Red Hat Enterprise Linux 8

Developing applications in RHEL 8

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

Last Updated: 2019-05-07

Red Hat Enterprise Linux 8 Developing applications in RHEL 8

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

Legal Notice

Abstract

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

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

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

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

CHAPTER 1. SETTING UP A DEVELOPMENT WORKSTATION

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

1.1. PREREQUISITES

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

1.2. ENABLING DEBUG AND SOURCE REPOSITORIES

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

Procedure

- Enable the source and debug information package channels:

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

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

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

1.3. SETTING UP TO MANAGE APPLICATION VERSIONS

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

Procedure

1. Install the **git** package:

```
# yum install git
```

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

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

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

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

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

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

Additional resources

- Linux manual pages for Git and tutorials:

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

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

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

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

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

Preregquisites

- The debug and source repositories must be enabled.

Procedure

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

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

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

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

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

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

1.5. SETTING UP TO DEBUG APPLICATIONS

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

Preregquisites

- The debug and source repositories must be enabled.

Procedure

1. Install the tools useful for debugging:

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

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

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

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

```
# stap-prep
```

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

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

1.6. SETTING UP TO MEASURE PERFORMANCE OF APPLICATIONS

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

Preregquisites

- The debug and source repositories must be enabled.

Procedure

1. Install the tools for performance measurement:

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

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

```
# stap-prep
```

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

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

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

APPENDIX A. DIFFERENCES IN DEVELOPMENT TOOLS IN RHEL 8

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

A.1. CHANGES IN TOOLCHAIN SINCE RHEL 7

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

A.1.1. Changes in GCC in RHEL 8

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

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

Security

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

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

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

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

Architecture and processor support

Improvements to architecture and processor support include:

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

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

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

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

Languages and standards

Notable changes related to languages and standards include:

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

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

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

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

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

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

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

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

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

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

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

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

Additional resources

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

A.1.2. Security enhancements in GCC in RHEL 8

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

New warnings

These warning options have been added:

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

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

Warning improvements

These GCC warnings have been improved:

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

New UndefinedBehaviorSanitizer

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

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

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

New options for AddressSanitizer

These options have been added to AddressSanitizer:

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

Other sanitizers and instrumentation

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

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

- A new bounds violation detector called Pointer Bounds Checker can be enabled with the **-fcheck-pointer-bounds** option. Memory accesses are instrumented with run-time checks of used pointers against their bounds to detect pointer bounds violations (overflows). This functionality is available on targets of the 64-bit Intel and AMD architectures with a new ISA extension Intel MPX support.

Additional resources

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

```
$ man gcc
```

A.2. COMPILER TOOLSETS

RHEL 8.0 provides the following compiler toolsets as Application Streams:

- Clang and LLVM Toolset 7.0.1, which provides the LLVM compiler infrastructure framework, the Clang compiler for the C and C++ languages, the LLDB debugger, and related tools for code analysis. See the Using Clang and LLVM Toolset document.

- Rust Toolset 1.31, which provides the Rust programming language compiler **rustc**, the **cargo** build tool and dependency manager, the **cargo-vendor** plugin, and required libraries. See the Using Rust Toolset document.

- Go Toolset 1.11.5, which provides the Go programming language tools and libraries. Go is alternatively known as **golang**. See the Using Go Toolset document.

A.3. COMPATIBILITY-BREAKING CHANGES IN GDB

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

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

A.3.1. GDBserver now starts inferiors with shell

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

To disable using the shell:

- When using the **target extended-remote** GDB command, disable shell with the **set startup-with-shell off** command.

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

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

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

- On RHEL 7:

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

- On RHEL 8:

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

A.3.2. gcj support removed

Support for debugging Java programs compiled with the **gcj** has been removed.

A.3.3. New syntax for symbol dumping maintenance commands

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

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

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

The new syntax for the symbol dumping maintenance commands is:

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

A.3.4. Thread numbers are no loger global

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

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

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

Example A.2. Example of GDB thread number changes

On Red Hat Enterprise Linux 7:

```
# debuginfo-install coreutils
$ gdb -batch -ex 'file echo' -ex start -ex 'add-inferior' -ex 'inferior 2' -ex 'file echo' -ex start -ex 'info
threads' -ex 'pring $_thread' -ex 'inferior 1' -ex 'pring $_thread'
(...)
  Id   Target Id         Frame
* 2    process 203923 "echo" main (argc=1, argv=0x7fffffffdb88) at src/echo.c:109
  1    process 203914 "echo" main (argc=1, argv=0x7fffffffdb88) at src/echo.c:109
$1 = 2
(...)
$2 = 1
```

On Red Hat Enterprise Linux 8:

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

A.3.5. Memory for value contents can be limited

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

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

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

A.3.6. Sun version of stabs format no longer supported

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

A.3.7. Sysroot handling changes

The **set sysroot *path*** command specifies system root when searching for files needed for debugging. Directory names supplied to this command may now be prefixed with the string **target:** to make GDB read the shared libraries from the target system (both local and remote). The formerly available **remote:**

prefix is now treated as **target:**. Additionally, the default system root value has changed from empty string to **target:** for backward compatibility.

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

A.3.8. HISTSIZE no longer controls GDB command history size

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

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

A.3.9. Completion limiting added

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

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

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

A.3.10. HP-UX XDB compatibility mode removed

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

A.3.11. Handling signals for threads

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

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

A.3.12. Breakpoint modes always-inserted off and auto merged

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

A.3.13. remotebaud commands no longer supported

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

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

A.4.1. C++ ABI change in std::string and std::list

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

A.4.2. librtkaio removed

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

As a result of the removal:

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

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

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

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

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

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

A.4.3. Sun RPC and NIS interfaces removed from glibc

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

A.4.4. Valgrind library for MPI debugging support removed

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

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

A.4.5. Development headers and static libraries removed from **valgrind-devel**

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

A.4.6. The **nosegneg** libraries for 32-bit Xen have been removed

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

A.4.7. GCC no longer builds Ada, Go, and Objective C/C++ code

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

To build Go code, use the Go Toolset instead.

A.4.8. **make** new operator **!=** causes a different interpretation of certain existing makefile syntax

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

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