Red Hat Developer Toolset 11

User Guide

Installing and Using Red Hat Developer Toolset

Last Updated: 2022-07-27

Red Hat Developer Toolset 11 User Guide

Installing and Using Red Hat Developer Toolset

Olga Tikhomirova
Red Hat Customer Content Services
otikhomi@redhat.com

Zuzana Zoubková
Red Hat Customer Content Services

Jaromír Hradílek
Red Hat Customer Content Services

Matt Newsome
Red Hat Software Engineering

Robert Krátký
Red Hat Customer Content Services

Vladimír Slávik
Red Hat Customer Content Services

Legal Notice

Copyright © 2022 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Red Hat Developer Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. The Red Hat Developer Toolset User Guide provides an overview of this product, explains how to invoke and use the Red Hat Developer Toolset versions of the tools, and links to resources with more in-depth information.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE 7

PART I. INTRODUCTION 8

CHAPTER 1. RED HAT DEVELOPER TOOLSET 9
1.1. ABOUT RED HAT DEVELOPER TOOLSET 9
What Is New in Red Hat Developer Toolset 11.0 9
1.2. MAIN FEATURES 10
1.3. COMPATIBILITY 11
Architecture support 11
1.4. GETTING ACCESS TO RED HAT DEVELOPER TOOLSET 11
1.4.1. Using Red Hat Software Collections 11
1.5. INSTALLING RED HAT DEVELOPER TOOLSET 13
1.5.1. Installing All Available Components 13
1.5.2. Installing Individual Package Groups 13
1.5.3. Installing Optional Packages 14
1.5.4. Installing Debugging Information 14
1.6. UPDATING RED HAT DEVELOPER TOOLSET 15
1.6.1. Updating to a Minor Version 15
1.6.2. Updating to a Major Version 15
1.7. UNINSTALLING RED HAT DEVELOPER TOOLSET 15
1.8. USING RED HAT DEVELOPER TOOLSET CONTAINER IMAGES 16
1.9. ADDITIONAL RESOURCES 16
Online Documentation 16
See Also 17

PART II. DEVELOPMENT TOOLS 18

CHAPTER 2. GNU COMPILER COLLECTION (GCC) 19
2.1. GNU C COMPILER 19
2.1.1. Installing the C Compiler 19
2.1.2. Using the C Compiler 19
2.1.3. Running a C Program 20
2.2. GNU C++ COMPILER 20
2.2.1. Installing the C++ Compiler 21
2.2.2. Using the C++ Compiler 21
2.2.3. Running a C++ Program 22
2.2.4. C++ Compatibility 22
2.2.4.1. C++ ABI 23
2.3. GNU FORTRAN COMPILER 24
2.3.1. Installing the Fortran Compiler 24
2.3.2. Using the Fortran Compiler 24
2.3.3. Running a Fortran Program 25
2.4. SPECIFICS OF GCC IN RED HAT DEVELOPER TOOLSET 25
2.5. ADDITIONAL RESOURCES 26
Installed Documentation 26
Online Documentation 26
See Also 27

CHAPTER 3. GNU MAKE 28
3.1. INSTALLING MAKE 28
3.2. USING MAKE 28

3.3. USING MAKEFILES 29
3.4. ADDITIONAL RESOURCES 30
Installed Documentation 30
Online Documentation 30
See Also 30

CHAPTER 4. BINUTILS 32
4.1. INSTALLING BINUTILS 33
4.2. USING THE GNU ASSEMBLER 33
4.3. USING THE GNU LINKER 33
4.4. USING OTHER BINARY TOOLS 34
4.5. SPECIFICS OF BINUTILS IN RED HAT DEVELOPER TOOLSET 35
4.6. ADDITIONAL RESOURCES 35
Installed Documentation 35
Online Documentation 35
See Also 35

CHAPTER 5. ELFUTILS 37
5.1. INSTALLING ELFUTILS 38
5.2. USING ELFUTILS 38
5.3. ADDITIONAL RESOURCES 38
See Also 38

CHAPTER 6. DWZ 39
6.1. INSTALLING DWZ 39
6.2. USING DWZ 39
6.3. ADDITIONAL RESOURCES 39
Installed Documentation 39
See Also 39

CHAPTER 7. ANNOBIN 41
7.1. INSTALLING ANNOBIN 41
7.2. USING ANNOBIN PLUGIN 41
7.3. USING ANNOCHECK 41
7.4. ADDITIONAL RESOURCES 42
Installed Documentation 42

PART III. DEBUGGING TOOLS 43

CHAPTER 8. GNU DEBUGGER (GDB) 44
8.1. INSTALLING THE GNU DEBUGGER 44
8.2. PREPARING A PROGRAM FOR DEBUGGING 44
Compiling Programs with Debugging Information 44
Installing Debugging Information for Existing Packages 45
8.3. RUNNING THE GNU DEBUGGER 45
8.4. LISTING SOURCE CODE 46
8.5. SETTING BREAKPOINTS 47
Setting a New Breakpoint 47
Listing Breakpoints 48
Deleting Existing Breakpoints 48
8.6. STARTING EXECUTION 49
8.7. DISPLAYING CURRENT VALUES 49
8.8. CONTINUING EXECUTION 50
8.9. ADDITIONAL RESOURCES 50
Installed Documentation 50

Online Documentation 51
See Also 51

CHAPTER 9. STRACE 52
9.1. INSTALLING STRACE 52
9.2. USING STRACE 52
9.2.1. Redirecting Output to a File 52
9.2.2. Tracing Selected System Calls 53
9.2.3. Displaying Time Stamps 54
9.2.4. Displaying a Summary 55
9.2.5. Tampering with System Call Results 55
9.3. ADDITIONAL RESOURCES 56
Installed Documentation 56
See Also 56

CHAPTER 10. LTRACE 57
10.1. INSTALLING LTRACE 57
10.2. USING LTRACE 57
10.2.1. Redirecting Output to a File 57
10.2.2. Tracing Selected Library Calls 58
10.2.3. Displaying Time Stamps 59
10.2.4. Displaying a Summary 59
10.3. ADDITIONAL RESOURCES 60
Installed Documentation 60
Online Documentation 60
See Also 60

CHAPTER 11. MEMSTOMP 61
11.1. INSTALLING MEMSTOMP 62
11.2. USING MEMSTOMP 62
11.3. ADDITIONAL RESOURCES 64
Installed Documentation 64
See Also 64

PART IV. PERFORMANCE MONITORING TOOLS 65

CHAPTER 12. SYSTEMTAP 66
12.1. INSTALLING SYSTEMTAP 66
12.2. USING SYSTEMTAP 67
12.3. ADDITIONAL RESOURCES 67
Installed Documentation 67
Online Documentation 67
See Also 68

CHAPTER 13. VALGRIND 69
13.1. INSTALLING VALGRIND 69
13.2. USING VALGRIND 70
13.3. ADDITIONAL RESOURCES 70
Installed Documentation 70
Online Documentation 70
See Also 71

CHAPTER 14. OPROFILE 72
14.1. INSTALLING OPROFILE 72
14.2. USING OPROFILE 72

14.3. ADDITIONAL RESOURCES 73
Installed Documentation 73
Online Documentation 73
See Also 74
CHAPTER 15. DYNINST 75
15.1. INSTALLING DYNINST 75
15.2. USING DYNINST 75
15.2.1. Using Dyninst with SystemTap 75
15.2.2. Using Dyninst as a Stand-alone Library 76
15.3. ADDITIONAL RESOURCES 80
Installed Documentation 80
Online Documentation 80
See Also 80
PART V. COMPILER TOOLSETS 82
CHAPTER 16. COMPILER TOOLSETS DOCUMENTATION 83
PART VI. GETTING HELP 84
CHAPTER 17. ACCESSING RED HAT PRODUCT DOCUMENTATION 85
Red Hat Developer Toolset 85
Red Hat Enterprise Linux 85
CHAPTER 18. CONTACTING GLOBAL SUPPORT SERVICES 86
18.1. GATHERING REQUIRED INFORMATION 86
Background Information 86
Diagnostics 86
Account and Contact Information 86
Issue Severity 87
18.2. ESCALATING AN ISSUE 87
18.3. RE-OPENING A SERVICE REQUEST 87
18.4. ADDITIONAL RESOURCES 88
Online Documentation 88
APPENDIX A. CHANGES IN VERSION 11.0 89
A.1. CHANGES IN GCC 89
General Improvements 89
Language-specific Improvements 89
Architecture-specific Improvements 91
A.2. CHANGES IN BINUTILS 91
The assembler 91
The linker 91
Other binary utilities 92
A.3. CHANGES IN ELFUTILS 92
A.4. CHANGES IN DWZ 93
A.5. CHANGES IN GDB 93
New features 93
New and improved commands 94
Python API 94
A.6. CHANGES IN LTRACE 94
A.7. CHANGES IN STRACE 94
Changes in Behavior 95
Improvements 95

Bug Fixes 96
A.8. CHANGES IN SYSTEMTAP 97
A.9. CHANGES IN VALGRIND 97
A.10. CHANGES IN DYNINST 97
A.11. CHANGES IN ANNOBIN 98
GCC plugin 98
Annocheck 98

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

PART I. INTRODUCTION

CHAPTER 1. RED HAT DEVELOPER TOOLSET

1.1. ABOUT RED HAT DEVELOPER TOOLSET

Red Hat Developer Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. It provides a complete set of development and performance analysis tools that can be installed and used on multiple versions of Red Hat Enterprise Linux. Executables built with the Red Hat Developer Toolset toolchain can then also be deployed and run on multiple versions of Red Hat Enterprise Linux. For detailed compatibility information, see Section 1.3, “Compatibility”.

Red Hat Developer Toolset does not replace the default system tools provided with Red Hat Enterprise Linux 7 when installed on those platforms. Instead, a parallel set of developer tools provides an alternative, newer version of those tools for optional use by developers. The default compiler and debugger, for example, remain those provided by the base Red Hat Enterprise Linux system.

What Is New in Red Hat Developer Toolset 11.0

Since Red Hat Developer Toolset 4.1, the Red Hat Developer Toolset content is also available in the ISO format together with the rest of Red Hat Software Collections content at https://access.redhat.com/downloads, specifically for Server and Workstation. Note that packages that require the *Optional* channel, which are discussed in Section 1.5.3, “Installing Optional Packages”, cannot be installed from the ISO image.

Table 1.1. Red Hat Developer Toolset Components

Name	Version	Description
GCC	11.2	A portable compiler suite with support for C, C++, and Fortran.
binutils	2.36	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
elfutils	0.185	A collection of binary tools and other utilities to inspect and manipulate ELF files.
dwz	0.14	A tool to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.
GDB	10.2	A command line debugger for programs written in C, C++, and Fortran.
ltrace	0.7.91	A debugging tool to display calls to dynamic libraries that a program makes. It can also monitor system calls executed by programs.
strace	5.13	A debugging tool to monitor system calls that a program uses and signals it receives.
memstomp	0.1.5	A debugging tool to identify calls to library functions with overlapping memory regions that are not allowed by various standards.
SystemTap	4.5	A tracing and probing tool to monitor the activities of the entire system without the need to instrument, recompile, install, and reboot.

Name	Version	Description
Valgrind	3.17.0	An instrumentation framework and a number of tools to profile applications in order to detect memory errors, identify memory management problems, and report any use of improper arguments in system calls.
OProfile	1.4.0	A system-wide profiler that uses the performance monitoring hardware on the processor to retrieve information about the kernel and executables on the system.
Dyninst	11.0.0	A library for instrumenting and working with user-space executables during their execution.
make	4.3	A dependency-tracking build automation tool.
annobin	9.82	A build security checking tool.

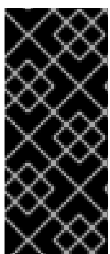
Red Hat Developer Toolset differs from "Technology Preview" compiler releases previously supplied in Red Hat Enterprise Linux in two important respects:

1. Red Hat Developer Toolset can be used on multiple major and minor releases of Red Hat Enterprise Linux, as detailed in Section 1.3, "Compatibility".

2. Unlike Technology Preview compilers and other tools shipped in earlier Red Hat Enterprise Linux, Red Hat Developer Toolset is fully supported under Red Hat Enterprise Linux Subscription Level Agreements, is functionally complete, and is intended for production use.

Important bug fixes and security errata are issued to Red Hat Developer Toolset subscribers in a similar manner to Red Hat Enterprise Linux for two years from the release of each major version release. A new major version of Red Hat Developer Toolset is released annually, providing significant updates for existing components and adding major new components. A single minor release, issued six months after each new major version release, provides a smaller update of bug fixes, security errata, and new minor components.

Additionally, the Red Hat Enterprise Linux Application Compatibility Specification also applies to Red Hat Developer Toolset (subject to some constraints on the use of newer C++11 language features, detailed in Section 2.2.4, "C++ Compatibility").



IMPORTANT

Applications and libraries provided by Red Hat Developer Toolset do not replace the Red Hat Enterprise Linux system versions, nor are they used in preference to the system versions. Using a framework called **Software Collections**, an additional set of developer tools is installed into the /**opt**/ directory and is explicitly enabled by the user on demand using the **scl** utility.

1.2. MAIN FEATURES

Red Hat Developer Toolset 11.0 brings the following changes:

- The Red Hat Developer Toolset version of the **GNU Compiler Collection** (**GCC**) has been upgraded to version 11.2 with many new features and bug fixes.

- The Red Hat Developer Toolset version of the **GNU Debugger** (**GDB**) has been upgraded to version 10.2 with many new features and bug fixes.

For a full list of changes and features introduced in this release, see Appendix A, *Changes in Version 11.0*.

1.3. COMPATIBILITY

Red Hat Developer Toolset 11.0 is available for Red Hat Enterprise Linux 7 for a number of architectures.

For ABI compatibility information, see Section 2.2.4, “C++ Compatibility”.

Table 1.2. Red Hat Developer Toolset 11.0 Compatibility

	Runs on Red Hat Enterprise Linux 7.7	Runs on Red Hat Enterprise Linux 7.9
Built with Red Hat Enterprise Linux 7.7	Supported	Supported
Built with Red Hat Enterprise Linux 7.9	Not Supported	Supported

Architecture support

Red Hat Developer Toolset is available on the following architectures:

- The 64-bit Intel and AMD architectures

- IBM Power Systems, big endian

- IBM Power Systems, little endian

- 64-bit IBM Z

1.4. GETTING ACCESS TO RED HAT DEVELOPER TOOLSET

Red Hat Developer Toolset is an offering distributed as a part of Red Hat Software Collections.

This content set is available to customers with Red Hat Enterprise Linux 7 subscriptions listed at https://access.redhat.com/solutions/472793.

Enable Red Hat Developer Toolset by using Red Hat Subscription Management. For information on how to register your system with this subscription management service, see the Red Hat Subscription Management collection of guides.

1.4.1. Using Red Hat Software Collections

Complete the following steps to attach a subscription that provides access to the repository for Red Hat Software Collections (which includes Red Hat Developer Toolset), and then enable that repository:

1. Determine the pool ID of a subscription that provides Red Hat Software Collections (and thus also Red Hat Developer Toolset). To do so, display a list of all subscriptions that are available for your system:

   ```
   # subscription-manager list --available
   ```

   For each available subscription, this command displays its name, unique identifier, expiration date, and other details related to your subscription. The pool ID is listed on a line beginning with **Pool ID**.

   For a complete list of subscriptions that provide access to Red Hat Developer Toolset, see https://access.redhat.com/solutions/472793.

2. Attach the appropriate subscription to your system:

   ```
   # subscription-manager attach --pool=pool_id
   ```

   Replace *pool_id* with the pool ID you determined in the previous step. To verify the list of subscriptions your system has currently attached, at any time:

   ```
   # subscription-manager list --consumed
   ```

3. Determine the exact name of the Red Hat Software Collections repository. Retrieve repository metadata and to display a list of available **Yum** repositories:

   ```
   # subscription-manager repos --list
   ```

   The repository names depend on the specific version of Red Hat Enterprise Linux you are using and are in the following format:

   ```
   rhel-variant-rhscl-version-rpms
   rhel-variant-rhscl-version-debug-rpms
   rhel-variant-rhscl-version-source-rpms
   ```

   In addition, certain packages, such as **devtoolset-11-gcc-plugin-devel**, depend on packages that are only available in the *Optional* channel. The repository names with these packages use the following format:

   ```
   rhel-version-variant-optional-rpms
   rhel-version-variant-optional-debug-rpms
   rhel-version-variant-optional-source-rpms
   ```

   For both the regular repositories and optional repositories, replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**), and *version* with the Red Hat Enterprise Linux system version (**7**).

4. Enable the repositories from step no. 3:

   ```
   # subscription-manager repos --enable repository
   ```

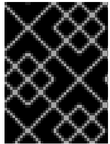
   Replace *repository* with the name of the repository to enable.

Once the subscription is attached to the system, you can install Red Hat Developer Toolset as described in Section 1.5, “Installing Red Hat Developer Toolset” . For more information on how to register your

system using Red Hat Subscription Management and associate it with subscriptions, see the Red Hat Subscription Management collection of guides.

1.5. INSTALLING RED HAT DEVELOPER TOOLSET

Red Hat Developer Toolset is distributed as a collection of RPM packages that can be installed, updated, uninstalled, and inspected by using the standard package management tools that are included in Red Hat Enterprise Linux. Note that a valid subscription that provides access to the Red Hat Software Collections content set is required in order to install Red Hat Developer Toolset on your system. For detailed instructions on how to associate your system with an appropriate subscription and get access to Red Hat Developer Toolset, see Section 1.4, "Getting Access to Red Hat Developer Toolset".



IMPORTANT

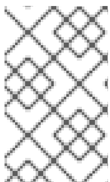
Before installing Red Hat Developer Toolset, install all available Red Hat Enterprise Linux updates.

1.5.1. Installing All Available Components

To install all components that are included in Red Hat Developer Toolset, install the **devtoolset-11** package:

```
# yum install devtoolset-11
```

This installs all development, debugging, and performance monitoring tools, and other dependent packages to the system. Alternatively, you can choose to install only a selected package group as described in Section 1.5.2, "Installing Individual Package Groups" .



NOTE

Note that since Red Hat Developer Toolset 3.0, the **scl-utils** package is not a part of Red Hat Developer Toolset, which is a change from preceding versions where the **scl** utility was installed along with the Red Hat Developer Toolset software collection.

1.5.2. Installing Individual Package Groups

To make it easier to install only certain components, such as the integrated development environment or the software development toolchain, Red Hat Developer Toolset is distributed with a number of meta packages that allow you to install selected package groups as described in Table 1.3, "Red Hat Developer Toolset Meta Packages".

Table 1.3. Red Hat Developer Toolset Meta Packages

Package Name	Description	Installed Components
devtoolset-11-perftools	Performance monitoring tools	SystemTap, Valgrind, OProfile, Dyninst
devtoolset-11-toolchain	Development and debugging tools	GCC, make, GDB, binutils, elfutils, dwz, memstomp, strace, ltrace

To install any of these meta packages:

```
# yum install package_name
```

Replace *package_name* with a space-separated list of meta packages you want to install. For example, to install only the development and debugging toolchain and packages that depend on it:

```
# yum install devtoolset-11-toolchain
```

Alternatively, you can choose to install all available components as described in Section 1.5.1, "Installing All Available Components".

1.5.3. Installing Optional Packages

Red Hat Developer Toolset is distributed with a number of optional packages that are not installed by default. To list all Red Hat Developer Toolset packages that are available to you but not installed on your system:

```
$ yum list available devtoolset-11-\*
```

To install any of these optional packages:

```
# yum install package_name
```

Replace *package_name* with a space-separated list of packages that you want to install. For example, to install the **devtoolset-11-gdb-gdbserver** and **devtoolset-11-gdb-doc** packages:

```
# yum install devtoolset-11-gdb-gdbserver devtoolset-11-gdb-doc
```

1.5.4. Installing Debugging Information

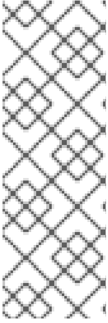
To install debugging information for any of the Red Hat Developer Toolset packages, make sure that the **yum-utils** package is installed and run:

```
# debuginfo-install package_name
```

For example, to install debugging information for the **devtoolset-11-dwz** package:

```
# debuginfo-install devtoolset-11-dwz
```

Note that in order to use this command, you need to have access to the repository with these packages. If your system is registered with Red Hat Subscription Management, enable the **rhel-*variant*-rhscl-*version*-debug-rpms** repository as described in Section 1.4, "Getting Access to Red Hat Developer Toolset". For more information on how to get access to debuginfo packages, see https://access.redhat.com/site/solutions/9907.



NOTE

The **devtoolset-11-*package_name*-debuginfo** packages can conflict with the corresponding packages from the base Red Hat Enterprise Linux system or from other versions of Red Hat Developer Toolset. This conflict also occurs in a multilib environment, where 64-bit debuginfo packages conflict with 32-bit debuginfo packages.

Manually uninstall the conflicting debuginfo packages prior to installing Red Hat Developer Toolset 11.0 and install only relevant debuginfo packages when necessary.

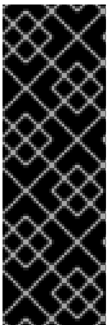
1.6. UPDATING RED HAT DEVELOPER TOOLSET

1.6.1. Updating to a Minor Version

When a new minor version of Red Hat Developer Toolset is available, update your Red Hat Enterprise Linux installation:

```
# yum update
```

This updates all packages on your Red Hat Enterprise Linux system, including the Red Hat Developer Toolset versions of development, debugging, and performance monitoring tools, and other dependent packages.



IMPORTANT

Use of Red Hat Developer Toolset requires the removal of any earlier pre-release versions of it. Additionally, it is not possible to update to Red Hat Developer Toolset 11.0 from a pre-release version of Red Hat Developer Toolset, including beta releases. If you have previously installed any pre-release version of Red Hat Developer Toolset, uninstall it from your system as described in Section 1.7, "Uninstalling Red Hat Developer Toolset" and install the new version as documented in Section 1.5, "Installing Red Hat Developer Toolset".

1.6.2. Updating to a Major Version

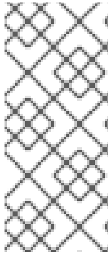
When a new major version of Red Hat Developer Toolset is available, you can install it in parallel with the previous version. For detailed instructions on how to install Red Hat Developer Toolset on your system, see Section 1.5, "Installing Red Hat Developer Toolset" .

1.7. UNINSTALLING RED HAT DEVELOPER TOOLSET

To uninstall Red Hat Developer Toolset packages from your system:

```
# yum remove devtoolset-11\* libasan libatomic libcilkrts libitm liblsan libtsan libubsan
```

This removes the **GNU Compiler Collection**, **GNU Debugger**, **binutils**, and other packages that are a part of Red Hat Developer Toolset from the system.



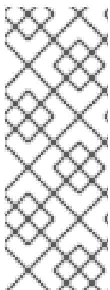
NOTE

Red Hat Developer Toolset 11.0 for Red Hat Enterprise Linux 7 no longer includes the **libatomic** and **libitm** libraries, which the above command attempts to remove, because they are not required for a proper function of Red Hat Developer Toolset components on that system. Nevertheless, the above command works as expected even on Red Hat Enterprise Linux 7.

Note that the uninstallation of the tools provided by Red Hat Developer Toolset does not affect the Red Hat Enterprise Linux system versions of these tools.

1.8. USING RED HAT DEVELOPER TOOLSET CONTAINER IMAGES

Docker-formatted container images can be used to run Red Hat Developer Toolset components inside virtual software containers, thus isolating them from the host system and allowing for their rapid deployment. For detailed description of the Red Hat Developer Toolset docker-formatted container images and Red Hat Developer Toolset *Dockerfiles*, see Using Red Hat Software Collections Container Images.



NOTE

The **docker** package, which contains the **Docker** daemon, command-line tool, and other necessary components for building and using docker-formatted container images, is currently available only for the Server variant of the Red Hat Enterprise Linux 7 product.

Follow the instructions outlined at Getting Docker in RHEL 7 to set up an environment for building and using docker-formatted container images.

1.9. ADDITIONAL RESOURCES

For more information about Red Hat Developer Toolset and Red Hat Enterprise Linux, see the resources listed below.

Online Documentation

- Red Hat Subscription Management collection of guides — The Red Hat Subscription Management collection of guides provides detailed information on how to manage subscriptions on Red Hat Enterprise Linux.
- Red Hat Developer Toolset 11.0 Release Notes — The *Release Notes* for Red Hat Developer Toolset 11.0 contain more information.
- Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guide* for Red Hat Enterprise Linux 7 provides more information on the **Eclipse** IDE, libraries and runtime support, compiling and building, debugging, and profiling on these systems.
- Red Hat Enterprise Linux 7 Installation Guide — The *Installation Guide* for Red Hat Enterprise Linux 7 explains how to obtain, install, and update the system.
- Red Hat Enterprise Linux 7 System Administrator's Guide — The *System Administrator's Guide* for Red Hat Enterprise Linux 7 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 7.
- Using Red Hat Software Collections Container Images — This book provides information on how to use container images based on Red Hat Software Collections. The available container images

include applications, daemons, databases, as well as the Red Hat Developer Toolset container images. The images can be run on Red Hat Enterprise Linux 7 Server and Red Hat Enterprise Linux Atomic Host.

- Getting Started with Containers — The guide contains a comprehensive overview of information about building and using container images on Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux Atomic Host.

See Also

- Appendix A, *Changes in Version 11.0* — A list of changes and improvements over the version of the Red Hat Developer Toolset tools in the previous version of Red Hat Developer Toolset.

PART II. DEVELOPMENT TOOLS

CHAPTER 2. GNU COMPILER COLLECTION (GCC)

The **GNU Compiler Collection**, commonly abbreviated **GCC**, is a portable compiler suite with support for a wide selection of programming languages.

Red Hat Developer Toolset is distributed with **GCC 11.2**. This version is more recent than the version included in Red Hat Enterprise Linux and provides a number of bug fixes and enhancements.

2.1. GNU C COMPILER

2.1.1. Installing the C Compiler

In Red Hat Developer Toolset, the GNU C compiler is provided by the **devtoolset-11-gcc** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, "Installing Red Hat Developer Toolset".

2.1.2. Using the C Compiler

To compile a C program on the command line, run the **gcc** compiler as follows:

```
$ scl enable devtoolset-11 'gcc -o output_file source_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line,:

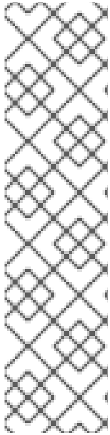
```
$ scl enable devtoolset-11 'gcc -o object_file -c source_file'
```

This creates an object file named *object_file*. If the **-o** option is omitted, the compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file:

```
$ scl enable devtoolset-11 'gcc -o output_file object_file...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **gcc** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **gcc** you are using at any point:

```
$ which gcc
```

Red Hat Developer Toolset's **gcc** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **gcc**:

```
$ gcc -v
```

Example 2.1. Compiling a C Program on the Command Line

Consider a source file named **hello.c** with the following contents:

```
#include <stdio.h>

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

Compile this source code on the command line by using the **gcc** compiler from Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'gcc -o hello hello.c'
```

This creates a new binary file called **hello** in the current working directory.

2.1.3. Running a C Program

When **gcc** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and run it:

```
$ ./file_name
```

Example 2.2. Running a C Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 2.1, "Compiling a C Program on the Command Line", you can run it by typing the following at a shell prompt:

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

2.2. GNU C++ COMPILER

2.2.1. Installing the C++ Compiler

In Red Hat Developer Toolset, the GNU C++ compiler is provided by the **devtoolset-11-gcc-c++** package and is automatically installed with the **devtoolset-11-toolchain** package as described in Section 1.5, "Installing Red Hat Developer Toolset".

2.2.2. Using the C++ Compiler

To compile a C++ program on the command line, run the **g++** compiler as follows:

```
$ scl enable devtoolset-11 'g++ -o output_file source_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the **g++** compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line:

```
$ scl enable devtoolset-11 'g++ -o object_file -c source_file'
```

This creates an object file named *object_file*. If the **-o** option is omitted, the **g++** compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file:

```
$ scl enable devtoolset-11 'g++ -o output_file object_file...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **g++** as default:

```
$ scl enable devtoolset-11 'bash'
```

NOTE

To verify the version of **g++** you are using at any point:

```
$ which g++
```

Red Hat Developer Toolset's **g++** executable path will begin with */opt*. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **g++**:

```
$ g++ -v
```

Example 2.3. Compiling a C++ Program on the Command Line

Consider a source file named **hello.cpp** with the following contents:

```
#include <iostream>
```

```
using namespace std;

int main(int argc, char *argv[]) {
  cout << "Hello, World!" << endl;
  return 0;
}
```

Compile this source code on the command line by using the **g++** compiler from Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'g++ -o hello hello.cpp'
```

This creates a new binary file called **hello** in the current working directory.

2.2.3. Running a C++ Program

When **g++** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and run it:

```
$ ./file_name
```

Example 2.4. Running a C++ Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 2.3, “Compiling a C++ Program on the Command Line”, you can run it:

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

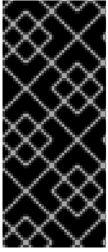
2.2.4. C++ Compatibility

All compilers from Red Hat Enterprise Linux versions 5, 6, and 7 and from Red Hat Developer Toolset versions 1 to 10 in any **-std** mode are compatible with any other of those compilers in C++98 mode.

A compiler in C++11, C++14, or C++17 mode is only guaranteed to be compatible with another compiler in those same modes if they are from the same release series.

Supported examples:

- C++11 and C++11 from Red Hat Developer Toolset 6.x
- C++14 and C++14 from Red Hat Developer Toolset 6.x
- C++17 and C++17 from Red Hat Developer Toolset 10.x



IMPORTANT

- The GCC compiler in Red Hat Developer Toolset 10.x can build code using C++20 but this capability is experimental and not supported by Red Hat.
- All compatibility information mentioned in this section is relevant only for Red Hat-supplied versions of the GCC C++ compiler.

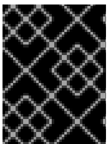
2.2.4.1. C++ ABI

Any C++98-compliant binaries or libraries built by the Red Hat Developer Toolset toolchain explicitly with **-std=c++98** or **-std=gnu++98** can be freely mixed with binaries and shared libraries built by the Red Hat Enterprise Linux 5, 6 or 7 system GCC.

The default language standard setting for Red Hat Developer Toolset 11.0 is C++17 with GNU extensions, equivalent to explicitly using option **-std=gnu++17**.

Using the C++14 language version is supported in Red Hat Developer Toolset when all C++ objects compiled with the respective flag have been built using Red Hat Developer Toolset 6 or later. Objects compiled by the system GCC in its default mode of C++98 are also compatible, but objects compiled with the system GCC in C++11 or C++14 mode are not compatible.

Starting with Red Hat Developer Toolset 10.x, using the C++17 language version is no longer experimental and is supported by Red Hat. All C++ objects compiled with C++17 must be built using Red Hat Developer Toolset 10.x or later.



IMPORTANT

Use of C++11, C++14, and C++17 features in your application requires careful consideration of the above ABI compatibility information.

The mixing of objects, binaries and libraries, built by the Red Hat Enterprise Linux 7 system toolchain GCC using the **-std=c++0x** or **-std=gnu++0x** flags, with those built with the C++11 or later language versions using the GCC in Red Hat Developer Toolset is explicitly not supported.

Aside from the C++11, C++14, and C++17 ABI, discussed above, the Red Hat Enterprise Linux Application Compatibility Specification is unchanged for Red Hat Developer Toolset. When mixing objects built with Red Hat Developer Toolset with those built with the Red Hat Enterprise Linux 7 toolchain (particularly **.o**/**.a** files), the Red Hat Developer Toolset toolchain should be used for any linkage. This ensures any newer library features provided only by Red Hat Developer Toolset are resolved at link-time.

A new standard mangling for SIMD vector types has been added to avoid name clashes on systems with vectors of varying lengths. The compiler in Red Hat Developer Toolset uses the new mangling by default. It is possible to use the previous standard mangling by adding the **-fabi-version=2** or **-fabi-version=3** options to GCC C++ compiler calls. To display a warning about code that uses the old mangling, use the **-Wabi** option.

On Red Hat Enterprise Linux 7, the GCC C++ compiler still uses the old mangling by default, but emits aliases with the new mangling on targets that support strong aliases. It is possible to use the new standard mangling by adding the **-fabi-version=4** option to compiler calls. To display a warning about code that uses the old mangling, use the **-Wabi** option.

On Red Hat Enterprise Linux 7, the GCC C++ compiler in Red Hat Developer Toolset still uses the old reference-counted implementation of **std::string**. This is done for compatibility with the Red Hat Enterprise Linux 7 system toolchain GCC. This means that some new C++17 features, such as

std::pmr::string, are not available on Red Hat Enterprise Linux 7, even when using the Red Hat Developer Toolset compiler.

2.3. GNU FORTRAN COMPILER

2.3.1. Installing the Fortran Compiler

In Red Hat Developer Toolset, the GNU Fortran compiler is provided by the **devtoolset-11-gcc-gfortran** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

2.3.2. Using the Fortran Compiler

To compile a Fortran program on the command line, run the **gfortran** compiler as follows:

```
$ scl enable devtoolset-11 'gfortran -o output_file source_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line:

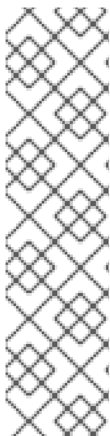
```
$ scl enable devtoolset-11 'gfortran -o object_file -c source_file'
```

This creates an object file named *object_file*. If the **-o** option is omitted, the compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file:

```
$ scl enable devtoolset-11 'gfortran -o output_file object_file...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **gfortran** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **gfortran** you are using at any point:

```
$ which gfortran
```

Red Hat Developer Toolset’s **gfortran** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **gfortran**:

```
$ gfortran -v
```

Example 2.5. Compiling a Fortran Program on the Command Line

Consider a source file named **hello.f** with the following contents:

```
program hello
  print *, "Hello, World!"
end program hello
```

Compile this source code on the command line by using the **gfortran** compiler from Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'gfortran -o hello hello.f'
```

This creates a new binary file called **hello** in the current working directory.

2.3.3. Running a Fortran Program

When **gfortran** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and run it:

```
$ ./file_name
```

Example 2.6. Running a Fortran Program on the Command Line

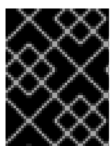
Assuming that you have successfully compiled the **hello** binary file as shown in Example 2.5, "Compiling a Fortran Program on the Command Line", you can run it:

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

2.4. SPECIFICS OF GCC IN RED HAT DEVELOPER TOOLSET

Static linking of libraries

Certain more recent library features are statically linked into applications built with Red Hat Developer Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk as 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 Red Hat Developer Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a

consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable devtoolset-11 'gcc -lsomelib objfile.o'
```

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

```
$ scl enable devtoolset-11 'gcc objfile.o -lsomelib'
```

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

2.5. ADDITIONAL RESOURCES

For more information about the GNU Compiler Collections and its features, see the resources listed below.

Installed Documentation

- *gcc*(1) — The manual page for the **gcc** compiler provides detailed information on its usage; with few exceptions, **g++** accepts the same command line options as **gcc**. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man gcc'
  ```

- *gfortran*(1) — The manual page for the **gfortran** compiler provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man gfortran'
  ```

- *C++ Standard Library Documentation* — Documentation on the *C++* standard library can be optionally installed:

  ```
  # yum install devtoolset-11-libstdc++-docs
  ```

  Once installed, HTML documentation is available at **/opt/rh/devtoolset-11/root/usr/share/doc/devtoolset-11-libstdC++-docs-11.2/html/index.html**.

Online Documentation

- Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guide* for Red Hat Enterprise Linux 7 provides in-depth information about **GCC**.

- Using the GNU Compiler Collection — The upstream GCC manual provides an in-depth description of the GNU compilers and their usage.

- The GNU C++ Library — The GNU C++ library documentation provides detailed information about the GNU implementation of the standard C++ library.

- The GNU Fortran Compiler — The GNU Fortran compiler documentation provides detailed information on **gfortran**'s usage.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 4, *binutils* — Instructions on using **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.
- Chapter 5, *elfutils* — Instructions on using **elfutils**, a collection of binary tools to inspect and manipulate **ELF** files.
- Chapter 6, *dwz* — Instructions on using the **dwz** tool to optimize DWARF debugging information contained in **ELF** shared libraries and **ELF** executables for size.
- Chapter 8, *GNU Debugger (GDB)* — Instructions on debugging programs written in C, C++, and Fortran.

CHAPTER 3. GNU 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.

Red Hat Developer Toolset is distributed with **make 4.3**. This version is more recent than the version included in Red Hat Enterprise Linux and provides a number of bug fixes and enhancements.

3.1. INSTALLING MAKE

In Red Hat Developer Toolset, **GNU make** is provided by the **devtoolset-11-make** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

3.2. USING MAKE

To build a program without using a Makefile, run the **make** tool as follows:

```
$ scl enable devtoolset-11 'make source_file_without_extension'
```

This command makes use of implicit rules that are defined for a number of programming languages, including C, C++, and Fortran. The result is a binary file named ***source_file_without_extension*** in the current working directory.

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **make** as default:

```
$ scl enable devtoolset-11 'bash'
```

> **NOTE**
>
> To verify the version of **make** you are using at any point:
>
> ```
> $ which make
> ```
>
> Red Hat Developer Toolset’s **make** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **make**:
>
> ```
> $ make -v
> ```

Example 3.1. Building a C Program Using make

Consider a source file named **hello.c** with the following contents:

```
#include <stdio.h>

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

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

Build this source code using the implicit rules defined by the **make** utility from Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'make hello'
cc     hello.c   -o hello
```

This creates a new binary file called **hello** in the current working directory.

3.3. USING MAKEFILES

To build complex programs that consist of a number of source files, **make** uses configuration files called *Makefiles* that control how to compile the components of a program and build the final executable. Makefiles can also contain instructions for cleaning the working directory, installing and uninstalling program files, and other operations.

make automatically uses files named **GNUmakefile**, **makefile**, or **Makefile** in the current directory. To specify another file name, use the **-f** option:

```
$ make -f make_file
```

Describing the details of Makefile syntax is beyond the scope of this guide. See GNU make, the upstream **GNU make** manual, which provides an in-depth description of the **GNU make** utility, Makefile syntax, and their usage.

The full **make** manual is also available in the Texinfo format as a part of your installation. To view this manual:

```
$ scl enable devtoolset-11 'info make'
```

Example 3.2. Building a C Program Using a Makefile

Consider the following universal Makefile named **Makefile** for building the simple C program introduced in Example 3.1, “Building a C Program Using make” . The Makefile defines some variables and specifies four *rules*, which consist of *targets* and their *recipes*. Note that the lines with recipes must start with the TAB character:

```
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)
```

To build the **hello.c** program using this Makefile, run the **make** utility:

```
$ scl enable devtoolset-11 'make'
gcc -c -Wall hello.c -o hello.o
gcc hello.o -o hello
```

This creates a new object file **hello.o** and a new binary file called **hello** in the current working directory.

To clean the working directory, run:

```
$ scl enable devtoolset-11 'make clean'
rm -rf hello.o hello
```

This removes the object and binary files from the working directory.

3.4. ADDITIONAL RESOURCES

For more information about the **GNU make** tool and its features, see the resources listed below.

Installed Documentation

- **make**(1) — The manual page for the **make** utility provides information on its usage. To display the manual page for the version included in Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'man make'
```

- The full **make** manual, which includes detailed information about Makefile syntax, is also available in the Texinfo format. To display the info manual for the version included in Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'info make'
```

Online Documentation

- GNU make — The upstream **GNU make** manual provides an in-depth description of the **GNU make** utility, Makefile syntax, and their usage.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 2, *GNU Compiler Collection (GCC)* — Instructions on using the **GNU Compiler Collection**, a portable compiler suite with support for a wide selection of programming languages.
- Chapter 4, *binutils* — Instructions on using **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.

- Chapter 5, *elfutils* – Instructions on using **elfutils**, a collection of binary tools to inspect and manipulate **ELF** files.
- Chapter 6, *dwz* – Instructions on using the **dwz** tool to optimize DWARF debugging information contained in **ELF** shared libraries and **ELF** executables for size.
- Chapter 8, *GNU Debugger (GDB)* – Instructions on debugging programs written in C, C++, and Fortran.

CHAPTER 4. BINUTILS

binutils is a collection of various binary tools, such as the **GNU linker**, **GNU assembler**, and other utilities that allow you to inspect and manipulate object files and binaries. See Table 4.1, “Tools Included in binutils for Red Hat Developer Toolset” for a complete list of binary tools that are distributed with the Red Hat Developer Toolset version of **binutils**.

Red Hat Developer Toolset is distributed with **binutils 2.36**. This version is more recent than the version included in Red Hat Enterprise Linux and the previous release of Red Hat Developer Toolset and provides bug fixes and enhancements.

Table 4.1. Tools Included in binutils for Red Hat Developer Toolset

Name	Description
addr2line	Translates addresses into file names and line numbers.
ar	Creates, modifies, and extracts files from archives.
as	The GNU assembler.
c++filt	Decodes mangled C++ symbols.
dwp	Combines DWARF object files into a single DWARF package file.
elfedit	Examines and edits ELF files.
gprof	Display profiling information.
ld	The GNU linker.
ld.bfd	An alternative to the GNU linker.
ld.gold	Another alternative to the GNU linker.
nm	Lists symbols from object files.
objcopy	Copies and translates object files.
objdump	Displays information from object files.
ranlib	Generates an index to the contents of an archive to make access to this archive faster.
readelf	Displays information about ELF files.
size	Lists section sizes of object or archive files.

Name	Description
strings	Displays printable character sequences in files.
strip	Discards all symbols from object files.

4.1. INSTALLING BINUTILS

In Red Hat Developer Toolset, **binutils** are provided by the **devtoolset-11-binutils** package and are automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

4.2. USING THE GNU ASSEMBLER

To produce an object file from an assembly language program, run the **as** tool as follows:

```
$ scl enable devtoolset-11 'as option ... -o object_file source_file'
```

This creates an object file named ***object_file*** in the current working directory.

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **as** as default:

```
$ scl enable devtoolset-11 'bash'
```

> **NOTE**
>
> To verify the version of **as** you are using at any point:
>
> ```
> $ which as
> ```
>
> Red Hat Developer Toolset's **as** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **as**:
>
> ```
> $ as -v
> ```

4.3. USING THE GNU LINKER

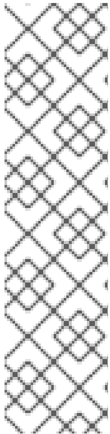
To create an executable binary file or a library from object files, run the **ld** tool as follows:

```
$ scl enable devtoolset-11 'ld option ... -o output_file object_file ...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **ld** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **ld** you are using at any point:

```
$ which ld
```

Red Hat Developer Toolset's **ld** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **ld**:

```
$ ld -v
```

4.4. USING OTHER BINARY TOOLS

The **binutils** provide many binary tools other than a linker and an assembler. For a complete list of these tools, see Table 4.1, "Tools Included in binutils for Red Hat Developer Toolset" .

To execute any of the tools that are a part of binutils:

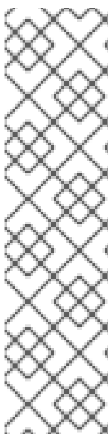
```
$ scl enable devtoolset-11 'tool option ... file_name'
```

See Table 4.1, "Tools Included in binutils for Red Hat Developer Toolset" for a list of tools that are distributed with **binutils**. For example, to use the **objdump** tool to inspect an object file:

```
$ scl enable devtoolset-11 'objdump option ... object_file'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset binary tools as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **binutils** you are using at any point:

```
$ which objdump
```

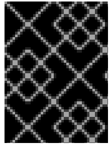
Red Hat Developer Toolset's **objdump** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **objdump**:

```
$ objdump -v
```

4.5. SPECIFICS OF BINUTILS IN RED HAT DEVELOPER TOOLSET

Static linking of libraries

Certain more recent library features are statically linked into applications built with Red Hat Developer Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk as 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 Red Hat Developer Toolset, libraries are linked using linker scripts which might specify some symbols through static archives. This is required to ensure compatibility with multiple versions of Red Hat Enterprise Linux. However, the linker scripts use the names of the respective shared object files. As a consequence, the linker uses different symbol handling rules than expected, and does not recognize symbols required by object files when the option adding the library is specified before options specifying the object files:

```
$ scl enable devtoolset-11 'ld -lsomelib objfile.o'
```

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

```
$ scl enable devtoolset-11 'ld objfile.o -lsomelib'
```

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

4.6. ADDITIONAL RESOURCES

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

Installed Documentation

- **as**(1), **ld**(1), **addr2line**(1), **ar**(1), **c++filt**(1), **dwp**(1), **elfedit**(1), **gprof**(1), **nm**(1), **objcopy**(1), **objdump**(1), **ranlib**(1), **readelf**(1), **size**(1), **strings**(1), **strip**(1), — Manual pages for various **binutils** tools provide more information about their respective usage. To display a manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man tool'
  ```

Online Documentation

- Documentation for binutils — The **binutils** documentation provides an in-depth description of the binary tools and their usage.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 5, *elfutils* — Information on how to use **elfutils**, a collection of binary tools to inspect and manipulate ELF files.
- Chapter 2, *GNU Compiler Collection (GCC)* — Information on how to compile programs written in C, C++, and Fortran.

CHAPTER 5. ELFUTILS

elfutils is a collection of various binary tools, such as **eu-objdump**, **eu-readelf**, and other utilities that allow you to inspect and manipulate **ELF** files. See Table 5.1, "Tools Included in elfutils for Red Hat Developer Toolset" for a complete list of binary tools that are distributed with the Red Hat Developer Toolset version of **elfutils**.

Red Hat Developer Toolset is distributed with **elfutils 0.185**. This version is more recent than the version included the previous release of Red Hat Developer Toolset and provides some bug fixes and enhancements.

Table 5.1. Tools Included in elfutils for Red Hat Developer Toolset

Name	Description
eu-addr2line	Translates addresses into file names and line numbers.
eu-ar	Creates, modifies, and extracts files from archives.
eu-elfcmp	Compares relevant parts of two **ELF** files for equality.
eu-elflint	Verifies that **ELF** files are compliant with the*generic ABI* (**gABI**) and *processor-specific supplement ABI* (**psABI**) specification.
eu-findtextrel	Locates the source of text relocations in files.
eu-make-debug-archive	Creates an offline archive for debugging.
eu-nm	Lists symbols from object files.
eu-objdump	Displays information from object files.
eu-ranlib	Generates an index to the contents of an archive to make access to this archive faster.
eu-readelf	Displays information about **ELF** files.
eu-size	Lists section sizes of object or archive files.
eu-stack	A new utility for unwinding processes and cores.
eu-strings	Displays printable character sequences in files.
eu-strip	Discards all symbols from object files.
eu-unstrip	Combines stripped files with separate symbols and debug information.

5.1. INSTALLING ELFUTILS

In Red Hat Developer Toolset, **elfutils** is provided by the **devtoolset-11-elfutils** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

5.2. USING ELFUTILS

To execute any of the tools that are part of **elfutils**, run the tool as follows:

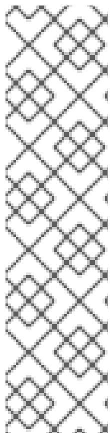
```
$ scl enable devtoolset-11 'tool option ... file_name'
```

See Table 5.1, “Tools Included in elfutils for Red Hat Developer Toolset” for a list of tools that are distributed with **elfutils**. For example, to use the **eu-objdump** tool to inspect an object file:

```
$ scl enable devtoolset-11 'eu-objdump option ... object_file'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset binary tools as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **elfutils** you are using at any point:

```
$ which eu-objdump
```

Red Hat Developer Toolset’s **eu-objdump** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **eu-objdump**:

```
$ eu-objdump -V
```

5.3. ADDITIONAL RESOURCES

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

See Also

- Chapter 1, *Red Hat Developer Toolset* – An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 2, *GNU Compiler Collection (GCC)* – Instructions on compiling programs written in C, C++, and Fortran.
- Chapter 4, *binutils* – Instructions on using **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.
- Chapter 6, *dwz* – Instructions on using the **dwz** tool to optimize DWARF debugging information contained in **ELF** shared libraries and **ELF** executables for size.

CHAPTER 6. DWZ

dwz is a command line tool that attempts to optimize DWARF debugging information contained in **ELF** shared libraries and **ELF** executables for size. To do so, **dwz** replaces DWARF information representation with equivalent smaller representation where possible and reduces the amount of duplication by using techniques from *Appendix E* of the *DWARF Standard*.

Red Hat Developer Toolset is distributed with **dwz 0.14**.

6.1. INSTALLING DWZ

In Red Hat Developer Toolset, the **dwz** utility is provided by the **devtoolset-11-dwz** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, "Installing Red Hat Developer Toolset".

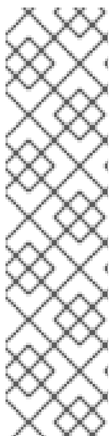
6.2. USING DWZ

To optimize DWARF debugging information in a binary file, run the **dwz** tool as follows:

```
$ scl enable devtoolset-11 'dwz option... file_name'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **dwz** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **dwz** you are using at any point:

```
$ which dwz
```

Red Hat Developer Toolset's **dwz** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **dwz**:

```
$ dwz -v
```

6.3. ADDITIONAL RESOURCES

For more information about **dwz** and its features, see the resources listed below.

Installed Documentation

- **dwz**(1) — The manual page for the **dwz** utility provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man dwz'
  ```

See Also

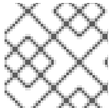
- Chapter 1, *Red Hat Developer Toolset* – An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 2, *GNU Compiler Collection (GCC)* – Instructions on compiling programs written in C, C++, and Fortran.
- Chapter 4, *binutils* – Instructions on using **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.
- Chapter 5, *elfutils* – Instructions on using **elfutils**, a collection of binary tools to inspect and manipulate **ELF** files.

CHAPTER 7. ANNOBIN

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.

7.1. INSTALLING ANNOBIN

In Red Hat Developer Toolset, the **annobin** plugin and the **annockeck** program are provided by the **devtoolset-11-gcc** package and are installed as described in Section 1.5.3, "Installing Optional Packages".

7.2. USING ANNOBIN PLUGIN

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

```
$ scl enable devtoolset-11 'gcc -fplugin=annobin -fplugin-arg-annobin-option file-name'
```

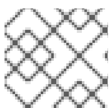
Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **as** as default:

```
$ scl enable devtoolset-11 'bash'
```

7.3. USING ANNOCHECK

To scan files, directories or RPM packages with the **annocheck** program:

```
$ scl enable devtoolset-11 'annocheck file-name'
```

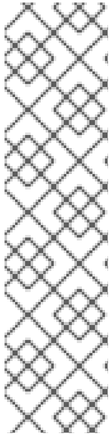


NOTE

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

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **as** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **annocheck** you are using at any point:

```
$ which annocheck
```

Red Hat Developer Toolset’s **annocheck** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **annocheck**:

```
$ annocheck --version
```

7.4. ADDITIONAL RESOURCES

For more information about **annocheck**, **annobin** and its features, see the resources listed below.

Installed Documentation

- **annocheck**(1) — The manual page for the **annocheck** utility provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'man annocheck'
```

- **annobin**(1) — The manual page for the **annobin** utility provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'man annobin'
```

PART III. DEBUGGING TOOLS

CHAPTER 8. GNU DEBUGGER (GDB)

The **GNU Debugger**, commonly abbreviated as **GDB**, is a command line tool that can be used to debug programs written in various programming languages. It allows you to inspect memory within the code being debugged, control the execution state of the code, detect the execution of particular sections of code, and much more.

Red Hat Developer Toolset is distributed with **GDB 10.2**. This version is more recent than the version included in Red Hat Enterprise Linux and the previous release of Red Hat Developer Toolset and provides some enhancements and numerous bug fixes.

8.1. INSTALLING THE GNU DEBUGGER

In Red Hat Developer Toolset, the **GNU Debugger** is provided by the **devtoolset-11-gdb** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

8.2. PREPARING A PROGRAM FOR DEBUGGING

Compiling Programs with Debugging Information

To compile a C program with debugging information that can be read by the **GNU Debugger**, make sure the **gcc** compiler is run with the **-g** option:

```
$ scl enable devtoolset-11 'gcc -g -o output_file input_file...'
```

Similarly, to compile a C++ program with debugging information:

```
$ scl enable devtoolset-11 'g++ -g -o output_file input_file...'
```

Example 8.1. Compiling a C Program With Debugging Information

Consider a source file named **fibonacci.c** that has the following contents:

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

int main (int argc, char *argv[]) {
  unsigned long int a = 0;
  unsigned long int b = 1;
  unsigned long int sum;

  while (b < LONG_MAX) {
    printf("%ld ", b);
    sum = a + b;
    a = b;
    b = sum;
  }

  return 0;
}
```

Compile this program on the command line using **GCC** from Red Hat Developer Toolset with debugging information for the **GNU Debugger**:

```
$ scl enable devtoolset-11 'gcc -g -o fibonacci fibonacci.c'
```

This creates a new binary file called **fibonacci** in the current working directory.

Installing Debugging Information for Existing Packages

To install debugging information for a package that is already installed on the system:

```
# debuginfo-install package_name
```

Note that the **yum-utils** package must be installed for the **debuginfo-install** utility to be available on your system.

Example 8.2. Installing Debugging Information for the glibc Package

Install debugging information for the **glibc** package:

```
# debuginfo-install glibc
Loaded plugins: product-id, refresh-packagekit, subscription-manager
--> Running transaction check
---> Package glibc-debuginfo.x86_64 0:2.17-105.el7 will be installed
...
```

8.3. RUNNING THE GNU DEBUGGER

To run the **GNU Debugger** on a program you want to debug:

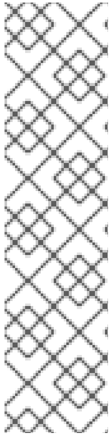
```
$ scl enable devtoolset-11 'gdb file_name'
```

This starts the **gdb** debugger in interactive mode and displays the default prompt, **(gdb)**. To quit the debugging session and return to the shell prompt, run the following command at any time:

```
(gdb) quit
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **gdb** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **gdb** you are using at any point:

```
$ which gdb
```

Red Hat Developer Toolset's **gdb** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **gdb**:

```
$ gdb -v
```

Example 8.3. Running the gdb Utility on the fibonacci Binary File

This example assumes that you have successfully compiled the **fibonacci** binary file as shown in Example 8.1, "Compiling a C Program With Debugging Information" .

Start debugging **fibonacci** with **gdb**:

```
$ scl enable devtoolset-11 'gdb fibonacci'
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-2.el7
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from fibonacci...done.
(gdb)
```

8.4. LISTING SOURCE CODE

To view the source code of the program you are debugging:

```
(gdb) list
```

Before you start the execution of the program you are debugging, **gdb** displays the first ten lines of the source code, and any subsequent use of this command lists another ten lines. Once you start the execution, **gdb** displays the lines that are surrounding the line on which the execution stops, typically when you set a breakpoint.

You can also display the code that is surrounding a particular line:

```
(gdb) list file_name:line_number
```

Similarly, to display the code that is surrounding the beginning of a particular function:

```
(gdb) list file_name:function_name
```

Note that you can change the number of lines the **list** command displays:

```
(gdb) set listsize number
```

Example 8.4. Listing the Source Code of the fibonacci Binary File

The **fibonacci.c** file listed in Example 8.1, “Compiling a C Program With Debugging Information” has exactly 17 lines. Assuming that you have compiled it with debugging information and you want the **gdb** utility to be capable of listing the entire source code, you can run the following command to change the number of listed lines to 20:

```
(gdb) set listsize 20
```

You can now display the entire source code of the file you are debugging by running the **list** command with no additional arguments:

```
(gdb) list
1       #include <stdio.h>
2       #include <limits.h>
3
4       int main (int argc, char *argv[]) {
5         unsigned long int a = 0;
6         unsigned long int b = 1;
7         unsigned long int sum;
8
9         while (b < LONG_MAX) {
10          printf("%ld ", b);
11          sum = a + b;
12          a = b;
13          b = sum;
14        }
15
16        return 0;
17      }
```

8.5. SETTING BREAKPOINTS

Setting a New Breakpoint

To set a new breakpoint at a certain line:

```
(gdb) break file_name:line_number
```

You can also set a breakpoint on a certain function:

```
(gdb) break file_name:function_name
```

Example 8.5. Setting a New Breakpoint

This example assumes that you have compiled the **fibonacci.c** file listed in Example 8.1, "Compiling a C Program With Debugging Information" with debugging information.

Set a new breakpoint at line 10:

```
(gdb) break 10
Breakpoint 1 at 0x4004e5: file fibonacci.c, line 10.
```

Listing Breakpoints

To display a list of currently set breakpoints:

```
(gdb) info breakpoints
```

Example 8.6. Listing Breakpoints

This example assumes that you have followed the instructions in Example 8.5, "Setting a New Breakpoint".

Display the list of currently set breakpoints:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004e5 in main at fibonacci.c:10
```

Deleting Existing Breakpoints

To delete a breakpoint that is set at a certain line:

```
(gdb) clear line_number
```

Similarly, to delete a breakpoint that is set on a certain function:

```
(gdb) clear function_name
```

Example 8.7. Deleting an Existing Breakpoint

This example assumes that you have compiled the **fibonacci.c** file listed in Example 8.1, "Compiling a C Program With Debugging Information" with debugging information.

Set a new breakpoint at line 7:

```
(gdb) break 7
Breakpoint 2 at 0x4004e3: file fibonacci.c, line 7.
```

Remove this breakpoint:

```
(gdb) clear 7
Deleted breakpoint 2
```

8.6. STARTING EXECUTION

To start an execution of the program you are debugging:

```
(gdb) run
```

If the program accepts any command line arguments, you can provide them as arguments to the **run** command:

```
(gdb) run argument…
```

The execution stops when the first breakpoint (if any) is reached, when an error occurs, or when the program terminates.

Example 8.8. Executing the fibonacci Binary File

This example assumes that you have followed the instructions in Example 8.5, “Setting a New Breakpoint”.

Execute the **fibonacci** binary file:

```
(gdb) run
Starting program: /home/john/fibonacci

Breakpoint 1, main (argc=1, argv=0x7fffffffe4d8) at fibonacci.c:10
10          printf("%ld ", b);
```

8.7. DISPLAYING CURRENT VALUES

The **gdb** utility allows you to display the value of almost anything that is relevant to the program, from a variable of any complexity to a valid expression or even a library function. However, the most common task is to display the value of a variable.

To display the current value of a certain variable:

```
(gdb) print variable_name
```

Example 8.9. Displaying the Current Values of Variables

This example assumes that you have followed the instructions in Example 8.8, “Executing the fibonacci Binary File” and the execution of the **fibonacci** binary stopped after reaching the breakpoint at line 10.

Display the current values of variables **a** and **b**:

```
(gdb) print a
$1 = 0
(gdb) print b
$2 = 1
```

8.8. CONTINUING EXECUTION

To resume the execution of the program you are debugging after it reached a breakpoint:

```
(gdb) continue
```

The execution stops again when another breakpoint is reached. To skip a certain number of breakpoints (typically when you are debugging a loop):

```
(gdb) continue number
```

The **gdb** utility also allows you to stop the execution after executing a single line of code:

```
(gdb) step
```

Finally, you can execute a certain number of lines:

```
(gdb) step number
```

Example 8.10. Continuing the Execution of the fibonacci Binary File

This example assumes that you have followed the instructions in Example 8.8, "Executing the fibonacci Binary File", and the execution of the **fibonacci** binary stopped after reaching the breakpoint at line 10.

Resume the execution:

```
(gdb) continue
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fffffffe4d8) at fibonacci.c:10
10          printf("%ld ", b);
```

The execution stops the next time the breakpoint is reached.

Execute the next three lines of code:

```
(gdb) step 3
13          b = sum;
```

This allows you to verify the current value of the **sum** variable before it is assigned to **b**:

```
(gdb) print sum
$3 = 2
```

8.9. ADDITIONAL RESOURCES

For more information about the **GNU Debugger** and all its features, see the resources listed below.

Installed Documentation

Installing the **devtoolset-11-gdb-doc** package provides the following documentation in HTML and PDF formats in the **/opt/rh/devtoolset-11/root/usr/share/doc/devtoolset-11-gdb-doc-10.2** directory:

- The *Debugging with GDB* book, which is a copy of the upstream material with the same name. The version of this document exactly corresponds to the version of **GDB** available in Red Hat Developer Toolset.
- The *GDB's Obsolete Annotations* document, which lists the obsolete **GDB** level 2 annotations.

Online Documentation

- Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guide* for Red Hat Enterprise Linux 7 provides more information on the **GNU Debugger** and debugging.
- GDB Documentation — The upstream **GDB** documentation includes the *GDB User Manual* and other reference material.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 2, *GNU Compiler Collection (GCC)* — Further information on how to compile programs written in C, C++, and Fortran.
- Chapter 9, *strace* — Instructions on using the **strace** utility to monitor system calls that a program uses and signals it receives.
- Chapter 11, *memstomp* — Instructions on using the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

CHAPTER 9. STRACE

strace is a diagnostic and debugging tool for the command line that can be used to trace system calls that are made and received by a running process. It records the name of each system call, its arguments, and its return value, as well as signals received by the process and other interactions with the kernel, and prints this record to standard error output or a selected file.

Red Hat Developer Toolset is distributed with **strace 5.13**.

9.1. INSTALLING STRACE

In Red Hat Enterprise Linux, the **strace** utility is provided by the **devtoolset-11-strace** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

9.2. USING STRACE

To run the **strace** utility on a program you want to analyze:

```
$ scl enable devtoolset-11 'strace program argument...'
```

Replace *program* with the name of the program you want to analyze, and *argument* with any command line options and arguments you want to supply to this program. Alternatively, you can run the utility on an already running process by using the **-p** command line option followed by the process ID:

```
$ scl enable devtoolset-11 'strace -p process_id'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **strace** as default:

```
$ scl enable devtoolset-11 'bash'
```

> **NOTE**
>
> To verify the version of **strace** you are using at any point:
>
> ```
> $ which strace
> ```
>
> Red Hat Developer Toolset’s **strace** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **strace**:
>
> ```
> $ strace -V
> ```

9.2.1. Redirecting Output to a File

By default, **strace** prints the name of each system call, its arguments and the return value to standard error output. To redirect this output to a file, use the **-o** command line option followed by the file name:

```
$ scl enable devtoolset-11 'strace -o file_name program argument...'
```

Replace *file_name* with the name of the file.

Example 9.1. Redirecting Output to a File

Consider a slightly modified version of the **fibonacci** file from Example 8.1, “Compiling a C Program With Debugging Information”. This executable file displays the Fibonacci sequence and optionally allows you to specify how many members of this sequence to list. Run the **strace** utility on this file and redirect the trace output to **fibonacci.log**:

```
$ scl enable devtoolset-11 'strace -o fibonacci.log ./fibonacci 20'
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

This creates a new plain-text file called **fibonacci.log** in the current working directory.

9.2.2. Tracing Selected System Calls

To trace only a selected set of system calls, run the **strace** utility with the **-e** command line option:

```
$ scl enable devtoolset-11 'strace -e expression program argument...'
```

Replace *expression* with a comma-separated list of system calls to trace or any of the keywords listed in Table 9.1, “Commonly Used Values of the -e Option” . For a detailed description of all available values, see the **strace**(1) manual page.

Table 9.1. Commonly Used Values of the -e Option

Value	Description
%file	System calls that accept a file name as an argument.
%process	System calls that are related to process management.
%network	System calls that are related to networking.
%signal	System calls that are related to signal management.
%ipc	System calls that are related to inter-process communication (**IPC**).
%desc	System calls that are related to file descriptors.

Note that the syntax **-e *expression*** is a shorthand for the full form **-e trace=*expression***.

Example 9.2. Tracing Selected System Calls

Consider the **employee** file from Example 11.1, “Using memstomp” . Run the **strace** utility on this executable file and trace only the **mmap** and **munmap** system calls:

```
$ scl enable devtoolset-11 'strace -e mmap,munmap ./employee'
```

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f896c744000
mmap(NULL, 61239, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f896c735000
mmap(0x3146a00000, 3745960, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3146a00000
mmap(0x3146d89000, 20480, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x189000) = 0x3146d89000
mmap(0x3146d8e000, 18600, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x3146d8e000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f896c734000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f896c733000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f896c732000
munmap(0x7f896c735000, 61239)          = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f896c743000
John,john@example.comDoe,
+++ exited with 0 +++
```

9.2.3. Displaying Time Stamps

To prefix each line of the trace with the exact time of the day in hours, minutes, and seconds, run the **strace** utility with the **-t** command line option:

```
$ scl enable devtoolset-11 'strace -t program argument...'
```

To also display milliseconds, supply the **-t** option twice:

```
$ scl enable devtoolset-11 'strace -tt program argument...'
```

To prefix each line of the trace with the time required to execute the respective system call, use the **-r** command line option:

```
$ scl enable devtoolset-11 'strace -r program argument...'
```

Example 9.3. Displaying Time Stamps

Consider an executable file named **pwd**. Run the **strace** utility on this file and include time stamps in the output:

```
$ scl enable devtoolset-11 'strace -tt pwd'
19:43:28.011815 execve("./pwd", ["./pwd"], [/* 36 vars */]) = 0
19:43:28.012128 brk(0)                  = 0xcd3000
19:43:28.012174 mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc869cb0000
19:43:28.012427 open("/etc/ld.so.cache", O_RDONLY) = 3
19:43:28.012446 fstat(3, {st_mode=S_IFREG|0644, st_size=61239, ...}) = 0
19:43:28.012464 mmap(NULL, 61239, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc869ca1000
```

```
19:43:28.012483 close(3)              = 0
...
19:43:28.013410 +++ exited with 0 +++
```

9.2.4. Displaying a Summary

To display a summary of how much time was required to execute each system call, how many times were these system calls executed, and how many errors were encountered during their execution, run the **strace** utility with the **-c** command line option:

```
$ scl enable devtoolset-11 'strace -c program argument...'
```

Example 9.4. Displaying a Summary

Consider an executable file named **lsblk**. Run the **strace** utility on this file and display a trace summary:

```
$ scl enable devtoolset-11 'strace -c lsblk > /dev/null'
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 80.88    0.000055           1       106        16 open
 19.12    0.000013           0       140           munmap
  0.00    0.000000           0       148           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0       258           close
  0.00    0.000000           0        37         2 stat
...
------ ----------- ----------- --------- --------- ----------------
100.00    0.000068                  1790        35 total
```

9.2.5. Tampering with System Call Results

Simulating errors returned from system calls can help identify missing error handling in programs.

To make a program receive a generic error as the result of a particular system call, run the **strace** utility with the **-e fault=** option and supply the system call:

```
$ scl enable devtoolset-11 'strace -e fault=syscall program argument...'
```

To specify the error type or return value, use the **-e inject=** option:

```
$ scl enable devtoolset-11 'strace -e inject=syscall:error=error-type program argument'
$ scl enable devtoolset-11 'strace -e inject=syscall:retval=return-value program argument'
```

Note that specifying the error type and return value is mutually exclusive.

Example 9.5. Tampering with System Call Results

Consider an executable file named **lsblk**. Run the **strace** utility on this file and make the **mmap()** system call return an error:

```
$ scl enable devtoolset-11 'strace -e fault=mmap:error=EPERM lsblk > /dev/null'
execve("/usr/bin/lsblk", ["lsblk"], 0x7fff1c0e02a0 /* 54 vars */) = 0
brk(NULL)                               = 0x55d9e8b43000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = -
1 EPERM (Operation not permitted) (INJECTED)
writev(2, [{iov_base="lsblk", iov_len=5}, {iov_base=": ", iov_len=2}, {iov_base="error while loading
shared libra"..., iov_len=36}, {iov_base=": ", iov_len=2}, {iov_base="", iov_len=0}, {iov_base="",
iov_len=0}, {iov_base="cannot create cache for search p"..., iov_len=35}, {iov_base=": ",
iov_len=2}, {iov_base="Cannot allocate memory", iov_len=22}, {iov_base="\n", iov_len=1}],
10lsblk: error while loading shared libraries: cannot create cache for search path: Cannot allocate
memory
) = 105
exit_group(127)                         = ?
+++ exited with 127 +++
```

9.3. ADDITIONAL RESOURCES

For more information about **strace** and its features, see the resources listed below.

Installed Documentation

- **strace**(1) — The manual page for the **strace** utility provides detailed information about its usage. To display the manual page for the version included in Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'man strace'
```

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 10, *ltrace* — Instructions on tracing program library calls using the **ltrace** tool.
- Chapter 8, *GNU Debugger (GDB)* — Instructions on debugging programs written in C, C++, and Fortran.
- Chapter 11, *memstomp* — Instructions on using the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

CHAPTER 10. LTRACE

ltrace is a diagnostic and debugging tool for the command line that can be used to display calls that are made to shared libraries. It uses the dynamic library hooking mechanism, which prevents it from tracing calls to statically linked libraries. **ltrace** also displays return values of the library calls. The output is printed to standard error output or to a selected file.

Red Hat Developer Toolset is distributed with **ltrace 0.7.91**. While the base version **ltrace** remains the same as in the previous release of Red Hat Developer Toolset, various enhancements and bug fixes have ported.

10.1. INSTALLING LTRACE

In Red Hat Enterprise Linux, the **ltrace** utility is provided by the **devtoolset-11-ltrace** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, "Installing Red Hat Developer Toolset".

10.2. USING LTRACE

To run the **ltrace** utility on a program you want to analyze:

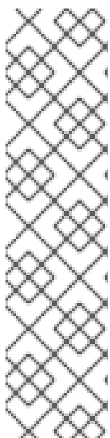
```
$ scl enable devtoolset-11 'ltrace program argument...'
```

Replace *program* with the name of the program you want to analyze, and *argument* with any command line options and arguments you want to supply to this program. Alternatively, you can run the utility on an already running process by using the **-p** command line option followed by the process ID:

```
$ scl enable devtoolset-11 'ltrace -p process_id'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **ltrace** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **ltrace** you are using at any point:

```
$ which ltrace
```

Red Hat Developer Toolset's **ltrace** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **ltrace**:

```
$ ltrace -V
```

10.2.1. Redirecting Output to a File

By default, **ltrace** prints the name of each system call, its arguments and the return value to standard error output. To redirect this output to a file, use the **-o** command line option followed by the file name:

```
$ scl enable devtoolset-11 'ltrace -o file_name program argument...'
```

Replace *file_name* with the name of the file.

Example 10.1. Redirecting Output to a File

Consider a slightly modified version of the **fibonacci** file from Example 8.1, "Compiling a C Program With Debugging Information". This executable file displays the Fibonacci sequence and optionally allows you to specify how many members of this sequence to list. Run the **ltrace** utility on this file and redirect the trace output to **fibonacci.log**:

```
$ scl enable devtoolset-11 'ltrace -o fibonacci.log ./fibonacci 20'
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

This creates a new plain-text file called **fibonacci.log** in the current working directory.

10.2.2. Tracing Selected Library Calls

To trace only a selected set of library calls, run the **ltrace** utility with the **-e** command line option:

```
$ scl enable devtoolset-11 'ltrace -e expression program argument...'
```

Replace *expression* with a chain of rules to specify the library calls to trace. The rules can consist of patterns that identify symbol names (such as **malloc** or **free**) and patterns that identify library SONAMEs (such as **libc.so**). For example, to trace call to the **malloc** and **free** function but to omit those that are done by the **libc** library:

```
$ scl enable devtoolset-11 'ltrace -e malloc+free-@libc.so* program'
```

Example 10.2. Tracing Selected Library Calls

Consider the **ls** command. Run the **ltrace** utility on this program and trace only the **opendir**, **readdir**, and **closedir** function calls:

```
$ scl enable devtoolset-11 'ltrace -e opendir+readdir+closedir ls'
ls->opendir(".")      = { 3 }
ls->readdir({ 3 })    = { 61533, "." }
ls->readdir({ 3 })    = { 131, ".." }
ls->readdir({ 3 })    = { 67185100, "BUILDROOT" }
ls->readdir({ 3 })    = { 202390772, "SOURCES" }
ls->readdir({ 3 })    = { 60249, "SPECS" }
ls->readdir({ 3 })    = { 67130110, "BUILD" }
ls->readdir({ 3 })    = { 136599168, "RPMS" }
ls->readdir({ 3 })    = { 202383274, "SRPMS" }
ls->readdir({ 3 })    = nil
ls->closedir({ 3 })   = 0
BUILD  BUILDROOT  RPMS  SOURCES  SPECS  SRPMS
+++ exited (status 0) +++
```

For a detailed description of available filter expressions, see the **ltrace**(1) manual page.

10.2.3. Displaying Time Stamps

To prefix each line of the trace with the exact time of the day in hours, minutes, and seconds, run the **ltrace** utility with the **-t** command line option:

```
$ scl enable devtoolset-11 'ltrace -t program argument...'
```

To also display milliseconds, supply the **-t** option twice:

```
$ scl enable devtoolset-11 'ltrace -tt program argument...'
```

To prefix each line of the trace with the time required to execute the respective system call, use the **-r** command line option:

```
$ scl enable devtoolset-11 'ltrace -r program argument...'
```

Example 10.3. Displaying Time Stamps

Consider the **pwd** command. Run the **ltrace** utility on this program and include time stamps in the output:

```
$ scl enable devtoolset-11 'ltrace -tt pwd'
13:27:19.631371 __libc_start_main([ "pwd" ] <unfinished ...>
13:27:19.632240 getenv("POSIXLY_CORRECT")                     = nil
13:27:19.632520 strrchr("pwd", '/')                           = nil
13:27:19.632786 setlocale(LC_ALL, "")                         = "en_US.UTF-8"
13:27:19.633220 bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
13:27:19.633471 textdomain("coreutils")                       = "coreutils"
(...)
13:27:19.637110  exited (status 0)
```

10.2.4. Displaying a Summary

To display a summary of how much time was required to execute each system call and how many times were these system calls executed, run the **ltrace** utility with the **-c** command line option:

```
$ scl enable devtoolset-11 'ltrace -c program argument...'
```

Example 10.4. Displaying a Summary

Consider the **lsblk** command. Run the **ltrace** utility on this program and display a trace summary:

```
$ scl enable devtoolset-11 'ltrace -c lsblk > /dev/null'
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 53.60    0.261644      261644         1 __libc_start_main
  4.48    0.021848          58       374 mbrtowc
  4.41    0.021524          57       374 wcwidth
  4.39    0.021409          57       374 __ctype_get_mb_cur_max
  4.38    0.021359          57       374 iswprint
  4.06    0.019838          74       266 readdir64
  3.21    0.015652          69       224 strlen
```

```
...
------ ----------- ----------- --------- --------------------
100.00    0.488135                3482 total
```

10.3. ADDITIONAL RESOURCES

For more information about **ltrace** and its features, see the resources listed below.

Installed Documentation

- **ltrace**(1) — The manual page for the **ltrace** utility provides detailed information about its usage. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man ltrace'
  ```

Online Documentation

- ltrace for RHEL 6 and 7 — This article on the Red Hat Developer Blog offers additional in-depth information (including practical examples) on how to use **ltrace** for application debugging.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 9, *strace* — Instructions on tracing program system calls using the **strace** tool.
- Chapter 8, *GNU Debugger (GDB)* — Instructions on debugging programs written in C, C++, and Fortran.
- Chapter 11, *memstomp* — Instructions on using the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

CHAPTER 11. MEMSTOMP

memstomp is a command line tool that can be used to identify function calls with overlapping memory regions in situations when such an overlap is not permitted by various standards. It intercepts calls to the library functions listed in Table 11.1, “Function Calls Inspected by memstomp” and for each memory overlap, it displays a detailed backtrace to help you debug the problem.

Similarly to **Valgrind**, the **memstomp** utility inspects applications without the need to recompile them. However, it is much faster than this tool and therefore serves as a convenient alternative to it.

Red Hat Developer Toolset is distributed with **memstomp 0.1.5**.

Table 11.1. Function Calls Inspected by memstomp

Function	Description
memcpy	Copies *n* bytes from one memory area to another and returns a pointer to the second memory area.
memccpy	Copies a maximum of *n* bytes from one memory area to another and stops when a certain character is found. It either returns a pointer to the byte following the last written byte, or NULL if the given character is not found.
mempcpy	Copies *n* bytes from one memory area to another and returns a pointer to the byte following the last written byte.
strcpy	Copies a string from one memory area to another and returns a pointer to the second string.
stpcpy	Copies a string from one memory area to another and returns a pointer to the terminating null byte of the second string.
strncpy	Copies a maximum of *n* characters from one string to another and returns a pointer to the second string.
stpncpy	Copies a maximum of *n* characters from one string to another. It either returns a pointer to the terminating null byte of the second string, or if the string is not null-terminated, a pointer to the byte following the last written byte.
strcat	Appends one string to another while overwriting the terminating null byte of the second string and adding a new one at its end. It returns a pointer to the new string.
strncat	Appends a maximum of *n* characters from one string to another while overwriting the terminating null byte of the second string and adding a new one at its end. It returns a pointer to the new string.
wmemcpy	The wide-character equivalent of the **memcpy**() function that copies*n* wide characters from one array to another and returns a pointer to the second array.

Function	Description
wmempcpy	The wide-character equivalent of the **mempcpy**() function that copies *n* wide characters from one array to another and returns a pointer to the byte following the last written wide character.
wcscpy	The wide-character equivalent of the **strcpy**() function that copies a wide-character string from one array to another and returns a pointer to the second array.
wcsncpy	The wide-character equivalent of the **strncpy**() function that copies a maximum of *n* wide characters from one array to another and returns a pointer to the second string.
wcscat	The wide-character equivalent of the **strcat**() function that appends one wide-character string to another while overwriting the terminating null byte of the second string and adding a new one at its end. It returns a pointer to the new string.
wcsncat	The wide-character equivalent of the **strncat**() function that appends a maximum of *n* wide characters from one array to another while overwriting the terminating null byte of the second wide-character string and adding a new one at its end. It returns a pointer to the new string.

11.1. INSTALLING MEMSTOMP

In Red Hat Developer Toolset, the **memstomp** utility is provided by the **devtoolset-11-memstomp** package and is automatically installed with **devtoolset-11-toolchain** as described in Section 1.5, "Installing Red Hat Developer Toolset".

11.2. USING MEMSTOMP

To run the **memstomp** utility on a program you want to analyze:

```
$ scl enable devtoolset-11 'memstomp program argument...'
```

To immediately terminate the analyzed program when a problem is detected, run the utility with the **--kill** (or **-k** for short) command line option:

```
$ scl enable devtoolset-11 'memstomp --kill program argument...'
```

The use of the **--kill** option is especially recommended if you are analyzing a multi-threaded program; the internal implementation of backtraces is not thread-safe and running the **memstomp** utility on a multi-threaded program without this command line option can therefore produce unreliable results.

Additionally, if you have compiled the analyzed program with the debugging information or this debugging information is available to you, you can use the **--debug-info** (or **-d**) command line option to produce a more detailed backtrace:

```
$ scl enable devtoolset-11 'memstomp --debug-info program argument...'
```

For detailed instructions on how to compile your program with the debugging information built in the binary file, see Section 8.2, "Preparing a Program for Debugging" . For information on how to install debugging information for any of the Red Hat Developer Toolset packages, see Section 1.5.4, "Installing

Debugging Information".

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **memstomp** as default:

```
$ scl enable devtoolset-11 'bash'
```

Example 11.1. Using memstomp

In the current working directory, create a source file named **employee.c** with the following contents:

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

#define BUFSIZE 80

int main(int argc, char *argv[]) {
  char employee[BUFSIZE] = "John,Doe,john@example.com";
  char name[BUFSIZE] = {0};
  char surname[BUFSIZE] = {0};
  char *email;
  size_t length;

  /* Extract the information: */
  memccpy(name, employee, ',', BUFSIZE);
  length = strlen(name);
  memccpy(surname, employee + length, ',', BUFSIZE);
  length += strlen(surname);
  email = employee + length;

  /* Compose the new entry: */
  strcat(employee, surname);
  strcpy(employee, name);
  strcat(employee, email);

  /* Print the result: */
  puts(employee);

  return 0;
}
```

Compile this program into a binary file named **employee**:

```
$ scl enable devtoolset-11 'gcc -rdynamic -g -o employee employee.c'
```

To identify erroneous function calls with overlapping memory regions:

```
$ scl enable devtoolset-11 'memstomp --debug-info ./employee'
memstomp: 0.1.4 successfully initialized for process employee (pid 14887).

strcat(dest=0x7fff13afc265, src=0x7fff13afc269, bytes=21) overlap for employee(14887)
        ??:0    strcpy()
        ??:0    strcpy()
```

```
        ??:0    _Exit()
        ??:0    strcat()
        employee.c:26   main()
        ??:0    __libc_start_main()
        ??:0    _start()
John,john@example.comDoe,
```

11.3. ADDITIONAL RESOURCES

For more information about **memstomp** and its features, see the resources listed below.

Installed Documentation

- **memstomp**(1) — The manual page for the **memstomp** utility provides detailed information about its usage. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man memstomp'
  ```

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 8, *GNU Debugger (GDB)* — Instructions on debugging programs written in C, C++, and Fortran.
- Chapter 9, *strace* — Instructions on using the strace utility to monitor system calls that a program uses and signals it receives.
- Chapter 13, *Valgrind* — Instructions on using the **Valgrind** tool to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.

PART IV. PERFORMANCE MONITORING TOOLS

CHAPTER 12. SYSTEMTAP

SystemTap is a tracing and probing tool that allows users to monitor the activities of the entire system without needing to instrument, recompile, install, and reboot. It is programmable with a custom scripting language, which gives it expressiveness (to trace, filter, and analyze) and reach (to look into the running kernel and applications).

SystemTap can monitor various types of events, such as function calls within the kernel or applications, timers, tracepoints, performance counters, and so on. Some included example scripts produce output similar to **netstat**, **ps**, **top**, and **iostat**, others include pretty-printed function callgraph traces or tools for working around security bugs.

Red Hat Developer Toolset is distributed with **SystemTap 4.5**. This version is more recent than the version included in the previous release of Red Hat Developer Toolset and provides numerous bug fixes and enhancements.

Table 12.1. Tools Distributed with SystemTap for Red Hat Developer Toolset

Name	Description
stap	Translates probing instructions into C code, builds a kernel module, and loads it into a running Linux kernel.
stapdyn	The **Dyninst** backend for **SystemTap**.
staprun	Loads, unloads, attaches to, and detaches from kernel modules built with the **stap** utility.
stapsh	Serves as a remote shell for **SystemTap**.
stap-prep	Determines and—if possible—downloads the kernel information packages that are required to run **SystemTap**.
stap-merge	Merges per-CPU files. This script is automatically executed when the **stap** utility is executed with the **-b** command line option.
stap-report	Gathers important information about the system for the purpose of reporting a bug in **SystemTap**.
stap-server	A compile server, which listens for requests from **stap** clients.

12.1. INSTALLING SYSTEMTAP

In Red Hat Developer Toolset, **SystemTap** is provided by the **devtoolset-11-systemtap** package and is automatically installed with **devtoolset-11-perftools** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

In order to place instrumentation into the Linux kernel, **SystemTap** may also require installation of additional packages with debugging information. To determine which packages to install, run the **stap-prep** utility as follows:

```
$ scl enable devtoolset-11 'stap-prep'
```

Note that if you execute this command as the **root** user, the utility automatically offers the packages for installation. For more information on how to install these packages on your system, see the *Red Hat Enterprise Linux 7 SystemTap Beginners Guide*.

12.2. USING SYSTEMTAP

To execute any of the tools that are part of **SystemTap**:

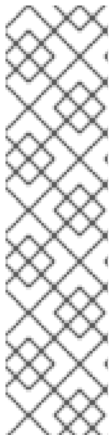
```
$ scl enable devtoolset-11 'tool option...'
```

See Table 12.1, "Tools Distributed with SystemTap for Red Hat Developer Toolset" for a list of tools that are distributed with **SystemTap**. For example, to run the **stap** tool to build an instrumentation module:

```
$ scl enable devtoolset-11 'stap option... argument...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **SystemTap** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **SystemTap** you are using at any point:

```
$ which stap
```

Red Hat Developer Toolset's **stap** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **SystemTap**:

```
$ stap -V
```

12.3. ADDITIONAL RESOURCES

For more information about **SystemTap** and its features, see the resources listed below.

Installed Documentation

- **stap**(1) — The manual page for the **stap** command provides detailed information on its usage, as well as references to other related manual pages. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man stap'
  ```

- **staprun**(8) — The manual page for the **staprun** command provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man staprun'
  ```

Online Documentation

- Red Hat Enterprise Linux 7 SystemTap Beginners Guide — The *SystemTap Beginners Guide* for Red Hat Enterprise Linux 7 provides an introduction to **SystemTap** and its usage.
- Red Hat Enterprise Linux 7 SystemTap Tapset Reference — The *SystemTap Tapset Reference* for Red Hat Enterprise Linux 7 provides further details about **SystemTap**.
- The SystemTap Documentation — The **SystemTap** documentation provides further documentation about **SystemTap**, and numerous examples of the **SystemTap** scripts.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 13, *Valgrind* — Instructions on using the **Valgrind** tool to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.
- Chapter 14, *OProfile* — Instructions on using the **OProfile** tool to determine which sections of code consume the greatest amount of **CPU** time and why.
- Chapter 15, *Dyninst* — Instructions on using the **Dyninst** library to instrument a user-space executable.

CHAPTER 13. VALGRIND

Valgrind is an instrumentation framework that ships with a number of tools for profiling applications. It can be used to detect various memory errors and memory-management problems, such as the use of uninitialized memory or an improper allocation and freeing of memory, or to identify the use of improper arguments in system calls. For a complete list of profiling tools that are distributed with the Red Hat Developer Toolset version of **Valgrind**, see Table 13.1, “Tools Distributed with Valgrind for Red Hat Developer Toolset”.

Valgrind profiles an application by rewriting it and instrumenting the rewritten binary. This allows you to profile your application without the need to recompile it, but it also makes **Valgrind** significantly slower than other profilers, especially when performing extremely detailed runs. It is therefore not suited to debugging time-specific issues, or kernel-space debugging.

Red Hat Developer Toolset is distributed with **Valgrind 3.17.0**. This version is more recent than the version included in the previous release of Red Hat Developer Toolset and provides numerous bug fixes and enhancements.

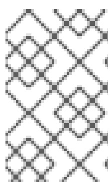
Table 13.1. Tools Distributed with Valgrind for Red Hat Developer Toolset

Name	Description
Memcheck	Detects memory management problems by intercepting system calls and checking all read and write operations.
Cachegrind	Identifies the sources of cache misses by simulating the level 1 instruction cache (**I1**), level 1 data cache (**D1**), and unified level 2 cache (**L2**).
Callgrind	Generates a call graph representing the function call history.
Helgrind	Detects synchronization errors in multithreaded C, C++, and Fortran programs that use **POSIX** threading primitives.
DRD	Detects errors in multithreaded C and C++ programs that use **POSIX** threading primitives or any other threading concepts that are built on top of these **POSIX** threading primitives.
Massif	Monitors heap and stack usage.

13.1. INSTALLING VALGRIND

In Red Hat Developer Toolset, **Valgrind** is provided by the **devtoolset-11-valgrind** package and is automatically installed with **devtoolset-11-perftools**.

For detailed instructions on how to install Red Hat Developer Toolset and related packages to your system, see Section 1.5, “Installing Red Hat Developer Toolset”.



NOTE

Note that if you use **Valgrind** in combination with the **GNU Debugger**, it is recommended that you use the version of **GDB** that is included in Red Hat Developer Toolset to ensure that all features are fully supported.

13.2. USING VALGRIND

To run any of the **Valgrind** tools on a program you want to profile:

```
$ scl enable devtoolset-11 'valgrind --tool=tool program argument...'
```

See Table 13.1, "Tools Distributed with Valgrind for Red Hat Developer Toolset" for a list of tools that are distributed with **Valgrind**. The argument of the **--tool** command line option must be specified in lower case, and if this option is omitted, **Valgrind** uses **Memcheck** by default. For example, to run **Cachegrind** on a program to identify the sources of cache misses:

```
$ scl enable devtoolset-11 'valgrind --tool=cachegrind program argument...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **Valgrind** as default:

```
$ scl enable devtoolset-11 'bash'
```

NOTE

To verify the version of Valgrind you are using at any point:

```
$ which valgrind
```

Red Hat Developer Toolset's **valgrind** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **Valgrind**:

```
$ valgrind --version
```

13.3. ADDITIONAL RESOURCES

For more information about **Valgrind** and its features, see the resources listed below.

Installed Documentation

- **valgrind**(1) — The manual page for the **valgrind** utility provides detailed information on how to use Valgrind. To display the manual page for the version included in Red Hat Developer Toolset:

  ```
  $ scl enable devtoolset-11 'man valgrind'
  ```

- *Valgrind Documentation* — **HTML** documentation for **Valgrind** is located at **/opt/rh/devtoolset-11/root/usr/share/doc/devtoolset-11-valgrind-3.17.0/html/index.html**.

Online Documentation

- Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guide* for Red Hat Enterprise Linux 7 provides more information about **Valgrind** and its **Eclipse** plug-in.
- Red Hat Enterprise Linux 7 Performance Tuning Guide — The *Performance Tuning Guide* for Red Hat Enterprise Linux 7 provide more detailed information about using **Valgrind** to profile applications.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 11, *memstomp* — Instructions on using the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.
- Chapter 12, *SystemTap* — An introduction to the **SystemTap** tool and instructions on how to use it to monitor the activities of a running system.
- Chapter 14, *OProfile* — Instructions on using the **OProfile** tool to determine which sections of code consume the greatest amount of CPU time and why.
- Chapter 15, *Dyninst* — Instructions on using the **Dyninst** library to instrument a user-space executable.

CHAPTER 14. OPROFILE

OProfile is a low overhead, system-wide profiler that uses the performance-monitoring hardware on the processor to retrieve information about the kernel and executables on the system, such as when memory is referenced, the number of level 2 cache (**L2**) requests, and the number of hardware interrupts received. It consists of a configuration utility, a daemon for collecting data, and a number of tools that can be used to transform the data into a human-readable form. For a complete list of tools that are distributed with the Red Hat Developer Toolset version of **OProfile**, see Table 14.1, “Tools Distributed with OProfile for Red Hat Developer Toolset”.

OProfile profiles an application without adding any instrumentation by recording the details of every nth event. This allows it to consume fewer resources than **Valgrind**, but it also causes its samples to be less precise. Unlike **Valgrind**, which only collects data for a single process and its children in user-space, **OProfile** is well suited to collect system-wide data on both user-space and kernel-space processes, and requires **root** privileges to run.

Red Hat Developer Toolset is distributed with **OProfile 1.4.0**.

Table 14.1. Tools Distributed with OProfile for Red Hat Developer Toolset

Name	Description
operf	Records samples either for a single process or system-wide using the Linux Performance Events subsystem.
opannotate	Generates an annotated source file or assembly listing from the profiling data.
oparchive	Generates a directory containing executable, debug, and sample files.
opgprof	Generates a summary of a profiling session in a format compatible with **gprof**.
ophelp	Displays a list of available events.
opimport	Converts a sample database file from a foreign binary format to the native format.
opjitconv	Converts a just-in-time (**JIT**) dump file to the Executable and Linkable Format (**ELF**).
opreport	Generates image and symbol summaries of a profiling session.
ocount	A new tool for counting the number of times particular events occur during the duration of a monitored command.

14.1. INSTALLING OPROFILE

In Red Hat Developer Toolset, **OProfile** is provided by the **devtoolset-11-oprofile** package and is automatically installed with **devtoolset-11-perftools** as described in Section 1.5, “Installing Red Hat Developer Toolset”.

14.2. USING OPROFILE

To run any of the tools that are distributed with **OProfile**:

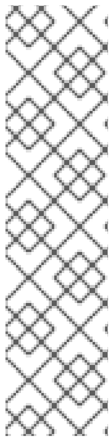
```
# scl enable devtoolset-11 'tool option...'
```

See Table 14.1, “Tools Distributed with OProfile for Red Hat Developer Toolset” for a list of tools that are distributed with **OProfile**. For example, to use the **ophelp** command to list available events in the **XML** format:

```
$ scl enable devtoolset-11 'ophelp -X'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **OProfile** as default:

```
$ scl enable devtoolset-11 'bash'
```



NOTE

To verify the version of **OProfile** you are using at any point:

```
$ which operf
```

Red Hat Developer Toolset’s **operf** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **OProfile**:

```
# operf --version
```

14.3. ADDITIONAL RESOURCES

For more information about **OProfile** and its features, see the resources listed below.

Installed Documentation

- **oprofile**(1) — The manual page named *oprofile* provides an overview of **OProfile** and available tools. To display the manual page for the version included in Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'man oprofile'
```

- **opannotate**(1), **oparchive**(1), **operf**(1), **opgprof**(1), **ophelp**(1), **opimport**(1), **opreport**(1) — Manual pages for various tools distributed with **OProfile** provide more information on their respective usage. To display the manual page for the version included in Red Hat Developer Toolset:

```
scl enable devtoolset-11 'man tool'
```

Online Documentation

- Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guide* for Red Hat Enterprise Linux 7 provides more information on **OProfile**.
- Red Hat Enterprise Linux 7 System Administrator’s Guide — The *System Administrator’s Guide* for Red Hat Enterprise Linux 7 documents how to use the **operf** tool.

See Also

- Chapter 1, *Red Hat Developer Toolset* — An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 12, *SystemTap* — An introduction to **SystemTap** and instructions on how to use it to monitor the activities of a running system.
- Chapter 13, *Valgrind* — Instructions on using the **Valgrind** tool to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.
- Chapter 15, *Dyninst* — Instructions on using the **Dyninst** library to instrument a user-space executable.

CHAPTER 15. DYNINST

The **Dyninst** library provides an *application programming interface* (**API**) for instrumenting and working with user-space executables during their execution. It can be used to insert code into a running program, change certain subroutine calls, or even remove them from the program. It serves as a valuable debugging and performance-monitoring tool. The **Dyninst API** is also commonly used along with **SystemTap** to allow non- **root** users to instrument user-space executables.

Red Hat Developer Toolset is distributed with **Dyninst 11.0.0**.

15.1. INSTALLING DYNINST

In Red Hat Developer Toolset, the **Dyninst** library is provided by the **devtoolset-11-dyninst** package and is automatically installed with **devtoolset-11-perftools** as described in Section 1.5, "Installing Red Hat Developer Toolset". In addition, it is recommended that you also install the **GNU Compiler Collection** provided by the **devtoolset-11-toolchain** package.

If you intend to write a custom instrumentation for binaries, install the relevant header files:

```
# yum install devtoolset-11-dyninst-devel
```

You can also install **API** documentation for this library:

```
# yum install devtoolset-11-dyninst-doc
```

For a complete list of documents that are included in the **devtoolset-11-dyninst-doc** package, see Section 15.3, "Additional Resources". For detailed instructions on how to install optional packages to your system, see Section 1.5, "Installing Red Hat Developer Toolset" .

15.2. USING DYNINST

15.2.1. Using Dyninst with SystemTap

To use **Dyninst** along with **SystemTap** to allow non- **root** users to instrument user-space executables, run the **stap** command with the **--dyninst** (or **--runtime=dyninst**) command line option. This tells **stap** to translate a **SystemTap** script into C code that uses the **Dyninst** library, compile this C code into a shared library, and then load the shared library and run the script. Note that when executed like this, the **stap** command also requires the **-c** or **-x** command line option to be specified.

To use the **Dyninst** runtime to instrument an executable file:

```
$ scl enable devtoolset-11 "stap --dyninst -c 'command' option... argument..."
```

Similarly, to use the **Dyninst** runtime to instrument a user's process:

```
$ scl enable devtoolset-11 "stap --dyninst -x process_id option... argument..."
```

See Chapter 12, *SystemTap* for more information about the Red Hat Developer Toolset version of **SystemTap**. For a general introduction to **SystemTap** and its usage, see the *SystemTap Beginners Guide* for Red Hat Enterprise Linux 7.

Example 15.1. Using Dyninst with SystemTap

Consider a source file named **exercise.C** that has the following contents:

```
#include <stdio.h>

void print_iteration(int value) {
  printf("Iteration number %d\n", value);
}

int main(int argc, char **argv) {
  int i;
  printf("Enter the starting number: ");
  scanf("%d", &i);
  for(; i>0; --i)
    print_iteration(i);
  return 0;
}
```

This program prompts the user to enter a starting number and then counts down to 1, calling the **print_iteration()** function for each iteration in order to print the number to the standard output. Compile this program on the command line using the **g++** compiler from Red Hat Developer Toolset:

```
$ scl enable devtoolset-11 'g++ -g -o exercise exercise.C'
```

Now consider another source file named **count.stp** with the following contents:

```
#!/usr/bin/stap

global count = 0

probe process.function("print_iteration") {
  count++
}

probe end {
  printf("Function executed %d times.\n", count)
}
```

This **SystemTap** script prints the total number of times the **print_iteration()** function was called during the execution of a process. Run this script on the **exercise** binary file:

```
$ scl enable devtoolset-11 "stap --dyninst -c './exercise' count.stp"
Enter the starting number: 5
Iteration number 5
Iteration number 4
Iteration number 3
Iteration number 2
Iteration number 1
Function executed 5 times.
```

15.2.2. Using Dyninst as a Stand-alone Library

Before using the **Dyninst** library as a part of your application, set the value of the **DYNINSTAPI_RT_LIB** environment variable to the path to the runtime library file:

```
$ export DYNINSTAPI_RT_LIB=/opt/rh/devtoolset-11/root/usr/lib64/dyninst/libdyninstAPI_RT.so
```

This sets the **DYNINSTAPI_RT_LIB** environment variable in the current shell session.

Example 15.2, "Using Dyninst as a Stand-alone Application" illustrates how to write and build a program to monitor the execution of a user-space process. For a detailed explanation of how to use **Dyninst**, see the resources listed in Section 15.3, "Additional Resources".

Example 15.2. Using Dyninst as a Stand-alone Application

Consider the **exercise.C** source file from Example 15.1, "Using Dyninst with SystemTap" : this program prompts the user to enter a starting number and then counts down to 1, calling the **print_iteration()** function for each iteration in order to print the number to standard output.

Now consider another source file named **count.C** with the following contents:

```
#include <stdio.h>
#include <fcntl.h>
#include "BPatch.h"
#include "BPatch_process.h"
#include "BPatch_function.h"
#include "BPatch_Vector.h"
#include "BPatch_thread.h"
#include "BPatch_point.h"

void usage() {
  fprintf(stderr, "Usage: count <process_id> <function>\n");
}

// Global information for counter
BPatch_variableExpr *counter = NULL;

void createCounter(BPatch_process *app, BPatch_image *appImage) {
  int zero = 0;
  counter = app->malloc(*appImage->findType("int"));
  counter->writeValue(&zero);
}

bool interceptfunc(BPatch_process *app,
                   BPatch_image *appImage,
                   char *funcName) {
  BPatch_Vector<BPatch_function *> func;
  appImage->findFunction(funcName, func);
  if(func.size() == 0) {
    fprintf(stderr, "Unable to find function to instrument()\n");
    exit (-1);
  }
  BPatch_Vector<BPatch_snippet *> incCount;
  BPatch_Vector<BPatch_point *> *points;
  points = func[0]->findPoint(BPatch_entry);
  if ((*points).size() == 0) {
    exit (-1);
  }
```

```
  BPatch_arithExpr counterPlusOne(BPatch_plus, *counter, BPatch_constExpr(1));
  BPatch_arithExpr addCounter(BPatch_assign, *counter, counterPlusOne);

  return app->insertSnippet(addCounter, *points);
}

void printCount(BPatch_thread *thread, BPatch_exitType) {
  int val = 0;
  counter->readValue(&val, sizeof(int));
  fprintf(stderr, "Function executed %d times.\n", val);
}

int main(int argc, char *argv[]) {
   int pid;
   BPatch bpatch;
   if (argc != 3) {
     usage();
     exit(1);
   }
   pid = atoi(argv[1]);
   BPatch_process *app = bpatch.processAttach(NULL, pid);
   if (!app) exit (-1);
   BPatch_image *appImage = app->getImage();
   createCounter(app, appImage);
   fprintf(stderr, "Finding function %s(): ", argv[2]);
   BPatch_Vector<BPatch_function*> countFuncs;
   fprintf(stderr, "OK\nInstrumenting function %s(): ", argv[2]);
   interceptfunc(app, appImage, argv[2]);
   bpatch.registerExitCallback(printCount);
   fprintf(stderr, "OK\nWaiting for process %d to exit...\n", pid);
   app->continueExecution();
   while (!app->isTerminated())
     bpatch.waitForStatusChange();
   return 0;
}
```

Note that a client application is expected to destroy all **Bpatch** objects before any of the **Dyninst** library destructors are called. Otherwise the mutator might terminate unexpectedly with a segmentation fault. To work around this problem, set the **BPatch** object of the mutator as a local variable in the **main()** function. Or, if you need to use **BPatch** as a global variable, manually detach all the mutatee processes before the mutator exits.

This program accepts a process ID and a function name as command line arguments and then prints the total number of times the function was called during the execution of the process. You can use the following **Makefile** to build these two files:

```
DTS      = /opt/rh/devtoolset-11/root
CXXFLAGS = -g -I$(DTS)/usr/include/dyninst
LBITS    := $(shell getconf LONG_BIT)

ifeq ($(LBITS),64)
  DYNINSTLIBS = $(DTS)/usr/lib64/dyninst
else
  DYNINSTLIBS = $(DTS)/usr/lib/dyninst
endif
```

```
.PHONY: all
all: count exercise

count: count.C
 g++ $(CXXFLAGS) count.C -I /usr/include/dyninst -c
 g++ $(CXXFLAGS) count.o -L $(DYNINSTLIBS) -ldyninstAPI -o count

exercise: exercise.C
 g++ $(CXXFLAGS) exercise.C -o exercise

.PHONY: clean
clean:
 rm -rf *~ *.o count exercise
```

To compile the two programs on the command line using the **g++** compiler from Red Hat Developer Toolset, run the **make** utility:

```
$ scl enable devtoolset-11 make
g++ -g -I/opt/rh/devtoolset-11/root/usr/include/dyninst count.C -c
g++ -g -I/opt/rh/devtoolset-11/root/usr/include/dyninst count.o -L /opt/rh/devtoolset-11/root/usr/lib64/dyninst -ldyninstAPI -o count
g++ -g -I/opt/rh/devtoolset-11/root/usr/include/dyninst exercise.C -o exercise
```

This creates new binary files called **exercise** and **count** in the current working directory.

In one shell session, execute the **exercise** binary file as follows and wait for it to prompt you to enter the starting number:

```
$ ./exercise
Enter the starting number:
```

Do not enter this number. Instead, start another shell session and type the following at its prompt to set the **DYNINSTAPI_RT_LIB** environment variable and execute the **count** binary file:

```
$ export DYNINSTAPI_RT_LIB=/opt/rh/devtoolset-11/root/usr/lib64/dyninst/libdyninstAPI_RT.so
$ ./count `pidof exercise` print_iteration
Finding function print_iteration(): OK
Instrumenting function print_iteration(): OK
Waiting for process 8607 to exit...
```

Now switch back to the first shell session and enter the starting number as requested by the **exercise** program. For example:

```
Enter the starting number: 5
Iteration number 5
Iteration number 4
Iteration number 3
Iteration number 2
Iteration number 1
```

When the **exercise** program terminates, the **count** program displays the number of times the **print_iteration()** function was executed:

Function executed 5 times.

15.3. ADDITIONAL RESOURCES

For more information about Dyninst and its features, see the resources listed below.

Installed Documentation

The **devtoolset-11-dyninst-doc** package installs the following documents in the **/opt/rh/devtoolset-11/root/usr/share/doc/devtoolset-11-dyninst-doc-11.0.0/** directory:

- *Dyninst Programmer's Guide* – A detailed description of the Dyninst **API** is stored in the **DyninstAPI.pdf** file.
- *DynC API Programmer's Guide* – An introduction to DynC API is stored in the **dynC_API.pdf** file.
- *ParseAPI Programmer's Guide* – An introduction to the ParseAPI is stored in the **ParseAPI.pdf** file.
- *PatchAPI Programmer's Guide* – An introduction to PatchAPI is stored in the **PatchAPI.pdf** file.
- *ProcControlAPI Programmer's Guide* – A detailed description of ProcControlAPI is stored in the **ProcControlAPI.pdf** file.
- *StackwalkerAPI Programmer's Guide* – A detailed description of StackwalkerAPI is stored in the **stackwalker.pdf** file.
- *SymtabAPI Programmer's Guide* – An introduction to SymtabAPI is stored in the **SymtabAPI.pdf** file.
- *InstructionAPI Reference Manual* – A detailed description of the InstructionAPI is stored in the **InstructionAPI.pdf** file.

For information on how to install this package on your system, see Section 15.1, "Installing Dyninst".

Online Documentation

- Dyninst Home Page – The project home page provides links to additional documentation and related publications.
- Red Hat Enterprise Linux 7 SystemTap Beginners Guide – The *SystemTap Beginners Guide* for Red Hat Enterprise Linux 7 provides an introduction to SystemTap and its usage.
- Red Hat Enterprise Linux 7 SystemTap Tapset Reference – The *SystemTap Tapset Reference* for Red Hat Enterprise Linux 7 provides further details about SystemTap.

See Also

- Chapter 1, *Red Hat Developer Toolset* – An overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 12, *SystemTap* – An introduction to **SystemTap** and instructions on how to use it to monitor the activities of a running system.

- Chapter 13, *Valgrind* — Instructions on using the **Valgrind** tool to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.

- Chapter 14, *OProfile* — Instructions on using the **OProfile** tool to determine which sections of code consume the greatest amount of **CPU** time and why.

PART V. COMPILER TOOLSETS

CHAPTER 16. COMPILER TOOLSETS DOCUMENTATION

The descriptions of the three compiler toolsets:

- LLVM Toolset
- Go Toolset
- Rust Toolset

have been moved to a separate documentation set under Red Hat Developer Tools.

PART VI. GETTING HELP

CHAPTER 17. ACCESSING RED HAT PRODUCT DOCUMENTATION

Red Hat Product Documentation located at https://access.redhat.com/site/documentation/ serves as a central source of information. It is currently translated in 23 languages, and for each product, it provides different kinds of books from release and technical notes to installation, user, and reference guides in **HTML**, **PDF**, and **EPUB** formats.

Below is a brief list of documents that are directly or indirectly relevant to this book.

Red Hat Developer Toolset

- Red Hat Developer Toolset 11.0 Release Notes – The *Release Notes* for Red Hat Developer Toolset 11.0 contain more information.
- Using Red Hat Software Collections Container Images – The *Using Red Hat Software Collections Container Images* provides instructions for obtaining, configuring, and using container images that are shipped with Red Hat Software Collections, including the Red Hat Developer Toolset container images.
- Red Hat Software Collections Packaging Guide – The *Software Collections Packaging Guide* explains the concept of Software Collections and documents how to create, build, and extend them.

Red Hat Enterprise Linux

- Red Hat Enterprise Linux 7 Developer Guide – The *Developer Guide* for Red Hat Enterprise Linux 7 provides more information about libraries and runtime support, compiling and building, debugging, and profiling.
- Red Hat Enterprise Linux 7 Installation Guide – The *Installation Guide* for Red Hat Enterprise Linux 7 explains how to obtain, install, and update the system.
- Red Hat Enterprise Linux 7 System Administrator's Guide – The *System Administrator's Guide* for Red Hat Enterprise Linux 7 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 7.

CHAPTER 18. CONTACTING GLOBAL SUPPORT SERVICES

Unless you have a Self-Support subscription, when both the Red Hat Documentation website and Customer Portal fail to provide the answers to your questions, you can contact *Global Support Services* (*GSS*).

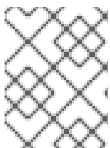
18.1. GATHERING REQUIRED INFORMATION

Several items of information should be gathered before contacting **GSS**.

Background Information

Ensure you have the following background information at hand before calling **GSS**:

- Hardware type, make, and model on which the product runs
- Software version
- Latest upgrades
- Any recent changes to the system
- An explanation of the problem and the symptoms
- Any messages or significant information about the issue



NOTE

If you ever forget your Red Hat login information, it can be recovered at https://access.redhat.com/site/help/LoginAssistance.html.

Diagnostics

The diagnostics report for Red Hat Enterprise Linux is required as well. This report is also known as a *sosreport* and the program to create the report is provided by the **sos** package. To install the **sos** package and all its dependencies on your system:

```
# yum install sos
```

To generate the report:

```
# sosreport
```

For more information, access the Knowledgebase article at https://access.redhat.com/kb/docs/DOC-3593.

Account and Contact Information

In order to help you, **GSS** requires your account information to customize their support, as well contact information to get back to you. When you contact **GSS** ensure you have your:

- Red Hat customer number or Red Hat Network (RHN) login name
- Company name
- Contact name

- Preferred method of contact (phone or email) and contact information (phone number or email address)

Issue Severity

Determining an issue's severity is important to allow the **GSS** team to prioritize their work. There are four levels of severity.

Severity 1 (urgent)
: A problem that severely impacts your use of the software for production purposes. It halts your business operations and has no procedural workaround.

Severity 2 (high)
: A problem where the software is functioning, but production is severely reduced. It causes a high impact to business operations, and no workaround exists.

Severity 3 (medium)
: A problem that involves partial, non-critical loss of the use of the software. There is a medium to low impact on your business, and business continues to function by utilizing a workaround.

Severity 4 (low)
: A general usage question, report of a documentation error, or a recommendation for a future product improvement.

For more information on determining the severity level of an issue, see https://access.redhat.com/support/policy/severity.

Once the issue severity has been determined, submit a service request through the Customer Portal under the **Connect** option, or at https://access.redhat.com/support/contact/technicalSupport.html. Note that you need your Red Hat login details in order to submit service requests.

If the severity is level 1 or 2, then follow up your service request with a phone call. Contact information and business hours are found at https://access.redhat.com/support/contact/technicalSupport.html.

If you have a premium subscription, then after hours support is available for Severity 1 and 2 cases.

Turn-around rates for both premium subscriptions and standard subscription can be found at https://access.redhat.com/support/offerings/production/sla.html.

18.2. ESCALATING AN ISSUE

If you feel an issue is not being handled correctly or adequately, you can escalate it. There are two types of escalations:

Technical escalation
: If an issue is not being resolved appropriately or if you need a more senior resource to attend to it.

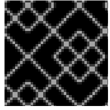
Management escalation
: If the issue has become more severe or you believe it requires a higher priority.

More information on escalation, including contacts, is available at https://access.redhat.com/support/policy/mgt_escalation.html.

18.3. RE-OPENING A SERVICE REQUEST

If there is more relevant information regarding a closed service request (such as the problem reoccurring), you can re-open the request via the Red Hat Customer Portal at

https://access.redhat.com/support/policy/mgt_escalation.html or by calling your local support center, the details of which can be found at https://access.redhat.com/support/contact/technicalSupport.html.



IMPORTANT

In order to re-open a service request, you need the original service-request number.

18.4. ADDITIONAL RESOURCES

For more information, see the resources listed below.

Online Documentation

- Getting Started – The *Getting Started* page serves as a starting point for people who purchased a Red Hat subscription and offers the *Red Hat Welcome Kit* and the *Quick Guide to Red Hat Support* for download.
- How can a RHEL Self-Support subscription be used? – A Knowledgebase article for customers with a Self-Support subscription.
- Red Hat Global Support Services and public mailing lists – A Knowledgebase article that answers frequent questions about public Red Hat mailing lists.

APPENDIX A. CHANGES IN VERSION 11.0

The following sections document features and compatibility changes introduced with Red Hat Developer Toolset 11.0. The list is not full and will be updated.

A.1. CHANGES IN GCC

Red Hat Developer Toolset 11.0 is distributed with **GCC 11.2**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

General Improvements

- GCC now defaults to the DWARF Version 5 debugging format.
- Column numbers shown in diagnostics represent real column numbers by default and respect multicolumn characters.
- The straight-line code vectorizer considers the whole function when vectorizing.
- A series of conditional expressions that compare the same variable can be transformed into a switch statement if each of them contains a comparison expression.
- Interprocedural optimization improvements:
  - A new IPA-modref pass, controlled by the **-fipa-modref** option, tracks side effects of function calls and improves the precision of points-to analysis.
  - The identical code folding pass, controlled by the **-fipa-icf** option, was significantly improved to increase the number of unified functions and reduce compile-time memory use.
- Link-time optimization improvements:
  - Memory allocation during linking was improved to reduce peak memory use.
- Using a new **GCC_EXTRA_DIAGNOSTIC_OUTPUT** environment variable in IDEs, you can request machine-readable “fix-it hints” without adjusting build flags.
- The static analyzer, run by the **-fanalyzer** option, is improved significantly with numerous bug fixes and enhancements provided.
- To mitigate CVE-2021-42574, a new warning was added to GCC with the release of the RHSA-2021:4669 advisory. This new **-Wbidirectional=[none|unpaired|any]** warning warns about possibly dangerous bidirectional (BiDi) Unicode characters and has three levels:
  - **-Wbidirectional=unpaired** (default) warns about improperly terminated BiDi contexts.
  - **-Wbidirectional=none** turns the warning off.
  - **-Wbidirectional=any** warns about any use of BiDi characters.

Language-specific Improvements

C family

- C and C++ compilers support non-rectangular loop nests in OpenMP constructs and the allocator routines of the OpenMP 5.0 specification.
- Attributes:
  - The new **no_stack_protector** attribute marks functions that should not be instrumented with stack protection (**-fstack-protector**).
  - The improved **malloc** attribute can be used to identify allocator and deallocator API pairs.
- New warnings:
  - **-Wsizeof-array-div**, enabled by the **-Wall** option, warns about divisions of two **sizeof** operators when the first one is applied to an array and the divisor does not equal the size of the array element.
  - **-Wstringop-overread**, enabled by default, warns about calls to string functions that try to read past the end of the arrays passed to them as arguments.
- Enhanced warnings:
  - **-Wfree-nonheap-object** detects more instances of calls to deallocation functions with pointers not returned from a dynamic memory allocation function.
  - **-Wmaybe-uninitialized** diagnoses the passing of pointers and references to uninitialized memory to functions that take **const**-qualified arguments.
  - **-Wuninitialized** detects reads from uninitialized dynamically allocated memory.

C

- Several new features from the upcoming C2X revision of the ISO C standard are supported with the **-std=c2x** and **-std=gnu2x** options. For example:
  - The standard attribute is supported.
  - The **__has_c_attribute** preprocessor operator is supported.
  - Labels may appear before declarations and at the end of a compound statement.

C++

- The default mode is changed to **-std=gnu++17**.
- The C++ library **libstdc++** has improved C++17 support now.
- Several new C++20 features are implemented. Note that C++20 support is experimental. For more information about the features, see C++20 Language Features.
- The C++ front end has experimental support for some of the upcoming C++23 draft features.
- New warnings:
  - **-Wctad-maybe-unsupported**, disabled by default, warns about performing class template argument deduction on a type with no deduction guides.
  - **-Wrange-loop-construct**, enabled by **-Wall**, warns when a range-based for loop is creating unnecessary and resource inefficient copies.

- **-Wmismatched-new-delete**, enabled by **-Wall**, warns about calls to operator delete with pointers returned from mismatched forms of operator new or from other mismatched allocation functions.

- **-Wvexing-parse**, enabled by default, warns about the most vexing parse rule: the cases when a declaration looks like a variable definition, but the C++ language requires it to be interpreted as a function declaration.

Architecture-specific Improvements

The 64-bit ARM architecture

- The Armv8-R architecture is supported through the **-march=armv8-r** option.

- GCC can autovectorize operations performing addition, subtraction, multiplication, and the accumulate and subtract variants on complex numbers.

AMD and Intel 64-bit architectures

- The following Intel CPUs are supported: Sapphire Rapids, Alder Lake, and Rocket Lake.

- New ISA extension support for Intel AVX-VNNI is added. The **-mavxvnni** compiler switch controls the AVX-VNNI intrinsics.

- AMD CPUs based on the znver3 core are supported with the new **-march=znver3** option.

- Three microarchitecture levels defined in the x86-64 psABI supplement are supported with the new **-march=x86-64-v2**, **-march=x86-64-v3**, and **-march=x86-64-v4** options.

A.2. CHANGES IN BINUTILS

Red Hat Developer Toolset 11.0 is distributed with **binutils 2.36**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

The assembler

- On Intel architectures, the AMX, AVX VNNI, HRESET, Key Locker, TDX, and UINTR instructions are supported.

- When setting the link order attribute of ELF sections, you can use a numeric section index instead of a symbol name.

- The following ARM cores are supported: Cortex-A78, Cortex-A78AE, Cortex-A78C, Cortex-X1, Cortex-R82, Neoverse V1, and Neoverse N2.

- On 64-bit ARM architectures, the Armv8-R and Armv8.7-A ISA extensions are supported.

- A **.nop** directive has been added that generates a single no-operation instruction that will work on any target.

- The **SHF_GNU_RETAIN** flag is supported. It specifies that the section should not be garbage collected by the linker. This flag can be applied to sections using the **R** flag in the **.section** directive.

The linker

- A new **libdep** plugin has been added. It records linking dependencies in static libraries and uses them when a final link is performed.
- A new **--error-handling-script=<NAME>** command-line option has been added. It runs a helper script when an undefined symbol or a missing library is encountered.
- The linker now deduplicates the types in the **.ctf** sections. You can specify how the linker does this by using the new **--ctf-share-types** command-line option. The default value of this option, which is **share-unconflicted**, produces the most compact output.
- The linker omits the variable section from the **.ctf** sections by default, saving space. This behavior can be unsuitable for projects that have their own analog of symbol tables, which are not reflected in ELF symbol tables.
- The **SHF_GNU_RETAIN ELF** section flag is supported. This flag specifies that the section should not be garbage collected by the linker.

Other binary utilities

- **nm**: a new command-line option **--ifunc-chars=CHARS** has been added that specifies a string of one or two characters. The first character is used as the type character when displaying global **ifunc** symbols. The second character, if present, is used when displaying local **ifunc** symbols.
- **ar**: the previously unused **l** modifier can be used for specifying dependencies of a static library. The arguments of this **l** option (or its longer form **--record-libdeps**) are stored verbatim in the **__.LIBDEP** member of the archive, which the linker may read at link time.
- **readelf**: using the **--lto-syms** command-line option you can display the contents of the LTO symbol table sections.
- **readelf** accepts the **-C** command-line option that enables demangling of symbol names. In addition, the **--demangle=<style>**, **--no-demangle**, **--recurse-limit**, and **--no-recurse-limit** options are have been added.
- To mitigate CVE-2021-42574, a new command-line option was added to **binutils** with the release of the RHSA-2021:4730 advisory.
  Tools which display names or strings (**readelf**, **strings**, **nm**, and **objdump**) now have a new **--unicode** (**-U**) command-line option, which controls how Unicode characters are handled. The following values can be set for the option:
  - **--unicode=default** treats BiDi characters as normal for the tool. This is the default behaviour when the **--unicode** option is not used.
  - **--unicode=locale** displays BiDi characters according to the current locale.
  - **--unicode=hex** displays BiDi characters as hex byte values.
  - **--unicode=escape** displays BiDi characters as Unicode escape sequences.
  - **--unicode=highlight** displays BiDi characters as Unicode escape sequences highlighted in red if it's supported by the output device.

A.3. CHANGES IN ELFUTILS

Red Hat Developer Toolset 11.0 is distributed with **elfutils 0.185**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

- The **eu-elflint** and **eu-readelf** tools now recognize and show the **SHF_GNU_RETAIN** and **SHT_X86_64_UNWIND** flags on ELF sections.
- The **DEBUGINFOD_SONAME** macro has been added to **debuginfod.h**. This macro can be used with the **dlopen** function to load the **libdebuginfod.so** library dynamically from an application.
- A new function **debuginfod_set_verbose_fd** has been added to the **debuginfod-client** library. This function enhances the **debuginfod_find_*** queries functionality by redirecting the verbose output to a separate file.
- Setting the **DEBUGINFOD_VERBOSE** environment variable now shows more information about which servers the **debuginfod** client connects to and the HTTP responses of those servers.
- The **debuginfod** server provides a new thread-busy metric and more detailed error metrics to make it easier to inspect processes that run on the **debuginfod** server.
- The **libdw** library transparently handles the **DW_FORM_indirect** location value so that the **dwarf_whatform** function returns the actual FORM of an attribute.
- To reduce network traffic, the **debuginfod-client** library stores negative results in a cache, and client objects can reuse an existing connection.

A.4. CHANGES IN DWZ

Red Hat Developer Toolset 11.0 is distributed with **dwz 0.14**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

- The DWARF Version 5 debugging format is supported.
- The DWARF supplementary object files can be produced using the **.debug_sup** section.
- A new experimental optimization has been added that exploits the One Definition Rule of C++.
- The **DW_OP_GNU_variable_value** expression opcode is supported.
- Numerous bugs have been fixed and performance improvements have been added.

A.5. CHANGES IN GDB

Red Hat Developer Toolset 11.0 is distributed with **GDB 10.2**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

New features

- Multithreaded symbol loading is enabled by default on architectures that support this feature. This change provides better performance for programs with many symbols.
- Text User Interface (TUI) windows can be arranged horizontally.

- GDB supports debugging multiple target connections simultaneously but this support is experimental and limited. For example, you can connect each inferior to a different remote server that runs on a different machine, or you can use one inferior to debug a local native process or a core dump or some other process.

New and improved commands

- A new **tui new-layout *name window weight* [*window weight…*]** command creates a new text user interface (TUI) layout, you can also specify a layout name and displayed windows.
- The improved **alias [-a] [--] *alias = command* [*default-args*]** command can specify default arguments when creating a new alias.
- The **set exec-file-mismatch** and **show exec-file-mismatch** commands set and show a new **exec-file-mismatch** option. When GDB attaches to a running process, this option controls how GDB reacts when it detects a mismatch between the current executable file loaded by GDB and the executable file used to start the process.

Python API

- The **gdb.register_window_type** function implements new TUI windows in Python.
- You can now query dynamic types. Instances of the **gdb.Type** class can have a new boolean attribute **dynamic** and the **gdb.Type.sizeof** attribute can have value **None** for dynamic types. If **Type.fields()** returns a field of a dynamic type, the value of its **bitpos** attribute can be **None**.
- A new **gdb.COMMAND_TUI** constant registers Python commands as members of the TUI help class of commands.
- A new **gdb.PendingFrame.architecture()** method retrieves the architecture of the pending frame.
- A new **gdb.Architecture.registers** method returns a **gdb.RegisterDescriptorIterator** object, an iterator that returns **gdb.RegisterDescriptor** objects. Such objects do not provide the value of a register but help understand which registers are available for an architecture.
- A new **gdb.Architecture.register_groups** method returns a **gdb.RegisterGroupIterator** object, an iterator that returns **gdb.RegisterGroup** objects. Such objects help understand which register groups are available for an architecture.

A.6. CHANGES IN LTRACE

Red Hat Developer Toolset 11.0 is distributed with **ltrace 0.7.91**.

The following feature has been modified since the previous release of Red Hat Developer Toolset:

- If a path is specified in the **$XDG_CONFIG_DIRS** patch file but does not exist, no diagnostic is given.

A.7. CHANGES IN STRACE

Red Hat Developer Toolset 11.0 is distributed with **strace 5.13**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

Changes in Behavior

- Modified **%process** class contains system calls associated with process lifecycle (creation, execution, and termination):
  - New calls: **kill**, **tkill**, **tgkill**, **pidfd_send_signal**, and **rt_sigqueueinfo**
  - Removed calls: **arch_prctl** and **unshare**

Improvements

- A new **-n** (**--syscall-number**) option prints system call numbers.
- A new **--secontext[=full]** option displays SELinux contexts.
- Poke injection is implemented and two new options are added: **--inject=SET:poke_enter=** and **--inject=SET:poke_exit=**.
- On IBM POWER architecture, System Call Vectored (SCV) ABI support is added.
- **libdw**-based stack tracing is enabled for non-native personalities.
- Netlink data is printed in a more structured way.
- Decoding of the following system calls is implemented: **close_range**, **epoll_pwait2**, **faccessat2**, **landlock_add_rule**, **landlock_create_ruleset**, **landlock_restrict_self**, **mount_setattr**, and **process_madvise**.
- Decoding of the following system calls is enhanced: **io_uring_setup**, **membarrier**, **perf_event_open**, and **pidfd_open**.
- Decoding of the **GPIO_*** and **TEE_*** **ioctl** commands is implemented.
- Decoding of the following **ioctl** commands is implemented: **FS_IOC_FS[GS]ETXATTR**, **FS_IOC_[GS]ETFLAGS**, **FS_IOC32_[GS]ETFLAGS**, **LOOP_CONFIGURE**, **SIOCADDMULTI**, **SIOCDELMULTI**, **SIOCGIFENCAP**, **SIOCOUTQNSD**, **SIOCSIFENCAP**, **SIOCSIFHWBROADCAST**, **UBI_IOCRPEB and UBI_IOCSPEB**, **V4L2_BUF_TYPE_META_CAPTURE**, **V4L2_BUF_TYPE_META_OUTPUT**, and **VIDIOC_QUERY_EXT_CTRL**.
- Decoding of the **NT_PRSTATUS** and **NT_FPREGSET** regsets of the **PTRACE_GETREGSET** and **PTRACE_SETREGSET** **ptrace** requests is implemented.
- Decoding of the **regs** argument of the following **ptrace** requests is implemented: **PTRACE_GETREGS**, **PTRACE_GETREGS64**, **PTRACE_SETREGS**, **PTRACE_SETREGS64**, **PTRACE_GETFPREGS**, and **PTRACE_SETFPREGS**.
- Decoding of the **struct msginfo** argument of the **IPC_INFO** and **MSG_INFO** **msgctl** system calls commands is implemented.
- Decoding of the **struct msqid_ds** argument of the **MSG_STAT** and **MSG_STAT_ANY** **msgctl** system calls commands is implemented.
- Decoding of the **struct seminfo** argument of the **IPC_INFO** and **SEM_INFO** **semctl** system calls commands is implemented.
- Decoding of the **struct semid_ds** argument of the **IPC_SET**, **IPC_STAT**, **SEM_STAT**, and **SEM_STAT_ANY** semctl system calls commands is implemented.

- Decoding of the **struct shminfo** argument of the **IPC_INFO shmctl** system calls command is implemented.
- Decoding of the **struct shm_info** argument of the **SHM_INFO shmctl** system calls command is implemented.
- Decoding of the **struct shmid_ds** argument of the **SHM_STAT** and **SHM_STAT_ANY shmctl** system calls commands is implemented.
- Decoding of the **IFLA_BRPORT_*** netlink attributes is updated to match the Linux 5.12 kernel.
- Lists of the following constants are updated: ***_MAGIC**, **ALG_***, **AUDIT_***, **BPF_***, **BTRFS_***, **CAP_***, **CLOSE_RANGE_***, **DEVCONF_***, **ETH_***, **FAN_***, **IFLA_***, **INET_DIAG_***, **IORING_***, **IPV6_***, **IP_***, **KEXEC_***, **KEYCTL_***, **KEY_***, **KVM_***, **LOOP_***, **MDBA_***, **MEMBARRIER_CMD_***, **MPOL_***, **MS_***, **MTD_***, **NDA_***, **NFT_MSG_***, **NLMSGERR_***, **NT_***, **PR_***, **PTP_PEROUT_***, **PTRACE_***, **RESOLVE_***, **RTAX_***, **RTA_***, **RTC_***, **RTM_***, **RTNH_***, **RTPROT_***, **SCTP_***, **SEGV_***, **SO_***, **STATX_***, **ST_***, **SYS_***, **TCA_***, **TRAP_***, **UFFDIO_***, **UFFD_***, and **V4L2_***.
- Lists of **ioctl** commands are updated to match such lists from the Linux 5.13 kernel update.
- With the release of the RHEA-2022:4635 advisory, **strace** can now display mismatches between the actual SELinux contexts and the definitions extracted from the SELinux context database. An existing **--secontext** option of **strace** has been extended with the **mismatch** parameter. This parameter enables to print the expected context along with the actual one upon mismatch only. The output is separated by double exclamation marks (**!!**), first the actual context, then the expected one. In the examples below, the **full,mismatch** parameters print the expected full context along with the actual one because the user part of the contexts mismatches. However, when using a solitary **mismatch**, it only checks the type part of the context. The expected context is not printed because the type part of the contexts matches.

```
[...]
$ strace --secontext=full,mismatch -e statx stat /home/user/file
statx(AT_FDCWD, "/home/user/file"
[system_u:object_r:user_home_t:s0!!unconfined_u:object_r:user_home_t:s0], ...

$ strace --secontext=mismatch -e statx stat /home/user/file
statx(AT_FDCWD, "/home/user/file" [user_home_t:s0], ...
```

  SELinux context mismatches often cause access control issues associated with SELinux. The mismatches printed in the system call traces can significantly expedite the checks of SELinux context correctness. The system call traces can also explain specific kernel behavior with respect to access control checks.

Bug Fixes

- Decoding of the **SIOCGIFINDEX**, **SIOCBRADDIF**, and **SIOCBRDELIF ioctl** commands is fixed.
- The **clock_gettime64**, **clock_settime64**, **clock_adjtime64**, and **lock_getres_time64** system calls are added to the **%clock** trace class.
- The **statx** system call is added to the **%fstat** trace class.
- Previously, **strace** used insufficient buffer sizes for network interface name printing. This led to assertions on attempts of printing interface names that require quoting, for example, names longer than 4 characters in **-xx** mode. With the release of the RHEA-2022:4635 advisory, this bug has been fixed.

A.8. CHANGES IN SYSTEMTAP

Red Hat Developer Toolset 11.0 is distributed with **SystemTap 4.5**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

- 32-bit floating-point variables are automatically widened to double variables and, as a result, can be accessed directly as **$context** variables.
- **enum** values can be accessed as **$context** variables.
- The BPF uconversions tapset has been extended and includes more tapset functions to access values in user space, for example **user_long_error()**.
- Concurrency control has been significantly improved to provide stable operation on large servers.

For further information about notable changes, see the upstream SystemTap 4.5 release notes.

A.9. CHANGES IN VALGRIND

Red Hat Developer Toolset 11.0 is distributed with **Valgrind 3.17.0**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

- Valgrind can read the DWARF Version 5 debugging format.
- Valgrind supports debugging queries to the **debuginfod** server.
- The ARMv8.2 processor instructions are partially supported.
- The Power ISA v.3.1 instructions on POWER10 processors are partially supported.
- The IBM z14 processor instructions are supported.
- Most IBM z15 instructions are supported. The Valgrind tool suite supports the miscellaneous-instruction-extensions facility 3 and the vector-enhancements facility 2 for the IBM z15 processor. As a result, Valgrind runs programs compiled with GCC **-march=z15** correctly and provides improved performance and debugging experience.
- The **--track-fds=yes option** respects **-q** (**--quiet**) and ignores the standard file descriptors **stdin**, **stdout**, and **stderr** by default. To track the standard file descriptors, use the **--track-fds=all** option.
- The DHAT tool has two new modes of operation: **--mode=copy** and **--mode=ad-hoc**.

A.10. CHANGES IN DYNINST

Red Hat Developer Toolset 11.0 is distributed with **Dyninst 11.0.0**.

The following features have been added since the previous release of Red Hat Developer Toolset 11.0:

- Support for the **debuginfod** server and for fetching separate **debuginfo** files.

- Improved detection of indirect calls to procedure linkage table (PLT) stubs.
- Improved C++ name demangling.
- Fixed memory leaks during code emitting.

A.11. CHANGES IN ANNOBIN

Red Hat Developer Toolset 11.0 is distributed with **Annobin 9.82**.

The following features have been added or modified since the previous release of Red Hat Developer Toolset:

GCC plugin

- ARM and RISCV targets are supported.
- The LTO compiler is supported.

Annocheck

- In verbose mode, the reason for skipping specific tests is reported.
- Some messages are highlighted with color.
- Some GO tests have been added.
- On 64-bit ARM architectures, tests for BTI and PAC security features have been added.
- To mitigate CVE-2021-42574, a new test is added to detect the presence of multibyte characters in symbol names. This change has been implemented in Annobin with the release of the RHSA-2021:4729 advisory.