Red Hat Enterprise Linux 9.0

Developing C and C++ applications in RHEL 9

Setting up a developer workstation, and developing and debugging C and C++ applications in Red Hat Enterprise Linux 9

Last Updated: 2024-11-13

Red Hat Enterprise Linux 9.0 Developing C and C++ applications in RHEL 9

Setting up a developer workstation, and developing and debugging C and C++ applications in Red Hat Enterprise Linux 9

Legal Notice

Abstract

Use the different features and utilities available in Red Hat Enterprise Linux 9 to develop and debug C and C++ applications.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting feedback through Jira (account required)

1. Log in to the Jira website.
2. Click **Create** in the top navigation bar
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Create** at the bottom of the dialogue.

CHAPTER 1. SETTING UP A DEVELOPMENT WORKSTATION

Red Hat Enterprise Linux 9 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-9-for-$(uname -i)-baseos-debug-rpms
# subscription-manager repos --enable rhel-9-for-$(uname -i)-baseos-source-rpms
# subscription-manager repos --enable rhel-9-for-$(uname -i)-appstream-debug-rpms
# subscription-manager repos --enable rhel-9-for-$(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
64-bit 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 shipped with Git, a distributed version control system.

Procedure

1. Install the **git** package:

```
# dnf 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:

```
# dnf group install "Development Tools"
```

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

```
# dnf install llvm-toolset
```

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

```
# dnf 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:

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

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

```
# dnf install dnf-utils
```

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

```
# stap-prep
```

Note that **stap-prep** installs packages relevant to the currently *running* kernel, which might not be the same as the actually installed kernel(s). To ensure **stap-prep** installs the correct **kernel-debuginfo** and **kernel-headers** packages, double-check the current kernel version by using the **uname -r** command and reboot your system if necessary.

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:

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

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

```
# stap-prep
```

Note that **stap-prep** installs packages relevant to the currently *running* kernel, which might not be the same as the actually installed kernel(s). To ensure **stap-prep** installs the correct **kernel-debuginfo** and **kernel-headers** packages, double-check the current kernel version by using the

uname -r command and reboot your system if necessary.

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

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

CHAPTER 2. 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.

2.1. GCC IN RHEL 9

Red Hat Enterprise Linux 9 is distributed with GCC 11 as the standard compiler.

The default language standard setting for GCC 11 is C++17. This is equivalent to explicitly using the command-line option **-std=gnu++17**.

Later language standards, such as C++20 and so on, and library features introduced in these later language standards are still considered experimental.

Additional resources

- Porting to GCC 11
- Porting your code to C++17 with GCC 11

2.2. BUILDING CODE WITH GCC

Learn about situations where source code must be transformed into executable code.

2.2.1. Relationship between code forms

Prerequisites

- Understanding the concepts of compiling and linking

Possible code forms

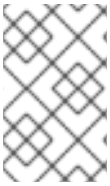
The C and C++ languages have three forms of code:

- **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 that uses the **.a** file name extension. Static linking is not recommended. See Section 2.3.2, "Static and dynamic linking".

Handling of code forms in GCC

Producing executable code from source code is performed in 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 **gcc** command for any of the required actions (compiling and linking). GCC automatically selects the actions and 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 so that it performs only compiling, only linking, or both compiling and linking in a single step. 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.

2.2.2. 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

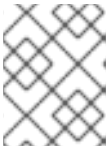
- C or C++ source code file(s)
- GCC installed on the system

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.

2.2.3. 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, and so on. This affects debugging negatively. For improved debugging experience, consider setting the optimization with the **-Og** option. However, changing the optimization level changes the executable code and may change the actual behaviour including removing some bugs.
- To also include 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
  ```

2.2.4. 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 **-O*level*** 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.

Level	Description
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 – Options That Control Optimization
- Linux manual page for GCC:

```
$ man gcc
```

2.2.5. 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

2.2.6. 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.

2.2.7. Example: Building a C program with GCC (compiling and linking in one step)

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

In this example, compiling and linking the code is done in one step.

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 and link the code with GCC:

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

This compiles the code, creates the object file **hello.o**, and links the executable file **helloworld** from the object file.

4. Run the resulting executable file:

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

2.2.8. Example: Building a C program with GCC (compiling and linking in two steps)

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

In this example, compiling and linking the code are two separate steps.

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!
```

2.2.9. Example: Building a C++ program with GCC (compiling and linking in one step)

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

In this example, compiling and linking the code is done in one step.

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 and link the code with **g++**:

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

This compiles the code, creates the object file **hello.o**, and links the executable file **helloworld** from the object file.

4. Run the resulting executable file:

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

2.2.10. Example: Building a C++ program with GCC (compiling and linking in two steps)

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

In this example, compiling and linking the code are two separate steps.

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!
```

2.3. USING LIBRARIES WITH GCC

Learn about using libraries in code.

2.3.1. 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.

2.3.2. Static and dynamic linking

Developers have a choice of using static or dynamic linking when building applications with fully compiled languages. It is important to understand the differences between static and dynamic linking, 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 introduced 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.

2.3.3. Link time optimization

Link time optimization (LTO) enables the compiler to perform various optimizations across all translation units of your program by using its intermediate representation at link time. As a result, your executable files and libraries are smaller and run faster. Also, you can analyze package source code at compile time more thoroughly by using LTO, which improves various GCC diagnostics for potential coding errors.

Known issues

- Violating the One Definition Rule (ODR) produces a **-Wodr** warning
  Violations of the ODR resulting in undefined behavior produce a **-Wodr** warning. This usually points to a bug in your program. The **-Wodr** warning is enabled by default.
- LTO causes increased memory consumption
  The compiler consumes more memory when it processes the translation units the program consists of. On systems with limited memory, disable LTO or lower the parallelism level when building your program.
- GCC removes seemingly unused functions
  GCC may remove functions it considers unused because the compiler is unable to recognize which symbols an asm() statement references. A compilation error may occur as a result. To prevent this, add **__attribute__((used))** to the symbols you use in your program.
- Compiling with the **-fPIC** option causes errors
  Because GCC does not parse the contents of asm() statements, compiling your code with the **-fPIC** command-line option can cause errors. To prevent this, use the **-fno-lto** option when compiling your translation unit.
  More information is available at LTO FAQ – Symbol usage from assembly language .
- Symbol versioning by using the **.symver** directive is not compatible with LTO
  Implementing symbol versioning by using the **.symver** directive in an asm() statement is not compatible with LTO. However, it is possible to implement symbol versioning using the **symver** attribute. For example:

  __attribute__ ((_symver_ ("<*symbol*>@VERS_1"))) void <*symbol*>_v1 (void) { }

Additional resources

- GCC Manual – Function Attributes
- GCC Wiki – Link Time Optimization

2.3.4. 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 a file name extension **.a**.
- Dynamic libraries are available as shared objects. They are a form of an executable file. A shared object has a file name extension **.so**.

To tell GCC where the archives or shared object files of a library 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

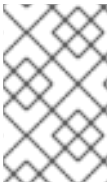
When the situation allows the code to be compiled and linked in one **gcc** command, use the options for both situations mentioned above at once.

Additional resources

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

2.3.5. 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 2.3.2, "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
```

WARNING

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 2.3.7, "Using both static and dynamic libraries with GCC" .

2.3.6. 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 run time

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*.

IMPORTANT

Do not add a 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. This value should represent the path where the shared library objects are located.

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
  ```

2.3.7. 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, and by default, **gcc** selects 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 a 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 the **foo** library 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.

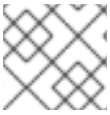
Using the -Wl option

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. The **-Wl** option is followed by a comma-separated list of linker options, while other **gcc** options require space-separated list of 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, link 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

2.4. CREATING LIBRARIES WITH GCC

Learn about the steps to creating libraries and the necessary concepts used by the Linux operating system for libraries.

2.4.1. 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.

2.4.2. 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.
- You must understand 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 **foo** library version *X.Y* is ABI-compatible with other versions with the same value of *X* in a version number. Minor changes preserving compatibility increase the number *Y*. Major changes that break compatibility increase the number *X*.

The actual **foo** library 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 that declares dependency on the library using the soname, not a name or a 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 a 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 their 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.

2.4.3. Creating dynamic libraries with GCC

Dynamically linked libraries (shared objects) allow:

- resource conservation through code reuse
- 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:

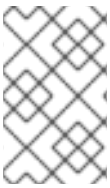
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

Additional resources

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

2.4.4. 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 the use of static linking for security reasons. Use static linking only when necessary, especially against libraries provided by Red Hat. See Section 2.3.2, "Static and dynamic linking" for more details.

Prerequisites

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

Procedure

1. Create intermediate object files with GCC.

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

Append more source files if 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
```

2.5. 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.

2.5.1. 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 example shows 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

2.5.2. Example: Building a C program using a Makefile

Build a sample C program using a Makefile by following the steps in this 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:

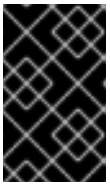
```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

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

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

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



IMPORTANT

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
```

2.5.3. 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

CHAPTER 3. DEBUGGING APPLICATIONS

Debugging applications is a very wide topic. This part provides a developer with the most common techniques for debugging in multiple situations.

3.1. ENABLING DEBUGGING WITH DEBUGGING INFORMATION

To debug applications and libraries, debugging information is required. The following sections describe how to obtain this information.

3.1.1. Debugging information

While debugging any executable code, two types of information allow the tools, and by extension the programmer, to comprehend the binary code:

- the source code text
- a description of how the source code text relates to the binary code

Such information is called debugging information.

Red Hat Enterprise Linux uses the ELF format for executable binaries, shared libraries, or **debuginfo** files. Within these ELF files, the DWARF format is used to hold the debug information.

To display DWARF information stored within an ELF file, run the **readelf -w *file*** command.

IMPORTANT

STABS is an older, less capable format, occasionally used with UNIX. Its use is discouraged by Red Hat. GCC and GDB provide STABS production and consumption on a best effort basis only. Some other tools such as Valgrind and **elfutils** do not work with STABS.

Additional resources

- The DWARF Debugging Standard

3.1.2. 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, and so on. This affects debugging negatively. For improved debugging experience, consider setting the optimization with the **-Og** option. However, changing the optimization level changes the executable code and may change the actual behaviour including removing some bugs.

- To also include 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
```

3.1.3. Debuginfo and debugsource packages

The **debuginfo** and **debugsource** packages contain debugging information and debug source code for programs and libraries. For applications and libraries installed in packages from the Red Hat Enterprise Linux repositories, you can obtain separate **debuginfo** and **debugsource** packages from an additional channel.

Debugging information package types

There are two types of packages available for debugging:

Debuginfo packages

The **debuginfo** packages provide debugging information needed to provide human-readable names for binary code features. These packages contain **.debug** files, which contain DWARF debugging information. These files are installed to the **/usr/lib/debug** directory.

Debugsource packages

The **debugsource** packages contain the source files used for compiling the binary code. With both respective **debuginfo** and **debugsource** package installed, debuggers such as GDB or LLDB can relate the execution of binary code to the source code. The source code files are installed to the **/usr/src/debug** directory.

3.1.4. Getting debuginfo packages for an application or library using GDB

Debugging information is required to debug code. For code that is installed from a package, the GNU Debugger (GDB) automatically recognizes missing debug information, resolves the package name and provides concrete advice on how to get the package.

Prerequisites

- The application or library you want to debug must be installed on the system.
- GDB and the **debuginfo-install** tool must be installed on the system. For details, see Setting up to debug applications.

- Repositories providing **debuginfo** and **debugsource** packages must be configured and enabled on the system. For details, see Enabling debug and source repositories.

Procedure

1. Start GDB attached to the application or library you want to debug. GDB automatically recognizes missing debugging information and suggests a command to run.

```
$ gdb -q /bin/ls
Reading symbols from /bin/ls...Reading symbols from .gnu_debugdata for /usr/bin/ls...(no debugging symbols found)...done.
(no debugging symbols found)...done.
Missing separate debuginfos, use: dnf debuginfo-install coreutils-8.30-6.el8.x86_64
(gdb)
```

2. Exit GDB: type **q** and confirm with **Enter**.

```
(gdb) q
```

3. Run the command suggested by GDB to install the required **debuginfo** packages:

```
# dnf debuginfo-install coreutils-8.30-6.el8.x86_64
```

The **dnf** package management tool provides a summary of the changes, asks for confirmation and once you confirm, downloads and installs all the necessary files.

4. In case GDB is not able to suggest the **debuginfo** package, follow the procedure described in Getting debuginfo packages for an application or library manually.

Additional resources

- How can I download or install debuginfo packages for RHEL systems? (Red Hat Knowledgebase)

3.1.5. Getting debuginfo packages for an application or library manually

You can determine manually which **debuginfo** packages you need to install by locating the executable file and then finding the package that installs it.

NOTE

Red Hat recommends that you use GDB to determine the packages for installation. Use this manual procedure only if GDB is not able to suggest the package to install.

Prerequisites

- The application or library must be installed on the system.
- The application or library was installed from a package.
- The **debuginfo-install** tool must be available on the system.
- Channels providing the **debuginfo** packages must be configured and enabled on the system.

Procedure

1. Find the executable file of the application or library.

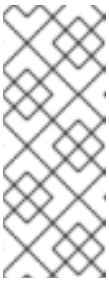
   a. Use the **which** command to find the application file.

   ```
   $ which less
   /usr/bin/less
   ```

   b. Use the **locate** command to find the library file.

   ```
   $ locate libz | grep so
   /usr/lib64/libz.so.1
   /usr/lib64/libz.so.1.2.11
   ```

   If the original reasons for debugging include error messages, pick the result where the library has the same additional numbers in its file name as those mentioned in the error messages. If in doubt, try following the rest of the procedure with the result where the library file name includes no additional numbers.

   NOTE

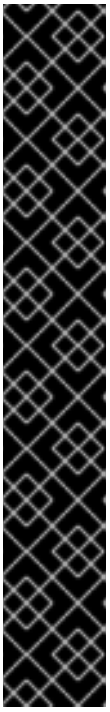
   The **locate** command is provided by the **mlocate** package. To install it and enable its use:

   ```
   # dnf install mlocate
   # updatedb
   ```

2. Search for a name and version of the package that provided the file:

   ```
   $ rpm -qf /usr/lib64/libz.so.1.2.7
   zlib-1.2.11-10.el8.x86_64
   ```

   The output provides details for the installed package in the *name*:*epoch*-*version*.*release*.*architecture* format.



IMPORTANT

If this step does not produce any results, it is not possible to determine which package provided the binary file. There are several possible cases:

- The file is installed from a package which is not known to package management tools in their *current* configuration.
- The file is installed from a locally downloaded and manually installed package. Determining a suitable **debuginfo** package automatically is impossible in that case.
- Your package management tools are misconfigured.
- The file is not installed from any package. In such a case, no respective **debuginfo** package exists.

Because further steps depend on this one, you must resolve this situation or abort this procedure. Describing the exact troubleshooting steps is beyond the scope of this procedure.

3. Install the **debuginfo** packages using the **debuginfo-install** utility. In the command, use the package name and other details you determined during the previous step:

```
# debuginfo-install zlib-1.2.11-10.el8.x86_64
```

Additional resources

- How can I download or install debuginfo packages for RHEL systems? (Red Hat Knowledgebase)

3.2. INSPECTING APPLICATION INTERNAL STATE WITH GDB

To find why an application does not work properly, control its execution and examine its internal state with a debugger. This section describes how to use the GNU Debugger (GDB) for this task.

3.2.1. GNU debugger (GDB)

Red Hat Enterprise Linux contains the GNU debugger (GDB) which lets you investigate what is happening inside a program through a command-line user interface.

GDB capabilities

A single GDB session can debug the following types of programs:

- Multithreaded and forking programs
- Multiple programs at once
- Programs on remote machines or in containers with the **gdbserver** utility connected over a TCP/IP network connection

Debugging requirements

To debug any executable code, GDB requires debugging information for that particular code:

- For programs developed by you, you can create the debugging information while building the code.
- For system programs installed from packages, you must install their debuginfo packages.

3.2.2. Attaching GDB to a process

In order to examine a process, GDB must be *attached* to the process.

Prerequisites

- GDB must be installed on the system

Starting a program with GDB

When the program is not running as a process, start it with GDB:

```
$ gdb program
```

Replace *program* with a file name or path to the program.

GDB sets up to start execution of the program. You can set up breakpoints and the **gdb** environment before beginning the execution of the process with the **run** command.

Attaching GDB to an already running process

To attach GDB to a program already running as a process:

1. Find the process ID (*pid*) with the **ps** command:

   ```
   $ ps -C program -o pid h
    pid
   ```

   Replace *program* with a file name or path to the program.

2. Attach GDB to this process:

   ```
   $ gdb -p pid
   ```

   Replace *pid* with an actual process ID number from the **ps** output.

Attaching an already running GDB to an already running process

To attach an already running GDB to an already running program:

1. Use the **shell** GDB command to run the **ps** command and find the program's process ID (*pid*):

   ```
   (gdb) shell ps -C program -o pid h
    pid
   ```

   Replace *program* with a file name or path to the program.

2. Use the **attach** command to attach GDB to the program:

   ```
   (gdb) attach pid
   ```

Replace *pid* by an actual process ID number from the **ps** output.

NOTE

In some cases, GDB might not be able to find the respective executable file. Use the **file** command to specify the path:

```
(gdb) file path/to/program
```

Additional resources

- Debugging with GDB — 2.1 Invoking GDB
- Debugging with GDB — 4.7 Debugging an Already-running Process

3.2.3. Stepping through program code with GDB

Once the **GDB** debugger is attached to a program, you can use a number of commands to control the execution of the program.

Prerequisites

- You must have the required debugging information available:
  - The program is compiled and built with debugging information, or
  - The relevant debuginfo packages are installed
- GDB must be attached to the program to be debugged

GDB commands to step through the code

r (run)

Start the execution of the program. If **run** is executed with any arguments, those arguments are passed on to the executable as if the program has been started normally. Users normally issue this command after setting breakpoints.

start

Start the execution of the program but stop at the beginning of the program's main function. If **start** is executed with any arguments, those arguments are passed on to the executable as if the program has been started normally.

c (continue)

Continue the execution of the program from the current state. The execution of the program will continue until one of the following becomes true:

- A breakpoint is reached.
- A specified condition is satisfied.
- A signal is received by the program.
- An error occurs.
- The program terminates.

n (next)

Continue the execution of the program from the current state, until the next line of code in the current source file is reached. The execution of the program will continue until one of the following becomes true:

- A breakpoint is reached.
- A specified condition is satisfied.
- A signal is received by the program.
- An error occurs.
- The program terminates.

s (step)

The **step** command also halts execution at each sequential line of code in the current source file. However, if the execution is currently stopped at a source line containing a **function call**, GDB stops the execution after entering the function call (rather than executing it).

until *location*

Continue the execution until the code location specified by the *location* option is reached.

fini (finish)

Resume the execution of the program and halt when execution returns from a function. The execution of the program will continue until one of the following becomes true:

- A breakpoint is reached.
- A specified condition is satisfied.
- A signal is received by the program.
- An error occurs.
- The program terminates.

q (quit)

Terminate the execution and exit GDB.

Additional resources

- Debugging with GDB — Starting your Program
- Debugging with GDB — Continuing and Stepping

3.2.4. Showing program internal values with GDB

Displaying the values of a program's internal variables is important for understanding of what the program is doing. GDB offers multiple commands that you can use to inspect the internal variables. The following are the most useful of these commands:

p (print)

Display the value of the argument given. Usually, the argument is the name of a variable of any complexity, from a simple single value to a structure. An argument can also be an expression valid in

the current language, including the use of program variables and library functions, or functions defined in the program being tested.
It is possible to extend GDB with *pretty-printer* Python or Guile scripts for customized display of data structures (such as classes, structs) using the **print** command.

bt (backtrace)

Display the chain of function calls used to reach the current execution point, or the chain of functions used up until execution was terminated. This is useful for investigating serious bugs (such as segmentation faults) with elusive causes.
Adding the **full** option to the **backtrace** command displays local variables, too.

It is possible to extend GDB with *frame filter* Python scripts for customized display of data displayed using the **bt** and **info frame** commands. The term *frame* refers to the data associated with a single function call.

info

The **info** command is a generic command to provide information about various items. It takes an option specifying the item to describe.

- The **info args** command displays options of the function call that is the currently selected frame.
- The **info locals** command displays local variables in the currently selected frame.

For a list of the possible items, run the command **help info** in a GDB session:

```
(gdb) help info
```

l (list)

Show the line in the source code where the program stopped. This command is available only when the program execution is stopped. While not strictly a command to show internal state, **list** helps the user understand what changes to the internal state will happen in the next step of the program's execution.

Additional resources

- The GDB Python API – Red Hat Developers Blog entry
- Debugging with GDB – Pretty Printing

3.2.5. Using GDB breakpoints to stop execution at defined code locations

Often, only small portions of code are investigated. Breakpoints are markers that tell GDB to stop the execution of a program at a certain place in the code. Breakpoints are most commonly associated with source code lines. In that case, placing a breakpoint requires specifying the source file and line number.

- To **place a breakpoint**:
  - Specify the name of the source code *file* and the *line* in that file:

    ```
    (gdb) br file:line
    ```

  - When *file* is not present, name of the source file at the current point of execution is used:

    ```
    -
    ```

```
(gdb) br line
```

- Alternatively, use a function name to put the breakpoint on its start:

```
(gdb) br function_name
```

- A program might encounter an error after a certain number of iterations of a task. To specify an additional **condition** to halt execution:

```
(gdb) br file:line if condition
```

Replace *condition* with a condition in the C or C++ language. The meaning of *file* and *line* is the same as above.

- To **inspect** the status of all breakpoints and watchpoints:

```
(gdb) info br
```

- To **remove** a breakpoint by using its *number* as displayed in the output of **info br**:

```
(gdb) delete number
```

- To **remove** a breakpoint at a given location:

```
(gdb) clear file:line
```

Additional resources

- Debugging with GDB – Breakpoints, Watchpoints, and Catchpoints

3.2.6. Using GDB watchpoints to stop execution on data access and changes

In many cases, it is advantageous to let the program execute until certain data changes or is accessed. The following examples are the most common use cases.

Prerequisites

- Understanding **GDB**

Using watchpoints in GDB

Watchpoints are markers which tell **GDB** to stop the execution of a program. Watchpoints are associated with data: placing a watchpoint requires specifying an expression that describes a variable, multiple variables, or a memory address.

- To **place** a watchpoint for data **change** (write):

```
(gdb) watch expression
```

Replace *expression* with an expression that describes what you want to watch. For variables, *expression* is equal to the name of the variable.

- To **place** a watchpoint for data **access** (read):

```
(gdb) rwatch expression
```

- To **place** a watchpoint for **any** data access (both read and write):

```
(gdb) awatch expression
```

- To **inspect** the status of all watchpoints and breakpoints:

```
(gdb) info br
```

- To **remove** a watchpoint:

```
(gdb) delete num
```

Replace the ***num*** option with the number reported by the **info br** command.

Additional resources

- Debugging with GDB – Setting Watchpoints

3.2.7. Debugging forking or threaded programs with GDB

Some programs use forking or threads to achieve parallel code execution. Debugging multiple simultaneous execution paths requires special considerations.

Prerequisites

- You must understand the concepts of process forking and threads.

Debugging forked programs with GDB

Forking is a situation when a program (**parent**) creates an independent copy of itself (**child**). Use the following settings and commands to affect what GDB does when a fork occurs:

- The **follow-fork-mode** setting controls whether GDB follows the parent or the child after the fork.

  set follow-fork-mode parent

  After a fork, debug the parent process. This is the default.

  set follow-fork-mode child

  After a fork, debug the child process.

  show follow-fork-mode

  Display the current setting of **follow-fork-mode**.

- The **set detach-on-fork** setting controls whether the GDB keeps control of the other (not followed) process or leaves it to run.

  set detach-on-fork on

  The process which is not followed (depending on the value of **follow-fork-mode**) is detached and runs independently. This is the default.

  set detach-on-fork off

GDB keeps control of both processes. The process which is followed (depending on the value of **follow-fork-mode**) is debugged as usual, while the other is suspended.

show detach-on-fork

Display the current setting of **detach-on-fork**.

Debugging Threaded Programs with GDB

GDB has the ability to debug individual threads, and to manipulate and examine them independently. To make GDB stop only the thread that is examined, use the commands **set non-stop on** and **set target-async on**. You can add these commands to the **.gdbinit** file. After that functionality is turned on, GDB is ready to conduct thread debugging.

GDB uses a concept of *current thread*. By default, commands apply to the current thread only.

info threads

Display a list of threads with their **id** and **gid** numbers, indicating the current thread.

thread *id*

Set the thread with the specified **id** as the current thread.

thread apply *ids command*

Apply the command ***command*** to all threads listed by ***ids***. The ***ids*** option is a space-separated list of thread ids. A special value **all** applies the command to all threads.

break *location* thread *id* if *condition*

Set a breakpoint at a certain ***location*** with a certain ***condition*** only for the thread number ***id***.

watch *expression* thread *id*

Set a watchpoint defined by ***expression*** only for the thread number ***id***.

command&

Execute command **command** and return immediately to the gdb prompt **(gdb)**, continuing any code execution in the background.

interrupt

Halt execution in the background.

Additional resources

- Debugging with GDB — 4.10 Debugging Programs with Multiple Threads
- Debugging with GDB — 4.11 Debugging Forks

3.3. RECORDING APPLICATION INTERACTIONS

The executable code of applications interacts with the code of the operating system and shared libraries. Recording an activity log of these interactions can provide enough insight into the application's behavior without debugging the actual application code. Alternatively, analyzing an application's interactions can help pinpoint the conditions in which a bug manifests.

3.3.1. Tools useful for recording application interactions

Red Hat Enterprise Linux offers multiple tools for analyzing an application's interactions.

strace

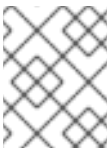
The **strace** tool primarily enables logging of system calls (kernel functions) used by an application.

- The **strace** tool can provide a detailed output about calls, because **strace** interprets parameters and results with knowledge of the underlying kernel code. Numbers are turned into the respective constant names, bitwise combined flags expanded to flag list, pointers to character arrays dereferenced to provide the actual string, and more. Support for more recent kernel features may be lacking.
- You can filter the traced calls to reduce the amount of captured data.
- The use of **strace** does not require any particular setup except for setting up the log filter.
- Tracing the application code with **strace** results in significant slowdown of the application's execution. As a result, **strace** is not suitable for many production deployments. As an alternative, consider using **ltrace** or SystemTap.
- The version of **strace** available in Red Hat Developer Toolset can also perform system call tampering. This capability is useful for debugging.

ltrace

The **ltrace** tool enables logging of an application's user space calls into shared objects (dynamic libraries).

- The **ltrace** tool enables tracing calls to any library.
- You can filter the traced calls to reduce the amount of captured data.
- The use of **ltrace** does not require any particular setup except for setting up the log filter.
- The **ltrace** tool is lightweight and fast, offering an alternative to **strace**: it is possible to trace the respective interfaces in libraries such as **glibc** with **ltrace** instead of tracing kernel functions with **strace**.
- Because **ltrace** does not handle a known set of calls like **strace**, it does not attempt to explain the values passed to library functions. The **ltrace** output contains only raw numbers and pointers. The interpretation of **ltrace** output requires consulting the actual interface declarations of the libraries present in the output.



NOTE

In Red Hat Enterprise Linux 9, a known issue prevents **ltrace** from tracing system executable files. This limitation does not apply to executable files built by users.

SystemTap

SystemTap is an instrumentation platform for probing running processes and kernel activity on the Linux system. SystemTap uses its own scripting language for programming custom event handlers.

- Compared to using **strace** and **ltrace**, scripting the logging means more work in the initial setup phase. However, the scripting capabilities extend SystemTap's usefulness beyond just producing logs.
- SystemTap works by creating and inserting a kernel module. The use of SystemTap is efficient and does not create a significant slowdown of the system or application execution on its own.
- SystemTap comes with a set of usage examples.

GDB

The GNU Debugger (GDB) is primarily meant for debugging, not logging. However, some of its features make it useful even in the scenario where an application's interaction is the primary activity of interest.

- With GDB, it is possible to conveniently combine the capture of an interaction event with immediate debugging of the subsequent execution path.
- GDB is best suited for analyzing response to infrequent or singular events, after the initial identification of problematic situation by other tools. Using GDB in any scenario with frequent events becomes inefficient or even impossible.

Additional resources

- Getting started with SystemTap
- Red Hat Developer Toolset User Guide

3.3.2. Monitoring an application's system calls with strace

The **strace** tool enables monitoring the system (kernel) calls performed by an application.

Prerequisites

- You must have **strace** installed on the system.

Procedure

1. Identify the system calls to monitor.
2. Start **strace** and attach it to the program.
   - If the program you want to monitor is not running, start **strace** and specify the *program*:

     ```
     $ strace -fvttTyy -s 256 -e trace=call program
     ```

   - If the program is already running, find its process id (*pid*) and attach **strace** to it:

     ```
     $ ps -C program
     (...)
     $ strace -fvttTyy -s 256 -e trace=call -ppid
     ```

   - Replace *call* with the system calls to be displayed. You can use the **-e trace=*call*** option multiple times. If left out, **strace** will display all system call types. See the *strace(1)* manual page for more information.
   - If you do not want to trace any forked processes or threads, leave out the **-f** option.
3. The **strace** tool displays the system calls made by the application and their details.
   In most cases, an application and its libraries make a large number of calls and **strace** output appears immediately, if no filter for system calls is set.
4. The **strace** tool exits when the program exits.
   To terminate the monitoring before the traced program exits, press **Ctrl+C**.

- If **strace** started the program, the program terminates together with **strace**.
- If you attached **strace** to an already running program, the program terminates together with **strace**.

5. Analyze the list of system calls done by the application.
   - Problems with resource access or availability are present in the log as calls returning errors.
   - Values passed to the system calls and patterns of call sequences provide insight into the causes of the application's behaviour.
   - If the application crashes, the important information is probably at the end of log.
   - The output contains a lot of unnecessary information. However, you can construct a more precise filter for the system calls of interest and repeat the procedure.

NOTE

It is advantageous to both see the output and save it to a file. Use the **tee** command to achieve this:

```
$ strace ... |& tee your_log_file.log
```

Additional resources

- The *strace(1)* manual page:

```
$ man strace
```

- How do I use strace to trace system calls made by a command? – Knowledgebase article
- Red Hat Developer Toolset User Guide – Chapter strace

3.3.3. Monitoring application's library function calls with ltrace

The **ltrace** tool enables monitoring an application's calls to functions available in libraries (shared objects).

NOTE

In Red Hat Enterprise Linux 9, a known issue prevents **ltrace** from tracing system executable files. This limitation does not apply to executable files built by users.

Prerequisites

- You must have **ltrace** installed on the system.

Procedure

1. Identify the libraries and functions of interest, if possible.
2. Start **ltrace** and attach it to the program.

- If the program you want to monitor is not running, start **ltrace** and specify *program*:

```
$ ltrace -f -l library -e function program
```

- If the program is already running, find its process id (*pid*) and attach **ltrace** to it:

```
$ ps -C program
(...)
$ ltrace -f -l library -e function -ppid program
```

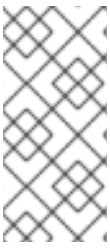
- Use the **-e**, **-f** and **-l** options to filter the output:
  - Supply the function names to be displayed as *function*. The **-e *function*** option can be used multiple times. If left out, **ltrace** displays calls to all functions.
  - Instead of specifying functions, you can specify whole libraries with the **-l *library*** option. This option behaves similarly to the **-e *function*** option.
  - If you do not want to trace any forked processes or threads, leave out the **-f** option.

  See the *ltrace*(1)_ manual page for more information.

3. **ltrace** displays the library calls made by the application.
   In most cases, an application makes a large number of calls and **ltrace** output displays immediately, if no filter is set.

4. **ltrace** exits when the program exits.
   To terminate the monitoring before the traced program exits, press **ctrl+C**.
   - If **ltrace** started the program, the program terminates together with **ltrace**.
   - If you attached **ltrace** to an already running program, the program terminates together with **ltrace**.

5. Analyze the list of library calls done by the application.
   - If the application crashes, the important information is probably at the end of log.
   - The output contains a lot of unnecessary information. However, you can construct a more precise filter and repeat the procedure.



NOTE

It is advantageous to both see the output and save it to a file. Use the **tee** command to achieve this:

```
$ ltrace ... |& tee your_log_file.log
```

Additional resources

- The *ltrace(1)* manual page:

```
$ man ltrace
```

- Red Hat Developer Toolset User Guide — Chapter ltrace

3.3.4. Monitoring application's system calls with SystemTap

The SystemTap tool enables registering custom event handlers for kernel events. In comparison with the **strace** tool, it is harder to use but more efficient and enables more complicated processing logic. A SystemTap script called **strace.stp** is installed together with SystemTap and provides an approximation of **strace** functionality using SystemTap.

Prerequisites

- SystemTap and the respective kernel packages must be installed on the system.

Procedure

1. Find the process ID (*pid*) of the process you want to monitor:

```
$ ps -aux
```

2. Run SystemTap with the **strace.stp** script:

```
# stap /usr/share/systemtap/examples/process/strace.stp -x pid
```

   The value of *pid* is the process id.

   The script is compiled to a kernel module, which is then loaded. This introduces a slight delay between entering the command and getting the output.

3. When the process performs a system call, the call name and its parameters are printed to the terminal.

4. The script exits when the process terminates, or when you press **Ctrl+C**.

3.3.5. Using GDB to intercept application system calls

GNU Debugger (GDB) lets you stop an execution in various situations that arise during program execution. To stop the execution when the program performs a system call, use a GDB *catchpoint*.

Prerequisites

- You must understand the usage of GDB breakpoints.
- GDB must be attached to the program.

Procedure

1. Set the catchpoint:

```
(gdb) catch syscall syscall-name
```

   The command **catch syscall** sets a special type of breakpoint that halts execution when the program performs a system call.

   The ***syscall-name*** option specifies the name of the call. You can specify multiple catchpoints for various system calls. Leaving out the ***syscall-name*** option causes GDB to stop on any system call.

2. Start execution of the program.
   - If the program has not started execution, start it:

     ```
     (gdb) r
     ```

   - If the program execution is halted, resume it:

     ```
     (gdb) c
     ```

3. GDB halts execution after the program performs any specified system call.

Additional resources

- Debugging with GDB – Setting Watchpoints

3.3.6. Using GDB to intercept handling of signals by applications

GNU Debugger (GDB) lets you stop the execution in various situations that arise during program execution. To stop the execution when the program receives a signal from the operating system, use a GDB *catchpoint*.

Prerequisites

- You must understand the usage of GDB breakpoints.
- GDB must be attached to the program.

Procedure

1. Set the catchpoint:

   ```
   (gdb) catch signal signal-type
   ```

   The command **catch signal** sets a special type of a breakpoint that halts execution when a signal is received by the program. The ***signal-type*** option specifies the type of the signal. Use the special value **'all'** to catch all signals.

2. Let the program run.
   - If the program has not started execution, start it:

     ```
     (gdb) r
     ```

   - If the program execution is halted, resume it:

     ```
     (gdb) c
     ```

3. GDB halts execution after the program receives any specified signal.

Additional resources

- Debugging With GDB – 5.1.3 Setting Catchpoints

3.4. DEBUGGING A CRASHED APPLICATION

Sometimes, it is not possible to debug an application directly. In these situations, you can collect information about the application at the moment of its termination and analyze it afterwards.

3.4.1. Core dumps: what they are and how to use them

A core dump is a copy of a part of the application's memory at the moment the application stopped working, stored in the ELF format. It contains all the application's internal variables and stack, which enables inspection of the application's final state. When augmented with the respective executable file and debugging information, it is possible to analyze a core dump file with a debugger in a way similar to analyzing a running program.

The Linux operating system kernel can record core dumps automatically, if this functionality is enabled. Alternatively, you can send a signal to any running application to generate a core dump regardless of its actual state.

WARNING

Some limits might affect the ability to generate a core dump. To see the current limits:

```
$ ulimit -a
```

3.4.2. Recording application crashes with core dumps

To record application crashes, set up core dump saving and add information about the system.

Procedure

1. To enable core dumps, ensure that the **/etc/systemd/system.conf** file contains the following lines:

   ```
   DumpCore=yes
   DefaultLimitCORE=infinity
   ```

   You can also add comments describing if these settings were previously present, and what the previous values were. This will enable you to reverse these changes later, if needed. Comments are lines starting with the **#** character.

   Changing the file requires administrator level access.

2. Apply the new configuration:

   ```
   # systemctl daemon-reexec
   ```

3. Remove the limits for core dump sizes:

   ```
   # ulimit -c unlimited
   ```

To reverse this change, run the command with value **0** instead of ***unlimited***.

4. Install the **sos** package which provides the **sosreport** utility for collecting system information:

```
# dnf install sos
```

5. When an application crashes, a core dump is generated and handled by **systemd-coredump**.

6. Create an SOS report to provide additional information about the system:

```
# sosreport
```

This creates a **.tar** archive containing information about your system, such as copies of configuration files.

7. Locate and export the core dump:

```
$ coredumpctl list executable-name
$ coredumpctl dump executable-name > /path/to/file-for-export
```

If the application crashed multiple times, output of the first command lists more captured core dumps. In that case, construct for the second command a more precise query using the other information. See the *coredumpctl(1)* manual page for details.

8. Transfer the core dump and the SOS report to the computer where the debugging will take place. Transfer the executable file, too, if it is known.

IMPORTANT

When the executable file is not known, subsequent analysis of the core file identifies it.

9. Optional: Remove the core dump and SOS report after transferring them, to free up disk space.

Additional resources

- Managing systemd in the document *Configuring basic system settings*
- How to enable core file dumps when an application crashes or segmentation faults (Red Hat Knowledgebase)
- What is a sosreport and how to create one in Red Hat Enterprise Linux 4.6 and later? (Red Hat Knowledgebase)

3.4.3. Inspecting application crash states with core dumps

Prerequisites

- You must have a core dump file and sosreport from the system where the crash occurred.
- GDB and elfutils must be installed on your system.

Procedure

1. To identify the executable file where the crash occurred, run the **eu-unstrip** command with the core dump file:

```
$ eu-unstrip -n --core=./core.9814
0x400000+0x207000 2818b2009547f780a5639c904cded443e564973e@0x400284
/usr/bin/sleep /usr/lib/debug/bin/sleep.debug [exe]
0x7fff26fff000+0x1000 1e2a683b7d877576970e4275d41a6aaec280795e@0x7fff26fff340 . -
linux-vdso.so.1
0x35e7e00000+0x3b6000
374add1ead31ccb449779bc7ee7877de3377e5ad@0x35e7e00280 /usr/lib64/libc-2.14.90.so
/usr/lib/debug/lib64/libc-2.14.90.so.debug libc.so.6
0x35e7a00000+0x224000
3ed9e61c2b7e707ce244816335776afa2ad0307d@0x35e7a001d8 /usr/lib64/ld-2.14.90.so
/usr/lib/debug/lib64/ld-2.14.90.so.debug ld-linux-x86-64.so.2
```

   The output contains details for each module on a line, separated by spaces. The information is listed in this order:

   1. The memory address where the module was mapped
   2. The build-id of the module and where in the memory it was found
   3. The module's executable file name, displayed as **-** when unknown, or as **.** when the module has not been loaded from a file
   4. The source of debugging information, displayed as a file name when available, as **.** when contained in the executable file itself, or as **-** when not present at all
   5. The shared library name (*soname*) or **[exe]** for the main module

   In this example, the important details are the file name **/usr/bin/sleep** and the build-id **2818b2009547f780a5639c904cded443e564973e** on the line containing the text **[exe]**. With this information, you can identify the executable file required for analyzing the core dump.

2. Get the executable file that crashed.

   - If possible, copy it from the system where the crash occurred. Use the file name extracted from the core file.
   - You can also use an identical executable file on your system. Each executable file built on Red Hat Enterprise Linux contains a note with a unique build-id value. Determine the build-id of the relevant locally available executable files:

     ```
     $ eu-readelf -n executable_file
     ```

     Use this information to match the executable file on the remote system with your local copy. The build-id of the local file and build-id listed in the core dump must match.

   - Finally, if the application is installed from an RPM package, you can get the executable file from the package. Use the **sosreport** output to find the exact version of the package required.

3. Get the shared libraries used by the executable file. Use the same steps as for the executable file.

4. If the application is distributed as a package, load the executable file in GDB, to display hints for

4. If the application is distributed as a package, load the executable file in GDB, to display hints for missing debuginfo packages. For more details, see Section 3.1.4, "Getting debuginfo packages for an application or library using GDB".

5. To examine the core file in detail, load the executable file and core dump file with GDB:

```
$ gdb -e executable_file -c core_file
```

Further messages about missing files and debugging information help you identify what is missing for the debugging session. Return to the previous step if needed.

If the application's debugging information is available as a file instead of as a package, load this file in GDB with the **symbol-file** command:

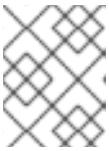
```
(gdb) symbol-file program.debug
```

Replace *program.debug* with the actual file name.

NOTE

It might not be necessary to install the debugging information for all executable files contained in the core dump. Most of these executable files are libraries used by the application code. These libraries might not directly contribute to the problem you are analyzing, and you do not need to include debugging information for them.

6. Use the GDB commands to inspect the state of the application at the moment it crashed. See Inspecting Application Internal State with GDB.



NOTE

When analyzing a core file, GDB is not attached to a running process. Commands for controlling execution have no effect.

Additional resources

- Debugging with GDB — 2.1.1 Choosing Files
- Debugging with GDB — 18.1 Commands to Specify Files
- Debugging with GDB — 18.3 Debugging Information in Separate Files

3.4.4. Creating and accessing a core dump with coredumpctl

The **coredumpctl** tool of **systemd** can significantly streamline working with core dumps on the machine where the crash happened. This procedure outlines how to capture a core dump of unresponsive process.

Prerequisites

- The system must be configured to use **systemd-coredump** for core dump handling. To verify this is true:

```
$ sysctl kernel.core_pattern
```

The configuration is correct if the output starts with the following:

```
kernel.core_pattern = |/usr/lib/systemd/systemd-coredump
```

Procedure

1. Find the PID of the hung process, based on a known part of the executable file name:

   ```
   $ pgrep -a executable-name-fragment
   ```

   This command will output a line in the form

   ```
   PID command-line
   ```

   Use the *command-line* value to verify that the *PID* belongs to the intended process.

   For example:

   ```
   $ pgrep -a bc
   5459 bc
   ```

2. Send an abort signal to the process:

   ```
   # kill -ABRT PID
   ```

3. Verify that the core has been captured by **coredumpctl**:

   ```
   $ coredumpctl list PID
   ```

   For example:

   ```
   $ coredumpctl list 5459
   TIME                            PID   UID   GID SIG COREFILE  EXE
   Thu 2019-11-07 15:14:46 CET    5459  1000  1000   6 present   /usr/bin/bc
   ```

4. Further examine or use the core file as needed.
   You can specify the core dump by PID and other values. See the *coredumpctl(1)* manual page for further details.

   - To show details of the core file:

     ```
     $ coredumpctl info PID
     ```

   - To load the core file in the GDB debugger:

     ```
     $ coredumpctl debug PID
     ```

     Depending on availability of debugging information, GDB will suggest commands to run, such as:

     ```
     Missing separate debuginfos, use: dnf debuginfo-install bc-1.07.1-5.el8.x86_64
     ```

For more details on this process, see Getting debuginfo packages for an application or library using GDB.

- To export the core file for further processing elsewhere:

```
$ coredumpctl dump PID > /path/to/file_for_export
```

Replace */path/to/file_for_export* with the file where you want to put the core dump.

3.4.5. Dumping process memory with gcore

The workflow of core dump debugging enables the analysis of the program's state offline. In some cases, you can use this workflow with a program that is still running, such as when it is hard to access the environment with the process. You can use the **gcore** command to dump memory of any process while it is still running.

Prerequisites

- You must understand what core dumps are and how they are created.
- GDB must be installed on the system.

Procedure

1. Find out the process id (*pid*). Use tools such as **ps**, **pgrep**, and **top**:

```
$ ps -C some-program
```

2. Dump the memory of this process:

```
$ gcore -o filename pid
```

This creates a file ***filename*** and dumps the process memory in it. While the memory is being dumped, the execution of the process is halted.

3. After the core dump is finished, the process resumes normal execution.
4. Create an SOS report to provide additional information about the system:

```
# sosreport
```

This creates a tar archive containing information about your system, such as copies of configuration files.

5. Transfer the program's executable file, core dump, and the SOS report to the computer where the debugging will take place.
6. Optional: Remove the core dump and SOS report after transferring them, to free up disk space.

Additional resources

- How to obtain a core file without restarting an application? (Red Hat Knowledgebase)

3.4.6. Dumping protected process memory with GDB

You can mark the memory of processes as not to be dumped. This can save resources and ensure additional security when the process memory contains sensitive data: for example, in banking or accounting applications or on whole virtual machines. Both kernel core dumps (**kdump**) and manual core dumps (**gcore**, GDB) do not dump memory marked this way.

In some cases, you must dump the whole contents of the process memory regardless of these protections. This procedure shows how to do this using the GDB debugger.

Prerequisites

- You must understand what core dumps are.
- GDB must be installed on the system.
- GDB must be already attached to the process with protected memory.

Procedure

1. Set GDB to ignore the settings in the **/proc/PID/coredump_filter** file:

```
(gdb) set use-coredump-filter off
```

2. Set GDB to ignore the memory page flag **VM_DONTDUMP**:

```
(gdb) set dump-excluded-mappings on
```

3. Dump the memory:

```
(gdb) gcore core-file
```

Replace *core-file* with name of file where you want to dump the memory.

Additional resources

- Debugging with GDB - How to Produce a Core File from Your Program

3.5. COMPATABILITY BREAKING CHANGES IN GDB

The version of GDB provided in Red Hat Enterprise Linux 9 contains a number of changes that break compatibility. The following sections provide more details about these changes.

Commands

- The **gdb -P python-script.py** command is no longer supported.
  Use the **gdb -ex 'source python-script.py'** command instead.
- The **gdb *COREFILE*** command is no longer supported.
  Use the **gdb *EXECUTABLE* --core *COREFILE*** command instead to load the executable specified in the core file.
- GDB now styles output by default.
  This new change might break scripts that try to parse the output of GDB. Use the **gdb -ex 'set style enabled off'** command to disable styling in scripts.

- Commands now define syntax for symbols according to the language.
  The **info functions**, **info types**, **info variables** and **rbreak** commands now define the syntax for entities according to the language chosen bybthe **set language** command. By setting it to **set language auto** means that GDB will automatically choose the language of the shown entities.

- The **set print raw frame-arguments** and **show print raw frame-arguments** commands have been deprecated.
  These commands are replaced with the **set print raw-frame-arguments** and **show print raw-frame-arguments** commands. The old commands may be removed in future versions.

- The following TUI commands are now case-sensitive:
  - **focus**
  - **winheight**
  - **+**
  - **-**
  - **>**
  - **<**

- The **help** and **apropos** commands now display command information only once.
  These commands now show the documentation of a command only once, even if that command has one or more aliases. These commands now show the command name, then all of its aliases, and finally the description of the command.

The MI interpreter

- The default version of the MI interpreter is now 3.
  The output of information about multi-location breakpoints (which is syntactically incorrect in MI 2) has changed in MI 3. This affects the following commands and events:
  - **-break-insert**
  - **-break-info**
  - **=breakpoint-created**
  - **=breakpoint-modified**

  Use the **-fix-multi-location-breakpoint-output** command to enable this behavior with previous MI versions.

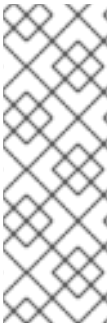
Python API

- The following symbols are now deprecated:
  - **gdb.SYMBOL_VARIABLES_DOMAIN**
  - **gdb.SYMBOL_FUNCTIONS_DOMAIN**
  - **gdb.SYMBOL_TYPES_DOMAIN**

- The **gdb.Value** type has a new constructor, which is used to construct a **gdb.Value** from a Python buffer object and a **gdb.Type**.
- The frame information printed by the Python frame filtering code is now consistent with what the **backtrace** command prints when there are no filters, or when using the **backtrace** command's **-no-filters** option.

3.6. DEBUGGING APPLICATIONS IN CONTAINERS

You can use various command-line tools tailored to different aspects of troubleshooting. The following provides categories along with common command-line tools.



NOTE

This is not a complete list of command-line tools. The choice of tool for debugging a container application is heavily based on the container image and your use case.

For instance, the **systemctl**, **journalctl**, **ip**, **netstat**, **ping**, **traceroute**, **perf**, **iostat** tools may need root access because they interact with system-level resources such as networking, systemd services, or hardware performance counters, which are restricted in rootless containers for security reasons.

Rootless containers operate without requiring elevated privileges, running as non-root users within user namespaces to provide improved security and isolation from the host system. They offer limited interaction with the host, reduced attack surface, and enhanced security by mitigating the risk of privilege escalation vulnerabilities.

Rootful containers run with elevated privileges, typically as the root user, granting full access to system resources and capabilities. While rootful containers offer greater flexibility and control, they pose security risks due to their potential for privilege escalation and exposure of the host system to vulnerabilities.

For more information about rootful and rootless containers, see Setting up rootless containers, Upgrading to rootless containers, and Special considerations for rootless containers.

Systemd and Process Management Tools

systemctl
: Controls systemd services within containers, allowing start, stop, enable, and disable operations.

journalctl
: Views logs generated by systemd services, aiding in troubleshooting container issues.

Networking Tools

ip
: Manages network interfaces, routing, and addresses within containers.

netstat
: Displays network connections, routing tables, and interface statistics.

ping
: Verifies network connectivity between containers or hosts.

traceroute

Identifies the path packets take to reach a destination, useful for diagnosing network issues.

Process and Performance Tools

ps

Lists currently running processes within containers.

top

Provides real-time insights into resource usage by processes within containers.

htop

Interactive process viewer for monitoring resource utilization.

perf

CPU performance profiling, tracing, and monitoring, aiding in pinpointing performance bottlenecks within the system or applications.

vmstat

Reports virtual memory statistics within containers, aiding in performance analysis.

iostat

Monitors input/output statistics for block devices within containers.

gdb (GNU Debugger)

A command-line debugger that helps in examining and debugging programs by allowing users to track and control their execution, inspect variables, and analyze memory and registers during runtime. For more information, see the Debugging applications within Red Hat OpenShift containers article.

strace

Intercepts and records system calls made by a program, aiding in troubleshooting by revealing interactions between the program and the operating system.

Security and Access Control Tools

sudo

Enables executing commands with elevated privileges.

chroot

Changes the root directory for a command, helpful in testing or troubleshooting within a different root directory.

Podman-Specific Tools

podman logs

Batch-retrieves whatever logs are present for one or more containers at the time of execution.

podman inspect

Displays the low-level information on containers and images as identified by name or ID.

podman events

Monitor and print events that occur in Podman. Each event includes a timestamp, a type, a status, a name (if applicable), and an image (if applicable). The default logging mechanism is **journald**.

podman run --health-cmd

Use the health check to determine the health or readiness of the process running inside the container.

podman top

Display the running processes of the container.

podman exec

Running commands in or attaching to a running container is extremely useful to get a better understanding of what is happening in the container.

podman export

When the container fails, it is basically impossible to know what happened. Exporting the filesystem structure from the container will allow for checking other logs files that may not be in the mounted volumes.

Additional resources

- Debugging applications within Red Hat OpenShift containers
  - **gdb**
- Debugging a Crashed Application
  - Core dump, **sosreport**, **gdb**, **ps**, **core**.
- Troubleshooting Kubernetes
  - Docker exec + env, **netstat**, **kubectl**, **etcdctl**, **journalctl**, docker logs
- Tips and Tricks for containerizing services
  - Watch, **podman logs**, **systemctl**, **podman exec**/**kill**/**restart**, **podman insect**, **podman top**, **podman exec**, **podman export**, **paunch**
- External links
  - Ten tips for debugging Docker containers

CHAPTER 4. ADDITIONAL TOOLSETS FOR DEVELOPMENT

4.1. USING GCC TOOLSET

4.1.1. What is GCC Toolset

Red Hat Enterprise Linux 9 continues support for GCC Toolset, an Application Stream containing more up-to-date versions of development and performance analysis tools. GCC Toolset is similar to Red Hat Developer Toolset for RHEL 7.

GCC Toolset is available as an Application Stream in the form of a software collection in the **AppStream** repository. GCC Toolset is fully supported under Red Hat Enterprise Linux Subscription Level Agreements, is functionally complete, and is intended for production use. Applications and libraries provided by GCC Toolset do not replace the Red Hat Enterprise Linux system versions, do not override them, and do not automatically become default or preferred choices. Using a framework called software collections, an additional set of developer tools is installed into the **/opt/** directory and is explicitly enabled by the user on demand using the **scl** utility. Unless noted otherwise for specific tools or features, GCC Toolset is available for all architectures supported by Red Hat Enterprise Linux.

For information about the length of support, see Red Hat Enterprise Linux Application Streams Life Cycle.

4.1.2. Installing GCC Toolset

Installing GCC Toolset on a system installs the main tools and all necessary dependencies. Note that some parts of the toolset are not installed by default and must be installed separately.

Procedure

- To install GCC Toolset version *N*:

```
# dnf install gcc-toolset-N
```

4.1.3. Installing individual packages from GCC Toolset

To install only certain tools from GCC Toolset instead of the whole toolset, list the available packages and install the selected ones with the **dnf** package management tool. This procedure is useful also for packages that are not installed by default with the toolset.

Procedure

1. List the packages available in GCC Toolset version *N*:

```
$ dnf list available gcc-toolset-N-\*
```

2. To install any of these packages:

```
# dnf install package_name
```

Replace *package_name* with a space-separated list of packages to install. For example, to install the **gcc-toolset-13-annobin-annocheck** and **gcc-toolset-13-binutils-devel** packages:

```
# dnf install gcc-toolset-13-annobin-annocheck gcc-toolset-13-binutils-devel
```

4.1.4. Uninstalling GCC Toolset

To remove GCC Toolset from your system, uninstall it using the **dnf** package management tool.

Procedure

- To uninstall GCC Toolset version *N*:

```
# dnf remove gcc-toolset-N\*
```

4.1.5. Running a tool from GCC Toolset

To run a tool from GCC Toolset, use the **scl** utility.

Procedure

- To run a tool from GCC Toolset version *N*:

```
$ scl enable gcc-toolset-N tool
```

4.1.6. Running a shell session with GCC Toolset

GCC Toolset allows running a shell session where the GCC Toolset tool versions are used instead of system versions of these tools, without explicitly using the **scl** command. This is useful when you need to interactively start the tools many times, such as when setting up or testing a development setup.

Procedure

- To run a shell session where tool versions from GCC Toolset version *N* override system versions of these tools:

```
$ scl enable gcc-toolset-N bash
```

4.1.7. Additional resources

- Red Hat Developer Toolset User Guide

4.2. GCC TOOLSET 12

Learn about information specific to GCC Toolset version 12 and the tools contained in this version.

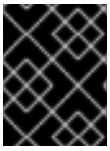
4.2.1. Tools and versions provided by GCC Toolset 12

GCC Toolset 12 provides the following tools and versions:

Table 4.1. Tool versions in GCC Toolset 12

Name	Version	Description
GCC	12.2.1	A portable compiler suite with support for C, C++, and Fortran.
GDB	11.2	A command-line debugger for programs written in C, C++, and Fortran.
binutils	2.38	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
dwz	0.14	A tool to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.
annobin	11.08	A build security checking tool.

4.2.2. C++ compatibility in GCC Toolset 12



IMPORTANT

The compatibility information presented here apply only to the GCC from GCC Toolset 12.

The GCC compiler in GCC Toolset can use the following C++ standards:

C++14

This language standard is available in GCC Toolset 12.
Using the C++14 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 6 or later.

C++11

This language standard is available in GCC Toolset 12.
Using the C++11 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 5 or later.

C++98

This language standard is available in GCC Toolset 12. Binaries, shared libraries and objects built using this standard can be freely mixed regardless of being built with GCC from GCC Toolset, Red Hat Developer Toolset, and RHEL 5, 6, 7 and 8.

C++17

This language standard is available in GCC Toolset 12.
This is the default language standard setting for GCC Toolset 12, with GNU extensions, equivalent to explicitly using option **-std=gnu++17**.

Using the C++17 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 10 or later.

C++20 and C++23

This language standard is available in GCC Toolset 12 only as an experimental, unstable, and unsupported capability. Additionally, compatibility of objects, binary files, and libraries built using this standard cannot be guaranteed.

To enable C++20 support, add the command-line option **-std=c++20** to your g++ command line.

To enable C++23 support, add the command-line option **-std=c++23** to your g++ command line.

All of the language standards are available in both the standard compliant variant or with GNU extensions.

When mixing objects built with GCC Toolset with those built with the RHEL toolchain (particularly **.o** or **.a** files), GCC Toolset toolchain should be used for any linkage. This ensures any newer library features provided only by GCC Toolset are resolved at link time.

4.2.3. Specifics of GCC in GCC Toolset 12

Static linking of libraries

Certain more recent library features are statically linked into applications built with GCC Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk because standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

IMPORTANT

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Specify libraries after object files when linking

In GCC Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable gcc-toolset-12 'gcc -lsomelib objfile.o'
```

Using a library from GCC Toolset in this manner results in the linker error message **undefined reference to symbol**. To prevent this problem, follow the standard linking practice and specify the option adding the library after the options specifying the object files:

```
$ scl enable gcc-toolset-12 'gcc objfile.o -lsomelib'
```

Note that this recommendation also applies when using the base Red Hat Enterprise Linux version of GCC.

4.2.4. Specifics of binutils in GCC Toolset 12

Static linking of libraries

Certain more recent library features are statically linked into applications built with GCC Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk because standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

IMPORTANT

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Specify libraries after object files when linking

In GCC Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable gcc-toolset-12 'ld -lsomelib objfile.o'
```

Using a library from GCC Toolset in this manner results in the linker error message **undefined reference to symbol**. To prevent this problem, follow the standard linking practice, and specify the option adding the library after the options specifying the object files:

```
$ scl enable gcc-toolset-12 'ld objfile.o -lsomelib'
```

Note that this recommendation also applies when using the base Red Hat Enterprise Linux version of **binutils**.

4.2.5. Specifics of annobin in GCC Toolset 12

Under some circumstances, due to a synchronization issue between **annobin** and **gcc** in GCC Toolset 12, your compilation can fail with an error message that looks similar to the following:

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

To work around the problem, create a symbolic link in the plugin directory from the **annobin.so** file to the **gcc-annobin.so** file:

```
# cd /opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin
# ln -s annobin.so gcc-annobin.so
```

Replace *architecture* with the architecture you use in your system:

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

4.3. GCC TOOLSET 13

Learn about information specific to GCC Toolset version 13 and the tools contained in this version.

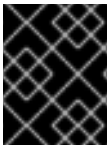
4.3.1. Tools and versions provided by GCC Toolset 13

GCC Toolset 13 provides the following tools and versions:

Table 4.2. Tool versions in GCC Toolset 13

Name	Version	Description
GCC	13.2.1	A portable compiler suite with support for C, C++, and Fortran.
GDB	12.1	A command-line debugger for programs written in C, C++, and Fortran.
binutils	2.40	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
dwz	0.14	A tool to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.
annobin	12.32	A build security checking tool.

4.3.2. C++ compatibility in GCC Toolset 13



IMPORTANT

The compatibility information presented here apply only to the GCC from GCC Toolset 13.

The GCC compiler in GCC Toolset can use the following C++ standards:

C++14

This language standard is available in GCC Toolset 13.
Using the C++14 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 6 or later.

C++11

This language standard is available in GCC Toolset 13.
Using the C++11 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 5 or later.

C++98

This language standard is available in GCC Toolset 13. Binaries, shared libraries and objects built using this standard can be freely mixed regardless of being built with GCC from GCC Toolset, Red Hat Developer Toolset, and RHEL 5, 6, 7 and 8.

C++17

This language standard is available in GCC Toolset 13.
This is the default language standard setting for GCC Toolset 13, with GNU extensions, equivalent to explicitly using option **-std=gnu++17**.

Using the C++17 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 10 or later.

C++20 and C++23

These language standards are available in GCC Toolset 13 only as an experimental, unstable, and unsupported capability. Additionally, compatibility of objects, binary files, and libraries built using this standard cannot be guaranteed.
To enable the C++20 standard, add the command-line option **-std=c++20** to your g++ command line.

To enable the C++23 standard, add the command-line option **-std=c++23** to your g++ command line.

All of the language standards are available in both the standard compliant variant or with GNU extensions.

When mixing objects built with GCC Toolset with those built with the RHEL toolchain (particularly **.o** or **.a** files), GCC Toolset toolchain should be used for any linkage. This ensures any newer library features provided only by GCC Toolset are resolved at link time.

4.3.3. Specifics of GCC in GCC Toolset 13

Static linking of libraries

Certain more recent library features are statically linked into applications built with GCC Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk because standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

IMPORTANT

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Specify libraries after object files when linking

In GCC Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable gcc-toolset-13 'gcc -lsomelib objfile.o'
```

Using a library from GCC Toolset in this manner results in the linker error message **undefined reference to symbol**. To prevent this problem, follow the standard linking practice and specify the option adding the library after the options specifying the object files:

```
$ scl enable gcc-toolset-13 'gcc objfile.o -lsomelib'
```

Note that this recommendation also applies when using the base Red Hat Enterprise Linux version of GCC.

4.3.4. Specifics of binutils in GCC Toolset 13

Static linking of libraries

Certain more recent library features are statically linked into applications built with GCC Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk because standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

IMPORTANT

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Specify libraries after object files when linking

In GCC Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable gcc-toolset-13 'ld -lsomelib objfile.o'
```

Using a library from GCC Toolset in this manner results in the linker error message **undefined reference to symbol**. To prevent this problem, follow the standard linking practice, and specify the option adding the library after the options specifying the object files:

```
$ scl enable gcc-toolset-13 'ld objfile.o -lsomelib'
```

Note that this recommendation also applies when using the base Red Hat Enterprise Linux version of **binutils**.

4.3.5. Specifics of annobin in GCC Toolset 13

Under some circumstances, due to a synchronization issue between **annobin** and **gcc** in GCC Toolset 13, your compilation can fail with an error message that looks similar to the following:

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-13/root/usr/lib/gcc/architecture-linux-gnu/13/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

To work around the problem, create a symbolic link in the plugin directory from the **annobin.so** file to the **gcc-annobin.so** file:

```
# cd /opt/rh/gcc-toolset-13/root/usr/lib/gcc/architecture-linux-gnu/13/plugin
# ln -s annobin.so gcc-annobin.so
```

Replace *architecture* with the architecture you use in your system:

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**

- **x86_64**

4.4. GCC TOOLSET 14

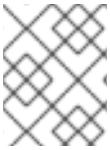
Learn about information specific to GCC Toolset version 14 and the tools contained in this version.

4.4.1. Tools and versions provided by GCC Toolset 14

GCC Toolset 14 provides the following tools and versions:

Table 4.3. Tool versions in GCC Toolset 14

Name	Version	Description
GCC	14.2.1	A portable compiler suite with support for C, C++, and Fortran.
binutils	2.41	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
dwz	0.14	A tool to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.
annobin	12.70	A build security checking tool.



NOTE

In RHEL 9.5, the system GDB was rebased to version 14.2, and GDB is no longer included in GCC Toolset.

4.4.2. C++ compatibility in GCC Toolset 14

IMPORTANT

The compatibility information presented here apply only to the GCC from GCC Toolset 14.

The GCC compiler in GCC Toolset can use the following C++ standards:

C++14

This language standard is available in GCC Toolset 14.
Using the C++14 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 6 or later.

C++11

This language standard is available in GCC Toolset 14.
Using the C++11 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 5 or later.

C++98

This language standard is available in GCC Toolset 14. Binaries, shared libraries and objects built using this standard can be freely mixed regardless of being built with GCC from GCC Toolset, Red Hat Developer Toolset, and RHEL 5, 6, 7 and 8.

C++17

This language standard is available in GCC Toolset 14.
This is the default language standard setting for GCC Toolset 14, with GNU extensions, equivalent to explicitly using option **-std=gnu++17**.

Using the C++17 language version is supported when all C++ objects compiled with the respective flag have been built using GCC version 10 or later.

C++20 and C++23

These language standards are available in GCC Toolset 14 only as an experimental, unstable, and unsupported capability. Additionally, compatibility of objects, binary files, and libraries built using this standard cannot be guaranteed.
To enable the C++20 standard, add the command-line option **-std=c++20** to your g++ command line.

To enable the C++23 standard, add the command-line option **-std=c++23** to your g++ command line.

All of the language standards are available in both the standard compliant variant or with GNU extensions.

When mixing objects built with GCC Toolset with those built with the RHEL toolchain (particularly **.o** or **.a** files), GCC Toolset toolchain should be used for any linkage. This ensures any newer library features provided only by GCC Toolset are resolved at link time.

4.4.3. Specifics of GCC in GCC Toolset 14

Static linking of libraries

Certain more recent library features are statically linked into applications built with GCC Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk because standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

IMPORTANT

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Specify libraries after object files when linking

In GCC Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable gcc-toolset-14 'gcc -lsomelib objfile.o'
```

Using a library from GCC Toolset in this manner results in the linker error message **undefined reference to symbol**. To prevent this problem, follow the standard linking practice and specify the option adding the library after the options specifying the object files:

```
$ scl enable gcc-toolset-14 'gcc objfile.o -lsomelib'
```

Note that this recommendation also applies when using the base Red Hat Enterprise Linux version of GCC.

4.4.4. Specifics of binutils in GCC Toolset 14

Static linking of libraries

Certain more recent library features are statically linked into applications built with GCC Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk because standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

IMPORTANT

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Specify libraries after object files when linking

In GCC Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable gcc-toolset-14 'ld -lsomelib objfile.o'
```

Using a library from GCC Toolset in this manner results in the linker error message **undefined reference to symbol**. To prevent this problem, follow the standard linking practice, and specify the option adding the library after the options specifying the object files:

```
$ scl enable gcc-toolset-14 'ld objfile.o -lsomelib'
```

Note that this recommendation also applies when using the base Red Hat Enterprise Linux version of **binutils**.

4.4.5. Specifics of annobin in GCC Toolset 14

Under some circumstances, due to a synchronization issue between **annobin** and **gcc** in GCC Toolset 14, your compilation can fail with an error message that looks similar to the following:

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-14/root/usr/lib/gcc/architecture-linux-gnu/14/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

To work around the problem, create a symbolic link in the plugin directory from the **annobin.so** file to the **gcc-annobin.so** file:

```
# cd /opt/rh/gcc-toolset-14/root/usr/lib/gcc/architecture-linux-gnu/14/plugin
# ln -s annobin.so gcc-annobin.so
```

Replace *architecture* with the architecture you use in your system:

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

4.5. USING THE GCC TOOLSET CONTAINER IMAGE

Only the GCC Toolset 14 container image is supported. Container images of earlier GCC Toolset versions are deprecated.

The GCC Toolset 14 components are available in the **GCC Toolset 14 Toolchain** container image.

The GCC Toolset container image is based on the **rhel9** base image and is available for all architectures supported by RHEL 9:

- AMD and Intel 64-bit architectures
- The 64-bit ARM architecture
- IBM Power Systems, Little Endian
- 64-bit IBM Z

4.5.1. GCC Toolset container image contents

Tools versions provided in the GCC Toolset 14 container image match the GCC Toolset 14 components versions.

The GCC Toolset 14 Toolchain contents

The **rhel9/gcc-toolset-14-toolchain** container image consists of the following components:

Component	Package
gcc	gcc-toolset-14-gcc
g++	gcc-toolset-14-gcc-c++
gfortran	gcc-toolset-14-gcc-gfortran

4.5.2. Accessing and running the GCC Toolset container image

The following section describes how to access and run the GCC Toolset container image.

Prerequisites

- Podman is installed.

Procedure

1. Access the Red Hat Container Registry using your Customer Portal credentials:

```
$ podman login registry.redhat.io
Username: username
Password: ********
```

2. Pull the container image you require by running a relevant command as root:

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-14-toolchain
```

NOTE

You can also set up your system to work with containers as a non-root user. For details, see Setting up rootless containers.

3. Optional: Check that pulling was successful by running a command that lists all container images on your local system:

```
# podman images
```

4. Run a container by launching a bash shell inside a container:

```
# podman run -it image_name /bin/bash
```

The **-i** option creates an interactive session; without this option the shell opens and instantly exits.

The **-t** option opens a terminal session; without this option you cannot type anything to the shell.

Additional resources

- Building, running, and managing Linux containers on RHEL 9
- Understanding root inside and outside a container (Red Hat Blog article)
- GCC Toolset container entries in the Red Hat Ecosystem Catalog

4.5.3. Example: Using the GCC Toolset 14 Toolchain container image

This example shows how to pull and start using the GCC Toolset 14 Toolchain container image.

Prerequisites

- Podman is installed.

Procedure

1. Access the Red Hat Container Registry using your Customer Portal credentials:

```
$ podman login registry.redhat.io
Username: username
Password: ********
```

2. Pull the container image as root:

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-14-toolchain
```

3. Launch the container image with an interactive shell as root:

```
# podman run -it registry.redhat.io/rhel9/gcc-toolset-14-toolchain /bin/bash
```

4. Run the GCC Toolset tools as expected. For example, to verify the **gcc** compiler version, run:

```
bash-4.4$ gcc -v
...
gcc version 14.2.1 20240801 (Red Hat 14.2.1-1) (GCC)
```

5. To list all packages provided in the container, run:

```
bash-4.4$ rpm -qa
```

4.6. COMPILER TOOLSETS

RHEL 9 provides the following compiler toolsets as Application Streams:

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

For more details and information about usage, see the compiler toolsets user guides on the Red Hat Developer Tools page.

4.7. THE ANNOBIN PROJECT

The Annobin project is an implementation of the Watermark specification project. Watermark specification project intends to add markers to Executable and Linkable Format (ELF) objects to determine their properties. The Annobin project consists of the **annobin** plugin and the **annockeck** program.

The **annobin** plugin scans the GNU Compiler Collection (GCC) command line, the compilation state, and the compilation process, and generates the ELF notes. The ELF notes record how the binary was built and provide information for the **annocheck** program to perform security hardening checks.

The security hardening checker is part of the **annocheck** program and is enabled by default. It checks the binary files to determine whether the program was built with necessary security hardening options

and compiled correctly. **annocheck** is able to recursively scan directories, archives, and RPM packages for ELF object files.

NOTE

The files must be in ELF format. **annocheck** does not handle any other binary file types.

The following section describes how to:

- Use the **annobin** plugin
- Use the **annocheck** program
- Remove redundant **annobin** notes

4.7.1. Using the annobin plugin

The following section describes how to:

- Enable the **annobin** plugin
- Pass options to the **annobin** plugin

4.7.1.1. Enabling the annobin plugin

The following section describes how to enable the **annobin** plugin via **gcc** and via **clang**.

Procedure

- To enable the **annobin** plugin with **gcc**, use:

```
$ gcc -fplugin=annobin
```

  - If **gcc** does not find the **annobin** plugin, use:

```
$ gcc -iplugindir=/path/to/directory/containing/annobin/
```

    Replace */path/to/directory/containing/annobin/* with the absolute path to the directory that contains **annobin**.

  - To find the directory containing the **annobin** plugin, use:

```
$ gcc --print-file-name=plugin
```

- To enable the **annobin** plugin with **clang**, use:

```
$ clang -fplugin=/path/to/directory/containing/annobin/
```

  Replace */path/to/directory/containing/annobin/* with the absolute path to the directory that contains **annobin**.

4.7.1.2. Passing options to the annobin plugin

The following section describes how to pass options to the **annobin** plugin via **gcc** and via **clang**.

Procedure

- To pass options to the **annobin** plugin with **gcc**, use:

```
$ gcc -fplugin=annobin -fplugin-arg-annobin-option file-name
```

Replace *option* with the **annobin** command line arguments and replace *file-name* with the name of the file.

Example

- To display additional details about what **annobin** it is doing, use:

```
$ gcc -fplugin=annobin -fplugin-arg-annobin-verbose file-name
```

Replace *file-name* with the name of the file.

- To pass options to the **annobin** plugin with **clang**, use:

```
$ clang -fplugin=/path/to/directory/containing/annobin/ -Xclang -plugin-arg-annobin -Xclang option file-name
```

Replace *option* with the **annobin** command line arguments and replace */path/to/directory/containing/annobin/* with the absolute path to the directory containing **annobin**.

Example

- To display additional details about what **annobin** it is doing, use:

```
$ clang -fplugin=/usr/lib64/clang/10/lib/annobin.so -Xclang -plugin-arg-annobin -Xclang verbose file-name
```

Replace *file-name* with the name of the file.

4.7.2. Using the annocheck program

The following section describes how to use **annocheck** to examine:

- Files
- Directories
- RPM packages
- **annocheck** extra tools

NOTE

annocheck recursively scans directories, archives, and RPM packages for ELF object files. The files have to be in the ELF format. **annocheck** does not handle any other binary file types.

4.7.2.1. Using annocheck to examine files

The following section describes how to examine ELF files using **annocheck**.

Procedure

- To examine a file, use:

```
$ annocheck file-name
```

  Replace *file-name* with the name of a file.

NOTE

The files must be in ELF format. **annocheck** does not handle any other binary file types. **annocheck** processes static libraries that contain ELF object files.

Additional information

- For more information about **annocheck** and possible command line options, see the **annocheck** man page on your system.

4.7.2.2. Using annocheck to examine directories

The following section describes how to examine ELF files in a directory using **annocheck**.

Procedure

- To scan a directory, use:

```
$ annocheck directory-name
```

  Replace *directory-name* with the name of a directory. **annocheck** automatically examines the contents of the directory, its sub-directories, and any archives and RPM packages within the directory.

NOTE

annocheck only looks for ELF files. Other file types are ignored.

Additional information

- For more information about **annocheck** and possible command line options, see the **annocheck** man page on your system.

4.7.2.3. Using annocheck to examine RPM packages

The following section describes how to examine ELF files in an RPM package using **annocheck**.

Procedure

- To scan an RPM package, use:

```
$ annocheck rpm-package-name
```

Replace *rpm-package-name* with the name of an RPM package. **annocheck** recursively scans all the ELF files inside the RPM package.

NOTE

annocheck only looks for ELF files. Other file types are ignored.

- To scan an RPM package with provided debug info RPM, use:

```
$ annocheck rpm-package-name --debug-rpm debuginfo-rpm
```

Replace *rpm-package-name* with the name of an RPM package, and *debuginfo-rpm* with the name of a debug info RPM associated with the binary RPM.

Additional information

- For more information about **annocheck** and possible command line options, see the **annocheck** man page on your system.

4.7.2.4. Using annocheck extra tools

annocheck includes multiple tools for examining binary files. You can enable these tools with the command-line options.

The following section describes how to enable the:

- **built-by** tool
- **notes** tool
- **section-size** tool

You can enable multiple tools at the same time.

NOTE

The hardening checker is enabled by default.

4.7.2.4.1. Enabling the built-by tool

You can use the **annocheck built-by** tool to find the name of the compiler that built the binary file.

Procedure

- To enable the **built-by** tool, use:

```
$ annocheck --enable-built-by
```

Additional information

- For more information about the **built-by** tool, see the **--help** command-line option.

4.7.2.4.2. Enabling the **notes** tool

You can use the **annocheck notes** tool to display the notes stored inside a binary file created by the **annobin** plugin.

Procedure

- To enable the **notes** tool, use:

```
$ annocheck --enable-notes
```

  The notes are displayed in a sequence sorted by the address range.

Additional information

- For more information about the **notes** tool, see the **--help** command-line option.

4.7.2.4.3. Enabling the **section-size** tool

You can use the **annocheck section-size** tool display the size of the named sections.

Procedure

- To enable the **section-size** tool, use:

```
$ annocheck --section-size=name
```

  Replace *name* with the name of the named section. The output is restricted to specific sections. A cumulative result is produced at the end.

Additional information

- For more information about the **section-size** tool, see the **--help** command-line option.

4.7.2.4.4. Hardening checker basics

The hardening checker is enabled by default. You can disable the hardening checker with the **--disable-hardened** command-line option.

4.7.2.4.4.1. Hardening checker options

The **annocheck** program checks the following options:

- Lazy binding is disabled using the **-z now** linker option.
- The program does not have a stack in an executable region of memory.
- The relocations for the GOT table are set to read only.
- No program segment has all three of the read, write and execute permission bits set.
- There are no relocations against executable code.
- The runpath information for locating shared libraries at runtime includes only directories rooted at /usr.

- The program was compiled with **annobin** notes enabled.
- The program was compiled with the **-fstack-protector-strong** option enabled.
- The program was compiled with **-D_FORTIFY_SOURCE=2**.
- The program was compiled with **-D_GLIBCXX_ASSERTIONS**.
- The program was compiled with **-fexceptions** enabled.
- The program was compiled with **-fstack-clash-protection** enabled.
- The program was compiled at **-O2** or higher.
- The program does not have any relocations held in a writeable.
- Dynamic executables have a dynamic segment.
- Shared libraries were compiled with **-fPIC** or **-fPIE**.
- Dynamic executables were compiled with **-fPIE** and linked with **-pie**.
- If available, the **-fcf-protection=full** option was used.
- If available, the **-mbranch-protection** option was used.
- If available, the **-mstackrealign** option was used.

4.7.2.4.4.2. Disabling the hardening checker

The following section describes how to disable the hardening checker.

Procedure

- To scan the notes in a file without the hardening checker, use:

```
$ annocheck --enable-notes --disable-hardened file-name
```

  Replace *file-name* with the name of a file.

4.7.3. Removing redundant annobin notes

Using **annobin** increases the size of binaries. To reduce the size of the binaries compiled with **annobin** you can remove redundant **annobin** notes. To remove the redundant **annobin** notes use the **objcopy** program, which is a part of the **binutils** package.

Procedure

- To remove the redundant **annobin** notes, use:

```
$ objcopy --merge-notes file-name
```

  Replace *file-name* with the name of the file.

4.7.4. Specifics of annobin in GCC Toolset 12

Under some circumstances, due to a synchronization issue between **annobin** and **gcc** in GCC Toolset 12, your compilation can fail with an error message that looks similar to the following:

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

To work around the problem, create a symbolic link in the plugin directory from the **annobin.so** file to the **gcc-annobin.so** file:

```
# cd /opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin
# ln -s annobin.so gcc-annobin.so
```

Replace *architecture* with the architecture you use in your system:

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

CHAPTER 5. SUPPLEMENTARY TOPICS

5.1. COMPATIBILITY BREAKING CHANGES IN COMPILERS AND DEVELOPMENT TOOLS

Non-constant PTHREAD_STACK_MIN, MINSIGSTKSZ, and SIGSTKSZ macros

In order to better support architectures that need a variable stack size for scalable vector registers, the constant value for the **PTHREAD_STACK_MIN**, **MINSIGSTKSZ**, and **SIGSTKSZ** macros have changed to a non-constant value, such as a **sysconf** call.

You can no longer use the **PTHREAD_STACK_MIN**, **MINSIGSTKSZ**, and **SIGSTKSZ** macros in a way that treats them like constant values. The value returned for the **PTHREAD_STACK_MIN**, **MINSIGSTKSZ**, and **SIGSTKSZ** macros is now of the long data type and might generate compiler warnings when compared against an unsigned value, such as **size_t**.

Libraries merged into libc.so.6

With this update, the following libraries have been merged into the **libc** library to provide a smoother in-place-upgrade experience, support safe use of threading at any time by a process, and to simplify the internal implementation:

- **libpthread**
- **libdl**
- **libutil**
- **libanl**

Additionally, parts of the **libresolv** library have been moved into libc to support moving the Name Switch Service (NSS) files and Domain Name System (DNS) plugins directly into the **libc** library. The NSS files and DNS plugins are now directly built into the **libc** library and can be used during an upgrade or across a **chroot** or container boundary. Their use across a **chroot** or container boundary supports safely querying Identity Management (IdM) data from those sources.

New location of zdump utility

/usr/bin/zdump is the new location of the **zdump** utility.

Deprecation of sys_siglist, _sys_siglist, and sys_sigabbrev symbols

The **sys_siglist**, **_sys_siglist**, and **sys_sigabbrev** symbols are exported only as compatibility symbols to support old binaries. All programs should use the **strsignal** symbol instead.

Using the **sys_siglist**, **_sys_siglist**, and **sys_sigabbrev** symbols creates issues such as copy relocations and an error-prone Application Binary Interface (ABI) with no explicit bound checks for the array access.

This change might affect building from source for some package. To fix the issue, rewrite the program to use the **strsignal** symbol instead. For example:

```
#include <signal.h>
#include <stdio.h>

static const char *strsig (int sig)
{
```

```
  return sys_siglist[sig];
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  return 0;
}
```

should be adjusted to:

```
#include <signal.h>
#include <stdio.h>
#include <string.h>

static const char *strsig (int sig)
{
  return strsignal(sig);
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  return 0;
}
```

Or, to use the **glibc-2.32** GNU extensions **sigabbrev_np** or **sigdescr_np**,:

```
#define _GNU_SOURCE
#include <signal.h>
#include <stdio.h>
#include <string.h>

static const char *strsig (int sig)
{
  const char *r = sigdescr_np (sig);
  return r == NULL ? "Unknown signal" : r;
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  printf ("%s\n", strsig (-1));
  return 0;
}
```

Both extensions are async-signal-safe and multithread-safe.

Deprecation of the sys_errlist, _sys_errlist, sys_nerr, and _sys_nerr symbols

The **sys_errlist**, **_sys_errlist**, **sys_nerr**, and **_sys_nerr** symbols are exported solely as compatibility symbols to support old binaries. All programs should use the **strerror** or **strerror_r** symbols instead.

Using the **sys_errlist**, **_sys_errlist**, **sys_nerr**, and **_sys_nerr** symbols creates issues such as copy relocations and an error-prone ABI with no explicit bound checks for the array access.

This change might affect building from source for some packages. To fix the problem, rewrite the program using the **strerror** or **strerror_r** symbols. For example:

```
#include <stdio.h>
#include <errno.h>

static const char *strerr (int err)
{
  if (err < 0 || err > sys_nerr)
    return "Unknown";
  return sys_errlist[err];
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

should be adjusted to:

```
#include <stdio.h>
#include <errno.h>

static const char *strerr (int err)
{
  return strerror (err);
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

Or, to use the **glibc-2.32** GNU extensions **strerrorname_np** or **strerrordesc_np**:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <string.h>

static const char *strerr (int err)
{
  const char *r = strerrordesc_np (err);
  return r == NULL ? "Unknown error" : r;
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
```

```
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

Both extensions are async-signal-safe and multithread-safe.

Userspace memory allocator, malloc, changes

The **mallwatch** and **tr_break** symbols are now deprecated and no longer used in the **mtrace** function. You can achieve similar functionality by using conditional breakpoints within **mtrace** functions from within GDB.

The **__morecore** and **__after_morecore_hook malloc** hooks and the default implementation, **__default_morecore**, have been removed from the API. Existing applications continue to link against these symbols but the interfaces no longer have any effect on **malloc**.

Debugging features in **malloc** such as the **MALLOC_CHECK_** environment variable (or the **glibc.malloc.check** tunable), **mtrace()**, and **mcheck()** have now been disabled by default in the main C library. To use these features, preload the new **libc_malloc_debug.so** debugging DSO.

The deprecated functions **malloc_get_state** and **malloc_set_state** have been moved from the core C library into the **libc_malloc_debug.so** library. Legacy applications that still use these functions must now preload the **libc_malloc_debug.so** library in their environment using the **LD_PRELOAD** environment variable.

The deprecated memory allocation hooks **__malloc_hook**, **__realloc_hook**, **__memalign_hook**, and **__free_hook** are now removed from the API. Compatibility symbols are present to support legacy programs, but new applications can no longer link to these symbols. These hooks no longer have any effect on **glibc** functionality. The **malloc** debugging DSO **libc_malloc_debug.so** currently supports hooks and can be preloaded to get this functionality back for older programs. However, this is a transitional measure and may be removed in a future release of the GNU C Library. You can port away from these hooks by writing and preloading your own **malloc** interposition library.

Lazy binding failures terminate the process

If a lazy binding failure happens during the **dlopen** function, during the execution of an ELF constructor, the process is now terminated. Previously, the dynamic loader returned **NULL** from **dlopen** with the lazy binding error captured in a **dlerror** message. In general, this is unsafe because resetting the stack in an arbitrary function call is not possible.

Deprecation of the stime function

The **stime** function is no longer available to newly linked binaries, and its declaration has been removed from **<time.h>**. Use the **clock_settime** function for programs that set the system time instead.

popen and system functions do not run atfork handlers anymore

Although it is a possible POSIX violation, the POSIX rationale in **pthread_atfork** documentation regarding **atfork** handlers is to handle inconsistent mutex states after a fork call in a multi-threaded process. In both the **popen** and **system** functions there is no direct access to user-defined mutexes.

Deprecated features in the C++ standard library

- **std::string::reserve(n)** will no longer reduce the string's capacity if called with an argument that is less than the string's current capacity. A string's capacity can be reduced by calling **reserve()** with no arguments, but that form is deprecated. The equivalent **shrink_to_fit()** should be used instead.

- The non-standard **std::__is_nullptr_t** type trait was deprecated. The standard **std::is_null_pointer** trait should be used instead.