

# Re: Solving the lib mismatch problem

---

*Source:* <http://unix.derkeiler.com/Newsgroups/comp.unix.programmer/2007-06/msg00419.html>

---

- *From:* Bin Chen <[binary.chen@xxxxxxxx](mailto:binary.chen@xxxxxxxx)>
  - *Date:* Fri, 15 Jun 2007 18:16:00 -0700
- 

On Jun 16, 12:18 am, \* Tong \* <[sun\\_tong\\_...@xxxxxxxxxxxxxxxxxxxxxxxx](mailto:sun_tong_...@xxxxxxxxxxxxxxxxxxxxxxxx)> wrote:

On Fri, 15 Jun 2007 13:51:32 +0000, \* Tong \* wrote:

```
$ xosview
xosview: /usr/lib/libstdc++.so.6: version `GLIBCXX_3.4.5' not found
(required by xosview)
```

What I meant to ask is,

is there any way to know what exactly version a library is?  
which tools can help me with it? I tried nm, but the lib is stripped.

I think that's the only knowledge for me to know to solve the problem.

Copy for you and please notice the "Startup sequence" section.

from Ulrich Drepper <[drepper@xxxxxxxx](mailto:drepper@xxxxxxxx)> ...

---

This is the most complete description on symbol versioning I have.  
It's the same the ia64 working group got submitted by me.

---

~~~~~  
ELF Symbol Versioning  
Ulrich Drepper <[drepper@xxxxxxxx](mailto:drepper@xxxxxxxx)>  
Cygnus Solutions

The symbol versioning implementation used on Linux with glibc 2.1 or up is an extension of Sun's versioning. It provides most of the functionality Sun has plus one decisive new element: symbol-level versioning with multiple definitions of a symbol.

## Re: Solving the lib mismatch problem

The implementation allows every DSO to either use versions for their symbols or not. Depending on whether the DSO an object is linked against had symbols or not, the reference to the DSO requires symbols or not. This is recorded in the binary by three new sections:

– SHT\_GNU\_verdef

This section is pointed to by an entry in the section table with the tag DT\_VERDEF. It points to a table with elements of type Elfxx\_Verdef. The number of entries in the table is determined by DT\_VERDEFNUM. GNU ld generated sections with the name '.gnu.version\_d'. The link in the section header points to the section with the strings used in the entries of this section.

Elfxx\_Verdef is defined as:

```
typedef struct
{
  Elfxx_Half vd_version; /* Version revision */
  Elfxx_Half vd_flags; /* Version information */
  Elfxx_Half vd_ndx; /* Version Index */
  Elfxx_Half vd_cnt; /* Number of associated aux entries */
  Elfxx_Word vd_hash; /* Version name hash value */
  Elfxx_Word vd_aux; /* Offset in bytes to verdaux array */
  Elfxx_Word vd_next; /* Offset in bytes to next verdef
entry */
} Elfxx_Verdef;
```

The meaning of the individual fields is as follows:

vd\_version

This field currently always has the value 1. It will be changed if the versioning implementation has to be changed in an incompatible way.

vd\_flags

This field contains a bitmask of flags. The following value is defined so far:

VER\_FLG\_BASE this is the version of the file itself. It must not be used for matching a symbol. It can be used to match references.

vd\_ndx

This is a numeric value which is used as an index into the

## Re: Solving the lib mismatch problem

SHT\_GNU\_versym  
section.

vd\_cnt

The element specifies the number of associated auxiliary entries.

These

auxiliary entries contain the actual version definition.

vd\_hash

Hash value of the name (computed using the ELF hash function)

vd\_aux

Offset in bytes to the corresponding auxiliary entries.

vd\_next

Offset in bytes to the next version definition

The entries referenced to by the vd\_aux elements are of this type:

```
typedef struct
```

```
{  
  Elfxx_Word vda_name; /* Version or dependency names */  
  Elfxx_Word vda_next; /* Offset in bytes to next verdaux  
entry */  
} Elfxx_Verdaux;
```

vda\_name

This is an index into the string section referenced in the section header to the point where the name string can be found.

vda\_next

byte offset to the next Elfxx\_Verdaux entry. The first entry (pointed to by the Elfxx\_Verdef entry, contains the actual defined name.

The

second and all later entries name predecessor versions.

– SHT\_GNU\_verneed

This section is pointed to by an entry in the section table with the tag

DT\_VERNEED. It points to a table with elements of type

Elfxx\_Verneed.

The number of entries in the table is determined by DT\_VERNEEDNUM.

GNU

ld generated sections with the name '.gnu.version\_r'. The link in the

section header points to the section with the strings used in the entries

## Re: Solving the lib mismatch problem

of this section.

Elfxx\_Verneed is defined as:

```
typedef struct
{
Elfxx_Half vn_version; /* Version of structure */
Elfxx_Half vn_cnt; /* Number of associated aux entries */
Elfxx_Word vn_file; /* Offset of filename for this
dependency */
Elfxx_Word vn_aux; /* Offset in bytes to vernaux array */
Elfxx_Word vn_next; /* Offset in bytes to next verneed
entry */
} Elfxx_Verneed;
```

The meaning of the individual fields is as follows:

vn\_version

This field currently always has the value 1. It will be changed if the versioning implementation has to be changed in an incompatible way.

The symbol VER\_NEED\_CURRENT is defined for this field.

vn\_cnt

The element specifies the number of associated auxiliary entries. These auxiliary entries contain the actual version definition.

vn\_file

Offset in the string section reference by the link in the section header for the string with the file name.

vn\_aux

Offset in bytes to the corresponding auxiliary entries.

vn\_next

Offset in bytes to the next version dependency record

The entries referenced to by the vn\_aux elements are of this type:

```
typedef struct
{
Elfxx_Word vna_hash; /* Hash value of dependency name */
Elfxx_Half vna_flags; /* Dependency specific information */
Elfxx_Half vna_other; /* Unused */
Elfxx_Word vna_name; /* Dependency name string offset */
Elfxx_Word vna_next; /* Offset in bytes to next vernaux
```

## Re: Solving the lib mismatch problem

```
entry */  
} Elfxx_Vernaux;
```

vna\_hash

Hash value (computed using the ELF hashing function) for the name referenced by vna\_name

vna\_flags

Bitmask of flags. Currently the following are defined:

VER\_FLG\_WEAK the reference to this version is weak.

vna\_other

Contains version index unique for the file which is used in the version symbol table. If the highest bit (bit 15) is set this is a hidden symbol which cannot be referenced from outside the object.

vna\_name

This is an index into the string section referenced in the section header to the point where the name string can be found.

vna\_next

byte offset to the next Elfxx\_Verdaux entry. The first entry (pointed to by the Elfxx\_Verdef entry, contains the actual defined name. The second and all later entries name predecessor versions.

– SHT\_GNU\_versym

This section contains an array of elements of type Elfxx\_Half (the same type as the vna\_other field). It has as many entries as the dynamic symbol table (DT\_SYMTAB). I.e., each symbol table entry has its associated entry in the symbol version table.

The values in the symbol version table entries are the vna\_other values from the definition of the required versions. The values 0 and 1 are reserved.

0 the symbol is local, not available outside the object

1 the symbol is defined in this object and globally available

All other values are used for versions in the own object or in any of the dependencies. This is the version the symbol is tight to. Not the

## Re: Solving the lib mismatch problem

numeric value is important but instead the string referenced by the `vna_name` element of the according `Elfxx_Verdaux/Elfxx_Vernaux` entry.

### Startup sequence

=====

We describe here only the additional activities required to implement the versioning in an already existing dynamic loader implementation. The exact place where the actions have to happen will vary.

After loading a shared object (but before making it generally available) the dynamic loader has to check whether the loading object fulfills all the version requirement of the object which caused it to load.

This has to be done by walking through all the entries of the loading object's `Elfxx_Verneed` array and examining whether the loaded object's `Elfxx_Verdef` array contains an appropriate version definition. Both, the version definition and the version requirement, are identified by strings. In both cases the structures contain beside the string (or string references) also ELF hashing values which can be compared before making the actual string comparison.

These tests have to be recursively performed for all objects and their dependencies. This way it is possible to recognize libraries which are too old and don't contain all the symbols or contain incompatible implementations. Without this kind of test one could end up with runtime errors which don't provide helpful information.

There is one situation where a missing symbol definition is not an error. This is when the `vna_flags` in the `Elfxx_Vernaux` entry has the `VER_FLG_WEAK` bit set. In this case only a warning is issued and the object is used normally.

Once all the tests for availability of the versions are performed successfully the object can be made available publicly. If the loaded objects contains no invalidly formed data this means that all versions referenced by undefined symbols are available.

### Symbol lookup

=====

During the relocations in an object using symbol versioning we have to extend the test for a matching definition. Not only is it now required that the strings with the symbol names are matching, it is now also required that the version name of the reference is the same as the name of the definition. To retrieve the names uses the same

## Re: Solving the lib mismatch problem

index as for the symbol table (both requirement and definition) and retrieves a value from the SHT\_GNU\_versym section. This section then can be used to get a string from the Elfxx\_Verneed entries (for the requirement) and the Elfxx\_Verdef entries (for the definition).

If the highest bit (no. 15) of the version symbol value is set, the object is hidden and must not be used. In this case the linker must treat the symbol as not present in the object.

Ideally both, the file having the requirement and the file with the definitions, are using symbol versioning. In this case the above matching must happen. In the case there is no matching definition in the currently searched object but the object is the one with the name from the Elfxx\_Verneed entry (referenced by the vn\_name element), then the missing of the symbol is a fatal error. An object must not simply lose the definition of a symbol.

In case only the object file with the reference does not use versioning but the object with the definition does, then the reference only matches the base definition. The base definition is the one with index numbers 1 and 2 (1 is the unspecified name, 2 is the name given later to the baseline of symbols once the library started using symbol versioning). The static linker is guaranteed to use this index for the base definition. If there is no symbol definition with such a version index and there is exactly one version for which this symbol is defined, then this version is accepted (this was mostly implemented for dlopen() calls as it will normally not happen when the static linker is used). Otherwise, if more than one version of the symbol is available, none of the definitions is accepted and the search continues with the next object.

The last case is if the object with the references uses symbol versions but the object with the definitions has none. In this case a matching symbol is accepted unless the object's name matches the one in the Elfxx\_Verneed entry. In the latter case this is a fatal error. In fact, this should never have happened in the first place since it would mean the list of required symbols was not correct and the steps required in the last section therefore haven't detected a too old version of an object file.

The case with two non-versioned objects is not new and simply is resolved according to existing rules.

### Static Linker

=====

The static linker has to do some extra work. It is required to generate the Elfxx\_Verdef and Elfxx\_Verneed records as well as the SHT\_GNU\_versym section.

## Re: Solving the lib mismatch problem

The Elfxx\_Verdef records are created from information contained in the object file. The assembler code contains lines of the form

```
.symver real, name@version  
or  
.symver real, name@@version
```

In both cases `real' must be the name of a defined symbol. The pseudo opcode generates an alias `name' which is associated with the version `version'. There can be arbitrary many definitions of the first kind but there must be exactly one definition of the later kind. The two `at' characters mean that this version is the default version.

Additional input comes from a map file given to the static linker. The map file also features the version names. For each version the programmer can specify whether a given symbol is exported or kept private. The .symver definitions must correspond to appropriate entries in the map file.

Example:

Assume the definitions

```
.symver __foo_old, foo@VER1  
.symver __foo_new, foo@@VER2  
.symver __bar_old, bar@@VER1  
....
```

The appropriate map file should look like this:

```
VER1 {  
global:  
foo; bar;  
local:  
*  
};
```

```
VER2 {  
global:  
foo;  
} VER1;
```

This means:

- there are two versions defined, VER1, and VER2. VER2 is a successor or VER1 (because of the line `} VER1;'
- the symbols foo@VER1 and bar@@VER1 are public
- the symbol foo@@VER2 is also public

## Re: Solving the lib mismatch problem

– all other symbols (which are matched by \*) are of version VER1 and are not exported (which means the version name is irrelevant but it somewhere has to be written down)

It makes no sense at all to associate versions with symbols which are not exported. Therefore the `local:' sections of all but the base version are empty and the `local:' section of the base version simply contains `\*'. This will match all symbols which are not explicitly mentioned in any `global:' list.

[For more information take a look at Solaris's map files which have a similar syntax. It's available in the Linker and Library Guide.]

The difference between a normal version definition and the default version definition is that when linking against a versioned object one always picks the default version. This is normally the latest version since it is the one currently developed. Please note that the undefined reference causing the binding cannot be associated in any way with a version. The association happens by finding the first default definition of a symbol with a matching name.

Once all references needed from a versioned object are identified, the linker collects all the version names and creates a derivation tree (it need not be a tree but it should be). This tree is then put into the form of an `Elfxx_Verneed` entry with as many `Elfxx_Verneed` entries as necessary. The name referenced in the `Elfxx_Verneed` entry is the SONAME of the object with the definitions. All the entries are then put in the `SHT_GNU_verneed` section of the newly created object.

In addition the static linker has to create the `SHT_GNU_versym` section if the created object contains a `DT_SYMTAB` section. It must have the same number of entries. Entries which correspond to symbol definitions in the own file have an index which can be found in an appropriate entry in the `Elfxx_Verdef` section. This entry then gives access to the version string associated with the symbol by the `.symver` pseudo op. If no version is defined (it always is: all symbols are defined or none) then the index is 1, meaning global.

In the case the symbol table entry is for an unreferenced symbol the index is in the `Elfxx_Verneed` section and it specifies the version name and the SONAME of the file the definition was found in at linktime. If the referenced symbol wasn't satisfied by any of the archive, objects, or DSOs used on the linker commandline or if the object the symbol was found in has no symbol versions, the used index is 1, meaning global.

.