pam_env(8)**.

SEE ALSO

systemd(1), **systemctl(1)**, **journalctl(8)**, **systemd.unit(5)**, **systemd.service(5)**, **systemd.socket(5)**, **systemd.swap(5)**, **systemd.mount(5)**, **systemd.kill(5)**, **systemd.resource-control(5)**, **systemd.time(7)**, **systemd.directives(7)**, **tmpfiles.d(5)**, **exec(3)**

NOTES

1. proc.txt
   https://www.kernel.org/doc/Documentation/filesystems/proc.txt