Red Hat Enterprise Linux 10

Developing C and C++ applications in RHEL 10

Setting up a developer workstation, and developing and debugging C and C++ applications in Red Hat Enterprise Linux 10

Last Updated: 2026-01-29

Red Hat Enterprise Linux 10 Developing C and C++ applications in RHEL 10

Setting up a developer workstation, and developing and debugging C and C++ applications in Red Hat Enterprise Linux 10

Legal Notice

Abstract

Use the different features and utilities available in Red Hat Enterprise Linux 10 to develop and debug C and C++ applications.

Table of Contents

CHAPTER 1. SETTING UP A DEVELOPMENT WORKSTATION

Red Hat Enterprise Linux 10 supports development of custom applications. This chapter describes common development use cases and the required tools. You must ensure that the RHEL system, including a graphical environment, is installed and subscribed.

1.1. ENABLING DEBUG AND SOURCE REPOSITORIES

To debug system components and measure their performance, enable the debug and source repositories. These repositories contain the required information for debugging and profiling but are disabled by default in a standard Red Hat Enterprise Linux installation.

Procedure

1. Enable the BaseOS debug information package channel:

```
# subscription-manager repos --enable rhel-10-for-$(uname -i)-baseos-debug-rpms
```

2. Enable the BaseOS source package channel:

```
# subscription-manager repos --enable rhel-10-for-$(uname -i)-baseos-source-rpms
```

3. Enable the AppStream debug information package channel:

```
# subscription-manager repos --enable rhel-10-for-$(uname -i)-appstream-debug-rpms
```

4. Enable the AppStream source package channel:

```
# subscription-manager repos --enable rhel-10-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

1.2. SETTING UP TO MANAGE APPLICATION VERSIONS

To manage application versions in multi-developer projects, use Git, a distributed version control system included in Red Hat Enterprise Linux.

Procedure

1. Install the **git** package:

```
# dnf install git
```

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

```
$ git config --global user.name "Full_Name"
```

3. Optional: Set the email address associated with your Git commits:

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

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

4. Optional: To change the default text editor started by Git, set the 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

- Pro Git
- Reference

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

To develop applications using C and C++, install the necessary development tools and compilers provided by Red Hat Enterprise Linux. This procedure describes how to install the standard development tools, including the GNU Compiler Collection (GCC), GNU Debugger (GDB), and the LLVM toolset.

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 the **lld** linker:

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

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

```
# dnf install gcc-gfortran
```

1.4. SETTING UP TO DEBUG APPLICATIONS

To analyze and troubleshoot internal application behavior, {ProductName} offers multiple debugging and instrumentation tools.

Prerequisites

- The debug and source repositories is enabled.

Procedure

1. Install the tools for debugging:

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

2. Install the **dnf-utils** package 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 installed kernel. 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.

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.

Additional resources

- Enabling debugging with debugging information

1.5. SETTING UP TO MEASURE PERFORMANCE OF APPLICATIONS

To identify the causes of application performance loss, use the performance measurement tools included in Red Hat Enterprise Linux, such as **perf**, **papi**, **valgrind**, and SystemTap.

Prerequisites

- The debug and source repositories are 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 installed kernel. 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 section lists some of the most common development tasks.

2.1. GCC IN RHEL 10

Red Hat Enterprise Linux 10 is distributed with the GNU Compiler Collection (GCC) 14 as the standard compiler.

The default language standard setting for GCC 14 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 remain experimental.

Additional resources

- Porting to GCC 12
- Porting to GCC 13
- Porting to GCC 14

2.2. BUILDING CODE WITH GCC

Transform source code into executable code. Compile source files, link object code, and understand the relationship between different code forms. Optimize, harden, and debug your applications by using the GCC.

2.2.1. Relationship between code forms

The C and C++ languages have three forms of code that are created through different stages of the build process. Understanding these relationships helps you work effectively with the GNU Compiler Collection (GCC).

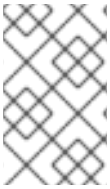
The code forms of C and C++ languages:

- **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 Static and dynamic linking.

Producing executable code from source code is performed in two steps, which require different applications or tools:

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

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.

You can run GCC to compile only, link only, or perform both steps. This is determined by the types of inputs and requested type of output.

Because larger projects require a build system that 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.

Additional resources

- Compiling source files to object code
- Linking code to create executable files
- Example: Building a C program with GCC (compiling and linking in one step)
- Example: Building a C program with GCC (compiling and linking in two steps)

2.2.2. Compiling source files to object code

To create object code files from source files without creating an executable file immediately, instruct GCC to create only object code files as its output. This is a basic operation of the build process for larger projects.

Prerequisites

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

Procedure

1. In Terminal, open 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.

Additional resources

- Options for hardening code with GCC
- Code optimization with GCC
- Building code with GCC

2.2.3. Enabling debugging of C and C++ applications with the GCC

To debug C and C++ applications effectively, generate debugging information during compilation. Use GCC's **-g** option to create this data. Debuggers use this data to map executable code to source lines for inspecting variables and logic.

Prerequisites

- You have the **gcc** package installed.

Procedure

1. Compile and link your code with the **-g** option to generate debugging information:

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

2. Optional: Set the optimization level to **-Og**:

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

Compiler optimizations can make executable code hard to relate to the source code. The **-Og** option optimizes the code without interfering with debugging. However, be aware that changing optimization levels can alter the program's behavior.

3. Optional: Use **-g** for moderate debugging information, or **-g3** to include macro definitions:

```
$ gcc ... -g3 ...
```

Verification

- Test the code by using the **-fcompare-debug** GCC option:

```
$ gcc -fcompare-debug ...
```

This option tests code compiled with and without debug information. If the resulting binaries are identical, the executable code is not affected by debugging options. By using the **-fcompare-debug** option significantly increases compilation time.

Additional resources

- Enabling debugging with debugging information
- Options for Debugging Your Program
- Debugging Information in Separate Files

2.2.4. Code optimization with the GCC

Compiler optimization transforms your code for efficiency. Use GCC options to balance compilation overhead with execution speed or binary size suitable for your deployment. By selecting the appropriate optimization level, you can achieve faster execution speeds or smaller binary sizes tailored to your application's deployment requirements.

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

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

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

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

GCC offers a large number of options to enable individual optimizations. For more information, see **gcc** man page for more details.

Additional resources

- Options That Control Optimization

2.2.5. Options for hardening code with the GCC

To add security checks during code compilation, you can use GNU Compiler Collection (GCC) compiler options. This helps produce more secure programs and libraries without changing 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=3 ...
```

- 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 ...
```

-fhardened

GCC 14 provides a new flag, **-fhardened**, which in turn enables several other flags to improve the security of generated code without impacting the ABI.

-fanalyzer

GCC provides a flag, **-fanalyzer**, which triggers warnings about potential issues in the source code, including security-related issues. Because **-fanalyzer** frequently has false positives and negatives, it should be used to locate potential bugs that should be investigated further and not as a formal analysis tool. This flag greatly increases the time and memory taken during compilation. Use only on C code.

Additional resources

- Defensive Coding Guide
- Compiler Options Hardening Guide for C and C++
- Memory Error Detection Using GCC

2.2.6. Linking code to create executable files

To create an executable file, linking combines all object files and libraries. This is the final step when building a C or C++ application.

Prerequisites

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

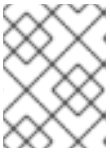
Procedure

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.

For more information, see Using libraries with GCC.



NOTE

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

Additional resources

- Building code with GCC
- Static and dynamic linking

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

To build a basic C program, use GCC to compile source code directly into an executable. This single-step compilation command creates a foundation for developing simple applications on Red Hat Enterprise Linux.

Procedure

1. Create a directory **hello-c**:

```
$ mkdir hello-c
```

2. Change to the created directory:

```
$ cd hello-c
```

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

```
#include <stdio.h>

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

4. 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.

5. Run the resulting executable file:

```
$ ./helloworld
```

```
Hello, World!
```

Additional resources

- Example: Building a C program using a Makefile

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

To build a simple C program, compile a source file into an object file and then link it to create an executable. This two-step process demonstrates the fundamentals of how to use the GNU Compiler Collection (GCC) compiler workflow for C development.

Procedure

1. Create a directory **hello-c**:

```
$ mkdir hello-c
```

2. Change to the created directory:

```
$ cd hello-c
```

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

```
#include <stdio.h>

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

4. Compile the code with GCC:

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

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

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

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

6. Run the resulting executable file:

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

7. Optional: Change back to the parent directory:

```
$ cd ..
```

8. Optional: Remove the **hello-c** directory:

```
$ rm -r hello-c
```

Additional resources

- Example: Building a C program using a Makefile

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

To build a minimal C++ program, use the following steps.

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

Procedure

1. Create a directory **hello-cpp**:

```
$ mkdir hello-cpp
```

2. Change to the created directory:

```
$ cd hello-cpp
```

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

```
#include <iostream>

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

4. 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.

5. Run the resulting executable file:

```
$ ./helloworld
```

```
Hello, World!
```

6. Optional: Change back to the parent directory:

```
$ cd ..
```

7. Optional: Remove the **hello-cpp** directory:

```
$ rm -r hello-cpp
```

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

To build a minimal C++ program by using a two-step process, first compile the source into an object file, and then link it to create the executable. This approach demonstrates modular building with the GNU Compiler Collection (GCC) compiler.

Procedure

1. Create a directory **hello-cpp**:

```
$ mkdir hello-cpp
```

2. Change to the created directory:

```
$ cd hello-cpp
```

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

```
#include <iostream>

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

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

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

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

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

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

6. Run the resulting executable file:

```
$ ./helloworld
```

```
Hello, World!
```

7. Optional: Change back to the parent directory:

```
$ cd ..
```

8. Optional: Remove the **hello-cpp** directory:

```
$ rm -r hello-cpp
```

2.3. CREATING LIBRARIES WITH GCC

Learn about the steps to creating libraries and the necessary concepts used by the Linux operating system for libraries. Also, you must learn a special file name convention used for libraries. See Section 2.4.1, “Library naming conventions” .

2.3.1. The soname mechanism

To manage multiple compatible versions of a library, dynamically loaded libraries (shared objects) use the *soname* mechanism.

- 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

The soname mechanism resolves these problems by using naming conventions to indicate compatibility. 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.

During the build, the linker searches for a symbolic link named **libfoo.so** that points to the library file. 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 by 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 that you want to examine.

Finding a library name and version in a file name

As an example, consider a library which is present as a file **libevent-2.0.so.5.1.9**. To find the actual components:

1. Start by ignoring the standard library file name prefix **lib**.
2. Break the remainder into the two parts preceding and following the string **.so.**.
3. The first part is **event-2.0**, which is the name of the library.
4. The second part is **5.1.9**. To find the *X* version component, take everything before first dot: **5**.
5. The rest is the *Y* version component: **1.9**.

Therefore the library's name is **event-2.0**, the *X* version component is 5, and *Y* is 1.9.

The soname of this library file is everything up to the *Y* component: **libevent-2.0.so.5**.

When a newer but still compatible version of the library is released, it uses the same soname, and the *Y* version component is increased. The new file name is **libevent-2.0.so.5.1.10**.

2.3.2. Creating dynamic libraries with the GCC

To build and install a dynamic library from the source code, you can use the GNU Compiler Collection (GCC). Dynamically linked libraries, also known as shared objects, help you conserve resources by reusing code and increase security by making library updates easier.

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 for the soname:

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

6. Create the symlink for the linker name:

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

Additional resources

- Shared Libraries

2.3.3. Creating static libraries with the GCC and ar

To create static libraries, bundle object files into an archive by using the **ar** utility. Use the resulting **.a** file for static linking and for distributing self-contained libraries without external dependencies.

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 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 ...
```

2.4. USING LIBRARIES WITH THE GCC

Libraries are collections of re-usable code, which can make coding easier and more effective. Understand library naming conventions and the distinction between static and dynamic linking. Link applications with static or dynamic libraries using the GCC, and optimize your code with Link Time Optimization (LTO).

2.4.1. Library naming conventions

System libraries require consistent naming. 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 the GNU Compiler Collection (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.

Additional resources

- The soname mechanism

2.4.2. Static and dynamic linking

When building C or C++ applications, you must use dynamic linking. Static linking reduces compatibility and prevents timely library security updates.

Comparison of static and dynamic linking

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

Static linking has numerous disadvantages and should be avoided, particularly for whole applications and the **glibc** and **libstdc++** libraries:

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.

However, static applications need fewer runtime 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 that 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 seems 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.

Cases for static linking

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

- When using a library that is not enabled for dynamic linking.

- When fully static linking is required for running code in an empty **chroot** environment or container. However, static linking by using the **glibc-static** package is not supported by Red Hat.

Additional resources

- The CodeReady Linux Builder repository

2.4.3. Link time optimization

Link time optimization (LTO) optimizes code across all translation units of the program by using intermediate representation at link time. This results in smaller and faster executable files and libraries. Additionally, LTO analyzes package source code at compile time more thoroughly, which improves various GCC diagnostics for potential coding errors.

Known issues

LTO has the following 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 might remove functions it considers unused because the compiler is unable to recognize which symbols an asm() statement references. A compilation error might 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 {mdash}; Symbol usage from assembly language.

  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 {mdash}; Function Attributes
- GCC Wiki {mdash}; Link Time Optimization

2.4.4. Using a library with the GCC

A library is a reusable package of code. A C or C++ library consists of the library code and 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.

For example, to specify a relative path **some/interesting/directory**:

```
$ gcc ... -Isome/interesting/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 you compile and link in a single **gcc** command, combine the compile-time and link-time options.

Additional resources

- Options for Directory Search
- Options for Linking

2.4.5. Linking static libraries with the GCC

To link static libraries, bundle them as archives that contain object files. After linking, they become part of the resulting executable file. Static linking overrides the default dynamic linking behavior.

NOTE

Red Hat discourages use of static linking for security reasons. See 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 that are part of Red Hat Enterprise Linux are supported for dynamic linking only. The steps below only work for libraries that are *not* enabled for dynamic linking.

See Static and 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**.

  a. Change to the directory containing your code.

  b. 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.

  c. 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**.

  d. 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 Using both static and dynamic libraries with GCC.

2.4.6. Using a dynamic library with the 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*.

Procedure

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

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

- To **use a run path value** stored in the executable file:
  The **run path** 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 uses the **run path** value to locate the library files.

  a. While linking with **GCC**, store the path *library_path* as **run path**:

```
$ gcc ... -Llibrary_path -lfoo -Wl,-run path=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,-run path=** option.

  b. Run the program:

```
$ ./program
```

  On Red Hat Enterprise Linux 10, the **run path** encoded in the program during linking is used only if the linked libraries are not found in **LD_LIBRARY_PATH**. You can use the **-Wl,--disable-new-dtags** option to restore the old behaviour in Red Hat Enterprise Linux 10,

where the run path is searched before the **LD_LIBRARY_PATH**.

- To **use the LD_LIBRARY_PATH environment variable**:
  Another way to set search paths to locate libraries is to 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 and must be set for every program invocation.

  a. Set the **LD_LIBRARY_PATH** environment variable:

  ```
  $ export LD_LIBRARY_PATH=library_path:$LD_LIBRARY_PATH
  ```

  b. Run the program:

  ```
  $ ./program
  ```

- Optional: **Place 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.

Additional resources

- Dynamic linker manual page
- Configuration file for the runtime linker
- Cache update and library directory listing with ldconfig

2.4.7. Using both static and dynamic libraries with GCC

Combining static and dynamic linking balances portability and efficiency. The GNU Compiler Collection (GCC) automatically selects shared objects over static archives unless configured otherwise. Understand this behavior to control exactly which library versions your application uses.

- Understanding static and dynamic linking

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, or 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 **-Bstatic** option to link libraries following this option statically, and **-Bdynamic** to link them dynamically. 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.

Link a program with 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

- Options for Linking
- Command Line Options

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.

What is 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.

To run **make** with a specific target:

```
$ make target
```

Makefile structure and syntax

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.

Basic Makefile example

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**.

Advanced Makefile with variables

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.

Installed documentation

- Use the **man** utility to view manual pages installed on your system:

```
$ man make
```

- Use the **info** utility to view information pages installed on your system:

```
$ info make
```

Additional resources

- An Introduction to Makefiles
- GNU Make Manual
- Building code with GCC

2.5.2. Example: Building a C program using a Makefile

To build a sample C program by using a Makefile, follow the steps in this example.

Prerequisites

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

Procedure

1. Create a directory **hellomake**:

```
$ mkdir hellomake
```

2. Change to the created directory:

```
$ cd hellomake
```

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

```
#include <stdio.h>

int main(int argc, char *argv[]) {
```

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

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

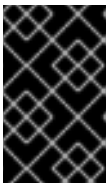
```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

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 might paste spaces instead of tabs. If this happens, correct the issue manually.

5. Run **make**:

```
$ make
```

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

This creates an executable file **hello**.

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

```
$ ./hello
```

```
Hello, World!
```

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

```
$ make clean
```

```
rm -rf hello.o hello
```

Additional resources

- Building code with GCC

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

Reliable debugging requires connecting binary code back to source code. Debugging information provides this link, essential for inspecting variables and execution flow. The GNU Compiler Collection (GCC) generates this data in the DWARF format within ELF files, which tools like the GNU Debugger (GDB) use to analyze program behavior.

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 the GCC

To debug C and C++ applications effectively, generate debugging information during compilation. Use GCC's **-g** option to create this data. Debuggers use this data to map executable code to source lines for inspecting variables and logic.

Prerequisites

- You have the **gcc** package installed.

Procedure

1. Compile and link your code with the **-g** option to generate debugging information:

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

2. Optional: Set the optimization level to **-Og**:

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

Compiler optimizations can make executable code hard to relate to the source code. The **-Og** option optimizes the code without interfering with debugging. However, be aware that changing optimization levels can alter the program's behavior.

3. Optional: Use **-g** for moderate debugging information, or **-g3** to include macro definitions:

```
$ gcc ... -g3 ...
```

Verification

- Test the code by using the **-fcompare-debug** GCC option:

```
$ gcc -fcompare-debug ...
```

This option tests code compiled with and without debug information. If the resulting binaries are identical, the executable code is not affected by debugging options. By using the **-fcompare-debug** option significantly increases compilation time.

Additional resources

- Enabling debugging with debugging information
- Options for Debugging Your Program
- Debugging Information in Separate Files

3.1.3. Debuginfo and debugsource packages

The **debuginfo** and **debugsource** packages contain debugging information and source code for programs and libraries. To debug Red Hat Enterprise Linux applications, install these packages from additional repositories.

Debugging information package types

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 **debuginfo** and **debugsource** packages 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.

Additional resources

- Debugging information
- Enabling debug and source repositories

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

To obtain the necessary debuginfo packages for troubleshooting installed applications or libraries, use the GNU Debugger (GDB). It automatically detects missing symbols and identifies the specific packages needed. Follow GDB's recommendations to install these packages and enable full debugging

capabilities.

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.
- 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-9.5-6.el10.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-9.5-6.el10.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

To manually determine which **debuginfo** packages you need to install, locate the executable file and find the package that installs it.

NOTE

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
```

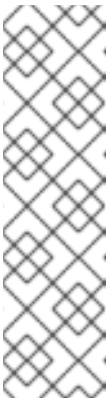
```
/usr/bin/less
```

   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
```

1. Update the database:

```
# updatedb
```

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

```
$ rpm -qf /usr/lib64/libz.so.1.3.1.zlib-ng
```

```
zlib-ng-compat-2.2.3-1.el10.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 corresponding **debuginfo** package exists.

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

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

```
# dnf debuginfo-install zlib-ng-compat-2.2.3-1.el10.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)

Use the GNU Debugger (GDB) to inspect program execution and post-crash states. Also, you can analyze internal data and control execution flow when tracking down runtime errors. This command-line tool shows the detailed application state needed to identify and fix bugs in complex applications.

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

To examine a system process, attach the **GNU Debugger (GDB)** to the process.

Prerequisites

- GDB must be installed on the system

Procedure

- Start a program with GDB.
  a. Launch a program using 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. For more information on setting breakpoints, see Using GDB breakpoints to stop execution at defined code locations.

  b. Attach GDB to an already running process.

  c. 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.

  d. Attach GDB to this process:

```
$ gdb -p pid
```

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

- Attach an active GDB session to a running program:
  a. 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.

b. 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 corresponding executable file. Use the **file** command to specify the path:

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

Additional resources

- Invoking GDB
- Debugging an Already-running Process

3.2.3. Controlling program execution with GDB

When the GNU Debugger (GDB) has been attached to a program, you can use several commands to control the execution of the program. These commands parse code, set breakpoints, and control program flow during debugging sessions.

To use these GDB commands effectively, you must have debugging information available through these methods:

- The program is compiled and built with debugging information
- The relevant debuginfo packages are installed
- GDB must be attached to the program to be debugged

GDB provides commands for stepping through and controlling program execution:

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

- Using GDB breakpoints to stop execution at defined code locations
- Starting your Program
- 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. The GNU Debugger (GDB) offers multiple commands that you can use to inspect the internal variables. The following list describes 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 program source code. When the program has been started, but is currently stopped, this command lists the source code where the program is currently stopped, along with a few lines of context. Before the program is started, this will list the main function. 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
- Pretty Printing

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

During a debugging session, you often need to investigate only specific sections of code. Breakpoints are markers that instruct GDB to stop the execution of a program at a defined location. Since breakpoints are typically associated with lines of source code, placing them requires you to specify the correct source file and line number.

Procedure

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

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

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

```
(gdb) br line
```

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

```
(gdb) br function_name
```

d. 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

- Breakpoints, Watchpoints, and Catchpoints

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

GDB watchpoints pause execution when data changes or is accessed. Use them to debug unexpected variable corruption when the cause is unknown. Setting a watchpoint stops the program at the exact moment of access to inspect the state and find the root cause.

Procedure

- 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

- Setting Watchpoints

3.2.7. Debugging forking or threaded programs with GDB

Some programs use forking or threads to achieve parallel code execution. To debug multiple simultaneous execution paths, you can use a variety of commands based on your use case.

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 Programs with Multiple Threads
- 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 for recording application interactions

To record application interactions, you can use several tools are available in RHEL. For system calls use **strace**, for library calls use **ltrace**, and for advanced probing SystemTap. Select the appropriate tool to log specific runtime behaviors and diagnose integration issues.

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 corresponding 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 might be lacking.

- To reduce the amount of captured data, filter the traced calls.
- 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.
- To reduce the amount of captured data, filter the traced calls.
- 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 corresponding 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 such as **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 10, 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 includes 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

To monitor the system (kernel) calls performed by an application, use the **strace** tool.

Prerequisites

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

Procedure

1. Identify the system calls to monitor.
   - 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*):

       ```
       $ ps -C program
       ```

     - Attach **strace** to the process:

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

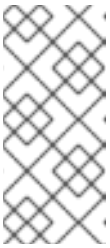
   - Replace *call* with the system calls to be displayed. You can use the **-e trace=*call*** option multiple times. If left out, **strace** will display all system call types. See the *strace(1)* manual page for more information.
   - If you do not want to trace any forked processes or threads, omit the **-f** option.
2. 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 displays immediately, if no filter for system calls is set.

3. 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**.

4. 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 large amount 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

- How do I use strace to trace system calls made by a command? (Red Hat Knowledgebase)
- strace (Red Hat Developer Toolset)

3.3.3. Monitoring application's library function calls with ltrace

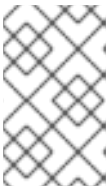
To monitor an application's calls to functions available in libraries (shared objects), use the **ltrace** tool.

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.



NOTE

In Red Hat Enterprise Linux 10, a known issue prevents **ltrace** from tracing system executable files. This limitation does not apply to executable files built by users.

- 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*):

```
$ ps -C program
```

- Attach **ltrace** to the process:

```
$ 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, omit the **-f** option.

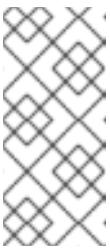
  See the *ltrace*(1)_ manual page for more information.

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. Stop **ltrace** by pressing **Ctrl**+**C**.

6. 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 large amount 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

- ltrace - (Red Hat Developer Toolset)

3.3.4. Monitoring application's system calls with SystemTap

To register custom event handlers for kernel events, use the SystemTap tool. SystemTap is more efficient than the **strace** tool, but requires more setup. The included **strace.stp** script provides **strace**-like functionality. Installing SystemTap also installs the **strace.stp** script, which provides an approximation of the **strace** functionality when using SystemTap.

Prerequisites

- SystemTap and the corresponding 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

To stop program execution when the program performs specific system calls, use GNU Debugger (GDB) *catchpoints*, then inspect the program state and system call parameters at those points.

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.
4. Use further GDB commands to examine the program state and advance execution, according to the particular situation.
5. To exit the GDB debugging session:

   ```
   (gdb) q
   ```

Additional resources

- Showing program internal values with GDB
- Controlling program execution with GDB
- Setting Watchpoints

3.3.6. Using GDB to intercept handling of signals by applications

To stop the execution of a program under specific circumstances, you can use the GNU Debugger (GDB). 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.
4. Continue by debugging the program code handling the signal, or resume execution with the knowledge of the signal being received.
5. Later, to exit the GDB debugging session:

```
(gdb) q
```

Additional resources

- Showing program internal values with GDB
- Controlling program execution with GDB
- 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 records parts of an application's memory when the application stops. After an application fails, you can analyze the core dump, along with the executable and debuginfo, by using a debugger.

The Linux operating system kernel can record core dumps automatically, if this functionality is enabled. Alternatively, 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 the core dump:

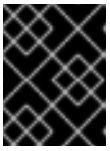
```
$ coredumpctl list executable-name
```

8. Export the core dump:

```
$ 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 by using the other information. See the *coredumpctl(1)* manual page for details.

9. 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.

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

Additional resources

- Managing systemd
- How to enable core file dumps when an application crashes or segmentation faults
- 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

To inspect the state of an application at the moment it terminated unexpectely, use 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 missing debuginfo packages. For more details, see 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:

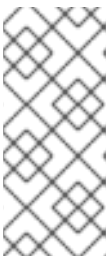
   ```
   $ gdb -e executable_file -c core_file
   ```

   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
   ```

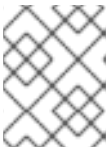
   Replace *program.debug* with the actual file name.



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.

7. To see only the stack of the application at the moment it terminated, open the core file with the **eu-stack** utility:

```
$ eu-stack --core core-file
```

This will display listing of the application's stack.

Additional resources

- Choosing Files
- Commands to Specify Files
- Debugging Information in Separate Files

3.4.4. Creating and accessing a core dump with coredumpctl

To manage and analyze core dumps directly on the affected system, use **coredumpctl**. This tool simplifies finding, capturing, and inspecting crash data. Identify an unresponsive process, force a core dump, and verify its successful capture to diagnose application failures.

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.

Next steps

- To show details of the core file, run:

```
$ coredumpctl info PID
```

- To load the core file in the GDB debugger, run:

```
$ coredumpctl debug PID
```

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

```
Missing separate debuginfos, use: dnf debuginfo-install bc-1.07.1-23.el10.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, run:

```
$ 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.

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

3.4.5. Dumping process memory with gcore

To capture the memory state of a running process without terminating it, use the **gcore** utility. This creates a core dump file for offline analysis. The result is a snapshot of the application's memory that helps you investigate issues while the service remains available.

Prerequisites

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

Procedure

1. Identify 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

To dump protected process memory, configure GNU Debugger (GDB) to ignore core dump filters. Capture memory regions flagged as non-dumpable, such as those conserving resources or holding sensitive data. Use the **gcore** command within GDB to generate the complete core file.

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
```

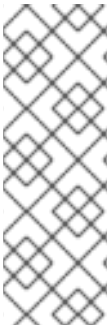
Replace *core-file* with name of file where you want to dump the memory.

Additional resources

- How to Produce a Core File from Your Program

3.5. DEBUGGING APPLICATIONS IN CONTAINERS

To troubleshoot container applications, you can use various 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 might 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 Creating a rootless container with bind mount by using the podman RHEL system role 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 difficult to know the reasons. Exporting the filesystem structure from the container will allow for checking other logs files that might not be in the mounted volumes.

Additional resources

- Debugging applications within Red Hat OpenShift containers
- Debugging a Crashed Application
- Troubleshooting Kubernetes
- Tips and Tricks for containerizing services

CHAPTER 4. ADDITIONAL TOOLSETS FOR DEVELOPMENT

Additional toolsets for C and C++ development provides capabilities for building, analyzing, and optimizing applications. By using these toolsets, you can define development workflows and improve application quality.

4.1. USING THE GCC TOOLSET

4.1.1. What is the GCC Toolset

Red Hat Enterprise Linux 10 introduces the GCC Toolset, which is an Application Stream containing updated versions of development and performance analysis tools. The GCC Toolset is similar to Red Hat Developer Toolset.

GCC Toolset is available as an Application Stream in the form of a software collection in the **AppStream** repository. The 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 the 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. By 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 by using the **scl** utility. Unless noted otherwise for specific tools or features, the 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 the GCC Toolset

Installing the 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 the GCC Toolset version *N*:

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

4.1.3. Installing individual packages from the GCC Toolset

To install only certain tools from the {gcct} instead of the whole toolset, list the available packages and install the selected ones with the **dnf** package management tool. Use selective installation to access packages not installed by default with the full toolset.

Procedure

1. List the packages available in the 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-15-annobin-annocheck** and **gcc-toolset-15-binutils-devel** packages:

```
# dnf install gcc-toolset-15-annobin-annocheck gcc-toolset-15-binutils-devel
```

4.1.4. Uninstalling the GCC Toolset

Remove the GCC Toolset from your system by uninstalling it using the **dnf** package management tool.

Procedure

- To uninstall the GCC Toolset version *N*:

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

4.1.5. Accessing the GCC Toolset

To access the {gcct}, you can run a specific tool using the **scl** utility, or start a shell session where the toolset versions override the system versions.

Procedure

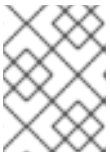
- To run a single tool from the {gcct} version *N*:

```
$ gcc-toolset-N-env tool
```

Replace *tool* with the command provided by the tool you want to run.

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

```
$ gcc-toolset-N-env bash
```



NOTE

The **scl** utility is not used for the GCC Toolset in Red Hat Enterprise Linux 10. The **scl enable** command does not work with the GCC Toolset.

4.1.6. Additional resources

- Developer Toolset User Guide

4.2. GCC TOOLSET 15

GCC Toolset 15 in Red Hat Enterprise Linux offers updated compilers and debuggers for C, C++, and Fortran. It enables building, testing, and optimizing applications with current features while maintaining system stability and support.

4.2.1. The GCC Toolset 15 tools and versions

The GCC Toolset 15 offers updated versions of development tools for building and debugging applications on RHEL.

- [list of tools]

Table 4.1. Tool versions in the GCC Toolset 15

Name	Version	Description
GCC	15.1.0	A portable compiler suite with support for C, C++, and Fortran.
binutils	2.44	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
dwz	0.16	A tool to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.

4.2.2. C++ compatibility in the GCC Toolset 15

GCC Toolset 15 supports a range of C++ language standards. The default standard is C++17, but you can choose other options such as C++98, C++11, C++14, or experimental versions including C++20, C++23, and C++26. To select a different standard, use the appropriate compiler flag when building your code.

IMPORTANT

This compatibility information applies only to GCC from the GCC Toolset 15.

The GCC compiler in the GCC Toolset 15 can use the following C++ standards:

C++98

This language standard is available in the GCC Toolset 15. Binaries, shared libraries, and objects built using this standard can be freely mixed regardless of being built with GCC from the GCC Toolset 15, Red Hat Developer Toolset, and RHEL 5, 6, 7, and 8.

C++11

This language standard is available in the GCC Toolset 15.
Using the C++11 language version is supported when all C++ objects compiled with the corresponding flag have been built using GCC version 5 or later.

C++14

This language standard is available in the GCC Toolset 15.
Using the C++14 language version is supported when all C++ objects compiled with the corresponding flag have been built using GCC version 6 or later.

C++17

This language standard is available in the GCC Toolset 15.
This is the default language standard setting for the GCC Toolset 15, 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 corresponding flag have been built using GCC version 10 or later.

C++20, C++23, and C++26

These language standards are available in the GCC Toolset 15 only as experimental, unstable, and unsupported capabilities. Additionally, the compatibility of objects, binary files, and libraries built using these standards 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.

To enable the C++26 standard, add the command-line option **-std=c++26** to your **g++** command line.

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

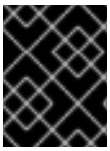
Use the GCC Toolset 15 for linking when you combine objects built with the GCC Toolset 15 and objects built with the system toolchain, particulary **.o** or **.a** files. This ensures any newer library features provided only by the GCC Toolset 15 are resolved at link time.

4.2.3. Specifics of GCC in the GCC Toolset 15

Certain behaviors and requirements of binutils in the GCC Toolset 15 differ from the base Red Hat Enterprise Linux binutils. These include automatic static linking of certain library features and the requirement to specify libraries after object files during linking.

Static linking of libraries

Certain more recent library features are statically linked into applications built with the GCC Toolset 15 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 the GCC Toolset 15, libraries are linked by 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 corresponding 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:

```
$ gcc-toolset-15-env gcc -lsomelib objfile.o
```

Using a library from the GCC Toolset 15 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 by adding the library after the options specifying the object files:

```
$ gcc-toolset-15-env 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 the GCC Toolset 15

Certain behaviors and requirements of binutils in the GCC Toolset 15 differ from the base Red Hat Enterprise Linux binutils. These include automatic static linking of certain library features and the requirement to specify libraries after object files during linking.

Static linking of libraries

GCC Toolset 15 statically links newer library features into applications to ensure compatibility across multiple Red Hat Enterprise Linux versions. Statically linked code can introduce minor security risks, because security updates require applications to be rebuilt. If a security vulnerability is discovered, Red Hat will notify developers to rebuild affected applications through a security advisory.

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 the GCC Toolset 15, libraries are linked by 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 corresponding 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:

```
$ gcc-toolset-15-env gcc ld -lsomelib objfile.o
```

Using a library from the GCC Toolset 15 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:

```
$ gcc-toolset-15-env 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 the GCC Toolset 15

Builds that use the GCC Toolset 15 and **annobin** can fail due to a synchronization issue between the **annobin** plug-in and **gcc**. This causes the compiler to fail locating the **gcc-annobin.so** plug-in file.

```
cc1: fatal error: inaccessible plugin file
```

```
opt/rh/gcc-toolset-15/root/usr/lib/gcc/_architecture_-linux-gnu/15/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

1. Change to the plug-in directory:

```
$ cd /opt/rh/gcc-toolset-15/root/usr/lib/gcc/architecture-linux-gnu/15/plugin
```

2. Create a symbolic link from **annobin.so** to **gcc-annobin.so**:

```
$ ln -s annobin.so gcc-annobin.so
```

Replace *architecture* with the architecture used on your system:

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

4.3. COMPILER TOOLSETS

RHEL 10 provides several compiler toolsets as Application Streams, including the LLVM Toolset, Rust Toolset, and Go Toolset. These toolsets provides compilers, debuggers, dependency managers, and other related tools and libraries for C, C++, Rust, and Go development.

The following compiler toolsets are available:

- 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** plug-in, 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.

4.4. 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

4.4.1. Using the annobin plugin

The following section describes how to:

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

4.4.1.1. Enabling the annobin plug-in

To add build security notes to binaries, enable the **annobin** plug-in by using command-line options with **gcc** or **clang** utilities.

Procedure

- To enable the **annobin** plug-in with **gcc**, use:

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

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

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

- To enable the **annobin** plug-in 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**.

- Optional: To remove the redundant **annobin** notes, use the **objcopy** utility:

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

4.4.1.2. Passing options to the annobin plug-in

To pass options to the **annobin** plug-in, use the appropriate command-line arguments with **gcc** or **clang**.

Procedure

- To pass options to the **annobin** plug-in 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.

  - For 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** plug-in 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**.

  - For 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.

4.4.2. 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.

4.4.2.1. Using annocheck to examine files

To verify hardening options and build security notes of ELF files, examine the files by using the **annocheck** tool.

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.

4.4.2.2. Using annocheck to examine directories

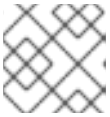
To examine ELF files in a directory, use the **annocheck** tool, which recursively scans directories, subdirectories, and archives.

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.

4.4.2.3. Using annocheck to examine RPM packages

To examine ELF files in an RPM package, use the **annocheck** tool, which recursively scans all ELF files inside the package.

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.

4.4.2.4. 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.

4.4.2.4.1. Enabling the built-by tool

To find the name of the compiler that built a specific binary file, you can use the **annocheck built-by** tool.

Procedure

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

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

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

4.4.2.4.2. Enabling the notes tool

To display the notes stored inside a binary file created by the **annobin** plug-in, you can use the **annocheck notes** tool.

Procedure

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

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

The notes are displayed in a sequence sorted by the address range. For more information about the **notes** tool, see the **--help** command-line option.

4.4.2.4.3. Enabling the section-size tool

To display the size of named sections, you can use the **annocheck section-size** tool.

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

4.4.2.4.4. Hardening checker basics

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

4.4.2.4.4.1. Hardening checker options

The **annocheck** tool verifies binaries for various hardening options, such as stack protection, PIC/PIE usage, and secure linker settings.

The following options are checked:

- 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.

4.4.2.4.4.2. Disabling the hardening checker

To skip security checks during binary analysis, disable the hardening checker by using the **annocheck** utility.

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.

4.4.3. Removing redundant annobin notes

Using **annobin** increases the size of binaries. To reduce the size of the binaries compiled with **annobin**, 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.

CHAPTER 5. NOTABLE CHANGES IN RHEL 10

The following are important changes that occur in RHEL 10.

5.1. COMPATIBILITY BREAKING CHANGES IN C++

In GNU Compiler Collection (GCC) 12 and later, **std::condition_variable::wait** is a thread cancellation point. This allows **pthread_cancel** to clean up the stack, whereas in GCC 11 and earlier, the function was **noexcept** and cancellation terminated the process. If you have code that depends on **wait** never throwing an exception, review the code and take appropriate action. Furthermore, newer C++ versions deprecate certain class templates, which now trigger warning diagnostics.

std::condition_variable::wait is now a thread cancellation point

In GCC 11 and earlier, the **std::condition_variable::wait** function was **noexcept**, which made it incompatible with thread cancellation. As a result, if a call to **pthread_cancel** canceled a thread blocked in **std::condition_variable::wait**, the process would be terminated. In GCC 12 and later, **std::condition_variable::wait** can be canceled by calls to the **pthread_cancel** function, which unwinds the stack. If you have code that depends on **wait** never throwing an exception, review the code and take appropriate action.

Deprecated class templates, Certain class templates have been deprecated in new versions of C++ and produce warning diagnostics in GCC 12 and later

- The following class templates have been deprecated in C++11 and later:
  - **std::unary_function**
  - **std::binary_function**
- The **std::iterator** class template has been deprecated in C++17 and later.

To prevent the warning diagnostics, you can take one of the following actions:

- When you do not want to make other changes to the code, silence the warning diagnostics by using GCC's diagnostic pragmas. For example:

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored “-Wdeprecated-declarations"
class Functor : public std::unary_function<int, int>
{ /* … */ };
#pragma GCC diagnostic pop
```

- When you want your code to be compatible with later versions of C++, replace these class templates in the code with nested typedefs. For example, you can replace a **std::unary_function** base class with **result_type** and **argument_type** typedefs:

```
class Functor
{
  using result_type = int;
  using argument_type = int;
  /* … */
};
```

5.2. COMPATIBILITY BREAKING CHANGES TO GLIBC

Updates to the GNU C Library (**glibc**) include compatibility breaking changes to the dynamic linker search algorithm, which no longer checks certain subdirectories for shared objects. This release also removes the **catchsegv** script and the **libSegFault.so** shared object.

Changes to how the dynamic linker finds shared objects

The following list describes changes to the dynamic linker search algorithm:

- The dynamic linker no longer loads shared objects from the **tls** subdirectories on the library search path or the subdirectory that corresponds to the **AT_PLATFORM** system name.
- The dynamic linker no longer searches subdirectories named after the legacy **AT_HWCAP** search mechanism.

Port your applications to the **glibc-hwcaps** mechanism, which has been available since RHEL 8.4.

Removed catchsegv script andlibSegFault.so shared object

The **catchsegv** script and associated **libSegFault.so** shared object have been removed. You can use an out-of-process alternative for intercepting coredumps and backtraces, such as **systemd-coredump** and **coredumpctl**.

5.3. C23 SUPPORT

The GNU C Library (**glibc**) supports new features introduced in the C23 standard, formerly known as C2X. For more information about these features, see the **glibc** manual.

5.4. RHEL 10 USES IEEE BINARY128 FOR LONG DOUBLE ON POWER

On the POWER (**ppc64le**) architecture, the default representation of the **long double** type changes from the IBM-specific double-double format to the standard IEEE 754 binary128 format in RHEL 10. Both formats use 16 bytes, but their internal representations differ. The C and C++ core runtime libraries in RHEL 10 (**glibc** and **libstdc++**) support both formats. The rest of the system uses binary128 by default. If your applications use the **long double** type on POWER, you must recompile them on RHEL 10 to ensure correct behavior.