NAME

　　　　ld.so, ld−linux.so − dynamic linker/loader

SYNOPSIS

　　　　The dynamic linker can be run either indirectly by running some dynamically linked program or shared ob-
　　　　ject (in which case no command-line options to the dynamic linker can be passed and, in the ELF case, the
　　　　dynamic linker which is stored in the **.interp** section of the program is executed) or directly by running:

　　　　/lib/ld−linux.so.* [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION

　　　　The programs **ld.so** and **ld−linux.so*** find and load the shared objects (shared libraries) needed by a pro-
　　　　gram, prepare the program to run, and then run it.

　　　　Linux binaries require dynamic linking (linking at run time) unless the **−static** option was given to **ld**(1)
　　　　during compilation.

　　　　The program **ld.so** handles a.out binaries, a binary format used long ago. The program **ld−linux.so***
　　　　(*/lib/ld−linux.so.1* for libc5, */lib/ld−linux.so.2* for glibc2) handles binaries that are in the more modern ELF
　　　　format. Both programs have the same behavior, and use the same support files and programs (**ldd**(1), **ld-**
　　　　config(8), and */etc/ld.so.conf*).

　　　　When resolving shared object dependencies, the dynamic linker first inspects each dependency string to see
　　　　if it contains a slash (this can occur if a shared object pathname containing slashes was specified at link
　　　　time). If a slash is found, then the dependency string is interpreted as a (relative or absolute) pathname, and
　　　　the shared object is loaded using that pathname.

　　　　If a shared object dependency does not contain a slash, then it is searched for in the following order:

　　　o　Using the directories specified in the DT_RPATH dynamic section attribute of the binary if present and
　　　　　DT_RUNPATH attribute does not exist. Use of DT_RPATH is deprecated.

　　　o　Using the environment variable **LD_LIBRARY_PATH**, unless the executable is being run in secure-
　　　　　execution mode (see below), in which case this variable is ignored.

　　　o　Using the directories specified in the DT_RUNPATH dynamic section attribute of the binary if present.
　　　　　Such directories are searched only to find those objects required by DT_NEEDED (direct dependen-
　　　　　cies) entries and do not apply to those objects' children, which must themselves have their own
　　　　　DT_RUNPATH entries. This is unlike DT_RPATH, which is applied to searches for all children in the
　　　　　dependency tree.

　　　o　From the cache file */etc/ld.so.cache*, which contains a compiled list of candidate shared objects previ-
　　　　　ously found in the augmented library path. If, however, the binary was linked with the **−z nodeflib**
　　　　　linker option, shared objects in the default paths are skipped. Shared objects installed in hardware capa-
　　　　　bility directories (see below) are preferred to other shared objects.

　　　o　In the default path */lib*, and then */usr/lib*. (On some 64-bit architectures, the default paths for 64-bit
　　　　　shared objects are */lib64*, and then */usr/lib64*.) If the binary was linked with the **−z nodeflib** linker op-
　　　　　tion, this step is skipped.

　　Rpath token expansion

　　　　The dynamic linker understands certain token strings in an rpath specification (DT_RPATH or DT_RUN-
　　　　PATH). Those strings are substituted as follows:

　　　　$ORIGIN (or equivalently *${ORIGIN}*)

　　　　　　　　This expands to the directory containing the program or shared object. Thus, an application lo-
　　　　　　　　cated in *somedir/app* could be compiled with

　　　　　　　　　gcc −Wl,−rpath,'$ORIGIN/../lib'

　　　　　　　　so that it finds an associated shared object in *somedir/lib* no matter where *somedir* is located in the
　　　　　　　　directory hierarchy. This facilitates the creation of "turn-key" applications that do not need to be
　　　　　　　　installed into special directories, but can instead be unpacked into any directory and still find their
　　　　　　　　own shared objects.

$LIB (or equivalently *${LIB}*)

   This expands to *lib* or *lib64* depending on the architecture (e.g., on x86-64, it expands to *lib64* and on x86-32, it expands to *lib*).

$PLATFORM (or equivalently *${PLATFORM}*)

   This expands to a string corresponding to the processor type of the host system (e.g., "x86_64"). On some architectures, the Linux kernel doesn't provide a platform string to the dynamic linker. The value of this string is taken from the **AT_PLATFORM** value in the auxiliary vector (see **getauxval**(3)).

OPTIONS

−−audit *list*

   Use objects named in *list* as auditors. The objects in *list* are delimited by colons.

−−inhibit−cache

   Do not use */etc/ld.so.cache*.

−−library−path *path*

   Use *path* instead of **LD_LIBRARY_PATH** environment variable setting (see below). The names *ORIGIN* , *LIB*, and *PLATFORM* are interpreted as for the **LD_LIBRARY_PATH** environment variable.

−−inhibit−rpath *list*

   Ignore RPATH and RUNPATH information in object names in *list*. This option is ignored when running in secure-execution mode (see below). The objects in *list* are delimited by colons or spaces.

−−list   List all dependencies and how they are resolved.

−−preload *list* (since glibc 2.30)

   Preload the objects specified in *list*. The objects in *list* are delimited by colons or spaces. The objects are preloaded as explained in the description of the **LD_PRELOAD** environment variable below.

   By contrast with **LD_PRELOAD**, the **−−preload** option provides a way to perform preloading for a single executable without affecting preloading performed in any child process that executes a new program.

−−verify

   Verify that program is dynamically linked and this dynamic linker can handle it.

ENVIRONMENT

Various environment variables influence the operation of the dynamic linker.

Secure-execution mode

For security reasons, if the dynamic linker determines that a binary should be run in secure-execution mode, the effects of some environment variables are voided or modified, and furthermore those environment variables are stripped from the environment, so that the program does not even see the definitions. Some of these environment variables affect the operation of the dynamic linker itself, and are described below. Other environment variables treated in this way include: **GCONV_PATH**, **GETCONF_DIR**, **HOSTALIASES**, **LOCALDOMAIN**, **LOCPATH**, **MALLOC_TRACE**, **NIS_PATH**, **NLSPATH**, **RESOLV_HOST_CONF**, **RES_OPTIONS**, **TMPDIR**, and **TZDIR**.

A binary is executed in secure-execution mode if the **AT_SECURE** entry in the auxiliary vector (see **getauxval**(3)) has a nonzero value. This entry may have a nonzero value for various reasons, including:

*   The process's real and effective user IDs differ, or the real and effective group IDs differ. This typically occurs as a result of executing a set-user-ID or set-group-ID program.

*   A process with a non-root user ID executed a binary that conferred capabilities to the process.

*   A nonzero value may have been set by a Linux Security Module.

Environment variables

Among the more important environment variables are the following:

LD_ASSUME_KERNEL (since glibc 2.2.3)

Each shared object can inform the dynamic linker of the minimum kernel ABI version that it requires. (This requirement is encoded in an ELF note section that is viewable via *readelf −n* as a section labeled **NT_GNU_ABI_TAG**.) At run time, the dynamic linker determines the ABI version of the running kernel and will reject loading shared objects that specify minimum ABI versions that exceed that ABI version.

LD_ASSUME_KERNEL can be used to cause the dynamic linker to assume that it is running on a system with a different kernel ABI version. For example, the following command line causes the dynamic linker to assume it is running on Linux 2.2.5 when loading the shared objects required by *myprog*:

```
$ LD_ASSUME_KERNEL=2.2.5 ./myprog
```

On systems that provide multiple versions of a shared object (in different directories in the search path) that have different minimum kernel ABI version requirements, **LD_ASSUME_KERNEL** can be used to select the version of the object that is used (dependent on the directory search order).

Historically, the most common use of the **LD_ASSUME_KERNEL** feature was to manually select the older LinuxThreads POSIX threads implementation on systems that provided both Linux-Threads and NPTL (which latter was typically the default on such systems); see **pthreads**(7).

LD_BIND_NOW (since glibc 2.1.1)

If set to a nonempty string, causes the dynamic linker to resolve all symbols at program startup instead of deferring function call resolution to the point when they are first referenced. This is useful when using a debugger.

LD_LIBRARY_PATH

A list of directories in which to search for ELF libraries at execution time. The items in the list are separated by either colons or semicolons, and there is no support for escaping either separator.

This variable is ignored in secure-execution mode.

Within the pathnames specified in **LD_LIBRARY_PATH**, the dynamic linker expands the tokens *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Rpath token expansion*. Thus, for example, the following would cause a library to be searched for in either the *lib* or *lib64* subdirectory below the directory containing the program to be executed:

```
$ LD_LIBRARY_PATH='$ORIGIN/$LIB' prog
```

(Note the use of single quotes, which prevent expansion of *$ORIGIN* and *$LIB* as shell variables!)

LD_PRELOAD

A list of additional, user-specified, ELF shared objects to be loaded before all others. This feature can be used to selectively override functions in other shared objects.

The items of the list can be separated by spaces or colons, and there is no support for escaping either separator. The objects are searched for using the rules given under DESCRIPTION. Objects are searched for and added to the link map in the left-to-right order specified in the list.

In secure-execution mode, preload pathnames containing slashes are ignored. Furthermore, shared objects are preloaded only from the standard search directories and only if they have set-user-ID mode bit enabled (which is not typical).

Within the names specified in the **LD_PRELOAD** list, the dynamic linker understands the tokens *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Rpath token expansion*. (See also the discussion of quoting under the description of **LD_LIBRARY_PATH**.)

There are various methods of specifying libraries to be preloaded, and these are handled in the following order:

(1) The **LD_PRELOAD** environment variable.

(2) The **−−preload** command-line option when invoking the dynamic linker directly.

(3) The */etc/ld.so.preload* file (described below).

LD_TRACE_LOADED_OBJECTS
If set (to any value), causes the program to list its dynamic dependencies, as if run by **ldd**(1), instead of running normally.

Then there are lots of more or less obscure variables, many obsolete or only for internal use.

LD_AUDIT (since glibc 2.4)
A list of user-specified, ELF shared objects to be loaded before all others in a separate linker namespace (i.e., one that does not intrude upon the normal symbol bindings that would occur in the process) These objects can be used to audit the operation of the dynamic linker. The items in the list are colon-separated, and there is no support for escaping the separator.

LD_AUDIT is ignored in secure-execution mode.

The dynamic linker will notify the audit shared objects at so-called auditing checkpoints—for example, loading a new shared object, resolving a symbol, or calling a symbol from another shared object—by calling an appropriate function within the audit shared object. For details, see **rtld−audit**(7). The auditing interface is largely compatible with that provided on Solaris, as described in its *Linker and Libraries Guide*, in the chapter *Runtime Linker Auditing Interface*.

Within the names specified in the **LD_AUDIT** list, the dynamic linker understands the tokens *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Rpath token expansion*. (See also the discussion of quoting under the description of **LD_LIBRARY_PATH**.)

Since glibc 2.13, in secure-execution mode, names in the audit list that contain slashes are ignored, and only shared objects in the standard search directories that have the set-user-ID mode bit enabled are loaded.

LD_BIND_NOT (since glibc 2.1.95)
If this environment variable is set to a nonempty string, do not update the GOT (global offset table) and PLT (procedure linkage table) after resolving a function symbol. By combining the use of this variable with **LD_DEBUG** (with the categories *bindings* and *symbols*), one can observe all run-time function bindings.

LD_DEBUG (since glibc 2.1)
Output verbose debugging information about operation of the dynamic linker. The content of this variable is one of more of the following categories, separated by colons, commas, or (if the value is quoted) spaces:

help   Specifying *help* in the value of this variable does not run the specified program, and displays a help message about which categories can be specified in this environment variable.

all   Print all debugging information (except *statistics* and *unused*; see below).

bindings   Display information about which definition each symbol is bound to.

files   Display progress for input file.

libs   Display library search paths.

reloc   Display relocation processing.

scopes   Display scope information.

       statistics   Display relocation statistics.

       symbols   Display search paths for each symbol look-up.

       unused   Determine unused DSOs.

       versions   Display version dependencies.

Since glibc 2.3.4, **LD_DEBUG** is ignored in secure-execution mode, unless the file */etc/suid−debug* exists (the content of the file is irrelevant).

LD_DEBUG_OUTPUT (since glibc 2.1)

By default, **LD_DEBUG** output is written to standard error. If **LD_DEBUG_OUTPUT** is defined, then output is written to the pathname specified by its value, with the suffix "." (dot) followed by the process ID appended to the pathname.

LD_DEBUG_OUTPUT is ignored in secure-execution mode.

LD_DYNAMIC_WEAK (since glibc 2.1.91)

By default, when searching shared libraries to resolve a symbol reference, the dynamic linker will resolve to the first definition it finds.

Old glibc versions (before 2.2), provided a different behavior: if the linker found a symbol that was weak, it would remember that symbol and keep searching in the remaining shared libraries. If it subsequently found a strong definition of the same symbol, then it would instead use that definition. (If no further symbol was found, then the dynamic linker would use the weak symbol that it initially found.)

The old glibc behavior was nonstandard. (Standard practice is that the distinction between weak and strong symbols should have effect only at static link time.) In glibc 2.2, the dynamic linker was modified to provide the current behavior (which was the behavior that was provided by most other implementations at that time).

Defining the **LD_DYNAMIC_WEAK** environment variable (with any value) provides the old (nonstandard) glibc behavior, whereby a weak symbol in one shared library may be overridden by a strong symbol subsequently discovered in another shared library. (Note that even when this variable is set, a strong symbol in a shared library will not override a weak definition of the same symbol in the main program.)

Since glibc 2.3.4, **LD_DYNAMIC_WEAK** is ignored in secure-execution mode.

LD_HWCAP_MASK (since glibc 2.1)

Mask for hardware capabilities.

LD_ORIGIN_PATH (since glibc 2.1)

Path where the binary is found.

Since glibc 2.4, **LD_ORIGIN_PATH** is ignored in secure-execution mode.

LD_POINTER_GUARD (glibc from 2.4 to 2.22)

Set to 0 to disable pointer guarding. Any other value enables pointer guarding, which is also the default. Pointer guarding is a security mechanism whereby some pointers to code stored in writable program memory (return addresses saved by **setjmp**(3) or function pointers used by various glibc internals) are mangled semi-randomly to make it more difficult for an attacker to hijack the pointers for use in the event of a buffer overrun or stack-smashing attack. Since glibc 2.23, **LD_POINTER_GUARD** can no longer be used to disable pointer guarding, which is now always enabled.

LD_PROFILE (since glibc 2.1)

The name of a (single) shared object to be profiled, specified either as a pathname or a soname. Profiling output is appended to the file whose name is: "*$LD_PROFILE_OUTPUT/$LD_PROFILE*.profile".

Since glibc 2.2.5, **LD_PROFILE** is ignored in secure-execution mode.

LD_PROFILE_OUTPUT (since glibc 2.1)

      Directory where **LD_PROFILE** output should be written. If this variable is not defined, or is defined as an empty string, then the default is */var/tmp*.

      LD_PROFILE_OUTPUT is ignored in secure-execution mode; instead */var/profile* is always used. (This detail is relevant only before glibc 2.2.5, since in later glibc versions, **LD_PROFILE** is also ignored in secure-execution mode.)

LD_SHOW_AUXV (since glibc 2.1)

      If this environment variable is defined (with any value), show the auxiliary array passed up from the kernel (see also **getauxval**(3)).

      Since glibc 2.3.4, **LD_SHOW_AUXV** is ignored in secure-execution mode.

LD_TRACE_PRELINKING (since glibc 2.4)

      If this environment variable is defined, trace prelinking of the object whose name is assigned to this environment variable. (Use **ldd**(1) to get a list of the objects that might be traced.) If the object name is not recognized, then all prelinking activity is traced.

LD_USE_LOAD_BIAS (since glibc 2.3.3)

      By default (i.e., if this variable is not defined), executables and prelinked shared objects will honor base addresses of their dependent shared objects and (nonprelinked) position-independent executables (PIEs) and other shared objects will not honor them. If **LD_USE_LOAD_BIAS** is defined with the value 1, both executables and PIEs will honor the base addresses. If **LD_USE_LOAD_BIAS** is defined with the value 0, neither executables nor PIEs will honor the base addresses.

      Since glibc 2.3.3, this variable is ignored in secure-execution mode.

LD_VERBOSE (since glibc 2.1)

      If set to a nonempty string, output symbol versioning information about the program if the **LD_TRACE_LOADED_OBJECTS** environment variable has been set.

LD_WARN (since glibc 2.1.3)

      If set to a nonempty string, warn about unresolved symbols.

LD_PREFER_MAP_32BIT_EXEC (x86-64 only; since glibc 2.23)

      According to the Intel Silvermont software optimization guide, for 64-bit applications, branch prediction performance can be negatively impacted when the target of a branch is more than 4 GB away from the branch. If this environment variable is set (to any value), the dynamic linker will first try to map executable pages using the **mmap**(2) **MAP_32BIT** flag, and fall back to mapping without that flag if that attempt fails. NB: MAP_32BIT will map to the low 2 GB (not 4 GB) of the address space.

      Because **MAP_32BIT** reduces the address range available for address space layout randomization (ASLR), **LD_PREFER_MAP_32BIT_EXEC** is always disabled in secure-execution mode.

FILES

    /lib/ld.so

      a.out dynamic linker/loader

    /lib/ld−linux.so.{1,2}

      ELF dynamic linker/loader

    /etc/ld.so.cache

      File containing a compiled list of directories in which to search for shared objects and an ordered list of candidate shared objects. See **ldconfig**(8).

    /etc/ld.so.preload

      File containing a whitespace-separated list of ELF shared objects to be loaded before the program. See the discussion of **LD_PRELOAD** above. If both **LD_PRELOAD** and */etc/ld.so.preload* are employed, the libraries specified by **LD_PRELOAD** are preloaded first. */etc/ld.so.preload* has a system-wide effect, causing the specified libraries to be preloaded for all programs that are executed on the system. (This is usually undesirable, and is typically employed only as an emergency

remedy, for example, as a temporary workaround to a library misconfiguration issue.)

lib*.so*

shared objects

NOTES

Hardware capabilities

Some shared objects are compiled using hardware-specific instructions which do not exist on every CPU. Such objects should be installed in directories whose names define the required hardware capabilities, such as *usr/lib/sse2/*. The dynamic linker checks these directories against the hardware of the machine and selects the most suitable version of a given shared object. Hardware capability directories can be cascaded to combine CPU features. The list of supported hardware capability names depends on the CPU. The following names are currently recognized:

Alpha   ev4, ev5, ev56, ev6, ev67

MIPS   loongson2e, loongson2f, octeon, octeon2

PowerPC

4xxmac, altivec, arch_2_05, arch_2_06, booke, cellbe, dfp, efpdouble, efpsingle, fpu, ic_snoop, mmu, notb, pa6t, power4, power5, power5+, power6x, ppc32, ppc601, ppc64, smt, spe, ucache, vsx

SPARC

flush, muldiv, stbar, swap, ultra3, v9, v9v, v9v2

s390   dfp, eimm, esan3, etf3enh, g5, highgprs, hpage, ldisp, msa, stfle, z900, z990, z9-109, z10, zarch

x86 (32-bit only)

acpi, apic, clflush, cmov, cx8, dts, fxsr, ht, i386, i486, i586, i686, mca, mmx, mtrr, pat, pbe, pge, pn, pse36, sep, ss, sse, sse2, tm

SEE ALSO

ld(1), **ldd**(1), **pldd**(1), **sprof**(1), **dlopen**(3), **getauxval**(3), **elf**(5), **capabilities**(7), **rtld−audit**(7), **ldconfig**(8), **sln**(8)

COLOPHON

This page is part of release 5.05 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.