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`.

Some options are:

`-q`/`--quiet`/`--silent`	don't print version number on startup
`-h`/`--help`	print help
`--tty=`*TTY*	use *TTY* for I/O by debugged program
`--nh`	do not read `~/.gdbinit`
`-x` *FILE*	execute GDB commands from *FILE*
`-ix` *FILE*	like `-x` but execute before loading inferior
`-ex` *CMD*	execute a single GDB command; may be used multiple times and with `-x`
`-iex` *CMD*	like `-ex` but before loading inferior
`-s` *SYMFILE*	read symbols from *SYMFILE*
`--write`	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*

You can abbreviate a gdb command to the first few letters of the command name, if that abbreviation is unambiguous; and you can repeat certain gdb commands by typing just *Return*. You can also use the *TAB* key to get gdb to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility).

You can always ask for information on commands by using `h`[`elp`].

Processes and threads

By default, when a program forks, gdb will continue to debug the parent process and the child process will run unimpeded. If you want to follow the child process instead of the parent process, use the command `set follow-fork-mode`. On Linux, if you want to debug both the parent and child processes, use the command `set detach-on-fork`.

If you issue a run command to gdb after an `exec` call executes, the new target restarts. To restart the original program, use the `file` command with the parent executable name as its argument. By default, after an `exec` call executes, gdb discards the symbols of the previous executable image. You can change this behaviour with the `set follow-exec-mode`.

gdb lets you run and debug multiple programs in a single session; in the most general case, you can have multiple threads of execution in each of multiple processes, launched from multiple executables. See the manual for details.

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`.

You can assign the result of an expression to an environment variable with `set`. For example, you can set the gdb prompt to a $-sign with `set prompt $`.

`show` is for describing the state of gdb itself. You can change most of things by using the related command `set`; for example, you can control what number system is used for displays with `set radix`, or simply inquire which is in use with `show radix`.

To display all settable parameters and their values, you can use `show` with no arguments; you may also use `info set`: both print the same.

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; you can get the same behavior by using:
`set exec-wrapper setarch 'uname -m' -R`.

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.

Checkpoint

On Linux gdb can save a snapshot of a program's state, called a *checkpoint*, and come back to it later. Returning to a checkpoint effectively undoes everything that has happened in the program since the checkpoint was saved. This includes changes in memory, registers, and even (within some limits) system state. Effectively, it is like going back in time to the moment when the checkpoint was saved.

`checkpoint` saves a snapshot; `info checkpoints` lists the checkpoints that have been saved, while `restart` *checkpoint-id* restores the state that was saved. All program variables, registers, stack frames etc. will be returned to the values that they had when the checkpoint was saved. Note that breakpoints, gdb variables, command history etc. are not affected by restoring a checkpoint. In general, 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 previously saved checkpoint will restore the user state of the program being debugged, plus a significant subset of the system (OS) state, including file pointers. It won't "un-write" data from a file, but it will rewind the file pointer to the previous location, so that the previously written data can be overwritten. For files opened in read mode, the pointer will also be restored so that the previously read data can be read again.