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.

IMPLICIT DEPENDENCIES

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

- Units with *WorkingDirectory=*, *RootDirectory=*, *RootImage=*, *RuntimeDirectory=*, *StateDirectory=*, *CacheDirectory=*, *LogsDirectory=* or *ConfigurationDirectory=* 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=*.
- Similarly, units with *PrivateTmp=* enabled automatically get mount unit dependencies for all mounts required to access /tmp/ and /var/tmp/. They will also gain an automatic *After=* dependency on **systemd-tmpfiles-setup.service**(8).
- Units whose standard output or error output is connected to **journal** or **kmsg** (or their combinations with console output, see below) automatically acquire dependencies of type *After=* on systemd−journald.socket.
- Units using *LogNamespace=* will automatically gain ordering and requirement dependencies on the two socket units associated with systemd−journald@.service instances.

PATHS

The following settings may be used to change a service's view of the filesystem. Please note that the paths must be absolute and must not contain a ".." path component.

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=*/*RootImage=* 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 *MountAPIVFS=* and *PrivateUsers=* settings are particularly useful in conjunction with *RootDirectory=*. For details, see below.

If *RootDirectory=*/*RootImage=* are used together with *NotifyAccess=* the notification socket is automatically mounted from the host into the root environment, to ensure the notification interface can work correctly.

Note that services using *RootDirectory=*/*RootImage=* will not be able to log via the syslog or journal protocols to the host logging infrastructure, unless the relevant sockets are mounted from the host, specifically:

Example 1. Mounting logging sockets into root environment

BindReadOnlyPaths=/dev/log /run/systemd/journal/socket /run/systemd/journal/stdout

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RootImage=

Takes a path to a block device node or regular file as argument. This call is similar to *RootDirectory=* however mounts a file system hierarchy from a block device node or loopback file instead of a directory. The device node or file system image file needs to contain a file system without a partition table, or a file system within an MBR/MS−DOS or GPT partition table with only a single Linux−compatible partition, or a set of file systems within a GPT partition table that follows the **Discoverable Partitions Specification**[1].

When *DevicePolicy=* is set to "closed" or "strict", or set to "auto" and *DeviceAllow=* is set, then this setting adds /dev/loop−control with **rw** mode, "block−loop" and "block−blkext" with **rwm** mode to *DeviceAllow=*. See **systemd.resource-control**(5) for the details about *DevicePolicy=* or *DeviceAllow=*. Also, see *PrivateDevices=* below, as it may change the setting of *DevicePolicy=*.

Units making use of *RootImage=* automatically gain an *After=* dependency on systemd−udevd.service.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RootImageOptions=

Takes a comma−separated list of mount options that will be used on disk images specified by *RootImage=*. Optionally a partition name can be prefixed, followed by colon, in case the image has multiple partitions, otherwise partition name "root" is implied. Options for multiple partitions can be specified in a single line with space separators. Assigning an empty string removes previous assignments. Duplicated options are ignored. For a list of valid mount options, please refer to **mount**(8).

Valid partition names follow the **Discoverable Partitions Specification**[1]: **root**, **usr**, **home**, **srv**, **esp**, **xbootldr**, **tmp**, **var**.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RootHash=

Takes a data integrity (dm−verity) root hash specified in hexadecimal, or the path to a file containing a root hash in ASCII hexadecimal format. This option enables data integrity checks using dm−verity, if the used image contains the appropriate integrity data (see above) or if *RootVerity=* is used. The specified hash must match the root hash of integrity data, and is usually at least 256 bits (and hence 64 formatted hexadecimal characters) long (in case of SHA256 for example). If this option is not specified, but the image file carries the "user.verity.roothash" extended file attribute (see **xattr**(7)), then the root hash is read from it, also as formatted hexadecimal characters. If the extended file attribute is not found (or is not supported by the underlying file system), but a file with the .roothash

suffix is found next to the image file, bearing otherwise the same name (except if the image has the .raw suffix, in which case the root hash file must not have it in its name), the root hash is read from it and automatically used, also as formatted hexadecimal characters.

If the disk image contains a separate /usr/ partition it may also be Verity protected, in which case the root hash may configured via an extended attribute "user.verity.usrhash" or a .usrhash file adjacent to the disk image. There's currently no option to configure the root hash for the /usr/ file system via the unit file directly.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RootHashSignature=

Takes a PKCS7 signature of the *RootHash=* option as a path to a DER−encoded signature file, or as an ASCII base64 string encoding of a DER−encoded signature prefixed by "base64:". The dm−verity volume will only be opened if the signature of the root hash is valid and signed by a public key present in the kernel keyring. If this option is not specified, but a file with the .roothash.p7s suffix is found next to the image file, bearing otherwise the same name (except if the image has the .raw suffix, in which case the signature file must not have it in its name), the signature is read from it and automatically used.

If the disk image contains a separate /usr/ partition it may also be Verity protected, in which case the signature for the root hash may configured via a .usrhash.p7s file adjacent to the disk image. There's currently no option to configure the root hash signature for the /usr/ via the unit file directly.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RootVerity=

Takes the path to a data integrity (dm−verity) file. This option enables data integrity checks using dm−verity, if *RootImage=* is used and a root−hash is passed and if the used image itself does not contains the integrity data. The integrity data must be matched by the root hash. If this option is not specified, but a file with the .verity suffix is found next to the image file, bearing otherwise the same name (except if the image has the .raw suffix, in which case the verity data file must not have it in its name), the verity data is read from it and automatically used.

This option is supported only for disk images that contain a single file system, without an enveloping partition table. Images that contain a GPT partition table should instead include both root file system and matching Verity data in the same image, implementing the **Discoverable Partitions Specification**[1].

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

MountAPIVFS=

Takes a boolean argument. If on, a private mount namespace for the unit's processes is created and the API file systems /proc/, /sys/, /dev/ and /run/ (as an empty "tmpfs") are mounted inside of it, unless they are already mounted. Note that this option has no effect unless used in conjunction with *RootDirectory=*/*RootImage=* as these four mounts are generally mounted in the host anyway, and unless the root directory is changed, the private mount namespace will be a 1:1 copy of the host's, and include these four mounts. Note that the /dev/ file system of the host is bind mounted if this option is used without *PrivateDevices=*. To run the service with a private, minimal version of /dev/, combine this option with *PrivateDevices=*.

In order to allow propagating mounts at runtime in a safe manner, /run/systemd/propagate on the host will be used to set up new mounts, and /run/host/incoming/ in the private namespace will be used as an

intermediate step to store them before being moved to the final mount point.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

ProtectProc=

Takes one of "noaccess", "invisible", "ptraceable" or "default" (which it defaults to). When set, this controls the "hidepid=" mount option of the "procfs" instance for the unit that controls which directories with process metainformation (/proc/*PID*) are visible and accessible: when set to "noaccess" the ability to access most of other users' process metadata in /proc/ is taken away for processes of the service. When set to "invisible" processes owned by other users are hidden from /proc/. If "ptraceable" all processes that cannot be **ptrace()**'ed by a process are hidden to it. If "default" no restrictions on /proc/ access or visibility are made. For further details see **The /proc Filesystem**[2]. It is generally recommended to run most system services with this option set to "invisible". This option is implemented via file system namespacing, and thus cannot be used with services that shall be able to install mount points in the host file system hierarchy. Note that the root user is unaffected by this option, so to be effective it has to be used together with *User=* or *DynamicUser=yes*, and also without the "CAP_SYS_PTRACE" capability, which also allows a process to bypass this feature. It cannot be used for services that need to access metainformation about other users' processes. This option implies *MountAPIVFS=*.

If the kernel doesn't support per−mount point **hidepid=** mount options this setting remains without effect, and the unit's processes will be able to access and see other process as if the option was not used.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

ProcSubset=

Takes one of "all" (the default) and "pid". If "pid", all files and directories not directly associated with process management and introspection are made invisible in the /proc/ file system configured for the unit's processes. This controls the "subset=" mount option of the "procfs" instance for the unit. For further details see **The /proc Filesystem**[2]. Note that Linux exposes various kernel APIs via /proc/, which are made unavailable with this setting. Since these APIs are used frequently this option is useful only in a few, specific cases, and is not suitable for most non−trivial programs.

Much like *ProtectProc=* above, this is implemented via file system mount namespacing, and hence the same restrictions apply: it is only available to system services, it disables mount propagation to the host mount table, and it implies *MountAPIVFS=*. Also, like *ProtectProc=* this setting is gracefully disabled if the used kernel does not support the "subset=" mount option of "procfs".

BindPaths=, *BindReadOnlyPaths=*

Configures unit−specific bind mounts. A bind mount makes a particular file or directory available at an additional place in the unit's view of the file system. Any bind mounts created with this option are specific to the unit, and are not visible in the host's mount table. This option expects a whitespace separated list of bind mount definitions. Each definition consists of a colon−separated triple of source path, destination path and option string, where the latter two are optional. If only a source path is specified the source and destination is taken to be the same. The option string may be either "rbind" or "norbind" for configuring a recursive or non−recursive bind mount. If the destination path is omitted, the option string must be omitted too. Each bind mount definition may be prefixed with "−", in which case it will be ignored when its source path does not exist.

BindPaths= creates regular writable bind mounts (unless the source file system mount is already marked read−only), while *BindReadOnlyPaths=* creates read−only bind mounts. These settings may be used more than once, each usage appends to the unit's list of bind mounts. If the empty string is assigned to either of these two options the entire list of bind mounts defined prior to this is reset. Note

that in this case both read−only and regular bind mounts are reset, regardless which of the two settings is used.

This option is particularly useful when *RootDirectory=*/*RootImage=* is used. In this case the source path refers to a path on the host file system, while the destination path refers to a path below the root directory of the unit.

Note that the destination directory must exist or systemd must be able to create it. Thus, it is not possible to use those options for mount points nested underneath paths specified in *InaccessiblePaths=*, or under /home/ and other protected directories if *ProtectHome=yes* is specified. *TemporaryFileSystem=* with ":ro" or *ProtectHome=tmpfs* should be used instead.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

MountImages=

This setting is similar to *RootImage=* in that it mounts a file system hierarchy from a block device node or loopback file, but the destination directory can be specified as well as mount options. This option expects a whitespace separated list of mount definitions. Each definition consists of a colon−separated tuple of source path and destination definitions, optionally followed by another colon and a list of mount options.

Mount options may be defined as a single comma−separated list of options, in which case they will be implicitly applied to the root partition on the image, or a series of colon−separated tuples of partition name and mount options. Valid partition names and mount options are the same as for *RootImageOptions=* setting described above.

Each mount definition may be prefixed with "−", in which case it will be ignored when its source path does not exist. The source argument is a path to a block device node or regular file. If source or destination contain a ":", it needs to be escaped as "\:". The device node or file system image file needs to follow the same rules as specified for *RootImage=*. Any mounts created with this option are specific to the unit, and are not visible in the host's mount table.

These settings may be used more than once, each usage appends to the unit's list of mount paths. If the empty string is assigned, the entire list of mount paths defined prior to this is reset.

Note that the destination directory must exist or systemd must be able to create it. Thus, it is not possible to use those options for mount points nested underneath paths specified in *InaccessiblePaths=*, or under /home/ and other protected directories if *ProtectHome=yes* is specified.

When *DevicePolicy=* is set to "closed" or "strict", or set to "auto" and *DeviceAllow=* is set, then this setting adds /dev/loop−control with **rw** mode, "block−loop" and "block−blkext" with **rwm** mode to *DeviceAllow=*. See **systemd.resource-control**(5) for the details about *DevicePolicy=* or *DeviceAllow=*. Also, see *PrivateDevices=* below, as it may change the setting of *DevicePolicy=*.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

ExtensionImages=

This setting is similar to *MountImages=* in that it mounts a file system hierarchy from a block device node or loopback file, but instead of providing a destination path, an overlay will be set up. This option expects a whitespace separated list of mount definitions. Each definition consists of a source path, optionally followed by a colon and a list of mount options.

A read−only OverlayFS will be set up on top of /usr/ and /opt/ hierarchies. The order in which the images are listed will determine the order in which the overlay is laid down: images specified first to

last will result in overlayfs layers bottom to top.

Mount options may be defined as a single comma−separated list of options, in which case they will be implicitly applied to the root partition on the image, or a series of colon−separated tuples of partition name and mount options. Valid partition names and mount options are the same as for *RootImageOptions*= setting described above.

Each mount definition may be prefixed with "−", in which case it will be ignored when its source path does not exist. The source argument is a path to a block device node or regular file. If the source path contains a ":", it needs to be escaped as "\:". The device node or file system image file needs to follow the same rules as specified for *RootImage*=. Any mounts created with this option are specific to the unit, and are not visible in the host's mount table.

These settings may be used more than once, each usage appends to the unit's list of image paths. If the empty string is assigned, the entire list of mount paths defined prior to this is reset.

When *DevicePolicy*= is set to "closed" or "strict", or set to "auto" and *DeviceAllow*= is set, then this setting adds /dev/loop−control with **rw** mode, "block−loop" and "block−blkext" with **rwm** mode to *DeviceAllow*=. See **systemd.resource-control**(5) for the details about *DevicePolicy*= or *DeviceAllow*=. Also, see *PrivateDevices*= below, as it may change the setting of *DevicePolicy*=.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

USER/GROUP IDENTITY

These options are only available for system services and are not supported for services running in per−user instances of the service manager.

User=, *Group*=

Set the UNIX user or group that the processes are executed as, respectively. Takes a single user or group name, or a 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 "+".

Note that this enforces only weak restrictions on the user/group name syntax, but will generate warnings in many cases where user/group names do not adhere to the following rules: the specified name should consist only of the characters a−z, A−Z, 0−9, "_" and "−", except for the first character which must be one of a−z, A−Z and "_" (i.e. digits and "−" are not permitted as first character). The user/group name must have at least one character, and at most 31. These restrictions are made in order to avoid ambiguities and to ensure user/group names and unit files remain portable among Linux systems. For further details on the names accepted and the names warned about see **User/Group Name Syntax**[3].

When used in conjunction with *DynamicUser*= the user/group name specified is dynamically allocated at the time the service is started, and released at the time the service is stopped — unless it is already allocated statically (see below). If *DynamicUser*= is not used the specified user and group must have been created statically in the user database no later than the moment the service is started, for example using the **sysusers.d**(5) facility, which is applied at boot or package install time. If the user does not exist by then program invocation will fail.

If the *User*= setting is used the supplementary group list is initialized from the specified user's default group list, as defined in the system's user and group database. Additional groups may be configured through the *SupplementaryGroups*= setting (see below).

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. Note that if *User=* is specified and the static group with the name exists, then it is required that the static user with the name already exists. Similarly, if *Group=* is specified and the static user with the name exists, then it is required that the static group with the name already exists. 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=* and *PrivateTmp=* are implied (and cannot be turned off). 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. Furthermore *NoNewPrivileges=* and *RestrictSUIDSGID=* are implicitly enabled (and cannot be disabled), to ensure that processes invoked cannot take benefit or create SUID/SGID files or directories. 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 allow−listed 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. Use *StateDirectory=*, *CacheDirectory=* and *LogsDirectory=* in order to assign a set of writable directories for specific purposes to the service in a way that they are protected from vulnerabilities due to UID reuse (see below). If this option is enabled, care should be taken that the unit's processes do not get access to directories outside of these explicitly configured and managed ones. Specifically, do not use *BindPaths=* and be careful with **AF_UNIX** file descriptor passing for directory file descriptors, as this would permit processes to create files or directories owned by the dynamic user/group that are not subject to the lifecycle and access guarantees of the service. 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 "+".

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, and is otherwise ignored. If not set, no PAM session will be opened for the executed processes. See **pam**(8) for details.

Note that for each unit making use of this option a PAM session handler process will be maintained as part of the unit and stays around as long as the unit is active, to ensure that appropriate actions can be

taken when the unit and hence the PAM session terminates. This process is named "(sd−pam)" and is an immediate child process of the unit's main process.

Note that when this option is used for a unit it is very likely (depending on PAM configuration) that the main unit process will be migrated to its own session scope unit when it is activated. This process will hence be associated with two units: the unit it was originally started from (and for which *PAMName=* was configured), and the session scope unit. Any child processes of that process will however be associated with the session scope unit only. This has implications when used in combination with *NotifyAccess*=**all**, as these child processes will not be able to affect changes in the original unit through notification messages. These messages will be considered belonging to the session scope unit and not the original unit. It is hence not recommended to use *PAMName=* in combination with *NotifyAccess*=**all**.

CAPABILITIES

These options are only available for system services and are not supported for services running in per−user instances of the service manager.

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 by **OR**, or by **AND** if the lines are prefixed with "~" (see below). 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 "+".

Use **systemd-analyze**(1)'s **capability** command to retrieve a list of capabilities defined on the local system.

Example: if a unit has the following,

CapabilityBoundingSet=CAP_A CAP_B
CapabilityBoundingSet=CAP_B CAP_C

then **CAP_A**, **CAP_B**, and **CAP_C** are set. If the second line is prefixed with "~", e.g.,

CapabilityBoundingSet=CAP_A CAP_B
CapabilityBoundingSet=~CAP_B CAP_C

then, only **CAP_A** is set.

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 (see the above examples in *CapabilityBoundingSet=*). 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 "+".

SECURITY

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 certain settings override this and ignore the value of this setting. This is the case when *DynamicUser=*, *LockPersonality=*, *MemoryDenyWriteExecute=*, *PrivateDevices=*, *ProtectClock=*, *ProtectHostname=*, *ProtectKernelLogs=*, *ProtectKernelModules=*, *ProtectKernelTunables=*, *RestrictAddressFamilies=*, *RestrictNamespaces=*, *RestrictRealtime=*, *RestrictSUIDSGID=*, *SystemCallArchitectures=*, *SystemCallFilter=*, or *SystemCallLog=* are specified. Note that even if this setting is overridden by them, **systemctl show** shows the original value of this setting. In case the service will be run in a new mount namespace anyway and SELinux is disabled, all file systems are mounted with **MS_NOSUID** flag. Also see **No New Privileges Flag**[4].

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.

MANDATORY ACCESS CONTROL

These options are only available for system services and are not supported for services running in per−user instances of the service manager.

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. If prefixed by "−", all errors will be ignored. This setting has no effect if AppArmor is not enabled. This setting 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 "+".

PROCESS PROPERTIES

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, *MemoryMax=* is a more powerful (and working) replacement for *LimitRSS=*.

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 below).

For system units these resource limits may be chosen freely. When these settings are configured in a user service (i.e. a service run by the per−user instance of the service manager) they cannot be used to raise the limits above those set for the user manager itself when it was first invoked, as the user's service manager generally lacks the privileges to do so. In user context these configuration options are hence only useful to lower the limits passed in or to raise the soft limit to the maximum of the hard limit as configured for the user. To raise the user's limits further, the available configuration mechanisms differ between operating systems, but typically require privileges. In most cases it is possible to configure higher per−user resource limits via PAM or by setting limits on the system service encapsulating the user's service manager, i.e. the user's instance of user@.service. After making such changes, make sure to restart the user's service manager.

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

UMask=

Controls the file mode creation mask. Takes an access mode in octal notation. See **umask**(2) for details. Defaults to 0022 for system units. For user units the default value is inherited from the per−user service manager (whose default is in turn inherited from the system service manager, and thus typically also is 0022 — unless overridden by a PAM module). In order to change the per−user mask for all user services, consider setting the *UMask=* setting of the user's user@.service system service instance. The per−user umask may also be set via the *umask* field of a user's **JSON User Record**[5] (for users managed by **systemd-homed.service**(8) this field may be controlled via **homectl −−umask=**). It may also be set via a PAM module, such as **pam_umask**(8).

CoredumpFilter=

Controls which types of memory mappings will be saved if the process dumps core (using the /proc/*pid*/coredump_filter file). Takes a whitespace−separated combination of mapping type names or numbers (with the default base 16). Mapping type names are **private−anonymous**, **shared−anonymous**, **private−file−backed**, **shared−file−backed**, **elf−headers**, **private−huge**, **shared−huge**, **private−dax**, **shared−dax**, and the special values **all** (all types) and **default** (the kernel default of "**private−anonymous shared−anonymous elf−headers private−huge**"). See **core**(5) for the meaning of the mapping types. When specified multiple times, all specified masks are ORed. When not set, or if the empty value is assigned, the inherited value is not changed.

Example 2. Add DAX pages to the dump filter

```
CoredumpFilter=default private−dax shared−dax
```

KeyringMode=

Controls how the kernel session keyring is set up for the service (see **session-keyring**(7) for details on the session keyring). Takes one of **inherit**, **private**, **shared**. If set to **inherit** no special keyring setup is done, and the kernel's default behaviour is applied. If **private** is used a new session keyring is allocated when a service process is invoked, and it is not linked up with any user keyring. This is the recommended setting for system services, as this ensures that multiple services running under the same system user ID (in particular the root user) do not share their key material among each other. If **shared** is used a new session keyring is allocated as for **private**, but the user keyring of the user configured with *User=* is linked into it, so that keys assigned to the user may be requested by the unit's processes. In this modes multiple units running processes under the same user ID may share key material. Unless **inherit** is selected the unique invocation ID for the unit (see below) is added as a

protected key by the name "invocation_id" to the newly created session keyring. Defaults to **private** for services of the system service manager and to **inherit** for non−service units and for services of the user service manager.

OOMScoreAdjust=
Sets the adjustment value for the Linux kernel's Out−Of−Memory (OOM) killer score for executed processes. Takes an integer between −1000 (to disable OOM killing of processes of this unit) and 1000 (to make killing of processes of this unit under memory pressure very likely). See **proc.txt**[6] for details. If not specified defaults to the OOM score adjustment level of the service manager itself, which is normally at 0.

Use the *OOMPolicy=* setting of service units to configure how the service manager shall react to the kernel OOM killer terminating a process of the service. See **systemd.service**(5) for details.

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.

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.

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.

SCHEDULING

Nice=
Sets the default nice level (scheduling priority) for executed processes. Takes an integer between −20 (highest priority) and 19 (lowest priority). In case of resource contention, smaller values mean more resources will be made available to the unit's processes, larger values mean less resources will be made available. See **setpriority**(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. In case of CPU resource contention, smaller values mean less CPU time is made available to the service, larger values mean more. 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 call **fork**(2), 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. Alternatively, takes a special "numa" value in which case systemd

automatically derives allowed CPU range based on the value of *NUMAMask=* option. 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.

NUMAPolicy=

Controls the NUMA memory policy of the executed processes. Takes a policy type, one of: **default**, **preferred**, **bind**, **interleave** and **local**. A list of NUMA nodes that should be associated with the policy must be specified in *NUMAMask=*. For more details on each policy please see, **set_mempolicy**(2). For overall overview of NUMA support in Linux see, **numa**(7).

NUMAMask=

Controls the NUMA node list which will be applied alongside with selected NUMA policy. Takes a list of NUMA nodes and has the same syntax as a list of CPUs for *CPUAffinity=* option or special "all" value which will include all available NUMA nodes in the mask. Note that the list of NUMA nodes is not required for **default** and **local** policies and for **preferred** policy we expect a single NUMA node.

IOSchedulingClass=

Sets the I/O scheduling class for executed processes. Takes one of the strings **realtime**, **best−effort** or **idle**. The kernel's default scheduling class is **best−effort** at a priority of 4. If the empty string is assigned to this option, all prior assignments to both *IOSchedulingClass=* and *IOSchedulingPriority=* have no effect. 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). In case of I/O contention, smaller values mean more I/O bandwidth is made available to the unit's processes, larger values mean less bandwidth. The available priorities depend on the selected I/O scheduling class (see above). If the empty string is assigned to this option, all prior assignments to both *IOSchedulingClass=* and *IOSchedulingPriority=* have no effect. For the kernel's default scheduling class (**best−effort**) this defaults to 4. See **ioprio_set**(2) for details.

SANDBOXING

The following sandboxing options are an effective way to limit the exposure of the system towards the unit's processes. It is recommended to turn on as many of these options for each unit as is possible without negatively affecting the process' ability to operate. Note that many of these sandboxing features are gracefully turned off on systems where the underlying security mechanism is not available. For example, *ProtectSystem=* has no effect if the kernel is built without file system namespacing or if the service manager runs in a container manager that makes file system namespacing unavailable to its payload. Similar, *RestrictRealtime=* has no effect on systems that lack support for SECCOMP system call filtering, or in containers where support for this is turned off.

Also note that some sandboxing functionality is generally not available in user services (i.e. services run by the per−user service manager). Specifically, the various settings requiring file system namespacing support (such as *ProtectSystem=*) are not available, as the underlying kernel functionality is only accessible to privileged processes. However, most namespacing settings, that will not work on their own in user services, will work when used in conjunction with *PrivateUsers=***true**.

ProtectSystem=

Takes a boolean argument or the special values "full" or "strict". If true, mounts the /usr/ and the boot loader directories (/boot and /efi) 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. This setting cannot ensure protection in all cases. In general it has the same limitations as *ReadOnlyPaths=*, see below. Defaults to off.

ProtectHome=
Takes a boolean argument or the special values "read−only" or "tmpfs". 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. If set to "tmpfs", temporary file systems are mounted on the three directories in read−only mode. The value "tmpfs" is useful to hide home directories not relevant to the processes invoked by the unit, while still allowing necessary directories to be made visible when listed in *BindPaths=* or *BindReadOnlyPaths=*.

Setting this to "yes" is mostly equivalent to set the three directories in *InaccessiblePaths=*. Similarly, "read−only" is mostly equivalent to *ReadOnlyPaths=*, and "tmpfs" is mostly equivalent to *TemporaryFileSystem=* with ":ro".

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. This setting cannot ensure protection in all cases. In general it has the same limitations as *ReadOnlyPaths=*, see below.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RuntimeDirectory=, *StateDirectory=*, *CacheDirectory=*, *LogsDirectory=*, *ConfigurationDirectory=*
These options take a whitespace−separated list of directory names. The specified directory names must be relative, and may not include "..". If set, when the unit is started, one or more directories by the specified names will be created (including their parents) below the locations defined in the following table. Also, the corresponding environment variable will be defined with the full paths of the directories. If multiple directories are set, then in the environment variable the paths are concatenated with colon (":").

Table 2. Automatic directory creation and environment variables

Directory	Below path for system units	Below path for user units	Environment variable set
RuntimeDirectory=	/run/	*$XDG_RUNTIME_DIR*	*$RUNTIME_DIRECTORY*
StateDirectory=	/var/lib/	*$XDG_CONFIG_HOME*	*$STATE_DIRECTORY*
CacheDirectory=	/var/cache/	*$XDG_CACHE_HOME*	*$CACHE_DIRECTORY*
LogsDirectory=	/var/log/	*$XDG_CONFIG_HOME*/log/	*$LOGS_DIRECTORY*
ConfigurationDirectory=	/etc/	*$XDG_CONFIG_HOME*	*$CONFIGURATION_DIRECTO*

In case of *RuntimeDirectory=* the innermost subdirectories are removed when the unit is stopped. It is possible to preserve the specified directories in this case if *RuntimeDirectoryPreserve=* is configured to **restart** or **yes** (see below). The directories specified with *StateDirectory=*, *CacheDirectory=*, *LogsDirectory=*, *ConfigurationDirectory=* are not removed when the unit is stopped.

Except in case of *ConfigurationDirectory=*, the innermost specified directories will be owned by the user and group specified in *User=* and *Group=*. If the specified directories already exist and their owning user or group do not match the configured ones, all files and directories below the specified directories as well as the directories themselves will have their file ownership recursively changed to match what is configured. As an optimization, if the specified directories are already owned by the right user and group, files and directories below of them are left as−is, even if they do not match what is requested. The innermost specified directories will have their access mode adjusted to the what is specified in *RuntimeDirectoryMode=*, *StateDirectoryMode=*, *CacheDirectoryMode=*, *LogsDirectoryMode=* and *ConfigurationDirectoryMode=*.

These options imply *BindPaths=* for the specified paths. When combined with *RootDirectory=* or *RootImage=* these paths always reside on the host and are mounted from there into the unit's file

system namespace.

If *DynamicUser=* is used, the logic for *CacheDirectory=*, *LogsDirectory=* and *StateDirectory=* is slightly altered: the directories are created below /var/cache/private, /var/log/private and /var/lib/private, respectively, which are host directories made inaccessible to unprivileged users, which ensures that access to these directories cannot be gained through dynamic user ID recycling. Symbolic links are created to hide this difference in behaviour. Both from perspective of the host and from inside the unit, the relevant directories hence always appear directly below /var/cache, /var/log and /var/lib.

Use *RuntimeDirectory=* to manage one or more runtime directories for the unit and bind their lifetime to the daemon runtime. 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).

The directories defined by these options are always created under the standard paths used by systemd (/var/, /run/, /etc/, ...). If the service needs directories in a different location, a different mechanism has to be used to create them.

tmpfiles.d(5) provides functionality that overlaps with these options. Using these options is recommended, because the lifetime of the directories is tied directly to the lifetime of the unit, and it is not necessary to ensure that the tmpfiles.d configuration is executed before the unit is started.

To remove any of the directories created by these settings, use the **systemctl clean ...** command on the relevant units, see **systemctl**(1) for details.

Example: if a system service unit has the following,

RuntimeDirectory=foo/bar baz

the service manager creates /run/foo (if it does not exist), /run/foo/bar, and /run/baz. The directories /run/foo/bar and /run/baz except /run/foo are owned by the user and group specified in *User=* and *Group=*, and removed when the service is stopped.

Example: if a system service unit has the following,

RuntimeDirectory=foo/bar
StateDirectory=aaa/bbb ccc

then the environment variable "RUNTIME_DIRECTORY" is set with "/run/foo/bar", and "STATE_DIRECTORY" is set with "/var/lib/aaa/bbb:/var/lib/ccc".

RuntimeDirectoryMode=, *StateDirectoryMode=*, *CacheDirectoryMode=*, *LogsDirectoryMode=*, *ConfigurationDirectoryMode=*

Specifies the access mode of the directories specified in *RuntimeDirectory=*, *StateDirectory=*, *CacheDirectory=*, *LogsDirectory=*, or *ConfigurationDirectory=*, respectively, as an octal number. Defaults to **0755**. See "Permissions" in **path_resolution**(7) for a discussion of the meaning of permission bits.

RuntimeDirectoryPreserve=

Takes a boolean argument or **restart**. If set to **no** (the default), the directories specified in *RuntimeDirectory=* are always removed when the service stops. If set to **restart** the directories are preserved when the service is both automatically and manually restarted. Here, the automatic restart means the operation specified in *Restart=*, and manual restart means the one triggered by **systemctl restart foo.service**. If set to **yes**, then the directories are not removed when the service is stopped. Note that since the runtime directory /run/ is a mount point of "tmpfs", then for system services the

directories specified in *RuntimeDirectory=* are removed when the system is rebooted.

TimeoutCleanSec=
Configures a timeout on the clean−up operation requested through **systemctl clean ...**, see **systemctl**(1) for details. Takes the usual time values and defaults to **infinity**, i.e. by default no timeout is applied. If a timeout is configured the clean operation will be aborted forcibly when the timeout is reached, potentially leaving resources on disk.

ReadWritePaths=, *ReadOnlyPaths=*, *InaccessiblePaths=*, *ExecPaths=*, *NoExecPaths=*
Sets up a new file system namespace for executed processes. These options may be used to limit access a process has to the file system. 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=*/*RootImage=*.

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 allow−list 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. This may be more restrictive than desired, because it is not possible to nest *ReadWritePaths=*, *ReadOnlyPaths=*, *BindPaths=*, or *BindReadOnlyPaths=* inside it. For a more flexible option, see *TemporaryFileSystem=*.

Content in paths listed in *NoExecPaths=* are not executable even if the usual file access controls would permit this. Nest *ExecPaths=* inside of *NoExecPaths=* in order to provide executable content within non−executable directories.

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=*, *InaccessiblePaths=*, *ExecPaths=* and *NoExecPaths=* may be prefixed with "−", in which case they will be ignored when they do not exist. If prefixed with "+" the paths are taken relative to the root directory of the unit, as configured with *RootDirectory=*/*RootImage=*, instead of relative to the root directory of the host (see above). When combining "−" and "+" on the same path make sure to specify "−" first, and "+" second.

Note that these settings will disconnect propagation of mounts from the unit's processes to the host. This means that this setting may not be used for services which shall be able to install mount points in the main mount namespace. For *ReadWritePaths=* and *ReadOnlyPaths=* propagation in the other direction is not affected, i.e. mounts created on the host generally appear in the unit processes' namespace, and mounts removed on the host also disappear there too. In particular, note that mount propagation from host to unit will result in unmodified mounts to be created in the unit's namespace, i.e. writable mounts appearing on the host will be writable in the unit's namespace too, even when propagated below a path marked with *ReadOnlyPaths=*! Restricting access with these options hence does not extend to submounts of a directory that are created later on. This means the lock−down offered by that setting is not complete, and does not offer full protection.

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*.

Simple allow−list example using these directives:

```
[Service]
ReadOnlyPaths=/
ReadWritePaths=/var /run
InaccessiblePaths=-/lost+found
NoExecPaths=/
ExecPaths=/usr/sbin/my_daemon /lib /lib64
```

These options are only available for system services and are not supported for services running in per−user instances of the service manager.

TemporaryFileSystem=

Takes a space−separated list of mount points for temporary file systems (tmpfs). If set, a new file system namespace is set up for executed processes, and a temporary file system is mounted on each mount point. This option may be specified more than once, in which case temporary file systems are mounted on all listed mount points. If the empty string is assigned to this option, the list is reset, and all prior assignments have no effect. Each mount point may optionally be suffixed with a colon (":") and mount options such as "size=10%" or "ro". By default, each temporary file system is mounted with "nodev,strictatime,mode=0755". These can be disabled by explicitly specifying the corresponding mount options, e.g., "dev" or "nostrictatime".

This is useful to hide files or directories not relevant to the processes invoked by the unit, while necessary files or directories can be still accessed by combining with *BindPaths=* or *BindReadOnlyPaths=*:

Example: if a unit has the following,

```
TemporaryFileSystem=/var:ro
BindReadOnlyPaths=/var/lib/systemd
```

then the invoked processes by the unit cannot see any files or directories under /var/ except for /var/lib/systemd or its contents.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

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 are 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 true, 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. Enabling this setting has the side effect of adding *Requires=* and *After=* dependencies on all mount units necessary to access /tmp/ and /var/tmp/. Moreover an implicitly *After=* ordering on **systemd-tmpfiles-setup.service**(8) is added.

Note that the implementation of this setting might be impossible (for example if mount namespaces are not available), and the unit should be written in a way that does not solely rely on this setting for security.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

PrivateDevices=

Takes a boolean argument. If true, sets up a new /dev/ mount 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 new /dev/ 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**. For this setting the same restrictions regarding mount propagation and privileges apply as for *ReadOnlyPaths=* and related calls, see above. If turned on and if running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied.

Note that the implementation of this setting might be impossible (for example if mount namespaces are not available), and the unit should be written in a way that does not solely rely on this setting for security.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

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 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, including **AF_NETLINK** and **AF_UNIX**. Effectively, for **AF_NETLINK** this means that device configuration events received from **systemd-udevd.service**(8) are not delivered to the unit's processes. And for **AF_UNIX** this has the effect that **AF_UNIX** sockets in the abstract socket namespace of the host will become unavailable to the unit's processes (however, those located in the file system will continue to be accessible).

Note that the implementation of this setting might be impossible (for example if network namespaces are not available), and the unit should be written in a way that does not solely rely on this setting for security.

When this option is used on a socket unit any sockets bound on behalf of this unit will be bound within a private network namespace. This may be combined with *JoinsNamespaceOf=* to listen on sockets inside of network namespaces of other services.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

NetworkNamespacePath=

Takes an absolute file system path refererring to a Linux network namespace pseudo−file (i.e. a file like /proc/$PID/ns/net or a bind mount or symlink to one). When set the invoked processes are added to the network namespace referenced by that path. The path has to point to a valid namespace file at the moment the processes are forked off. If this option is used *PrivateNetwork=* has no effect. If this option is used together with *JoinsNamespaceOf=* then it only has an effect if this unit is started before any of the listed units that have *PrivateNetwork=* or *NetworkNamespacePath=* configured, as otherwise the network namespace of those units is reused.

When this option is used on a socket unit any sockets bound on behalf of this unit will be bound within the specified network namespace.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

PrivateIPC=
Takes a boolean argument. If true, sets up a new IPC namespace for the executed processes. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue file system. This is useful to avoid name clash of IPC identifiers. Defaults to false. It is possible to run two or more units within the same private IPC namespace by using the *JoinsNamespaceOf=* directive, see **systemd.unit**(5) for details.

Note that IPC namespacing does not have an effect on **AF_UNIX** sockets, which are the most common form of IPC used on Linux. Instead, **AF_UNIX** sockets in the file system are subject to mount namespacing, and those in the abstract namespace are subject to network namespacing. IPC namespacing only has an effect on SysV IPC (which is mostly legacy) as well as POSIX message queues (for which **AF_UNIX**/**SOCK_SEQPACKET** sockets are typically a better replacement). IPC namespacing also has no effect on POSIX shared memory (which is subject to mount namespacing) either. See **ipc_namespaces**(7) for the details.

Note that the implementation of this setting might be impossible (for example if IPC namespaces are not available), and the unit should be written in a way that does not solely rely on this setting for security.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

IPCNamespacePath=
Takes an absolute file system path refererring to a Linux IPC namespace pseudo−file (i.e. a file like /proc/$PID/ns/ipc or a bind mount or symlink to one). When set the invoked processes are added to the network namespace referenced by that path. The path has to point to a valid namespace file at the moment the processes are forked off. If this option is used *PrivateIPC=* has no effect. If this option is used together with *JoinsNamespaceOf=* then it only has an effect if this unit is started before any of the listed units that have *PrivateIPC=* or *IPCNamespacePath=* configured, as otherwise the network namespace of those units is reused.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

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.

When this setting is set up by a per−user instance of the service manager, the mapping of the "root" user and group to itself is omitted (unless the user manager is root). Additionally, in the per−user

instance manager case, the user namespace will be set up before most other namespaces. This means that combining *PrivateUsers=***true** with other namespaces will enable use of features not normally supported by the per−user instances of the service manager.

This setting is particularly useful in conjunction with *RootDirectory=*/*RootImage=*, 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.

Note that the implementation of this setting might be impossible (for example if user namespaces are not available), and the unit should be written in a way that does not solely rely on this setting for security.

ProtectHostname=

Takes a boolean argument. When set, sets up a new UTS namespace for the executed processes. In addition, changing hostname or domainname is prevented. Defaults to off.

Note that the implementation of this setting might be impossible (for example if UTS namespaces are not available), and the unit should be written in a way that does not solely rely on this setting for security.

Note that when this option is enabled for a service hostname changes no longer propagate from the system into the service, it is hence not suitable for services that need to take notice of system hostname changes dynamically.

If this setting is on, but the unit doesn't have the **CAP_SYS_ADMIN** capability (e.g. services for which *User=* is set), *NoNewPrivileges=yes* is implied.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

ProtectClock=

Takes a boolean argument. If set, writes to the hardware clock or system clock will be denied. It is recommended to turn this on for most services that do not need modify the clock. Defaults to off. Enabling this option removes **CAP_SYS_TIME** and **CAP_WAKE_ALARM** from the capability bounding set for this unit, installs a system call filter to block calls that can set the clock, and *DeviceAllow=char−rtc r* is implied. This ensures /dev/rtc0, /dev/rtc1, etc. are made read−only to the service. See **systemd.resource-control**(5) for the details about *DeviceAllow=*. If this setting is on, but the unit doesn't have the **CAP_SYS_ADMIN** capability (e.g. services for which *User=* is set), *NoNewPrivileges=yes* is implied.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

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 be initialized only at boot−time, for example with the **sysctl.d**(5) mechanism. Few 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. If this setting is on, but the unit doesn't have the **CAP_SYS_ADMIN** capability (e.g. services for which *User=* is set), *NoNewPrivileges=yes* is implied. Note that this option does not prevent indirect changes to kernel tunables effected by IPC calls to other processes. However, *InaccessiblePaths=* may be used to make relevant IPC file system objects inaccessible. If *ProtectKernelTunables=* is set, *MountAPIVFS=yes* is implied.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

ProtectKernelModules=
Takes a boolean argument. If true, explicit module loading will be denied. This allows module load and unload operations to be turned off 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. Defaults 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. If this setting is on, but the unit doesn't have the **CAP_SYS_ADMIN** capability (e.g. services for which *User=* is set), *NoNewPrivileges=yes* is implied.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

ProtectKernelLogs=
Takes a boolean argument. If true, access to the kernel log ring buffer will be denied. It is recommended to turn this on for most services that do not need to read from or write to the kernel log ring buffer. Enabling this option removes **CAP_SYSLOG** from the capability bounding set for this unit, and installs a system call filter to block the **syslog**(2) system call (not to be confused with the libc API **syslog**(3) for userspace logging). The kernel exposes its log buffer to userspace via /dev/kmsg and /proc/kmsg. If enabled, these are made inaccessible to all the processes in the unit. If this setting is on, but the unit doesn't have the **CAP_SYS_ADMIN** capability (e.g. services for which *User=* is set), *NoNewPrivileges=yes* is implied.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

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. If *ProtectControlGroups=* is set, *MountAPIVFS=yes* is implied.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

RestrictAddressFamilies=
Restricts the set of socket address families accessible to the processes of this unit. Takes "none", or a space−separated list of address family names to allow−list, such as **AF_UNIX**, **AF_INET** or **AF_INET6**. When "none" is specified, then all address families will be denied. When prefixed with "~" the listed address families will be applied as deny list, otherwise as allow list. 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, ppc64, ppc64−le and is ignored (but works correctly on other ABIs, including x86−64). Note that on systems supporting multiple ABIs (such as x86/x86−64) it is recommended to turn off alternative ABIs for services, so that they cannot be used to circumvent the restrictions of this option. Specifically, it is recommended to combine this option with *SystemCallArchitectures=native* or similar. If running in

user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied. By default, no restrictions apply, all address families are accessible to processes. If assigned the empty string, any previous address family 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 allow list 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 (allow−listing). By prepending the list with a single tilde character ("~") the effect may be inverted: only the listed namespace types will be made inaccessible, all unlisted ones are permitted (deny−listing). If the empty string is assigned, the default namespace restrictions are applied, which is equivalent to false. This option may appear more than once, in which case the namespace types are merged by **OR**, or by **AND** if the lines are prefixed with "~" (see examples below). 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, mips, mips−le, mips64, mips64−le, mips64−n32, mips64−le−n32, ppc64, ppc64−le, 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.

Example: if a unit has the following,

```
RestrictNamespaces=cgroup ipc
RestrictNamespaces=cgroup net
```

then **cgroup**, **ipc**, and **net** are set. If the second line is prefixed with "~", e.g.,

```
RestrictNamespaces=cgroup ipc
RestrictNamespaces=~cgroup net
```

then, only **ipc** is set.

LockPersonality=

Takes a boolean argument. If set, locks down the **personality**(2) system call so that the kernel execution domain may not be changed from the default or the personality selected with *Personality=* directive. This may be useful to improve security, because odd personality emulations may be poorly tested and source of vulnerabilities. If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied.

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) or **pkey_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 and libraries that generate program code dynamically at runtime, including JIT execution engines, executable stacks, and code "trampoline" feature of various C compilers. This option improves service security, as it makes harder for software exploits to change running code dynamically. However, the protection can be circumvented, if the service can write to a filesystem, which is not mounted with **noexec** (such as /dev/shm), or it can use **memfd_create()**. This can be prevented by making such file systems inaccessible to the service (e.g. *InaccessiblePaths=/dev/shm*) and installing further system call filters (*SystemCallFilter=~memfd_create*). Note that this feature is fully available on x86−64, and partially on x86. Specifically, the **shmat()** protection is not available on x86. Note that on systems supporting multiple ABIs (such as x86/x86−64) it is recommended to turn off alternative ABIs for services, so that they cannot be used to circumvent the restrictions of this option. Specifically, it is recommended to combine this option with *SystemCallArchitectures=native* or similar. 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. If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied. 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.

RestrictSUIDSGID=

Takes a boolean argument. If set, any attempts to set the set−user−ID (SUID) or set−group−ID (SGID) bits on files or directories will be denied (for details on these bits see **inode**(7)). If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied. As the SUID/SGID bits are mechanisms to elevate privileges, and allows users to acquire the identity of other users, it is recommended to restrict creation of SUID/SGID files to the few programs that actually require them. Note that this restricts marking of any type of file system object with these bits, including both regular files and directories (where the SGID is a different meaning than for files, see documentation). This option is implied if *DynamicUser=* is enabled. Defaults to off.

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.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

PrivateMounts=

Takes a boolean parameter. If set, the processes of this unit will be run in their own private file system (mount) namespace with all mount propagation from the processes towards the host's main file system namespace turned off. This means any file system mount points established or removed by the unit's processes will be private to them and not be visible to the host. However, file system mount points established or removed on the host will be propagated to the unit's processes. See **mount_namespaces**(7) for details on file system namespaces. Defaults to off.

When turned on, this executes three operations for each invoked process: a new **CLONE_NEWNS**

namespace is created, after which all existing mounts are remounted to **MS_SLAVE** to disable propagation from the unit's processes to the host (but leaving propagation in the opposite direction in effect). Finally, the mounts are remounted again to the propagation mode configured with *MountFlags=*, see below.

File system namespaces are set up individually for each process forked off by the service manager. Mounts established in the namespace of the process created by *ExecStartPre=* will hence be cleaned up automatically as soon as that process exits and will not be available to subsequent processes forked off for *ExecStart=* (and similar applies to the various other commands configured for units). Similarly, *JoinsNamespaceOf=* does not permit sharing kernel mount namespaces between units, it only enables sharing of the /tmp/ and /var/tmp/ directories.

Other file system namespace unit settings — *PrivateMounts=*, *PrivateTmp=*, *PrivateDevices=*, *ProtectSystem=*, *ProtectHome=*, *ReadOnlyPaths=*, *InaccessiblePaths=*, *ReadWritePaths=*, ... — also enable file system namespacing in a fashion equivalent to this option. Hence it is primarily useful to explicitly request this behaviour if none of the other settings are used.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

MountFlags=

Takes a mount propagation setting: **shared**, **slave** or **private**, which controls whether file system mount points in the file system namespaces set up for this unit's processes will receive or propagate mounts and unmounts from other file system namespaces. See **mount**(2) for details on mount propagation, and the three propagation flags in particular.

This setting only controls the *final* propagation setting in effect on all mount points of the file system namespace created for each process of this unit. Other file system namespacing unit settings (see the discussion in *PrivateMounts=* above) will implicitly disable mount and unmount propagation from the unit's processes towards the host by changing the propagation setting of all mount points in the unit's file system namespace to **slave** first. Setting this option to **shared** does not reestablish propagation in that case.

If not set – but file system namespaces are enabled through another file system namespace unit setting – **shared** mount propagation is used, but — as mentioned — as **slave** is applied first, propagation from the unit's processes to the host is still turned off.

It is not recommended to use **private** mount propagation for units, as this means temporary mounts (such as removable media) of the host will stay mounted and thus indefinitely busy in forked off processes, as unmount propagation events won't be received by the file system namespace of the unit.

Usually, it is best to leave this setting unmodified, and use higher level file system namespacing options instead, in particular *PrivateMounts=*, see above.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

SYSTEM CALL FILTERING

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 (allow−listing). (See *SystemCallErrorNumber=* below for changing the default action). If the first character of the list is "~", the effect is inverted: only the listed system calls will result in immediate process termination (deny−listing). Deny−listed system calls and system call groups may optionally be suffixed with a colon (":") and "errno" error number (between 0 and 4095) or errno name such as **EPERM**, **EACCES** or **EUCLEAN** (see **errno**(3) for a full list). This value will

be returned when a deny−listed system call is triggered, instead of terminating the processes immediately. Special setting "kill" can be used to explicitly specify killing. This value takes precedence over the one given in *SystemCallErrorNumber=*, see below. If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *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 allow−listed 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 on systems supporting multiple ABIs (such as x86/x86−64) it is recommended to turn off alternative ABIs for services, so that they cannot be used to circumvent the restrictions of this option. Specifically, it is recommended to combine this option with *SystemCallArchitectures=native* or similar.

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. allow−listing and deny−listing), 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 an allow list rule for **read()** and **write()**, and right after it add a deny list rule for **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 3. Currently predefined system call sets

Note, that as new system calls are added to the kernel, additional system calls might be added to the groups above. Contents of the sets may also change between systemd versions. In addition, the list of system calls depends on the kernel version and architecture for which systemd was compiled. Use **systemd−analyze syscall−filter** to list the actual list of system calls in each filter.

Generally, allow−listing system calls (rather than deny−listing) is the safer mode of operation. It is recommended to enforce system call allow lists for all long−running system services. Specifically, the following lines are a relatively safe basic choice for the majority of system services:

```
[Service]
SystemCallFilter=@system−service
SystemCallErrorNumber=EPERM
```

Note that various kernel system calls are defined redundantly: there are multiple system calls for executing the same operation. For example, the **pidfd_send_signal()** system call may be used to execute operations similar to what can be done with the older **kill()** system call, hence blocking the latter without the former only provides weak protection. Since new system calls are added regularly to the kernel as development progresses, keeping system call deny lists comprehensive requires constant work. It is thus recommended to use allow−listing instead, which offers the benefit that new system calls are by default implicitly blocked until the allow list is updated.

Also note that a number of system calls are required to be accessible for the dynamic linker to work. The dynamic linker is required for running most regular programs (specifically: all dynamic ELF binaries, which is how most distributions build packaged programs). This means that blocking these system calls (which include **open()**, **openat()** or **mmap()**) will make most programs typically shipped with generic distributions unusable.

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=*, *ProtectKernelLogs=*, *ProtectClock=*, *ReadOnlyPaths=*, *InaccessiblePaths=* and *ReadWritePaths=*.

SystemCallErrorNumber=

Takes an "errno" error number (between 1 and 4095) or errno name such as **EPERM**, **EACCES** or **EUCLEAN**, to return when the system call filter configured with *SystemCallFilter=* is triggered, instead of terminating the process immediately. See **errno**(3) for a full list of error codes. When this setting is not used, or when the empty string or the special setting "kill" 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**. The special identifier **native** implicitly maps to the native architecture of the system (or more precisely: 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=*), *NoNewPrivileges=yes* is implied. By default, this option is set to the empty list, i.e. no filtering is applied.

If this setting is used, processes of this unit will only be permitted to call native system calls, and system calls of the specified architectures. For the purposes of this option, the x32 architecture is treated as including x86−64 system calls. However, this setting still fulfills its purpose, as explained below, on x32.

System call filtering is not equally effective on all architectures. For example, on x86 filtering of network socket−related calls is not possible, due to ABI limitations — a limitation that x86−64 does

not have, however. On systems supporting multiple ABIs at the same time — such as x86/x86−64 — it is hence recommended to limit the set of permitted system call architectures so that secondary ABIs may not be used to circumvent the restrictions applied to the native ABI of the system. In particular, setting *SystemCallArchitectures=native* is a good choice for disabling non−native ABIs.

System call architectures may also be restricted system−wide via the *SystemCallArchitectures=* option in the global configuration. See **systemd-system.conf**(5) for details.

SystemCallLog=

Takes a space−separated list of system call names. If this setting is used, all system calls executed by the unit processes for the listed ones will be logged. If the first character of the list is "~", the effect is inverted: all system calls except the listed system calls will be logged. If running in user mode, or in system mode, but without the **CAP_SYS_ADMIN** capability (e.g. setting *User=*), *NoNewPrivileges=yes* is implied. This feature makes use of the Secure Computing Mode 2 interfaces of the kernel ('seccomp filtering') and is useful for auditing or setting up a minimal sandboxing environment. 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 "+".

ENVIRONMENT

Environment=

Sets environment variables for executed processes. Each line is unquoted using the rules described in "Quoting" section in **systemd.syntax**(7) and becomes a list of variable assignments. If you need to assign a value containing spaces or the equals sign to a variable, put quotes around the whole assignment. Variable expansion is not performed inside the strings and the "$" character has no special meaning. Specifier expansion is performed, see the "Specifiers" section in **systemd.unit**(5).

This option may be specified more than once, in which case all listed variables will be set. If the same variable is listed 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.

The names of the variables can contain ASCII letters, digits, and the underscore character. Variable names cannot be empty or start with a digit. In variable values, most characters are allowed, but non−printable characters are currently rejected.

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.

Note that environment variables are not suitable for passing secrets (such as passwords, key material, ...) to service processes. Environment variables set for a unit are exposed to unprivileged clients via D−Bus IPC, and generally not understood as being data that requires protection. Moreover, environment variables are propagated down the process tree, including across security boundaries (such as setuid/setgid executables), and hence might leak to processes that should not have access to the secret data. Use *LoadCredential=* (see below) to pass data to unit processes securely.

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 (").

C escapes[7] are supported, but not **most control characters**[8]. "\t" and "\n" can be used to insert tabs and newlines within *EnvironmentFile=*.

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. The files are read from the file system of the service manager, before any file system changes like bind mounts take place).

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 set for the system service 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 passed. If the empty string is assigned to this option, the list of environment variables to pass is reset, all prior assignments have no effect. Variables specified that are not set for the system manager will not be passed and will be silently ignored. Note that this option is only relevant for the system service manager, as system services by default do not automatically inherit any environment variables set for the service manager itself. However, in case of the user service manager all environment variables are passed to the executed processes anyway, hence this option is without effect for the user service manager.

Variables set for invoked processes due to this setting are subject to being overridden by those configured with *Environment=* or *EnvironmentFile=*.

C escapes[7] are supported, but not **most control characters**[8]. "\t" and "\n" can be used to insert tabs and newlines within *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.

UnsetEnvironment=

Explicitly unset environment variable assignments that would normally be passed from the service manager to invoked processes of this unit. Takes a space−separated list of variable names or variable assignments. This option may be specified more than once, in which case all listed variables/assignments will be unset. If the empty string is assigned to this option, the list of environment variables/assignments to unset is reset. If a variable assignment is specified (that is: a variable name, followed by "=", followed by its value), then any environment variable matching this precise assignment is removed. If a variable name is specified (that is a variable name without any following "=" or value), then any assignment matching the variable name, regardless of its value is removed. Note that the effect of *UnsetEnvironment=* is applied as final step when the environment list

passed to executed processes is compiled. That means it may undo assignments from any configuration source, including assignments made through *Environment=* or *EnvironmentFile=*, inherited from the system manager's global set of environment variables, inherited via *PassEnvironment=*, set by the service manager itself (such as *$NOTIFY_SOCKET* and such), or set by a PAM module (in case *PAMName=* is used).

See "Environment Variables in Spawned Processes" below for a description of how those settings combine to form the inherited environment. See **environ**(7) for general information about environment variables.

LOGGING AND STANDARD INPUT/OUTPUT

StandardInput=

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

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 **data** option may be used to configure arbitrary textual or binary data to pass via standard input to the executed process. The data to pass is configured via *StandardInputText=*/*StandardInputData=* (see below). Note that the actual file descriptor type passed (memory file, regular file, UNIX pipe, ...) might depend on the kernel and available privileges. In any case, the file descriptor is read−only, and when read returns the specified data followed by EOF.

The **file:***path* option may be used to connect a specific file system object to standard input. An absolute path following the ":" character is expected, which may refer to a regular file, a FIFO or special file. If an **AF_UNIX** socket in the file system is specified, a stream socket is connected to it. The latter is useful for connecting standard input of processes to arbitrary system services.

The **socket** option is valid in socket−activated services only, and requires the relevant socket unit file (see **systemd.socket**(5) for details) to have *Accept=yes* set, or to specify 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) socket activation daemon.

The **fd:***name* option connects standard input to a specific, named file descriptor provided by a socket unit. The name may be specified as part of this option, following a ":" character (e.g. "fd:foobar"). If no name is specified, the name "stdin" is implied (i.e. "fd" is equivalent to "fd:stdin"). At least one socket unit defining the specified name must be provided via the *Sockets=* option, and the 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 file descriptors and their ordering.

This setting defaults to **null**, unless *StandardInputText=*/*StandardInputData=* are set, in which case it

defaults to **data**.

StandardOutput=

Controls where file descriptor 1 (stdout) of the executed processes is connected to. Takes one of **inherit**, **null**, **tty**, **journal**, **kmsg**, **journal+console**, **kmsg+console**, **file:***path*, **append:***path*, **truncate:***path*, **socket** or **fd:***name*.

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 kmsg (see below) is implicitly stored in the journal as well, the specific option listed below is hence a superset of this one. (Also note that any external, additional syslog daemons receive their log data from the journal, too, hence this is the option to use when logging shall be processed with such a daemon.)

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

The **file:***path* option may be used to connect a specific file system object to standard output. The semantics are similar to the same option of *StandardInput=*, see above. If *path* refers to a regular file on the filesystem, it is opened (created if it doesn't exist yet) for writing at the beginning of the file, but without truncating it. If standard input and output are directed to the same file path, it is opened only once, for reading as well as writing and duplicated. This is particularly useful when the specified path refers to an **AF_UNIX** socket in the file system, as in that case only a single stream connection is created for both input and output.

append:*path* is similar to **file:***path* above, but it opens the file in append mode.

truncate:*path* is similar to **file:***path* above, but it truncates the file when opening it. For units with multiple command lines, e.g. *Type=oneshot* services with multiple *ExecStart=*, or services with *ExecCondition=*, *ExecStartPre=* or *ExecStartPost=*, the output file is reopened and therefore re−truncated for each command line. If the output file is truncated while another process still has the file open, e.g. by an *ExecReload=* running concurrently with an *ExecStart=*, and the other process continues writing to the file without adjusting its offset, then the space between the file pointers of the two processes may be filled with **NUL** bytes, producing a sparse file. Thus, **truncate:***path* is typically only useful for units where only one process runs at a time, such as services with a single *ExecStart=* and no *ExecStartPost=*, *ExecReload=*, *ExecStop=* or similar.

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

The **fd:***name* option connects standard output to a specific, named file descriptor provided by a socket unit. A name may be specified as part of this option, following a ":" character (e.g. "fd:foobar"). If no name is specified, the name "stdout" is implied (i.e. "fd" is equivalent to "fd:stdout"). At least one socket unit defining the specified name must be provided via the *Sockets=* option, and the 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 their ordering.

If the standard output (or error output, see below) of a unit is connected to the journal or the kernel log buffer, the unit will implicitly gain a dependency of type *After=* on systemd−journald.socket (also see the "Implicit Dependencies" section above). Also note that in this case stdout (or stderr, see below) will be an **AF_UNIX** stream socket, and not a pipe or FIFO that can be re−opened. This means when executing shell scripts the construct **echo "hello" > /dev/stderr** for writing text to stderr will not work. To mitigate this use the construct **echo "hello" >&2** instead, which is mostly equivalent and avoids this pitfall.

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:***name* will use a default file descriptor name of "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).

StandardInputText=, *StandardInputData=*

Configures arbitrary textual or binary data to pass via file descriptor 0 (STDIN) to the executed processes. These settings have no effect unless *StandardInput=* is set to **data** (which is the default if *StandardInput=* is not set otherwise, but *StandardInputText=*/*StandardInputData=* is). Use this option to embed process input data directly in the unit file.

StandardInputText= accepts arbitrary textual data. C−style escapes for special characters as well as the usual "%"−specifiers are resolved. Each time this setting is used the specified text is appended to the per−unit data buffer, followed by a newline character (thus every use appends a new line to the end of the buffer). Note that leading and trailing whitespace of lines configured with this option is removed. If an empty line is specified the buffer is cleared (hence, in order to insert an empty line, add an additional "\n" to the end or beginning of a line).

StandardInputData= accepts arbitrary binary data, encoded in **Base64**[9]. No escape sequences or specifiers are resolved. Any whitespace in the encoded version is ignored during decoding.

Note that *StandardInputText=* and *StandardInputData=* operate on the same data buffer, and may be mixed in order to configure both binary and textual data for the same input stream. The textual or binary data is joined strictly in the order the settings appear in the unit file. Assigning an empty string to either will reset the data buffer.

Please keep in mind that in order to maintain readability long unit file settings may be split into multiple lines, by suffixing each line (except for the last) with a "\" character (see **systemd.unit**(5) for details). This is particularly useful for large data configured with these two options. Example:

```
...
StandardInput=data
StandardInputData=SWNrIHNpdHplIGRhIHVuJyBlc3NlIEtsb3BzLAp1ZmYgZWVtYWwga2xvcHAncy4KSWNrI
                  LCBzdGF1bmUsIHd1bmRyZSBtaXIsCnVmZiBlZW1hbCBqZWh0IHNlIHVmZiBkaWUgVMO8ci4KT
                  dSwgZGVuayBpY2ssIGljayBkZW5rIG5hbnUhCkpldHogaXNzZSB1ZmYsIGVyc2NodCB3YXIgc2Ug
```

```
        enUhCkljayBqZWhlIHJhdXMgdW5kIGJsaWNrZSDigJQKdW5kIHdlciBzdGVodCBkcmF1w59lbj8g \
        SWNrZSEK
    ...
```

LogLevelMax=
: Configures filtering by log level of log messages generated by this unit. Takes a **syslog** log level, one of **emerg** (lowest log level, only highest priority messages), **alert**, **crit**, **err**, **warning**, **notice**, **info**, **debug** (highest log level, also lowest priority messages). See **syslog**(3) for details. By default no filtering is applied (i.e. the default maximum log level is **debug**). Use this option to configure the logging system to drop log messages of a specific service above the specified level. For example, set *LogLevelMax=***info** in order to turn off debug logging of a particularly chatty unit. Note that the configured level is applied to any log messages written by any of the processes belonging to this unit, as well as any log messages written by the system manager process (PID 1) in reference to this unit, sent via any supported logging protocol. The filtering is applied early in the logging pipeline, before any kind of further processing is done. Moreover, messages which pass through this filter successfully might still be dropped by filters applied at a later stage in the logging subsystem. For example, *MaxLevelStore=* configured in **journald.conf**(5) might prohibit messages of higher log levels to be stored on disk, even though the per−unit *LogLevelMax=* permitted it to be processed.

LogExtraFields=
: Configures additional log metadata fields to include in all log records generated by processes associated with this unit. This setting takes one or more journal field assignments in the format "FIELD=VALUE" separated by whitespace. See **systemd.journal-fields**(7) for details on the journal field concept. Even though the underlying journal implementation permits binary field values, this setting accepts only valid UTF−8 values. To include space characters in a journal field value, enclose the assignment in double quotes ("). The usual specifiers are expanded in all assignments (see below). Note that this setting is not only useful for attaching additional metadata to log records of a unit, but given that all fields and values are indexed may also be used to implement cross−unit log record matching. Assign an empty string to reset the list.

LogRateLimitIntervalSec=, *LogRateLimitBurst=*
: Configures the rate limiting that is applied to messages generated by this unit. If, in the time interval defined by *LogRateLimitIntervalSec=*, more messages than specified in *LogRateLimitBurst=* are logged by a service, all further messages within the interval are dropped until the interval is over. A message about the number of dropped messages is generated. The time specification for *LogRateLimitIntervalSec=* may be specified in the following units: "s", "min", "h", "ms", "us" (see **systemd.time**(7) for details). The default settings are set by *RateLimitIntervalSec=* and *RateLimitBurst=* configured in **journald.conf**(5).

LogNamespace=
: Run the unit's processes in the specified journal namespace. Expects a short user−defined string identifying the namespace. If not used the processes of the service are run in the default journal namespace, i.e. their log stream is collected and processed by systemd−journald.service. If this option is used any log data generated by processes of this unit (regardless if via the **syslog()**, journal native logging or stdout/stderr logging) is collected and processed by an instance of the systemd−journald@.service template unit, which manages the specified namespace. The log data is stored in a data store independent from the default log namespace's data store. See **systemd-journald.service**(8) for details about journal namespaces.

  Internally, journal namespaces are implemented through Linux mount namespacing and over−mounting the directory that contains the relevant **AF_UNIX** sockets used for logging in the unit's mount namespace. Since mount namespaces are used this setting disconnects propagation of mounts from the unit's processes to the host, similar to how *ReadOnlyPaths=* and similar settings (see above) work. Journal namespaces may hence not be used for services that need to establish mount points on the host.

  When this option is used the unit will automatically gain ordering and requirement dependencies on

the two socket units associated with the systemd−journald@.service instance so that they are automatically established prior to the unit starting up. Note that when this option is used log output of this service does not appear in the regular **journalctl**(1) output, unless the **−−namespace=** option is used.

This option is only available for system services and is not supported for services running in per−user instances of the service manager.

SyslogIdentifier=
Sets the process name ("**syslog** tag") 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 **journal** or **kmsg** (or to the same settings in combination with **+console**) and only applies to log messages written to stdout or stderr.

SyslogFacility=
Sets the **syslog** facility identifier to use when logging. 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 **journal** or **kmsg** (or to the same settings in combination with **+console**), and only applies to log messages written to stdout or stderr. Defaults to **daemon**.

SyslogLevel=
The default **syslog** log level to use when logging to the logging system 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 **journal** or **kmsg** (or to the same settings in combination with **+console**), and only applies to log messages written to stdout or stderr. Note that individual lines output by executed processes may 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 **journal** or **kmsg** (or to the same settings in combination with **+console**), log lines written by the executed process that are prefixed with a log level will be processed 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. This only applies to log messages written to stdout or stderr. For details about this prefixing see **sd-daemon**(3). Defaults to true.

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".

CREDENTIALS

LoadCredential=ID[:*PATH*]
Pass a credential to the unit. Credentials are limited−size binary or textual objects that may be passed to unit processes. They are primarily used for passing cryptographic keys (both public and private) or certificates, user account information or identity information from host to services. The data is accessible from the unit's processes via the file system, at a read−only location that (if possible and

permitted) is backed by non−swappable memory. The data is only accessible to the user associated with the unit, via the *User=*/*DynamicUser=* settings (as well as the superuser). When available, the location of credentials is exported as the *$CREDENTIALS_DIRECTORY* environment variable to the unit's processes.

The *LoadCredential=* setting takes a textual ID to use as name for a credential plus a file system path, separated by a colon. The ID must be a short ASCII string suitable as filename in the filesystem, and may be chosen freely by the user. If the specified path is absolute it is opened as regular file and the credential data is read from it. If the absolute path refers to an **AF_UNIX** stream socket in the file system a connection is made to it (only once at unit start−up) and the credential data read from the connection, providing an easy IPC integration point for dynamically providing credentials from other services. If the specified path is not absolute and itself qualifies as valid credential identifier it is understood to refer to a credential that the service manager itself received via the *$CREDENTIALS_DIRECTORY* environment variable, which may be used to propagate credentials from an invoking environment (e.g. a container manager that invoked the service manager) into a service. The contents of the file/socket may be arbitrary binary or textual data, including newline characters and **NUL** bytes. If the file system path is omitted it is chosen identical to the credential name, i.e. this is a terse way do declare credentials to inherit from the service manager into a service. This option may be used multiple times, each time defining an additional credential to pass to the unit.

The credential files/IPC sockets must be accessible to the service manager, but don't have to be directly accessible to the unit's processes: the credential data is read and copied into separate, read−only copies for the unit that are accessible to appropriately privileged processes. This is particularly useful in combination with *DynamicUser=* as this way privileged data can be made available to processes running under a dynamic UID (i.e. not a previously known one) without having to open up access to all users.

In order to reference the path a credential may be read from within a *ExecStart=* command line use "${CREDENTIALS_DIRECTORY}/mycred", e.g. "ExecStart=cat ${CREDENTIALS_DIRECTORY}/mycred".

Currently, an accumulated credential size limit of 1 MB per unit is enforced.

If referencing an **AF_UNIX** stream socket to connect to, the connection will originate from an abstract namespace socket, that includes information about the unit and the credential ID in its socket name. Use **getpeername**(2) to query this information. The returned socket name is formatted as **NUL** *RANDOM* "/unit/" *UNIT* "/" *ID*, i.e. a **NUL** byte (as required for abstract namespace socket names), followed by a random string (consisting of alphadecimal characters), followed by the literal string "/unit/", followed by the requesting unit name, followed by the literal character "/", followed by the textual credential ID requested. Example: "\0adf9d86b6eda275e/unit/foobar.service/credx" in case the credential "credx" is requested for a unit "foobar.service". This functionality is useful for using a single listening socket to serve credentials to multiple consumers.

SetCredential=ID:*VALUE*

The *SetCredential=* setting is similar to *LoadCredential=* but accepts a literal value to use as data for the credential, instead of a file system path to read the data from. Do not use this option for data that is supposed to be secret, as it is accessible to unprivileged processes via IPC. It's only safe to use this for user IDs, public key material and similar non−sensitive data. For everything else use *LoadCredential=*. In order to embed binary data into the credential data use C−style escaping (i.e. "\n" to embed a newline, or "\x00" to embed a **NUL** byte).

If a credential of the same ID is listed in both *LoadCredential=* and *SetCredential=*, the latter will act as default if the former cannot be retrieved. In this case not being able to retrieve the credential from the path specified in *LoadCredential=* is not considered fatal.

SYSTEM V COMPATIBILITY

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".

ENVIRONMENT VARIABLES IN SPAWNED PROCESSES

Processes started by the service manager are executed with an environment variable block assembled from multiple sources. Processes started by the system service manager generally do not inherit environment variables set for the service manager itself (but this may be altered via *PassEnvironment=*), but processes started by the user service manager instances generally do inherit all environment variables set for the service manager itself.

For each invoked process the list of environment variables set is compiled from the following sources:

- Variables globally configured for the service manager, using the *DefaultEnvironment=* setting in **systemd-system.conf**(5), the kernel command line option *systemd.setenv=* understood by **systemd**(1), or via **systemctl**(1) **set−environment** verb.
- Variables defined by the service manager itself (see the list below).
- Variables set in the service manager's own environment variable block (subject to *PassEnvironment=* for the system service manager).
- Variables set via *Environment=* in the unit file.
- Variables read from files specified via *EnvironmentFile=* in the unit file.
- Variables set by any PAM modules in case *PAMName=* is in effect, cf. **pam_env**(8).

If the same environment variable is set by multiple of these sources, the later source — according to the order of the list above — wins. Note that as the final step all variables listed in *UnsetEnvironment=* are removed from the compiled environment variable list, immediately before it is passed to the executed process.

The general philosophy is to expose a small curated list of environment variables to processes. Services started by the system manager (PID 1) will be started, without additional service−specific configuration, with just a few environment variables. The user manager inherits environment variables as any other system service, but in addition may receive additional environment variables from PAM, and, typically, additional imported variables when the user starts a graphical session. It is recommended to keep the environment blocks in both the system and user managers managers lean. Importing all variables inherited by the graphical session or by one of the user shells is strongly discouraged.

Hint: **systemd−run −P env** and **systemd−run −−user −P env** print the effective system and user service environment blocks.

Environment Variables Set or Propagated by the Service Manager

The following environment variables are propagated by the service manager or generated internally for each invoked process:

$PATH
: Colon−separated list of directories to use when launching executables. **systemd** uses a fixed value of "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin" in the system manager. When compiled for systems with "unmerged /usr/" (/bin is not a symlink to /usr/bin), ":/sbin:/bin" is appended. In case of the user manager, a different path may be configured by the distribution. It is recommended to not rely on the order of entries, and have only one program with a given name in *$PATH*.

$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 to use for runtime objects (such as IPC objects) and volatile state. Set for all services run by the user **systemd** instance, as well as any system services that use *PAMName=* with a PAM stack that includes **pam_systemd**. See below and **pam_systemd**(8) for more information.

$RUNTIME_DIRECTORY, *$STATE_DIRECTORY*, *$CACHE_DIRECTORY*, *$LOGS_DIRECTORY*, *$CONFIGURATION_DIRECTORY*
: Absolute paths to the directories defined with *RuntimeDirectory=*, *StateDirectory=*, *CacheDirectory=*, *LogsDirectory=*, and *ConfigurationDirectory=* when those settings are used.

$CREDENTIALS_DIRECTORY
: An absolute path to the per−unit directory with credentials configured via *LoadCredential=*/*SetCredential=*. The directory is marked read−only and is placed in unswappable memory (if supported and permitted), and is only accessible to the UID associated with the unit via *User=* or *DynamicUser=* (and the superuser).

$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).

$SYSTEMD_EXEC_PID
: The PID of the unit process (e.g. process invoked by *ExecStart=*). The child process can use this information to determine whether the process is directly invoked by the service manager or indirectly as a child of another process by comparing this value with the current PID (as similar to the scheme used in **sd_listen_fds**(3) with *$LISTEN_PID* and *$LISTEN_FDS*).

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

$LOG_NAMESPACE
Contains the name of the selected logging namespace when the *LogNamespace=* service setting is used.

$JOURNAL_STREAM
If the standard output or standard error output of the executed processes are connected to the journal (for example, by setting *StandardError=journal*) *$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.

If both standard output and standard error of the executed processes are connected to the journal via a stream socket, this environment variable will contain information about the standard error stream, as that's usually the preferred destination for log data. (Note that typically the same stream is used for both standard output and standard error, hence very likely the environment variable contains device and inode information matching both stream file descriptors.)

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:

Table 4. Defined *$SERVICE_RESULT* values

Value	Meaning
"success"	The service ran successfully and exited cleanly.
"protocol"	A protocol violation occurred: the service did not take the steps required by its unit configuration (specifically what is configured in its *Type=* setting).
"timeout"	One of the steps timed out.
"exit−code"	Service process exited with a non−zero exit code; see *$EXIT_CODE* below for the actual exit code returned.
"signal"	A service process was terminated abnormally by a signal, without dumping core. See *$EXIT_CODE* below for the actual signal causing the termination.
"core−dump"	A service process terminated abnormally with a signal and dumped core. See *$EXIT_CODE* below for the signal causing the termination.
"watchdog"	Watchdog keep−alive ping was enabled for the service, but the deadline was missed.
"start−limit−hit"	A start limit was defined for the unit and it was hit, causing the unit to fail to start. See **systemd.unit**(5)'s *StartLimitIntervalSec=* and *StartLimitBurst=* for details.
"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 5. Summary of possible service result variable values

$SERVICE_RESULT	*$EXIT_CODE*	*$EXIT_STATUS*
"success"	"killed"	"HUP", "INT", "TERM", "PIPE"
	"exited"	"0"
"protocol"	not set	not set
	"exited"	"0"
"timeout"	"killed"	"TERM", "KILL"
	"exited"	"0", "1", "2", "3", ..., "255"
"exit−code"	"exited"	"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"
"exec−condition"	"exited"	"1", "2", "3", "4", ..., "254"
"oom−kill"	"killed"	"TERM", "KILL"
"start−limit−hit"	not set	not set
"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. Moreover, using *SuccessExitStatus=* additional exit statuses may be declared to indicate clean termination, which is not reflected by this table.		

$PIDFILE
: The path to the configured PID file, in case the process is forked off on behalf of a service that uses the *PIDFile=* setting, see **systemd.service**(5) for details. Service code may use this environment variable to automatically generate a PID file at the location configured in the unit file. This field is set to an absolute path in the file system.

For system services, when *PAMName=* is enabled and **pam_systemd** is part of the selected PAM stack, additional environment variables defined by systemd may be set for services. Specifically, these are *$XDG_SEAT*, *$XDG_VTNR*, see **pam_systemd**(8) for details.

PROCESS EXIT CODES

When invoking a unit process the service manager possibly fails to apply the execution parameters configured with the settings above. In that case the already created service process will exit with a non−zero exit code before the configured command line is executed. (Or in other words, the child process possibly exits with these error codes, after having been created by the **fork**(2) system call, but before the matching **execve**(2) system call is called.) Specifically, exit codes defined by the C library, by the LSB specification and by the systemd service manager itself are used.

The following basic service exit codes are defined by the C library.

Table 6. Basic C library exit codes

Exit Code	Symbolic Name	Description
0	**EXIT_SUCCESS**	Generic success code.
1	**EXIT_FAILURE**	Generic failure or unspecified error.

The following service exit codes are defined by the **LSB specification**[10].

Table 7. LSB service exit codes

Exit Code	Symbolic Name	Description
2	**EXIT_INVALIDARGUMENT**	Invalid or excess arguments.
3	**EXIT_NOTIMPLEMENTED**	Unimplemented feature.
4	**EXIT_NOPERMISSION**	The user has insufficient privileges.
5	**EXIT_NOTINSTALLED**	The program is not installed.
6	**EXIT_NOTCONFIGURED**	The program is not configured.
7	**EXIT_NOTRUNNING**	The program is not running.

The LSB specification suggests that error codes 200 and above are reserved for implementations. Some of them are used by the service manager to indicate problems during process invocation:

Table 8. systemd−specific exit codes

Finally, the BSD operating systems define a set of exit codes, typically defined on Linux systems too:

Table 9. BSD exit codes

Exit Code	Symbolic Name	Description
64	**EX_USAGE**	Command line usage error
65	**EX_DATAERR**	Data format error
66	**EX_NOINPUT**	Cannot open input
67	**EX_NOUSER**	Addressee unknown
68	**EX_NOHOST**	Host name unknown
69	**EX_UNAVAILABLE**	Service unavailable
70	**EX_SOFTWARE**	internal software error
71	**EX_OSERR**	System error (e.g., can't fork)
72	**EX_OSFILE**	Critical OS file missing
73	**EX_CANTCREAT**	Can't create (user) output file
74	**EX_IOERR**	Input/output error
75	**EX_TEMPFAIL**	Temporary failure; user is invited to retry
76	**EX_PROTOCOL**	Remote error in protocol
77	**EX_NOPERM**	Permission denied
78	**EX_CONFIG**	Configuration error

SEE ALSO

systemd(1), **systemctl**(1), **systemd-analyze**(1), **journalctl**(1), **systemd-system.conf**(5), **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), **fork**(2)

NOTES

1. Discoverable Partitions Specification
   https://systemd.io/DISCOVERABLE_PARTITIONS
2. The /proc Filesystem
   https://www.kernel.org/doc/html/latest/filesystems/proc.html#mount-options
3. User/Group Name Syntax
   https://systemd.io/USER_NAMES
4. No New Privileges Flag
   https://www.kernel.org/doc/html/latest/userspace-api/no_new_privs.html
5. JSON User Record
   https://systemd.io/USER_RECORD
6. proc.txt
   https://www.kernel.org/doc/Documentation/filesystems/proc.txt
7. C escapes
   https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences
8. most control characters
   https://en.wikipedia.org/wiki/Control_character#In_ASCII
9. Base64
   https://tools.ietf.org/html/rfc2045#section-6.8
10. LSB specification
    https://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/iniscrptact.html