Rocky Enterprise Linux 9.2 Manual Pages on command 'rtld-audit.7'

$ man rtld-audit.7

RTLD-AUDIT(7)   Linux Programmer's Manual   RTLD-AUDIT(7)

NAME

    rtld-audit - auditing API for the dynamic linker

SYNOPSIS

    #define _GNU_SOURCE   /* See feature_test_macros(7) */

    #include <link.h>

DESCRIPTION

    The GNU dynamic linker (run-time linker) provides an auditing API that

    allows an application to be notified when various dynamic linking

    events occur. This API is very similar to the auditing interface pro‐

    vided by the Solaris run-time linker. The necessary constants and pro‐

    totypes are defined by including <link.h>.

    To use this interface, the programmer creates a shared library that im‐

    plements a standard set of function names. Not all of the functions

    need to be implemented: in most cases, if the programmer is not inter‐

    ested in a particular class of auditing event, then no implementation

    needs to be provided for the corresponding auditing function.

    To employ the auditing interface, the environment variable LD_AUDIT

must be defined to contain a colon-separated list of shared libraries,
each of which can implement (parts of) the auditing API. When an au?
ditable event occurs, the corresponding function is invoked in each li?
brary, in the order that the libraries are listed.

la_version()

unsigned int la_version(unsigned int version);

This is the only function that must be defined by an auditing library:
it performs the initial handshake between the dynamic linker and the
auditing library.   When invoking this function, the dynamic linker
passes, in version, the highest version of the auditing interface that
the linker supports.

A   typical implementation of this function simply returns the constant
LAV_CURRENT, which indicates the version of <link.h> that was used to
build the audit module. If the dynamic linker does not support this
version of the audit interface, it will refuse to activate this audit
module. If the function returns zero, the dynamic linker also does not
activate this audit module.

In order to enable backwards compatibility with older dynamic linkers,
an audit module can examine the version argument and return an earlier
version than LAV_CURRENT, assuming the module can adjust its implemen?
tation to match the requirements of the previous version of the audit
interface. The la_version function should not return the value of ver?
sion without further checks because it could correspond to an interface
that does not match the <link.h> definitions used to build the audit
module.

la_objsearch()

char *la_objsearch(const char *name, uintptr_t *cookie,
                unsigned int flag);

The dynamic linker invokes this function to inform the auditing library
that it is about to search for a shared object. The name argument is
the filename or pathname that is to be searched for. cookie identifies
the shared object that initiated the search. flag is set to one of the
following values:

LA_SER_ORIG   This is the original name that is being searched for. Typically, this name comes from an ELF DT_NEEDED en? try, or is the filename argument given to dlopen(3).

LA_SER_LIBPATH   name was created using a directory specified in LD_LI? BRARY_PATH.

LA_SER_RUNPATH   name was created using a directory specified in an ELF DT_RPATH or DT_RUNPATH list.

LA_SER_CONFIG   name   was   found via the ldconfig(8) cache (/etc/ld.so.cache).

LA_SER_DEFAULT   name was found via a search of one of the default di? rectories.

LA_SER_SECURE   name is specific to a secure object (unused on Linux).

As its function result, la_objsearch() returns the pathname that the dynamic linker should use for further processing. If NULL is returned, then this pathname is ignored for further processing. If this audit library simply intends to monitor search paths, then name should be re? turned.

la_activity()

    void la_activity(uintptr_t *cookie, unsigned int flag);

    The dynamic linker calls this function to inform the auditing library that link-map activity is occurring. cookie identifies the object at the head of the link map. When the dynamic linker invokes this func? tion, flag is set to one of the following values:

    LA_ACT_ADD   New objects are being added to the link map.

    LA_ACT_DELETE   Objects are being removed from the link map.

    LA_ACT_CONSISTENT Link-map activity has been completed: the map is once again consistent.

la_objopen()

    unsigned int la_objopen(struct link_map *map, Lmid_t lmid,

                    uintptr_t *cookie);

    The dynamic linker calls this function when a new shared object is loaded. The map argument is a pointer to a link-map structure that de? scribes the object. The lmid field has one of the following values

LM_ID_BASE   Link map is part of the initial namespace.

LM_ID_NEWLM   Link map is part of a new namespace requested via dl?

mopen(3).

cookie is a pointer to an identifier for this object.   The identifier

is provided to later calls to functions in the auditing library in or?

der to identify this object. This identifier is initialized to point

to object's link map, but the audit library can change the identifier

to some other value that it may prefer to use to identify the object.

As its return value, la_objopen() returns a bit mask created by ORing

zero or more of the following constants, which allow the auditing li?

brary to select the objects to be monitored by la_symbind*():

LA_FLG_BINDTO   Audit symbol bindings to this object.

LA_FLG_BINDFROM Audit symbol bindings from this object.

A return value of 0 from la_objopen() indicates that no symbol bindings

should be audited for this object.

la_objclose()

unsigned int la_objclose(uintptr_t *cookie);

The dynamic linker invokes this function after any finalization code

for the object has been executed, before the object is unloaded.   The

cookie argument is the identifier obtained from a previous invocation

of la_objopen().

In the current implementation, the value returned by la_objclose() is

ignored.

la_preinit()

void la_preinit(uintptr_t *cookie);

The dynamic linker invokes this function after all shared objects have

been loaded, before control is passed to the application (i.e., before

calling main()). Note that main() may still later dynamically load ob?

jects using dlopen(3).

la_symbind*()

uintptr_t la_symbind32(Elf32_Sym *sym, unsigned int ndx,

uintptr_t *refcook, uintptr_t *defcook,

unsigned int *flags, const char *symname);

```
uintptr_t la_symbind64(Elf64_Sym *sym, unsigned int ndx,

                uintptr_t *refcook, uintptr_t *defcook,

                unsigned int *flags, const char *symname);
```

The dynamic linker invokes one of these functions when a symbol binding

occurs between two shared objects that have been marked for auditing

notification by la_objopen(). The la_symbind32() function is employed

on 32-bit platforms; the la_symbind64() function is employed on 64-bit

platforms.

The sym argument is a pointer to a structure that provides information

about the symbol being bound. The structure definition is shown in

<elf.h>. Among the fields of this structure, st_value indicates the

address to which the symbol is bound.

The ndx argument gives the index of the symbol in the symbol table of

the bound shared object.

The refcook argument identifies the shared object that is making the

symbol reference; this is the same identifier that is provided to the

la_objopen() function that returned LA_FLG_BINDFROM. The defcook argu?

ment identifies the shared object that defines the referenced symbol;

this is the same identifier that is provided to the la_objopen() func?

tion that returned LA_FLG_BINDTO.

The symname argument points a string containing the name of the symbol.

The flags argument is a bit mask that both provides information about

the symbol and can be used to modify further auditing of this PLT (Pro?

cedure Linkage Table) entry. The dynamic linker may supply the follow?

ing bit values in this argument:

LA_SYMB_DLSYM   The binding resulted from a call to dlsym(3).

LA_SYMB_ALTVALUE   A previous la_symbind*() call returned an alter?

                nate value for this symbol.

By default, if the auditing library implements la_pltenter() and la_pl?

texit() functions (see below), then these functions are invoked, after

la_symbind(), for PLT entries, each time the symbol is referenced. The

following flags can be ORed into *flags to change this default behav?

ior:

LA_SYMB_NOPLTENTER   Don't call la_pltenter() for this symbol.

LA_SYMB_NOPLTEXIT   Don't call la_pltexit() for this symbol.

The return value of la_symbind32() and la_symbind64() is the address to which control should be passed after the function returns. If the au? diting library is simply monitoring symbol bindings, then it should re? turn sym->st_value. A different value may be returned if the library wishes to direct control to an alternate location.

la_pltenter()

The precise name and argument types for this function depend on the hardware platform. (The appropriate definition is supplied by <link.h>.) Here is the definition for x86-32:

```
Elf32_Addr la_i86_gnu_pltenter(Elf32_Sym *sym, unsigned int ndx,
              uintptr_t *refcook, uintptr_t *defcook,
              La_i86_regs *regs, unsigned int *flags,
              const char *symname, long *framesizep);
```

This function is invoked just before a PLT entry is called, between two shared objects that have been marked for binding notification.

The sym, ndx, refcook, defcook, and symname are as for la_symbind*().

The regs argument points to a structure (defined in <link.h>) contain? ing the values of registers to be used for the call to this PLT entry.

The flags argument points to a bit mask that conveys information about, and can be used to modify subsequent auditing of, this PLT entry, as for la_symbind*().

The framesizep argument points to a long int buffer that can be used to explicitly set the frame size used for the call to this PLT entry.   If different la_pltenter() invocations for this symbol return different values, then the maximum returned value is used. The la_pltexit() function is called only if this buffer is explicitly set to a suitable value.

The return value of la_pltenter() is as for la_symbind*().

la_pltexit()

The precise name and argument types for this function depend on the hardware platform. (The appropriate definition is supplied by

<link.h>.) Here is the definition for x86-32:

unsigned int la_i86_gnu_pltexit(Elf32_Sym *sym, unsigned int ndx,

          uintptr_t *refcook, uintptr_t *defcook,

          const La_i86_regs *inregs, La_i86_retval *outregs,

          const char *symname);

This function is called when a PLT entry, made between two shared ob?
jects that have been marked for binding notification, returns. The
function is called just before control returns to the caller of the PLT
entry.

The sym, ndx, refcook, defcook, and symname are as for la_symbind*().
The inregs argument points to a structure (defined in <link.h>) con?
taining the values of registers used for the call to this PLT entry.
The outregs argument points to a structure (defined in <link.h>) con?
taining return values for the call to this PLT entry. These values can
be modified by the caller, and the changes will be visible to the
caller of the PLT entry.

In the current GNU implementation, the return value of la_pltexit() is
ignored.

CONFORMING TO

This API is nonstandard, but very similar to the Solaris API, described
in the Solaris Linker and Libraries Guide, in the chapter Runtime
Linker Auditing Interface.

NOTES

Note the following differences from the Solaris dynamic linker auditing
API:

* The Solaris la_objfilter() interface is not supported by the GNU im?
  plementation.

* The Solaris la_symbind32() and la_pltexit() functions do not provide
  a symname argument.

* The Solaris la_pltexit() function does not provide inregs and out?
  regs arguments (but does provide a retval argument with the function
  return value).

BUGS

In glibc versions up to and include 2.9, specifying more than one audit library in LD_AUDIT results in a run-time crash. This is reportedly fixed in glibc 2.10.

EXAMPLES

```c
#include <link.h>
#include <stdio.h>

unsigned int
la_version(unsigned int version)
{
    printf("la_version(): version = %u; LAV_CURRENT = %u\n",
           version, LAV_CURRENT);

    return LAV_CURRENT;
}

char *
la_objsearch(const char *name, uintptr_t *cookie, unsigned int flag)
{
    printf("la_objsearch(): name = %s; cookie = %p", name, cookie);
    printf("; flag = %s\n",
           (flag == LA_SER_ORIG) ?    "LA_SER_ORIG" :
           (flag == LA_SER_LIBPATH) ? "LA_SER_LIBPATH" :
           (flag == LA_SER_RUNPATH) ? "LA_SER_RUNPATH" :
           (flag == LA_SER_DEFAULT) ? "LA_SER_DEFAULT" :
           (flag == LA_SER_CONFIG) ?  "LA_SER_CONFIG" :
           (flag == LA_SER_SECURE) ?  "LA_SER_SECURE" :
           "???");

    return name;
}

void
la_activity (uintptr_t *cookie, unsigned int flag)
{
    printf("la_activity(): cookie = %p; flag = %s\n", cookie,
           (flag == LA_ACT_CONSISTENT) ? "LA_ACT_CONSISTENT" :
           (flag == LA_ACT_ADD) ?        "LA_ACT_ADD" :
```

```c
            (flag == LA_ACT_DELETE) ?    "LA_ACT_DELETE" :
            "???");
}
unsigned int
la_objopen(struct link_map *map, Lmid_t lmid, uintptr_t *cookie)
{
    printf("la_objopen(): loading \"%s\"; lmid = %s; cookie=%p\n",
        map->l_name,
        (lmid == LM_ID_BASE) ?  "LM_ID_BASE" :
        (lmid == LM_ID_NEWLM) ? "LM_ID_NEWLM" :
        "???",
        cookie);
    return LA_FLG_BINDTO | LA_FLG_BINDFROM;
}
unsigned int
la_objclose (uintptr_t *cookie)
{
    printf("la_objclose(): %p\n", cookie);
    return 0;
}
void
la_preinit(uintptr_t *cookie)
{
    printf("la_preinit(): %p\n", cookie);
}
uintptr_t
la_symbind32(Elf32_Sym *sym, unsigned int ndx, uintptr_t *refcook,
        uintptr_t *defcook, unsigned int *flags, const char *symname)
{
    printf("la_symbind32(): symname = %s; sym->st_value = %p\n",
        symname, sym->st_value);
    printf("        ndx = %u; flags = %#x", ndx, *flags);
    printf("; refcook = %p; defcook = %p\n", refcook, defcook);
```

```
       return sym->st_value;

   }

   uintptr_t

   la_symbind64(Elf64_Sym *sym, unsigned int ndx, uintptr_t *refcook,

        uintptr_t *defcook, unsigned int *flags, const char *symname)

   {

      printf("la_symbind64(): symname = %s; sym->st_value = %p\n",

          symname, sym->st_value);

      printf("        ndx = %u; flags = %#x", ndx, *flags);

      printf("; refcook = %p; defcook = %p\n", refcook, defcook);

      return sym->st_value;

   }

   Elf32_Addr

   la_i86_gnu_pltenter(Elf32_Sym *sym, unsigned int ndx,

        uintptr_t *refcook, uintptr_t *defcook, La_i86_regs *regs,

        unsigned int *flags, const char *symname, long *framesizep)

   {

      printf("la_i86_gnu_pltenter(): %s (%p)\n", symname, sym->st_value);

      return sym->st_value;

   }
```

SEE ALSO

   ldd(1), dlopen(3), ld.so(8), ldconfig(8)

COLOPHON

   This page is part of release 5.10 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/.

Linux   2020-11-01   RTLD-AUDIT(7)