NAME

dpkg-buildflags - returns build flags to use during package build

SYNOPSIS

dpkg-buildflags [*option*...] [*command*]

DESCRIPTION

dpkg-buildflags is a tool to retrieve compilation flags to use during build of Debian packages. The default flags are defined by the vendor but they can be extended/overriden in several ways:

1. system-wide with **/etc/dpkg/buildflags.conf**;
2. for the current user with **$XDG_CONFIG_HOME/dpkg/buildflags.conf** where **$XDG_CONFIG_HOME** defaults to **$HOME/.config**;
3. temporarily by the user with environment variables (see section **ENVIRONMENT**);
4. dynamically by the package maintainer with environment variables set via **debian/rules** (see section **ENVIRONMENT**).

The configuration files can contain two types of directives:

SET *flag value*
: Override the flag named *flag* to have the value *value*.

STRIP *flag value*
: Strip from the flag named *flag* all the build flags listed in *value*.

APPEND *flag value*
: Extend the flag named *flag* by appending the options given in *value*. A space is prepended to the appended value if the flag's current value is non-empty.

PREPEND *flag value*
: Extend the flag named *flag* by prepending the options given in *value*. A space is appended to the prepended value if the flag's current value is non-empty.

The configuration files can contain comments on lines starting with a hash (#). Empty lines are also ignored.

COMMANDS

--dump
: Print to standard output all compilation flags and their values. It prints one flag per line separated from its value by an equal sign (*flag*=*value*). This is the default action.

--list
: Print the list of flags supported by the current vendor (one per line). See the **SUPPORTED FLAGS** section for more information about them.

--status
: Display any information that can be useful to explain the behaviour of **dpkg-buildflags**: relevant environment variables, current vendor, state of all feature flags. Also print the resulting compiler flags with their origin.

  This is intended to be run from **debian/rules**, so that the build log keeps a clear trace of the build flags used. This can be useful to diagnose problems related to them.

--export=*format*
: Print to standard output commands that can be used to export all the compilation flags for some particular tool. If the *format* value is not given, **sh** is assumed. Only compilation flags starting with an upper case character are included, others are assumed to not be suitable for the environment. Supported formats:

  sh
  : Shell commands to set and export all the compilation flags in the environment. The flag values are quoted so the output is ready for evaluation by a shell.

cmdline
Arguments to pass to a build program's command line to use all the compilation flags (since dpkg 1.17.0). The flag values are quoted in shell syntax.

configure
This is a legacy alias for **cmdline**.

make Make directives to set and export all the compilation flags in the environment. Output can be written to a makefile fragment and evaluated using an **include** directive.

--get *flag*
Print the value of the flag on standard output. Exits with 0 if the flag is known otherwise exits with 1.

--origin *flag*
Print the origin of the value that is returned by **--get**. Exits with 0 if the flag is known otherwise exits with 1. The origin can be one of the following values:

vendor
the original flag set by the vendor is returned;

system
the flag is set/modified by a system-wide configuration;

user the flag is set/modified by a user-specific configuration;

env the flag is set/modified by an environment-specific configuration.

--query-features *area*
Print the features enabled for a given area. The only currently recognized areas are **qa**, **reproducible** and **hardening**, see the **FEATURE AREAS**section for more details. Exits with 0 if the area is known otherwise exits with 1.

The output is in RFC822 format, with one section per feature. For example:

Feature: pie
Enabled: no

Feature: stackprotector
Enabled: yes

--help Show the usage message and exit.

--version
Show the version and exit.

SUPPORTED FLAGS

CFLAGS
Options for the C compiler. The default value set by the vendor includes **-g** and the default optimization level (**-O2** usually, or **-O0** if the **DEB_BUILD_OPTIONS** environment variable defines *noopt*).

CPPFLAGS
Options for the C preprocessor. Default value: empty.

CXXFLAGS
Options for the C++ compiler. Same as **CFLAGS**.

OBJCFLAGS
Options for the Objective C compiler. Same as **CFLAGS**.

OBJCXXFLAGS
Options for the Objective C++ compiler. Same as **CXXFLAGS**.

GCJFLAGS
Options for the GNU Java compiler (gcj). A subset of **CFLAGS**.

FFLAGS
Options for the Fortran 77 compiler. A subset of **CFLAGS**.

FCFLAGS
Options for the Fortran 9x compiler. Same as **FFLAGS**.

LDFLAGS
Options passed to the compiler when linking executables or shared objects (if the linker is called directly, then **-Wl** and **,** have to be stripped from these options). Default value: empty.

New flags might be added in the future if the need arises (for example to support other languages).

FEATURE AREAS

Each area feature can be enabled and disabled in the **DEB_BUILD_OPTIONS** and **DEB_BUILD_MAINT_OPTIONS** environment variable's area value with the + and - modifier. For example, to enable the **hardening** pie feature and disable the fortify feature you can do this in **debian/rules**:

export DEB_BUILD_MAINT_OPTIONS=hardening=+pie,-fortify

The special feature **all** (valid in any area) can be used to enable or disable all area features at the same time. Thus disabling everything in the **hardening** area and enabling only format and fortify can be achieved with:

export DEB_BUILD_MAINT_OPTIONS=hardening=-all,+format,+fortify

Quality Assurance (QA)

Several compile-time options (detailed below) can be used to help detect problems in the source code or build system.

bug This setting (disabled by default) adds any warning option that reliably detects problematic source code. The warnings are fatal.

canary
This setting (disabled by default) adds dummy canary options to the build flags, so that the build logs can be checked for how the build flags propagate and to allow finding any omission of normal build flag settings. The only currently supported flags are **CPPFLAGS**, **CFLAGS**, **OBJCFLAGS**, **CXXFLAGS** and **OBJCXXFLAGS** with flags set to **-D__DEB_CANARY_***flag_random-id*__, and **LDFLAGS** set to **-Wl,-z,deb-canary-***random-id*.

Hardening

Several compile-time options (detailed below) can be used to help harden a resulting binary against memory corruption attacks, or provide additional warning messages during compilation. Except as noted below, these are enabled by default for architectures that support them.

format
This setting (enabled by default) adds **-Wformat -Werror=format-security** to **CFLAGS**, **CXXFLAGS**, **OBJCFLAGS** and **OBJCXXFLAGS**. This will warn about improper format string uses, and will fail when format functions are used in a way that represent possible security problems. At present, this warns about calls to **printf** and **scanf** functions where the format string is not a string literal and there are no format arguments, as in **printf(foo);** instead of **printf(%s, foo);**This ma y be a security hole if the format string came from untrusted input and contains %n.

fortify
This setting (enabled by default) adds **-D_FORTIFY_SOURCE=2** to **CPPFLAGS**. During code generation the compiler knows a great deal of information about buffer sizes

(where possible), and attempts to replace insecure unlimited length buffer function calls with length-limited ones. This is especially useful for old, crufty code. Additionally, format strings in writable memory that contain '%n' are blocked. If an application depends on such a format string, it will need to be worked around.

Note that for this option to have any effect, the source must also be compiled with **-O1** or higher. If the environment variable **DEB_BUILD_OPTIONS** contains *noopt*, then **fortify** support will be disabled, due to new warnings being issued by glibc 2.16 and later.

stackprotector
This setting (enabled by default if stackprotectorstrong is not in use) adds **-fstack-protector --param=ssp-buffer-size=4** to **CFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS** and **FCFLAGS**. This adds safety checks against stack overwrites. This renders many potential code injection attacks into aborting situations. In the best case this turns code injection vulnerabilities into denial of service or into non-issues (depending on the application).

This feature requires linking against glibc (or another provider of **__stack_chk_fail**), so needs to be disabled when building with **-nostdlib** or **-ffreestanding** or similar.

stackprotectorstrong
This setting (enabled by default) adds **-fstack-protector-strong** to **CFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS** and **FCFLAGS**. This is a stronger variant of **stackprotector**, but without significant performance penalties.

Disabling **stackprotector** will also disable this setting.

This feature has the same requirements as **stackprotector**, and in addition also requires gcc 4.9 and later.

relro This setting (enabled by default) adds **-Wl,-z,relro** to **LDFLAGS**. During program load, several ELF memory sections need to be written to by the linker. This flags the loader to turn these sections read-only before turning over control to the program. Most notably this prevents GOT overwrite attacks. If this option is disabled, **bindnow** will become disabled as well.

bindnow
This setting (disabled by default) adds **-Wl,-z,now** to **LDFLAGS**. During program load, all dynamic symbols are resolved, allowing for the entire PLT to be marked read-only (due to **relro** above). The option cannot become enabled if **relro** is not enabled.

pie This setting (disabled by default) adds **-fPIE** to **CFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS** and **FCFLAGS**, and **-fPIE -pie**to **LDFLA GS**. Position Independent Executable are needed to take advantage of Address Space Layout Randomization, supported by some kernel versions. While ASLR can already be enforced for data areas in the stack and heap (brk and mmap), the code areas must be compiled as position-independent. Shared libraries already do this (-fPIC), so they gain ASLR automatically, but binary .text regions need to be build PIE to gain ASLR. When this happens, ROP (Return Oriented Programming) attacks are much harder since there are no static locations to bounce off of during a memory corruption attack.

This is not compatible with **-fPIC** so care must be taken when building shared objects.

Additionally, since PIE is implemented via a general register, some architectures (most notably i386) can see performance losses of up to 15% in very text-segment-heavy application workloads; most workloads see less than 1%. Architectures with more general registers (e.g. amd64) do not see as high a worst-case penalty.

Reproducibility

The compile-time options detailed below can be used to help improve build reproducibility or provide additional warning messages during compilation. Except as noted below, these are enabled by default for architectures that support them.

timeless

This setting (disabled by default) adds **-Wdate-time** to **CPPFLAGS**. This will cause warnings when the **__TIME__**, **__DATE__** and **__TIMESTAMP__** macros are used.

ENVIRONMENT

There are 2 sets of environment variables doing the same operations, the first one (DEB_*flag*_*op*) should never be used within **debian/rules**. It's meant for any user that wants to rebuild the source package with different build flags. The second set (DEB_*flag*_MAINT_*op*) should only be used in **debian/rules** by package maintainers to change the resulting build flags.

DEB_*flag*_SET
DEB_*flag*_MAINT_SET

This variable can be used to force the value returned for the given *flag*.

DEB_*flag*_STRIP
DEB_*flag*_MAINT_STRIP

This variable can be used to provide a space separated list of options that will be stripped from the set of flags returned for the given *flag*.

DEB_*flag*_APPEND
DEB_*flag*_MAINT_APPEND

This variable can be used to append supplementary options to the value returned for the given *flag*.

DEB_*flag*_PREPEND
DEB_*flag*_MAINT_PREPEND

This variable can be used to prepend supplementary options to the value returned for the given *flag*.

DEB_BUILD_OPTIONS
DEB_BUILD_MAINT_OPTIONS

These variables can be used by a user or maintainer to disable/enable various area features that affect build flags. The **DEB_BUILD_MAINT_OPTIONS** variable overrides any setting in the **DEB_BUILD_OPTIONS** feature areas. See the **FEATURE AREAS** section for details.

FILES

Configuration files

/etc/dpkg/buildflags.conf

System wide configuration file.

$XDG_CONFIG_HOME/dpkg/buildflags.conf or **$HOME/.config/dpkg/buildflags.conf**

User configuration file.

Packaging support

/usr/share/dpkg/buildflags.mk

Makefile snippet that will load (and optionally export) all flags supported by **dpkg-buildflags** into variables (since dpkg 1.16.1).

EXAMPLES

To pass build flags to a build command in a makefile:

```
$(MAKE) $(shell dpkg-buildflags --export=cmdline)

./configure $(shell dpkg-buildflags --export=cmdline)
```

To set build flags in a shell script or shell fragment, eval can be used to interpret the output and

to export the flags in the environment:

```
eval $(dpkg-buildflags --export=sh) && make
```

or to set the positional parameters to pass to a command:

```
eval set -- $(dpkg-buildflags --export=cmdline)
for dir in a b c; do (cd $dir && ./configure $@ && make); done
```

Usage in debian/rules

You should call **dpkg-buildflags** or include **buildflags.mk** from the **debian/rules** file to obtain the needed build flags to pass to the build system. Note that older versions of **dpkg-buildpackage** (before dpkg 1.16.1) exported these flags automatically. However, you should not rely on this, since this breaks manual invocation of **debian/rules**.

For packages with autoconf-like build systems, you can pass the relevant options to configure or **make(1)** directly, as shown above.

For other build systems, or when you need more fine-grained control about which flags are passed where, you can use **--get**. Or you can include **buildflags.mk** instead, which takes care of calling **dpkg-buildflags** and storing the build flags in make variables.

If you want to export all buildflags into the environment (where they can be picked up by your build system):

```
DPKG_EXPORT_BUILDFLAGS = 1
include /usr/share/dpkg/buildflags.mk
```

For some extra control over what is exported, you can manually export the variables (as none are exported by default):

```
include /usr/share/dpkg/buildflags.mk
export CPPFLAGS CFLAGS LDFLAGS
```

And you can of course pass the flags to commands manually:

```
include /usr/share/dpkg/buildflags.mk
build-arch:
 $(CC) -o hello hello.c $(CPPFLAGS) $(CFLAGS) $(LDFLAGS)
```