Red Hat Enterprise Linux 8

Developing applications in RHEL 8

An introduction to application development tools in Red Hat Enterprise Linux 8

Last Updated: 2019-07-30

Red Hat Enterprise Linux 8 Developing applications in RHEL 8

An introduction to application development tools in Red Hat Enterprise Linux 8

Legal Notice

Abstract

This document describes the different features and utilities that make Red Hat Enterprise Linux 8 an ideal enterprise platform for application development.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Please let us know how we could make it better. To do so:

- For simple comments on specific passages, make sure you are viewing the documentation in the Multi-page HTML format. Highlight the part of text that you want to comment on. Then, click the **Add Feedback** pop-up that appears below the highlighted text, and follow the displayed instructions.
- For submitting more complex feedback, create a Bugzilla ticket:
  1. Go to the Bugzilla website.
  2. As the Component, use **Documentation**.
  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click **Submit Bug**.

CHAPTER 1. SETTING UP A DEVELOPMENT WORKSTATION

Red Hat Enterprise Linux 8 supports development of custom applications. To allow developers to do so, the system must be set up with the required tools and utilities. This chapter lists the most common use cases for development and the items to install.

1.1. PREREQUISITES

- The system must be installed, including a graphical environment, and subscribed.

1.2. ENABLING DEBUG AND SOURCE REPOSITORIES

A standard installation of Red Hat Enterprise Linux does not enable the debug and source repositories. These repositories contain information needed to debug the system components and measure their performance.

Procedure

- Enable the source and debug information package channels:

```
# subscription-manager repos --enable rhel-8-for-$(uname -i)-baseos-debug-rpms
# subscription-manager repos --enable rhel-8-for-$(uname -i)-baseos-source-rpms
# subscription-manager repos --enable rhel-8-for-$(uname -i)-appstream-debug-rpms
# subscription-manager repos --enable rhel-8-for-$(uname -i)-appstream-source-rpms
```

The **$(uname -i)** part is automatically replaced with a matching value for architecture of your system:

Architecture name	Value
64-bit Intel and AMD	x86_64
64-bit ARM	aarch64
IBM POWER	ppc64le
IBM Z	s390x

1.3. SETTING UP TO MANAGE APPLICATION VERSIONS

Effective version control is essential to all multi-developer projects. Red Hat Enterprise Linux is distributed with Git, a distributed version control system.

Procedure

1. Install the **git** package:

```
# yum install git
```

2. Optional: Set the full name and email address associated with your Git commits:

```
$ git config --global user.name "Full Name"
$ git config --global user.email "email@example.com"
```

Replace *Full Name* and *email@example.com* with your actual name and email address.

3. Optional: To change the default text editor started by Git, set value of the **core.editor** configuration option:

```
$ git config --global core.editor command
```

Replace *command* with the command to be used to start the selected text editor.

Additional resources

- Linux manual pages for Git and tutorials:

```
$ man git
$ man gittutorial
$ man gittutorial-2
```

Note that many Git commands have their own manual pages. As an example see *git-commit(1)*.

- *Git User's Manual* – HTML documentation for Git is located at **/usr/share/doc/git/user-manual.html**.
- Pro Git – The online version of the *Pro Git* book provides a detailed description of Git, its concepts and its usage.
- Reference – Online version of the Linux manual pages for Git

1.4. SETTING UP TO DEVELOP APPLICATIONS USING C AND C++

Red Hat Enterprise Linux includes tools for creating C and C++ applications.

Prerequisites

- The debug and source repositories must be enabled.

Procedure

1. Install the **Development Tools** package group including GNU Compiler Collection (GCC), GNU Debugger (GDB), and other development tools:

```
# yum group install "Development Tools"
```

2. Install the LLVM-based toolchain including the **clang** compiler and **lldb** debugger:

```
# yum install llvm-toolset
```

3. Optional: For Fortran dependencies, install the GNU Fortran compiler:

```
# yum install gcc-gfortran
```

1.5. SETTING UP TO DEBUG APPLICATIONS

Red Hat Enterprise Linux offers multiple debugging and instrumentation tools to analyze and troubleshoot internal application behavior.

Prerequisites

- The debug and source repositories must be enabled.

Procedure

1. Install the tools useful for debugging:

```
# yum install gdb valgrind systemtap ltrace strace
```

2. Install the **yum-utils** package in order to use the **debuginfo-install** tool:

```
# yum install yum-utils
```

3. Run a SystemTap helper script for setting up the environment.

```
# stap-prep
```

Running this script installs kernel debuginfo packages. The size of these packages exceeds 2 GiB.

4. Make sure **SELinux** policies allow the relevant applications to run not only normally, but in the debugging situations, too. For more information, see Using SELinux.

1.6. SETTING UP TO MEASURE PERFORMANCE OF APPLICATIONS

Red Hat Enterprise Linux includes several applications that can help a developer identify the causes of application performance loss.

Prerequisites

- The debug and source repositories must be enabled.

Procedure

1. Install the tools for performance measurement:

```
# yum install perf papi pcp-zeroconf valgrind strace sysstat systemtap
```

2. Run a SystemTap helper script for setting up the environment.

```
# stap-prep
```

Running this script installs kernel debuginfo packages. The size of these packages exceeds 2 GiB.

3. Enable and start the Performance Co-Pilot (PCP) collector service:

```
# systemctl enable pmcd && systemctl start pmcd
```

CHAPTER 2. OPTIONS FOR RUNNING A RHEL 6 OR 7 APPLICATION ON RHEL 8

To run a Red Hat Enterprise Linux 6 or 7 application on Red Hat Enterprise Linux 8, a spectrum of options is available. A system administrator needs detailed guidance from the application developer. The following list outlines the options, considerations, and resources provided by Red Hat.

Run the application in a virtual machine with a matching RHEL version guest OS

Resource costs are high for this option, but the environment is a close match to the application's requirements, and this approach does not require many additional considerations. This is the currently recommended option.

Run the application in a container based on the respective RHEL version

Resource costs are lower than in the previous cases, while configuration requirements are stricter. For details on the relationship between container hosts and guest user spaces, see the Red Hat Enterprise Linux Container Compatibility Matrix.

Run the application natively on RHEL 8

This option offers the lowest resource costs, but also the most strict requirements. The application developer must determine a correct configuration of the RHEL 8 system. The following resources can help the developer in this task:

- Red Hat Enterprise Linux 8: Application Compatibility Guide
- Red Hat Enterprise Linux 7: Application Compatibility Guide
- Release notes for Red Hat Enterprise Linux 8.0
- Considerations in adopting RHEL 8

Note that this list is not a complete set of resources needed to determine application compatibility. These are only starting points with lists of known incompatible changes and Red Hat policies related to compatibility.

Additionally, the *What is Kernel Application Binary Interface (kABI)?* Knowledge Centered Support article contains information relevant to kernel and compatibility.

PART I. CREATING C OR C++ APPLICATIONS

Red Hat offers multiple tools for creating applications using the C and C++ languages. This part of the book lists some of the most common development tasks.

CHAPTER 3. BUILDING CODE WITH GCC

This chapter deals with situations where source code must be transformed into executable code.

3.1. RELATIONSHIP BETWEEN CODE FORMS

Prerequisites

- Understanding the concepts of compiling and linking

Possible code forms

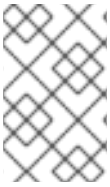
When using the C and C++ languages, there are three forms of code:

- **Source code** written in the C or C++ language, present as plain text files.
  The files typically use extensions such as **.c**, **.cc**, **.cpp**, **.h**, **.hpp**, **.i**, **.inc**. For a complete list of supported extensions and their interpretation, see the gcc manual pages:

  ```
  $ man gcc
  ```

- **Object code**, created by *compiling* the source code with a *compiler*. This is an intermediate form.
  The object code files use the **.o** extension.

- **Executable code**, created by *linking* object code with a *linker*.
  Linux application executable files do not use any file name extension. Shared object (library) executable files use the **.so** file name extension.



NOTE

Library archive files for static linking also exist. This is a variant of object code and uses the **.a** file name extension. Static linking is not recommended. See Section 4.2, "Static and dynamic linking".

Handling of code forms in GCC

Producing executable code from source code requires two steps, which require different applications or tools. GCC can be used as an intelligent driver for both compilers and linkers. This allows you to use a single command **gcc** for any of the required actions. GCC automatically selects the actions required (compiling and linking), as well as their sequence:

1. Source files are compiled to object files.
2. Object files and libraries are linked (including the previously compiled sources).

It is possible to run GCC such that only step 1 happens, only step 2 happens, or both steps 1 and 2 happen. This is determined by the types of inputs and requested type of output(s).

Because larger projects require a build system which usually runs GCC separately for each action, it is better to always consider compilation and linking as two distinct actions, even if GCC can perform both at once.

3.2. COMPILING SOURCE FILES TO OBJECT CODE

To create object code files from source files and not an executable file immediately, GCC must be instructed to create only object code files as its output. This action represents the basic operation of the build process for larger projects.

Prerequisites

- C or C++ source code file(s)
- GCC installed on the system

Procedure

1. Change to the directory containing the source code file(s).
2. Run **gcc** with the **-c** option:

```
$ gcc -c source.c another_source.c
```

Object files are created, with their file names reflecting the original source code files: **source.c** results in **source.o**.

NOTE

With C++ source code, replace the **gcc** command with **g++** for convenient handling of C++ Standard Library dependencies.

3.3. ENABLING DEBUGGING OF C AND C++ APPLICATIONS WITH GCC

Because debugging information is large, it is not included in executable files by default. To enable debugging of your C and C++ applications with it, you must explicitly instruct the compiler to create it.

To enable creation of debugging information with **GCC** when compiling and linking code, use the **-g** option:

```
$ gcc ... -g ...
```

- Optimizations performed by the compiler and linker can result in executable code which is hard to relate to the original source code: Variables may be optimized out, loops unrolled, operations merged into the surrounding ones etc. This affects debugging negatively. For improved debuging experience, consider setting the optimization with the **-Og** option. However, changing the optimization level changes the executable code and may change the actual behaviour so as to remove some bugs.
- To include also macro definitions in the debug information, use the **-g3** option instead of **-g**.
- The **-fcompare-debug** GCC option tests code compiled by GCC with debug information and without debug information. The test passes if the resulting two binary files are identical. This test ensures that executable code is not affected by any debugging options, which further ensures that there are no hidden bugs in the debug code. Note that using the **-fcompare-debug** option significantly increases compilation time. See the GCC manual page for details about this option.

Additional resources

- Using the GNU Compiler Collection (GCC) — Options for Debugging Your Program
- Debugging with GDB — Debugging Information in Separate Files
- The GCC manual page:

```
$ man gcc
```

3.4. CODE OPTIMIZATION WITH GCC

A single program can be transformed into more than one sequence of machine instructions. You can achieve a more optimal result if you allocate more resources to analyzing the code during compilation.

With GCC, you can set the optimization level using the ***-Olevel*** option. This option accepts a set of values in place of the *level*.

Level	Description
0	Optimize for compilation speed - no code optimization (default).
1, **2**, **3**	Optimize to increase code execution speed (the larger the number, the greater the speed).
s	Optimize for file size.
fast	Same as a level **3** setting, plus **fast** disregards strict standards compliance to allow for additional optimizations.
g	Optimize for debugging experience.

For release builds, use the optimization option **-O2**.

During development, the **-Og** option is useful for debugging the program or library in some situations. Because some bugs manifest only with certain optimization levels, test the program or library with the release optimization level.

GCC offers a large number of options to enable individual optimizations. For more information, see the following Additional resources.

Additional resources

- Using GNU Compiler Collection — 3.10 Options That Control Optimization
- Linux manual page for GCC:

```
$ man gcc
```

3.5. OPTIONS FOR HARDENING CODE WITH GCC

When the compiler transforms source code to object code, it can add various checks to prevent commonly exploited situations and increase security. Choosing the right set of compiler options can help produce more secure programs and libraries, without having to change the source code.

Release version options

The following list of options is the recommended minimum for developers targeting Red Hat Enterprise Linux:

```
$ gcc ... -O2 -g -Wall -Wl,-z,now,-z,relro -fstack-protector-strong -fstack-clash-protection -D_FORTIFY_SOURCE=2 ...
```

- For programs, add the **-fPIE** and **-pie** Position Independent Executable options.
- For dynamically linked libraries, the mandatory **-fPIC** (Position Independent Code) option indirectly increases security.

Development options

Use the following options to detect security flaws during development. Use these options in conjunction with the options for the release version:

```
$ gcc ... -Walloc-zero -Walloca-larger-than -Wextra -Wformat-security -Wvla-larger-than ...
```

Additional resources

- Defensive Coding Guide
- Memory Error Detection Using GCC – Red Hat Developers Blog post

3.6. LINKING CODE TO CREATE EXECUTABLE FILES

Linking is the final step when building a C or C++ application. Linking combines all object files and libraries into an executable file.

Prerequisites

- One or more object file(s)
- GCC must be installed on the system

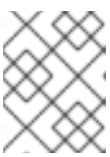
Procedure

1. Change to the directory containing the object code file(s).
2. Run **gcc**:

   ```
   $ gcc ... objfile.o another_object.o ... -o executable-file
   ```

   An executable file named ***executable-file*** is created from the supplied object files and libraries.

   To link additional libraries, add the required options after the list of object files.



NOTE

With C++ source code, replace the **gcc** command with **g++** for convenient handling of C++ Standard Library dependencies.

3.7. EXAMPLE: BUILDING A C PROGRAM WITH GCC

This example shows the exact steps to build a simple sample C program.

Prerequisites

- You must understand how to use GCC.

Procedure

1. Create a directory **hello-c** and change to it:

```
$ mkdir hello-c
$ cd hello-c
```

2. Create file **hello.c** with the following contents:

```
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

3. Compile the code with GCC:

```
$ gcc -c hello.c
```

   The object file **hello.o** is created.

4. Link an executable file **helloworld** from the object file:

```
$ gcc hello.o -o helloworld
```

5. Run the resulting executable file:

```
$ ./helloworld
Hello, World!
```

3.8. EXAMPLE: BUILDING A C++ PROGRAM WITH GCC

This example shows the exact steps to build a sample minimal C++ program.

Prerequisites

- You must understand the difference between **gcc** and **g++**.

Procedure

1. Create a directory **hello-cpp** and change to it:

```
$ mkdir hello-cpp
$ cd hello-cpp
```

2. Create file **hello.cpp** with the following contents:

```
#include <iostream>

int main() {
  std::cout << "Hello, World!\n";
  return 0;
}
```

3. Compile the code with **g++**:

```
$ g++ -c hello.cpp
```

The object file **hello.o** is created.

4. Link an executable file **helloworld** from the object file:

```
$ g++ hello.o -o helloworld
```

5. Run the resulting executable file:

```
$ ./helloworld
Hello, World!
```

CHAPTER 4. USING LIBRARIES WITH GCC

This chapter describes using libraries in code.

4.1. LIBRARY NAMING CONVENTIONS

A special file name convention is used for libraries: A library known as **foo** is expected to exist as file **lib*foo*.so** or **lib*foo*.a**. This convention is automatically understood by the linking input options of GCC, but not by the output options:

- When linking against the library, the library can be specified only by its name **foo** with the **-l** option as **-l*foo***:

```
$ gcc ... -lfoo ...
```

- When creating the library, the full file name **lib*foo*.so** or **lib*foo*.a** must be specified.

4.2. STATIC AND DYNAMIC LINKING

Developers have a choice of using static or dynamic linking when building applications with fully compiled languages. This section lists the differences, particularly in the context using the C and C++ languages on Red Hat Enterprise Linux. To summarize, Red Hat discourages the use of static linking in applications for Red Hat Enterprise Linux.

Comparison of static and dynamic linking

Static linking makes libraries part of the resulting executable file. Dynamic linking keeps these libraries as separate files.

Dynamic and static linking can be compared in a number of ways:

Resource use

Static linking results in larger executable files which contain more code. This additional code coming from libraries cannot be shared across multiple programs on the system, increasing file system usage and memory usage at run time. Multiple processes running the same statically linked program will still share the code.
On the other hand, static applications need fewer run-time relocations, leading to reduced startup time, and require less private resident set size (RSS) memory. Generated code for static linking can be more efficient than for dynamic linking due to the overhead introduce by position-independent code (PIC).

Security

Dynamically linked libraries which provide ABI compatibility can be updated without changing the executable files depending on these libraries. This is especially important for libraries provided by Red Hat as part of Red Hat Enterprise Linux, where Red Hat provides security updates. Static linking against any such libraries is strongly discouraged.

Compatibility

Static linking appears to provide executable files independent of the versions of libraries provided by the operating system. However, most libraries depend on other libraries. With static linking, this dependency becomes inflexible and as a result, both forward and backward compatibility is lost. Static linking is guaranteed to work only on the system where the executable file was built.

WARNING

Applications linking statically libraries from the GNU C library (**glibc**) still require **glibc** to be present on the system as a dynamic library. Furthermore, the dynamic library variant of **glibc** available at the application's run time must be a bitwise identical version to that present while linking the application. As a result, static linking is guaranteed to work only on the system where the executable file was built.

Support coverage

Most static libraries provided by Red Hat are in the *CodeReady Linux Builder* channel and not supported by Red Hat.

Functionality

Some libraries, notably the GNU C Library (**glibc**), offer reduced functionality when linked statically. For example, when statically linked, **glibc** does not support threads and any form of calls to the **dlopen()** function in the same program.

As a result of the listed disadvantages, static linking should be avoided at all costs, particularly for whole applications and the **glibc** and **libstdc++** libraries.

Cases for static linking

Static linking might be a reasonable choice in some cases, such as:

- Using a library which is not enabled for dynamic linking.
- Fully static linking can be required for running code in an empty **chroot** environment or container. However, static linking using the **glibc-static** package is not supported by Red Hat.

Additional resources

- Red Hat Enterprise Linux 8: Application Compatibility GUIDE
- Description of the The CodeReady Linux Builder repository in the *Package manifest*

4.3. USING A LIBRARY WITH GCC

A library is a package of code which can be reused in your program. A C or C++ library consists of two parts:

- The library code
- Header files

Compiling code that uses a library

The header files describe the interface of the library: The functions and variables available in the library. Information from the header files is needed for compiling the code.

Typically, header files of a library will be placed in a different directory than your application's code. To tell GCC where the header files are, use the **-I** option:

```
$ gcc ... -Iinclude_path ...
```

Replace *include_path* with the actual path to the header file directory.

The **-I** option can be used multiple times to add multiple directories with header files. When looking for a header file, these directories are searched in the order of their appearance in the **-I** options.

Linking code that uses a library

When linking the executable file, both the object code of your application and the binary code of the library must be available. The code for static and dynamic libraries is present in different forms:

- Static libraries are available as archive files. They contain a group of object files. The archive file has an file name extension **.a**.
- Dynamic libraries are available as shared objects. They are a form of an executable file. A shared object has an file name extension **.so**.

To tell GCC where the archives or shared object files of a are, use the **-L** option:

```
$ gcc ... -Llibrary_path -lfoo ...
```

Replace *library_path* with the actual path to the library directory.

The **-L** option can be used multiple times to add multiple directories. When looking for a library, these directories are searched in the order of their **-L** options.

The order of options matters: GCC cannot link against a library **foo** unless it knows the directory with this library. Therefore, use the **-L** options to specify library directories before using the **-l** options for linking against libraries.

Compiling and linking code which uses a library in one step

When the situation allows the code to be compiled and linked in one **gcc** command, use the options for both situations mentioned above at once.

Additional resources

- Using the GNU Compiler Collection (GCC) — 3.15 Options for Directory Search
- Using the GNU Compiler Collection (GCC) — 3.14 Options for Linking

4.4. USING A STATIC LIBRARY WITH GCC

Static libraries are available as archives containing object files. After linking, they become part of the resulting executable file.

NOTE

Red Hat discourages use of static linking for security reasons. See Section 4.2, "Static and dynamic linking". Use static linking only when necessary, especially against libraries provided by Red Hat.

Prerequisites

- GCC must be installed on your system.
- You must understand static and dynamic linking.
- You have a set of source or object files forming a valid program, requiring some static library **foo** and no other libraries.
- The **foo** library is available as a file **libfoo.a**, and no file **libfoo.so** is provided for dynamic linking.

NOTE

Most libraries which are part of Red Hat Enterprise Linux are supported for dynamic linking only. The steps below only work for libraries which are *not* enabled for dynamic linking.

Procedure

To link a program from source and object files, adding a statically linked library **foo**, which is to be found as a file **libfoo.a**:

1. Change to the directory containing your code.
2. Compile the program source files with headers of the **foo** library:

```
$ gcc ... -Iheader_path -c ...
```

   Replace *header_path* with a path to a directory containing the header files for the **foo** library.

3. Link the program with the **foo** library:

```
$ gcc ... -Llibrary_path -lfoo ...
```

   Replace *library_path* with a path to a directory containing the file **libfoo.a**.

4. To run the program later, simply:

```
$ ./program
```

CAUTION

The **-static** GCC option related to static linking forbids all dynamic linking. Instead, use the **-Wl,-Bstatic** and **-Wl,-Bdynamic** options to control linker behavior more precisely. See Section 4.6, "Using both static and dynamic libraries with GCC".

4.5. USING A DYNAMIC LIBRARY WITH GCC

Dynamic libraries are available as standalone executable files, required at both linking time and run time. They stay independent of your application's executable file.

Prerequisites

- GCC must be installed on the system.

- A set of source or object files forming a valid program, requiring some dynamic library **foo** and no other libraries.
- The **foo** library must be available as a file *libfoo.so*.

Linking a program against a dynamic library

To link a program against a dynamic library **foo**:

```
$ gcc ... -Llibrary_path -lfoo ...
```

When a program is linked against a dynamic library, the resulting program must always load the library at run time. There are two options for locating the library:

- Using a **rpath** value stored in the executable file itself
- Using the **LD_LIBRARY_PATH** variable at runtime

Using a rpath Value Stored in the Executable File

The **rpath** is a special value saved as a part of an executable file when it is being linked. Later, when the program is loaded from its executable file, the runtime linker will use the **rpath** value to locate the library files.

While linking with **GCC**, to store the path *library_path* as **rpath**:

```
$ gcc ... -Llibrary_path -lfoo -Wl,-rpath=library_path ...
```

The path *library_path* must point to a directory containing the file *libfoo.so*.

CAUTION

There is no space after the comma in the **-Wl,-rpath=** option!

To run the program later:

```
$ ./program
```

Using the LD_LIBRARY_PATH environment variable

If no **rpath** is found in the program's executable file, the runtime linker will use the **LD_LIBRARY_PATH** environment variable. The value of this variable must be changed for each program according to the path where the shared library objects are to be found.

To run the program without **rpath** set, with libraries present in path *library_path*:

```
$ export LD_LIBRARY_PATH=library_path:$LD_LIBRARY_PATH
$ ./program
```

Leaving out the **rpath** value offers flexibility, but requires setting the **LD_LIBRARY_PATH** variable every time the program is to run.

Placing the Library into the Default Directories

The runtime linker configuration specifies a number of directories as a default location of dynamic library files. To use this default behaviour, copy your library to the appropriate directory.

A full description of the dynamic linker behavior is out of scope of this document. For more information, see the following resources:

- Linux manual pages for the dynamic linker:

```
$ man ld.so
```

- Contents of the **/etc/ld.so.conf** configuration file:

```
$ cat /etc/ld.so.conf
```

- Report of the libraries recognized by the dynamic linker without additional configuration, which includes the directories:

```
$ ldconfig -v
```

4.6. USING BOTH STATIC AND DYNAMIC LIBRARIES WITH GCC

Sometimes it is required to link some libraries statically and some dynamically. This situation brings some challenges.

Prerequisites

- Understanding static and dynamic linking

Introduction

gcc recognizes both dynamic and static libraries. When the **-l*foo*** option is encountered, **gcc** will first attempt to locate a shared object (a **.so** file) containing a dynamically linked version of the **foo** library, and then look for the archive file (**.a**) containing a static version of the library. Thus, the following situations can result from this search:

- Only the shared object is found and **gcc** links against it dynamically
- Only the archive is found and **gcc** links against it statically
- Both the shared object and archive are found; **gcc** selects by default dynamic linking against the shared object
- Neither shared object nor archive is found and linking fails

Because of these rules, the best way to select the static or dynamic version of library for linking is having only that version found by **gcc**. This can be controlled to some extent by using or leaving out directories containing the library versions, when specifying the **-L*path*** options.

Additionally, because dynamic linking is the default, the only situation where linking must be explicitly specified is when a library with both versions present should be linked statically. There are two possible resolutions:

- Specifying the static libraries by file path instead of the **-l** option
- Using the **-Wl** option to pass options to the linker

Specifying the static libraries by file

Usually, **gcc** is instructed to link against a library **foo** with the **-l*foo*** option. However, it is possible to specify the full path to file **lib*foo*.a** containing the library instead:

```
$ gcc ... path/to/libfoo.a ...
```

From the file extension **.a**, **gcc** will understand that this is a library to link with the program. However, specifying the full path to the library file is a less flexible method.

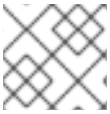
Using the -Wl option

The **gcc** option **-Wl** is a special option for passing options to the underlying linker. Syntax of this option differs from the other **gcc** options: It is followed by a comma-separated list of linker options, so that the linker options do not get mixed up with space-separated **gcc** options.

The **ld** linker used by **gcc** offers the options **-Bstatic** and **-Bdynamic** to specify whether libraries following this option should be linked statically or dynamically, respectively. After passing **-Bstatic** and a library to the linker, the default dynamic linking behaviour must be restored manually for the following libraries to be linked dynamically with the **-Bdynamic** option.

To link a program, linking library **first** statically (**libfirst.a**) and **second** dynamically (**libsecond.so**):

```
$ gcc ... -Wl,-Bstatic -lfirst -Wl,-Bdynamic -lsecond ...
```



NOTE

gcc can be configured to use linkers other than the default **ld**.

Additional resources

- Using the GNU Compiler Collection (GCC) — 3.14 Options for Linking
- Documentation for binutils 2.27 — 2.1 Command Line Options

CHAPTER 5. CREATING LIBRARIES WITH GCC

This chapter describes steps for creating libraries and explains the necessary concepts used by the Linux operating system for libraries.

5.1. LIBRARY NAMING CONVENTIONS

A special file name convention is used for libraries: A library known as **foo** is expected to exist as file **lib*foo*.so** or **lib*foo*.a**. This convention is automatically understood by the linking input options of GCC, but not by the output options:

- When linking against the library, the library can be specified only by its name **foo** with the **-l** option as **-l*foo***:

```
$ gcc ... -lfoo ...
```

- When creating the library, the full file name **lib*foo*.so** or **lib*foo*.a** must be specified.

5.2. THE SONAME MECHANISM

Dynamically loaded libraries (shared objects) use a mechanism called *soname* to manage multiple compatible versions of a library.

Prerequisites

- You must understand dynamic linking and libraries.
- Understanding the concept of ABI compatibility
- You must understand library naming conventions.
- You must understand symbolic links.

Problem introduction

A dynamically loaded library (shared object) exists as an independent executable file. This makes it possible to update the library without updating the applications that depend on it. However, the following problems arise with this concept:

- Identification of the actual version of the library
- Need for multiple versions of the same library present
- Signalling ABI compatibility of each of the multiple versions

The soname mechanism

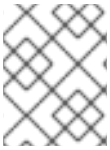
To resolve this, Linux uses a mechanism called soname.

A library **foo** version *X.Y* is ABI-compatible with other versions with the same value of *X* in version number. Minor changes preserving compatibility increase the number *Y*. Major changes that break compatibility increase the number *X*.

The actual library **foo** version *X.Y* exists as a file **libfoo.so.*x.y***. Inside the library file, a soname is recorded with value **libfoo.so.x** to signal the compatibility.

When applications are built, the linker looks for the library by searching for the file **libfoo.so**. A symbolic link with this name must exist, pointing to the actual library file. The linker then reads the soname from the library file and records it into the application executable file. Finally, the linker creates the application such that it declares dependency on the library using the soname, not name or file name.

When the runtime dynamic linker links an application before running, it reads the soname from application's executable file. This soname is **libfoo.so.*x***. A symbolic link with this name must exist, pointing to the actual library file. This allows loading the library, regardless of the *Y* component of version, because the soname does not change.



NOTE

The *Y* component of the version number is not limited to just a single number. Additionally, some libraries encode version in their name.

Reading soname from a file

To display the soname of a library file **somelibrary**:

```
$ objdump -p somelibrary | grep SONAME
```

Replace *somelibrary* with the actual file name of the library you wish to examine.

5.3. CREATING DYNAMIC LIBRARIES WITH GCC

Dynamically linked libraries (shared objects) allow resource conservation through code reuse and increased security by making it easier to update the library code. Follow these steps to build and install a dynamic library from source.

Prerequisites

- You must understand the soname mechanism.
- GCC must be installed on the system.
- You must have source code for a library.

Procedure

1. Change to the directory with library sources.
2. Compile each source file to an object file with the Position independent code option **-fPIC**:

   ```
   $ gcc ... -c -fPIC some_file.c ...
   ```

   The object files have the same file names as the original source code files, but their extension is **.o**.
3. Link the shared library from the object files:

   ```
   $ gcc -shared -o libfoo.so.x.y -Wl,-soname,libfoo.so.x some_file.o ...
   ```

   The used major version number is X and minor version number Y.

4. Copy the ***libfoo.so.x.y*** file to an appropriate location, where the system's dynamic linker can find it. On Red Hat Enterprise Linux, the directory for libraries is **/usr/lib64**:

```
# cp libfoo.so.x.y /usr/lib64
```

   Note that you need root permissions to manipulate files in this directory.

5. Create the symlink structure for soname mechanism:

```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

Additional resources

- The Linux Documentation Project – Program Library HOWTO – 3. Shared Libraries

5.4. CREATING STATIC LIBRARIES WITH GCC AND AR

Creating libraries for static linking is possible through conversion of object files into a special type of archive file.

NOTE

Red Hat discourages use of static linking for security reasons. Use static linking only when necessary, especially against libraries provided by Red Hat. See Section 4.2, "Static and dynamic linking" for more details.

Prerequisites

- GCC and binutils must be installed on the system.
- You must undserstand static and dynamic linking.
- Source file(s) with functions to be shared as a library

Procedure

1. Create intermediate object files with GCC.

```
$ gcc -c source_file.c ...
```

   Append more source files as required. The resulting object files share the file name but use the **.o** file name extension.

2. Turn the object files into a static library (archive) using the **ar** tool from the **binutils** package.

```
$ ar rcs libfoo.a source_file.o ...
```

   File **libfoo.a** is created.

3. Use the **nm** command to inspect the resulting archive:

```
$ nm libfoo.a
```

4. Copy the static library file to the appropriate directory.

5. When linking against the library, GCC will automatically recognize from the **.a** file name extension that the library is an archive for static linking.

```
$ gcc ... -lfoo ...
```

Additional resources

- Linux manual page for *ar(1)*:

```
$ man ar
```

CHAPTER 6. MANAGING MORE CODE WITH MAKE

The GNU make utility, commonly abbreviated **make**, is a tool for controlling the generation of executables from source files. **make** automatically determines which parts of a complex program have changed and need to be recompiled. **make** uses configuration files called Makefiles to control the way programs are built.

6.1. GNU MAKE AND MAKEFILE OVERVIEW

To create a usable form (usually executable files) from the source files of a particular project, perform several necessary steps. Record the actions and their sequence to be able to repeat them later.

Red Hat Enterprise Linux contains GNU **make**, a build system designed for this purpose.

Prerequisites

- Understanding the concepts of compiling and linking

GNU make

GNU **make** reads Makefiles which contain the instructions describing the build process. A Makefile contains multiple *rules* that describe a way to satisfy a certain condition (*target*) with a specific action (*recipe*). Rules can hierarchically depend on another rule.

Running **make** without any options makes it look for a Makefile in the current directory and attempt to reach the default target. The actual Makefile file name can be one of **Makefile**, **makefile**, and **GNUmakefile**. The default target is determined from the Makefile contents.

Makefile details

Makefiles use a relatively simple syntax for defining *variables* and *rules*, which consists of a *target* and a *recipe*. The target specifies what is the output if a rule is executed. The lines with recipes must start with the TAB character.

Typically, a Makefile contains rules for compiling source files, a rule for linking the resulting object files, and a target that serves as the entry point at the top of the hierarchy.

Consider the following **Makefile** for building a C program which consists of a single file, **hello.c**.

```
all: hello

hello: hello.o
	gcc hello.o -o hello

hello.o: hello.c
	gcc -c hello.c -o hello.o
```

This specifies that to reach the target **all**, file **hello** is required. To get **hello**, one needs **hello.o** (linked by **gcc**), which in turn is created from **hello.c** (compiled by **gcc**).

The target **all** is the default target because it is the first target that does not start with a period (.). Running **make** without any arguments is then identical to running **make all**, when the current directory contains this **Makefile**.

Typical makefile

A more typical Makefile uses variables for generalization of the steps and adds a target "clean" - remove everything but the source files.

```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
	$(CC) $(OBJ) -o $@

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJ) $(EXE)
```

Adding more source files to such Makefile requires only adding them to the line where the SOURCE variable is defined.

Additional resources

- GNU make: Introduction — 2 An Introduction to Makefiles

6.2. EXAMPLE: BUILDING A C PROGRAM USING A MAKEFILE

Build a sample C program using a Makefile by following the steps in the following example.

Prerequisites

- You must understand the concepts of Makefiles and **make**.

Procedure

1. Create a directory **hellomake** and change to this directory:

```
$ mkdir hellomake
$ cd hellomake
```

2. Create a file **hello.c** with the following contents:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

3. Create a file **Makefile** with the following contents:

```
CC=gcc
```

```
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
	$(CC) $(OBJ) -o $@

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJ) $(EXE)
```

CAUTION

The Makefile recipe lines must start with the tab character! When copying the text above from the documentation, the cut-and-paste process may paste spaces instead of tabs. If this happens, correct the issue manually.

4. Run **make**:

```
$ make
gcc -c -Wall hello.c -o hello.o
gcc hello.o -o hello
```

This creates an executable file **hello**.

5. Run the executable file **hello**:

```
$ ./hello
Hello, World!
```

6. Run the Makefile target **clean** to remove the created files:

```
$ make clean
rm -rf hello.o hello
```

6.3. DOCUMENTATION RESOURCES FOR MAKE

For more information about **make**, see the resources listed below.

Installed documentation

- Use the **man** and **info** tools to view manual pages and information pages installed on your system:

```
$ man make
$ info make
```

Online documentation

- The GNU Make Manual hosted by the Free Software Foundation
- Red Hat Developer Toolset User Guide — Chapter 3. GNU make

APPENDIX A. DIFFERENCES IN DEVELOPMENT TOOLS IN RHEL 8

The following sections list important differences between Red Hat Enterprise Linux 8 and 7.

A.1. CHANGES IN TOOLCHAIN SINCE RHEL 7

The following sections list changes in toolchain since the release of the described components in Red Hat Enterprise Linux 7. See also Release notes for Red Hat Enterprise Linux 8.0 .

A.1.1. Changes in GCC in RHEL 8

In Red Hat Enterprise Linux 8, the GCC toolchain is based on the GCC 8.2 release series. Notable changes since Red Hat Enterprise Linux 7 include:

- Numerous general optimizations have been added, such as alias analysis, vectorizer improvements, identical code folding, inter-procedural analysis, store merging optimization pass, and others.
- The Address Sanitizer has been improved.
- The Leak Sanitizer for detection of memory leaks has been added.
- The Undefined Behavior Sanitizer for detection of undefined behavior has been added.
- Debug information can now be produced in the DWARF5 format. This capability is experimental.
- The source code coverage analysis tool GCOV has been extended with various improvements.
- Support for the OpenMP 4.5 specification has been added. Additionally, the offloading features of the OpenMP 4.0 specification are now supported by the C, C++, and Fortran compilers.
- New warnings and improved diagnostics have been added for static detection of certain likely programming errors.
- Source locations are now tracked as ranges rather than points, which allows much richer diagnostics. The compiler now offers “fix-it” hints, suggesting possible code modifications. A spell checker has been added to offer alternative names and ease detecting typos.

Security

GCC has been extended to provide tools to ensure additional hardening of the generated code. Improvements related to security include:

- The **__builtin_add_overflow**, **__builtin_sub_overflow**, and **__builtin_mul_overflow** built-in functions for arithmetics with overflow checking have been added.
- The **-fstack-clash-protection** option has been added to generate additional code guarding against stack clash.
- The **-fcf-protection** option was introduced to check target addresses of control-flow instructions for increased program security.
- The new **-Wstringop-truncation** warning option lists calls to bounded string manipulation functions such as **strncat**, **strncpy**, or **stpncpy** that might truncate the copied string or leave the destination unchanged.

- The **-Warray-bounds** warning option has been improved to detect out-of-bounds array indices and pointer offsets better.

- The **-Wclass-memaccess** warning option has been added to warn about potentially unsafe manipulation of objects of non-trivial class types by raw memory access functions such as **memcpy** or **realloc**.

Architecture and processor support

Improvements to architecture and processor support include:

- Multiple new architecture-specific options for the Intel AVX-512 architecture, a number of its microarchitectures, and Intel Software Guard Extensions (SGX) have been added.

- Code generation can now target the 64-bit ARM architecture LSE extensions, ARMv8.2-A 16-bit Floating-Point Extensions (FPE), and ARMv8.2-A, ARMv8.3-A, and ARMv8.4-A architecture versions.

- Handling of the **-march=native** option on the ARM and 64-bit ARM architectures has been fixed.

- Support for the z13 and z14 processors of the IBM Z architecture has been added.

Languages and standards

Notable changes related to languages and standards include:

- The default standard used when compiling code in the C language has changed to C17 with GNU extensions.

- The default standard used when compiling code in the C++ language has changed to C++14 with GNU extensions.

- The C++ runtime library now supports the C++11 and C++14 standards.

- The C++ compiler now implements the C++14 standard with many new features such as variable templates, aggregates with non-static data member initializers, the extended **constexpr** specifier, sized deallocation functions, generic lambdas, variable-length arrays, digit separators, and others.

- Support for the C language standard C11 has been improved: ISO C11 atomics, generic selections, and thread-local storage are now available.

- The new **__auto_type** GNU C extension provides a subset of the functionality of C++11 **auto** keyword in the C language.

- The **_FloatN** and **_FloatNx** type names specified by the ISO/IEC TS 18661-3:2015 standard are now recognized by the C front end.

- The default standard used when compiling code in the C language has changed to C17 with GNU extensions. This has the same effect as using the **--std=gnu17** option. Previously, the default was C89 with GNU extensions.

- GCC can now experimentally compile code using the C++17 language standard, and certain features from the C++20 standard.

- Passing an empty class as an argument now takes up no space on the Intel 64 and AMD64 architectures, as required by the platform ABI. Passing or returning a class with only deleted

copy and move constructors now uses the same calling convention as a class with a non-trivial copy or move constructor.

- The value returned by the C++11 **alignof** operator has been corrected to match the C **_Alignof** operator and return minimum alignment. To find the preferred alignment, use the GNU extension **__alignof__**.
- The main version of the **libgfortran** library for Fortran language code has been changed to 5.
- Support for the Ada (GNAT), GCC Go, and Objective C/C++ languages has been removed. Use the Go Toolset for Go code development.

Additional resources

- See also the Red Hat Enterprise Linux 8 Release Notes .
- Using Go Toolset

A.1.2. Security enhancements in GCC in RHEL 8

This section decribes in detail the changes in GCC related to security and added since the release of Red Hat Enterprise Linux 7.0.

New warnings

These warning options have been added:

Option	Displays warnings for
-Wstringop-truncation	Calls to bounded string manipulation functions such as **strncat**, **strncpy**, and **stpncpy** that might either truncate the copied string or leave the destination unchanged.
-Wclass-memaccess	Objects of non-trivial class types manipulated in potentially unsafe ways by raw memory functions such as **memcpy**, or **realloc**. The warning helps detect calls that bypass user-defined constructors or copy-assignment operators, corrupt virtual table pointers, data members of const-qualified types or references, or member pointers. The warning also detects calls that would bypass access controls to data members.
-Wmisleading-indentation	Places where the indentation of the code gives a misleading idea of the block structure of the code to a human reader.
-Walloc-size-larger-than=*size*	Calls to memory allocation functions where the amount of memory to allocate exceeds *size*. Works also with functions where the allocation is specified by multiplying two parameters, and with any functions decorated with attribute **alloc_size**.
-Walloc-zero	Calls to memory allocation functions that attempt to allocate zero amount of memory. Works also with functions where the allocation is specified by multiplying two parameters, and with any functions decorated with attribute **alloc_size**.

Option	Displays warnings for
-Walloca	All calls to the **alloca** function.
-Walloca-larger-than=*size*	Calls to the **alloca** function where the requested memory is more than*size*.
-Wvla-larger-than=*size*	Definitions of Variable Length Arrays (VLA) that can either exceed the specified size or whose bound is not known to be sufficiently constrained.
-Wformat-overflow=*level*	Both certain and likely buffer overflow in calls to the **sprintf** family of formatted output functions. For more details and explanation of the *level* value, see the *gcc(1)* manual page.
-Wformat-truncation=*level*	Both certain and likely output truncation in calls to the **snprintf** family of formatted output functions. For more details and explanation of the *level* value, see the *gcc(1)* manual page.
-Wstringop-overflow=*type*	Buffer overflow in calls to string handling functions such as **memcpy** and **strcpy**. For more details and explanation of the*level* value, see the *gcc(1)* manual page.

Warning improvements

These GCC warnings have been improved:

- The **-Warray-bounds** option has been improved to detect more instances of out-of-bounds array indices and pointer offsets. For example, negative or excessive indices into flexible array members and string literals are detected.
- The **-Wrestrict** option introduced in GCC 7 has been enhanced to detect many more instances of overlapping accesses to objects via restrict-qualified arguments to standard memory and string manipulation functions such as **memcpy** and **strcpy**.
- The **-Wnonnull** option has been enhanced to detect a broader set of cases of passing null pointers to functions that expect a non-null argument (decorated with attribute **nonnull**).

New UndefinedBehaviorSanitizer

A new run-time sanitizer for detecting undefined behavior called UndefinedBehaviorSanitizer has been added. The following options are noteworthy:

Option	Check
-fsanitize=float-divide-by-zero	Detect floating-point division by zero.
-fsanitize=float-cast-overflow	Check that the result of floating-point type to integer conversions do not overflow.

Option	Check
-fsanitize=bounds	Enable instrumentation of array bounds and detect out-of-bounds accesses.
-fsanitize=alignment	Enable alignment checking and detect various misaligned objects.
-fsanitize=object-size	Enable object size checking and detect various out-of-bounds accesses.
-fsanitize=vptr	Enable checking of C++ member function calls, member accesses and some conversions between pointers to base and derived classes. Additonally, detect when referenced objects do not have correct dynamic type.
-fsanitize=bounds-strict	Enable strict checking of array bounds. This enables **-fsanitize=bounds** as well as instrumentation of flexible array member-like arrays.
-fsanitize=signed-integer-overflow	Diagnose arithmetic overflows even on arithmetic operations with generic vectors.
-fsanitize=builtin	Diagnose at run time invalid arguments to **__builtin_clz** or **__builtin_ctz** prefixed builtins. Includes checks from **-fsanitize=undefined**.
-fsanitize=pointer-overflow	Perform cheap run time tests for pointer wrapping. Includes checks from **-fsanitize=undefined**.

New options for AddressSanitizer

These options have been added to AddressSanitizer:

Option	Check
-fsanitize=pointer-compare	Warn about comparison of pointers that point to a different memory object.
-fsanitize=pointer-subtract	Warn about subtraction of pointers that point to a different memory object.
-fsanitize-address-use-after-scope	Sanitize variables whose address is taken and used after a scope where the variable is defined.

Other sanitizers and instrumentation

- The option **-fstack-clash-protection** has been added to insert probes when stack space is allocated statically or dynamically to reliably detect stack overflows and thus mitigate the attack vector that relies on jumping over a stack guard page provided by the operating system.

- A new option **-fcf-protection=[full|branch|return|none]** has been added to perform code instrumentation and increase program security by checking that target addresses of control-flow transfer instructions (such as indirect function call, function return, indirect jump) are valid.

Additional resources

- For more details and explanation of the values supplied to some of the options above, see the *gcc(1)* manual page:

```
$ man gcc
```

A.2. COMPILER TOOLSETS

RHEL 8.0 provides the following compiler toolsets as Application Streams:

- Clang and LLVM Toolset 7.0.1, which provides the LLVM compiler infrastructure framework, the Clang compiler for the C and C++ languages, the LLDB debugger, and related tools for code analysis. See the Using Clang and LLVM Toolset document.
- Rust Toolset 1.31, which provides the Rust programming language compiler **rustc**, the **cargo** build tool and dependency manager, the **cargo-vendor** plugin, and required libraries. See the Using Rust Toolset document.
- Go Toolset 1.11.5, which provides the Go programming language tools and libraries. Go is alternatively known as **golang**. See the Using Go Toolset document.

A.3. COMPATIBILITY-BREAKING CHANGES IN GDB

The version of GDB provided in Red Hat Enterprise Linux 8 contains a number of changes that break compatibility, especially for cases where the GDB output is read directly from the terminal. The following sections provide more details about these changes.

Parsing output of GDB is not recommended. Prefer scripts using the Python GDB API or the GDB Machine Interface (MI).

GDBserver now starts inferiors with shell

To enable expansion and variable substitution in inferior command line arguments, GDBserver now starts the inferior in a shell, same as GDB.

To disable using the shell:

- When using the **target extended-remote** GDB command, disable shell with the **set startup-with-shell off** command.
- When using the **target remote** GDB command, disable shell with the **--no-startup-with-shell** option of GDBserver.

Example A.1. Example of shell expansion in remote GDB inferiors

This example shows how running the /**bin**/**echo** /* command through GDBserver differs on Red Hat Enterprise Linux versions 7 and 8:

- On RHEL 7:

```
$ gdbserver --multi :1234
```

```
$ gdb -batch -ex 'target extended-remote :1234' -ex 'set remote exec-file /bin/echo' -ex
'file /bin/echo' -ex 'run /*'
/*
```

- On RHEL 8:

```
$ gdbserver --multi :1234
$ gdb -batch -ex 'target extended-remote :1234' -ex 'set remote exec-file /bin/echo' -ex
'file /bin/echo' -ex 'run /*'
/bin /boot (...) /tmp /usr /var
```

gcj support removed

Support for debugging Java programs compiled with the GNU Compiler for Java (**gcj**) has been removed.

New syntax for symbol dumping maintenance commands

The symbol dumping maintenance commands syntax now includes options before file names. As a result, commands that worked with GDB in RHEL 7 do not work in RHEL 8.

As an example, the following command no longer stores symbols in a file, but produces an error message:

```
(gdb) maintenance print symbols /tmp/out main.c
```

The new syntax for the symbol dumping maintenance commands is:

```
maint print symbols [-pc address] [--] [filename]
maint print symbols [-objfile objfile] [-source source] [--] [filename]
maint print psymbols [-objfile objfile] [-pc address] [--] [filename]
maint print psymbols [-objfile objfile] [-source source] [--] [filename]
maint print msymbols [-objfile objfile] [--] [filename]
```

Thread numbers are no loger global

Previously, GDB used only global thread numbering. The numbering has been extended to be displayed per inferior in the form **inferior_num.thread_num**, such as **2.1**. As consequence, thread numbers in the **$_thread** convenience variable and in the **InferiorThread.num** Python attribute are no longer unique between inferiors.

GDB now stores a second thread ID per thread, called the global thread ID, which is the new equivalent of thread numbers in previous releases. To access the global thread number, use the **$_gthread** convenience variable and **InferiorThread.global_num** Python attribute.

For backwards compatibility, the Machine Interface (MI) thread IDs always contains the global IDs.

Example A.2. Example of GDB thread number changes

On Red Hat Enterprise Linux 7:

```
# debuginfo-install coreutils
$ gdb -batch -ex 'file echo' -ex start -ex 'add-inferior' -ex 'inferior 2' -ex 'file echo' -ex start -ex 'info
threads' -ex 'pring $_thread' -ex 'inferior 1' -ex 'pring $_thread'
(...)
  Id   Target Id         Frame
* 2    process 203923 "echo" main (argc=1, argv=0x7fffffffdb88) at src/echo.c:109
```

```
  1    process 203914 "echo" main (argc=1, argv=0x7fffffffdb88) at src/echo.c:109
$1 = 2
(...)
$2 = 1
```

On Red Hat Enterprise Linux 8:

```
# dnf debuginfo-install coreutils
$ gdb -batch -ex 'file echo' -ex start -ex 'add-inferior' -ex 'inferior 2' -ex 'file echo' -ex start -ex 'info
threads' -ex 'pring $_thread' -ex 'inferior 1' -ex 'pring $_thread'
(...)
  Id   Target Id         Frame
  1.1  process 4106488 "echo" main (argc=1, argv=0x7fffffffce58) at ../src/echo.c:109
* 2.1  process 4106494 "echo" main (argc=1, argv=0x7fffffffce58) at ../src/echo.c:109
$1 = 1
(...)
$2 = 1
```

Memory for value contents can be limited

Previously, GDB did not limit the amount of memory allocated for value contents. As a consequence, debugging incorrect programs could cause GDB to allocate too much memory. The **max-value-size** setting has been added to enable limiting the amount of allocated memory. The default value of this limit is 64 KiB. As a result, GDB in Red Hat Enterprise Linux 8 will not display too large values, but report that the value is too large instead.

As an example, printing a value defined as **char s[128*1024];** produces different results:

- On Red Hat Enterprise Linux 7, **$1 = 'A' <repeats 131072 times>**
- On Red Hat Enterprise Linux 8, **value requires 131072 bytes, which is more than max-value-size**

Sun version of stabs format no longer supported

Support for the Sun version of the **stabs** debug file format has been removed. The **stabs** format produced by GCC in RHEL with the **gcc -gstabs** option is still supported by GDB.

Sysroot handling changes

The **set sysroot *path*** command specifies system root when searching for files needed for debugging. Directory names supplied to this command may now be prefixed with the string **target:** to make GDB read the shared libraries from the target system (both local and remote). The formerly available **remote:** prefix is now treated as **target:**. Additionally, the default system root value has changed from empty string to **target:** for backward compatibility.

The specified system root is prepended to the file name of the main executable, when GDB starts processes remotely, or when it attaches to already running processes (both local and remote). This means that for remote processes, the default value **target:** makes GDB always try to load the debugging information from the remote system. To prevent this, run the **set sysroot** command before the **target remote** command so that local symbol files are found before the remote ones.

HISTSIZE no longer controls GDB command history size

Previously, GDB used the **HISTSIZE** environment variable to determine how long command history should be kept. GDB has been changed to use the **GDBHISTSIZE** environment variable instead. This variable is specific only to GDB. The possible values and their effects are:

- a positive number - use command history of this size,
- **-1** or an empty string - keep history of all commands,
- non-numeric values - ignored.

Completion limiting added

The maximum number of candidates considered during completion can now be limited using the **set max-completions** command. To show the current limit, run the **show max-completions** command. The default value is 200. This limit prevents GDB from generating excessively large completion lists and becoming unresponsive.

As an example, the output after the input **p <tab><tab>** is:

- on RHEL 7: **Display all 29863 possibilities? (y or n)**
- on RHEL 8: **Display all 200 possibilities? (y or n)**

HP-UX XDB compatibility mode removed

The **-xdb** option for the HP-UX XDB compatibility mode has been removed from GDB.

Handling signals for threads

Previously, GDB could deliver a signal to the current thread instead of the thread for which the signal was actually sent. This bug has been fixed, and GDB now always passes the signal to the correct thread when resuming execution.

Additionally, the **signal** command now always correctly delivers the requested signal to the current thread. If the program is stopped for a signal and the user switched threads, GDB asks for confirmation.

Breakpoint modes always-inserted off and auto merged

The **breakpoint always-inserted** setting has been changed. The **auto** value and corresponding behavior has been removed. The default value is now **off**. Additionally, the **off** value now causes GDB to not remove breakpoints from the target until all threads stop.

remotebaud commands no longer supported

The **set remotebaud** and **show remotebaud** commands are no longer supported. Use the **set serial baud** and **show serial baud** commands instead.

A.4. COMPATIBILITY-BREAKING CHANGES IN COMPILERS AND DEVELOPMENT TOOLS

C++ ABI change in std::string and std::list

The Application Binary Interface (ABI) of the **std::string** and **std::list** classes from the **libstdc++** library changed between RHEL 7 (GCC 4.8) and RHEL 8 (GCC 8) to conform to the C++11 standard. The **libstdc++** library supports both the old and new ABI, but some other C++ system libraries do not. As a consequence, applications that dynamically link against these libraries will need to be rebuilt. This affects all C++ standard modes, including C++98. It also affects applications built with Red Hat Developer Toolset compilers for RHEL 7, which kept the old ABI to maintain compatibility with the system libraries.

librtkaio removed

With this update, the **librtkaio** library has been removed. This library provided high performance real time asynchronous I/O access for some files, which was based on Linux kernel Asynchronous I/O support (KAIO).

As a result of the removal:

- Applications using the **LD_PRELOAD** method to load **librtkaio** display a warning about a missing library, load the **librt** library instead and run correctly.
- Applications using the **LD_LIBRARY_PATH** method to load **librtkaio** load the **librt** library instead and run correctly, without any warning.
- Applications using the **dlopen()** system call to access **librtkaio** directly load the **librt** library instead.

Users of **librtkaio** have the following options:

- Use the fallback mechanism described above, without any changes to their applications.
- Change code of their applications to use the **librt** library, which offers a compatible POSIX-compliant API.
- Change code of their applications to use the **libaio** library, which offers a compatible API.

Both **librt** and **libaio** can provide comparable features and performance under specific conditions.

Note that the **libaio** package has Red Hat compatibility level of 2, while **librtk** and the removed **librtkaio** level 1.

For more details, see https://fedoraproject.org/wiki/Changes/GLIBC223_librtkaio_removal

Sun RPC and NIS interfaces removed from glibc

The **glibc** library no longer provides Sun RPC and NIS interfaces for new applications. These interfaces are now available only for running legacy applications. Developers must change their applications to use the **libtirpc** library instead of Sun RPC and **libnsl2** instead of NIS. Applications can benefit from IPv6 support in the replacement libraries.

Valgrind library for MPI debugging support removed

The **libmpiwrap.so** wrapper library for **Valgrind** provided by the **valgrind-openmpi** package has been removed. This library enabled **Valgrind** to debug programs using the Message Passing Interface (MPI). This library was specific to the Open MPI implementation version in previous versions of Red Hat Enterprise Linux.

Users of **libmpiwrap.so** are encouraged to build their own version from upstream sources specific to their MPI implementation and version. Supply these custom-built libraries to **Valgrind** using the **LD_PRELOAD** technique.

Development headers and static libraries removed from valgrind-devel

Previously, the **valgrind-devel** sub-package used to include development files for developing custom valgrind tools. This update removes these files because they do not have a guaranteed API, have to be linked statically, and are unsupported. The **valgrind-devel** package still does contain the development files for valgrind-aware programs and header files such as **valgrind.h**, **callgrind.h**, **drd.h**, **helgrind.h**, and **memcheck.h**, which are stable and well supported.

The nosegneg libraries for 32-bit Xen have been removed

Previously, the **glibc** i686 packages contained an alternative **glibc** build, which avoided the use of the thread descriptor segment register with negative offsets (**nosegneg**). This alternative build was only used in the 32-bit version of the Xen Project hypervisor without hardware virtualization support, as an optimization to reduce the cost of full paravirtualization. These alternative builds are no longer used and they have been removed.

GCC no longer builds Ada, Go, and Objective C/C++ code

Capability for building code in the Ada (GNAT), GCC Go, and Objective C/C++ languages has been removed from the GCC compiler.

To build Go code, use the Go Toolset instead.

make new operator != causes a different interpretation of certain existing makefile syntax
The **!=** shell assignment operator has been added to GNU **make** as an alternative to the **$(shell ...)** function to increase compatibility with BSD makefiles. As a consequence, variables with name ending in exclamation mark and immediately followed by assignment such as **variable!=value** are now interpreted as the shell assignment. To restore the previous behavior, add a space after the exclamation mark, such as **variable! =value**.

For more details and differences between the operator and the function, see the GNU **make** manual.