

# Reversing with GDB and GEF

## Starting GDB

```
gdb program [core|pid]
gdb gdb-options [--args program args]
```

At startup, gdb reads the following init files and executes their commands: `/etc/gdb/gdbinit` and `~/gdbinit`, plus `./gdbinit`, if `set auto-load local-gdbinit` is set to `on`. In addition to reading its config from `~/gef.rc`, GEF can also be configured with `gef config`.

Some options are:

<code>-q/--quiet/--silent</code>	don't print version number on startup
<code>-h/--help</code>	print help
<code>--tty=TTY</code>	use TTY for I/O by debugged program
<code>--nh</code>	do not read <code>~/gdbinit</code>
<code>-x FILE</code>	execute GDB commands from <i>FILE</i>
<code>-ix FILE</code>	like <code>-x</code> but execute before loading inferior
<code>-ex CMD</code>	execute a single GDB command
<code>-iex CMD</code>	like <code>-ex</code> but before loading inferior
<code>-s SYMFILE</code>	read symbols from <i>SYMFILE</i>
<code>--write</code>	set writing into executable and core files

To quit, `q[uit]` or `Ctrl-D`. You can invoke commands on the standard shell by using: `shell command-string` or simply: `!command-string`. Commands can be abbreviated, when unambiguous, and some commands can be repeated by typing just *Return*. *Tab* autocompletes. You can ask for information on commands by using `h[elp]` and `gef help`, or use `apropos regexp` to search for commands matching *regexp*.

When GDB is spawn from a tmux session, `tmux-setup` splits the pane vertically, and configure the context to be redirected to the new pane.

**Debugging targets** `target type param` connects to machines, processes, or files; e.g. `target remote | sshpass -ppw ssh -T [-p port] [user@host] gdbserver - prog [args]`

`gef-remote` is a wrapper for the `target remote` command, which includes a QEMU-mode with `-q`.

**Processes and threads** When a program forks, gdb will continue to debug the parent. To follow the child, use `set follow-fork-mode`; to debug both processes, use `set detach-on-fork off`. Then, you can list the inferiors with `info inferiors` and switch among them with `inferior id`. If you issue a `run` after an `exec` call executes, the new target restarts. To restart the original one, use `file` with parent executable name as its argument.

`info threads` lists current threads, `thread id` selects the current one, `thread name name` gives a name and `thread apply range|all cmd` runs *cmd* for each specified thread; e.g. `thread apply all bt`.

In (default) all-stop mode, whenever your program stops, all threads stop. Conversely, whenever you restart the program, all threads start executing, *even when single-stepping*. Set `scheduler-locking to on` or `step` to avoid this. To enable non-stop mode, use `set pagination off` and `set non-stop on`, before starting/connecting to a target.

**Logging output** Logging can be enabled/disabled with `set logging on/off`. `set logging file file` changes the current logfile (default: `gdb.txt`). `show logging` shows current logging settings; other settings are `logging overwrite`, and `logging redirect` to choose whether the output goes to both terminal and logfile.

## Starting your program

`r[un] [args]`, `start [args]` and `starti [args]`

`start` does the equivalent of setting a temporary breakpoint at the beginning of *main* and invoking `run`. `starti` does the equivalent of setting a temporary breakpoint at the first instruction of a program's execution and invoking `run`. For programs containing an elaboration phase, `starti` will stop execution at the start of the elaboration phase.

*args* may include `"*"`, or `"[...]"`; they are expanded using the shell that will start the program (specified by the `$SHELL` environment variable). Input/output redirection with `">"`, `"<"`, or `">>"` are also allowed. With no arguments these commands use arguments last specified; to cancel previous arguments, use `set args` without arguments. To start the inferior without using a shell, use `set startup-with-shell off`.

`set disable-randomization on` (enabled by default) turns off ASLR; equivalently: `set exec-wrapper setarch 'uname -m' -R`. Also, `aslr`.

`elf[-info]` provides basic information on currently loaded program. `checksec`, inspired from `checksec.sh`, provides a convenient way to determine which security protections are enabled in a binary.

## Environment

`show environment [varname]`, `set environment varname [=value]` and `unset environment varname`

The changes are for your program (and the shell gdb uses to launch it), not for gdb itself. If your shell runs an initialization file when started non-interactively, it may affect your program.

`set cwd [dir]` sets the inferior's working directory to *dir*, while `cd [dir]` changes gdb working directory.

`tty` is an alias for `set inferior-tty`, which can be used to set the terminal terminal that will be used for future runs.

If debugged program was compiled with `-fstack-protector`, `canary` displays the value of the canary. `hijack-fd fd-num newfile` can be used to modify file descriptors of the debugged process. Permission rights on specific pages can be set by using `set-permission`.

## Stopping execution

- breakpoints* make your program stop whenever a certain point in the program is reached; you can add conditions to control whether your program stops. `[t]b[reak] location [thread id] [if condition]` sets a [temporary] breakpoint at the given *location* [, and given *condition*]. When called without any arguments, `break` sets a breakpoint at the next instruction to be executed. To use an address as a location, you must prepend it with `*`; e.g. `*0x1234`.

- `[t]hbreak args` sets [temporary] *hardware* breakpoints
- `rbreak [file:]regex` set breakpoints on all functions [of *file*] matching the regular expression *regex*.

`info break[points] [list]` prints all breakpoints, watchpoints, and catchpoints (or the ones specified in *list*). `clear location` and `delete [breakpoints] list` delete breakpoints.

`ignore bnum n` sets the ignore count of breakpoint *bnum* to *n* (0 to make it stop the next time breakpoint *bnum* is reached).

- `format-string-helper` creates a specific type of breakpoints dedicated to detecting potentially insecure format strings.
- `pie ...` handles PIE breakpoints.
- watchpoints*, AKA *data breakpoints*, are special breakpoints that stop your program when the value of an expression changes. `watch [-l|location] expr [thread thread-id]` sets a watchpoint for expression *expr*. If the command includes *thread-id* argument, gdb breaks only when the corresponding thread changes the value of *expr*. Ordinarily a watchpoint respects the scope of variables in *expr*; however, `-location` tells gdb to instead watch the memory referred to by *expr*.

- `rwatch` sets a *read* watchpoint
- `awatch` sets an *access* (i.e., *read or write*) watchpoint
- `info watchpoints [list]` prints a list of watchpoints

- catchpoints* stop your program when a certain kind of event occurs, such as throwing a C++ exception or loading a library. `catch event` stops when *event* occurs; it can be: an exception event; `syscall [name | number | g[roup]:groupname]`, `exec, fork`, `[un]load regexp`, `signal [signal... | all]`. Use `set stop-on-solib-events 1` to stop the target when a shared library is loaded or unloaded.

- To stop when your program receives a signal, use `handle`

A \*point can have any of several different states of enablement:

- Enabled/Disabled — `{enable|disable} [breakpoints] [list]`
- Enabled once — `enable [breakpoints] once [list]`
- Enabled for a count — `enable [breakpoints] count n [list]`
- Enabled for deletion — `enable [breakpoints] delete [list]`

`condition bnum [expr]` sets or removes a condition for \*point *bnum*.

When a \*point is hit, GEF displays the context (current instruction, registers, stack, ...) via its `context` command; to configure its layout see `gef config context.layout`. Any \*point can execute a series of commands when hit; see `commands`.

`save breakpoints filename` saves all current \*point definitions together with their commands and ignore counts, into *filename*. To read the saved breakpoint definitions, use `source`.

## Checkpoints

`checkpoint` saves a snapshot; `info checkpoints` lists the checkpoints that have been saved, while `restart checkpoint-id` restores the state that was saved. A checkpoint only restores things that reside in the program being debugged, not in the debugger. Finally, `delete checkpoint checkpoint-id` deletes the corresponding checkpoint.

Returning to a checkpoint restores the user state of the program being debugged, and a subset of the OS state including file pointers.

## Getting information

**i[nfo]** is for describing the state of your program. For example, you can show the arguments passed to a function with **info args**; you can get a complete list of the info sub-commands with **help info**.

**info program** displays information about the status of your program. **process-status** provides an exhaustive description of the current running process, by extending the information provided by **info proc**. **got** displays the status of the GOT inside the process.

**vmmap** displays the entire memory space mapping (**info shared** just the status of shared libraries); **xfiles** is a more convenient representation of **info files** to list the targets and files being debugged.

You can assign the result of an expression to an environment variable with **set**. For example, to set the prompt to a \$-sign: **set prompt \$**.

**show** is for describing the state of gdb itself. You can change most of things by using **set**; for example, what number system is used with **set radix**, or inquire it with **show radix**. To display all settable parameters and their values, you can use **show** with no arguments.

**info address symbol** shows where data for *symbol* is stored. **info symbol addr** prints the name of a symbol stored at the address *addr*. If no symbol is stored exactly at *addr*, gdb prints the nearest symbol and an offset from it. **xinfo** displays all information about a specific address. **[add-]symbol-file** loads different symbol tables.

## Examining the stack

One of the stack frames is selected and many commands refer implicitly to it. There are commands to select whichever frame you are interested in (**f[ra]me** **[n]****addr**, **up** **[n]** and **down** **[n]**). When your program stops, gdb automatically selects the currently executing frame and describes it briefly, similar to the **f[ra]me** command (without argument; for a more verbose description, **info f[ra]me** **[addr]**).

**info args** prints the arguments of selected frame, while **info locals** the local variables.

A backtrace is a summary of how your program got where it is. To print a backtrace of the entire stack, use **bt / backtrace** **[full]**; for dumping the stacks of all threads: **thread apply bt** **[full]**.

## Examining code

**disassemble** dumps a range of memory as machine instructions, with the raw instructions in hex with the **/r** modifier. The arguments specify a range of addresses to dump, in one of two forms: *start*, *end* to disassemble from *start* (inclusive) to *end* (exclusive), and *start*, *length* to disassemble from *start* (inclusive) to *start+length* (exclusive). See **cs / capstone-disassemble** (note: different args) for using *Capstone*.

If you have *Keystone*, then GEF can assemble native instructions directly with **asm / assembler**.

## Examining data

**registers** **[r<sub>1</sub> [r<sub>2</sub> ...]]** displays registers and dereference any pointers (in plain gdb you can see register values with **info registers**); **[edit-]flags** **[{+|-}~] flag-name [...]** shows/edits the flag register.

**p[rint]** **[/f] expr** prints the value of expression *expr* (in the source language). By default it's printed in a format appropriate to its type;

you can choose a different format by specifying **/f**, where *f* is one of: o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), c(char), s(string) and z(hex, zero padded on the left).

An expression in the form **{type}addr** refers to an object of type *type* stored at address *addr* in memory.

printed expressions may include function calls in the program being debugged; use **call addr** for calling functions without printing.

**x[/fmt] [addr]** examines memory; default *addr* is usually just after the last address examined. *fmt* is a repeat count followed by a format letter and a size letter. Format letters are like **print** plus i(instruction). Sizes are b(byte), h(halfword), w(word), g(giant, 8 bytes). If a negative number is specified, memory is examined backward. With GEF you can use **hexdump {qword[dword[word]byte]} location [lsize] [up/down]**.

**dprintf location,format-str[,a<sub>1</sub>[,a<sub>2</sub>...]]** sets a dynamic printf at specified *location*; this is actually a dprintf-breakpoint and you can use **cond** to add a condition, for instance.

**memory ...** adds or removes address ranges to the memory view.

**dereference / telescope** simplifies the dereferencing of an address in GDB to determine the content it actually points to.

The **\$** command mimics WinDBG ? command: when provided one argument, it evaluates the expression and displays the result in various formats. With two arguments, the delta between them.

**heap ...** is a base command to get information about the Glibc heap structure; moreover, **heap-analysis-helper** aims to help the process of identifying Glibc heap inconsistencies by tracking and analyzing allocations and deallocations.

**nop** and **patch** writes values into memory.

**dump**, **append**, and **restore** copy data between target memory and a file. **dump** and **append** write data to a file, whereas **restore** reads data from a file back into the inferior's memory.

**print-format / pf** dumps an arbitrary location as an array of bytes following the syntax of the programming language specified.

**shellcode** is a client for @JonathanSalwan shellcodes database.

**stub** stubs out functions, optionally specifying the return value.

**Automatic display** If you find that you want to print the value of an expression frequently you might want to add it to the automatic display list so that gdb prints its value each time your program stops. **display[/fmt] {expr}addr** adds the expression/address to such a list.

## Searching in memory

**find** **[/size-char] [/max-count] start-address, {end-address+length}, expr1 [, expr2 ...]** where *size-char* is one of b,h,w,g for 8,16,32,64 bit values respectively, and if not specified the size is taken from the type of the expression in the current language; in the case of C-like languages a search for an untyped 0x42 will search for "(int) 0x42" which is typically four bytes, and a search for a string "hello" will include the trailing '\0'. The null terminator can be removed from searching by using casts, e.g.: **char[5]"hello"**.

The address of the last match is stored as the value of **\$\_**. Convenience variable **\$numfound** is set to the number of matches.

**search-pattern**, alias **grep**, allows you to search for a specific pattern at runtime in all the segments of your process memory layout.

**pattern ...** creates or searches a De Bruijn cyclic pattern to facilitate determining offsets in memory; the algorithm is the same as the one in *puntools*, and can therefore be used in conjunction.

**scan haystack needle** searches for addresses that are located in memory mapping *haystack* that belonging to another, *needle*; its arguments are grepped against the processes memory mappings (just like **vmmap** to determine the memory ranges to search).

## Resuming execution

**c[ontinue] [ignore-count]** resumes program execution; the optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location.

To resume execution at a different place, use **return [expr]** to go back to the calling function; or **j[ump]** to go to an arbitrary location. Note that **return** does not resume execution: it leaves the program stopped in the state that would exist if the function had just returned. In contrast, **fin[ish]** continues running until just after function in the selected stack frame returns. **u[n]til**, without arguments, continues running until a source line past the current line is reached. It is like **next**, except that when **until** encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump. **u[n]til location** continue running until either the specified *location* is reached, or the current stack frame returns.

## Stepping

**s[tep] [count]** continue running your program until control reaches a different source line [*count* times]. **n[ext] [count]** continues to the next source line in the current stack frame. **stepi/si count** and **nexti/ni count** work on machine instructions, instead of source lines.

## Emulation

If you have installed *Unicorn* emulation engine and its Python bindings, **unicorn-emulate / emu** replicates the current memory mapping, and by default (i.e. without any additional argument), it will emulate the execution of the instruction about to be executed (i.e. the one pointed by **\$pc**) and display which register(s) is(are) tainted by it.

## Record, replay and reverse execution

**record** starts the process record and replay target; **record stop** stops.

**record goto {begin|end|n}** goes to {begin, end, instruction number *n* in instruction log}.

**rc / reverse-continue count** starts executing *in reverse*; similarly, for **reverse-{step[i],next[i]} [count]**. **reverse-finish** returns to a point where current function was called.

**set exec-direction {reverse,forward}** affects commands **step[i]**, **next[i]**, **continue** and **finish**; note **return** cannot be used in reverse.