TD 8: Using gdb

Get the TD tarball from
`https://dept-info.labri.fr/~thibault/SecuLog/td8.tgz`
unpack it, run `make`

The goal of this practice lesson is to make yourself more familiar with how to inspect the execution of a program with gdb (and valgrind), in preparation of the forensic exercices of next week.

Note that all `gdb` commands can be abbreviated to their minimum number of letters that is not ambiguous with other commands

Note that if you are working on your own machine, make sure to have the libc debugging symbols for 32bit, e.g. by installing the `libc6-dbg:i386` package.

Ignore the `__attribute__((noinline))` showing up in some places, it is just there for the examples to remain simple, and yet prevent gcc from trivially optimizing everything away.

1 Playing in the stack

— Start `gdb stack`
— Set a breakpoint on `f` :
`b f`
(full command name : `break`)
— Run the program :
`r`
(full command name : `run`)
— Use `p` (full command name `print`) to print `x` , `*x` , `&x` and make sure to match the obtained values with the source code and what `pframe` shows.
— Note that the output of `p` can be tuned : `p/x x` (hexa), `p/u x` (unsigned int), `p/a x` (address), `p/t x` (binary) `p/c x` (character)
— Use `bt` (full command name `backtrace`) to print the call backtrace. It is ordered with most recent call first : `f` is shown at the beginning of the output. We also conveniently see the function parameters.
— Check the return address of `f` back to `g` : in the `pframe output` you can see it with `@ret`, it also shows up in the backtrace on the `g` line.
— Note that you cannot print `y` or `z` since they are in the context of the `g` function
— Use `up` to get "up" to the context of the `g` function.
— Print `y` , `&y` , `z` , `&z` , match this again with the source code and what `pframe` shows. Note that `pframe` also switched to looking at the context of `g`.
— Use `up` to get "up" to the context of the `main` function.
— Use `down` to get back to `g`.
— Use `bt` to look at the backtrace again.
— Use `frame 0` , `frame 1` , and `frame 2` to jump directly between `f`, `g`, and `main`.
— Use `bt full` to look at the backtrace, this time with the local variables printed in addition to the parameters.

2 Damn optimizations

— Start `gdb optim.O2`
— Set a breakpoint on `f`
— Run the program
— Try to print the address of the `z` variable in the `g` function. gdb says it can't...
— Use `disas` (full command name `disassemble`) to check the code of `g` . Why can't gdb print the address of `z` ?
— Try to print `a` or its address. Why can't gdb do it ?
— Also see that `bt full` cannot print `a` .

3 Watchpoints

We now want to catch the modification of a variable. In our case it's trivial to find out that `f` is doing it, but let's pretend we don't know.

— Start `gdb modif`
— Set a breakpoint on `main` and run the program
— Use `wa a` (full command name `watch`) to set a watch point on the `a` variable. Note that gdb says it is a *hardware* watchpoint : since `a` is only 4 bytes, gdb was able to just take the address of `a` and tell the processor to watch for that memory location in hardware, i.e. we will run the program at full speed and the processor will stop at the right time.
— Use `c` (full command name `continue`) to let gdb continue execution until the watch point is triggered.
— Notice that it simply stopped at the initialization of the variable. Note that gdb actually stopped right after the initialization. Use `p $eip` and `disas` to see that the instruction just before `eip` is indeed the one that does the initialization.
— Continue again, notice that here we are now in `f`. Use `disas` again to check that it indeed stopped just after the instruction that modified `a`. Notice that the actual incrementation is done a bit before that.

— Restart `gdb modif` again from scratch
— Set a breakpoint on `main`
— Print the address of `a`
— Now we will assume that we only know this address, and not the name of `a`. We thus cannot use `watch a`, instead use `watch *(int*) 0x12345` to tell gdb to watch at the address of `a` (note that you can basically type `watch *` and the paste the output of `p &a`)
— Notice that we indeed get gdb to stop as well. That will be useful for forensic.

Note : there are also `info break` to get the list of breakpoints and watchpoints currently defined, `delete 1` to delete breakpoint/watchpoint number 1, and `disable 1` to disable breakpoint/watchpoint number 1 without deleting it (and `enable` to re-enable it).

4 Real debugging with watchpoints

Try to run `modif2`, see that one of the integers is apparently getting overwritten. We want to find out how (initially without looking the source code of `f` and `g`).

— Start `gdb modif2`
— Set a breakpoint on `main`
— Run the program
— Watch the two integers
— Continue the execution

— See where an integer does indeed get overwritten.
— See that this is in the middle of (assembly!) implementation of `memset`.
— You can have a quick look at the `disas` to see what an SSE-based `memset` looks like, but let's not try to understand that part, use `up` to get to the context of the libc function call.
— gdb shows us the C source line, but let's pretend we have not seen it :) Rather use `disas` to see the function being called : `memset`.
— Put a breakpoint on the `call memset` instruction itself, (with `b * 0x12345678` where `0x12345678` is the address of the instruction), and restart the program.
— Print the parameters passed to memset, compute addresses and compare them with the addresses of `a`, `b`, `c`,
— Now look at the backtrace, you now know that the path to these parameters is through `g` and `f`, so you can now read their source code, and understand the bug.

5 Combining valgrind and gdb

— Try to run modif3, it seems alright.
— Try to run it in valgrind, no it is not alright actually.
— It is not necessarily so easy to find out what is happening with only the valgrind output, we'd want to see the addresses of all of this.
— Let's combine with gdb then : run `valgrind --vgdb-error=1 ./modif3`
— Valgrind is waiting for you to attach gdb externally, run in another terminal : `gdb ./modif3`
— and there, use `target remote | /usr/bin/vgdb --pid=1234` as Valgrind advised you (really use `/usr/bin/vgdb`, not the (bogus) path that Valgrind tells you).
— Now you have gdb stopped at the moment of the issue. You can now use `bt` etc. to check the content of the stack, the positions of the variables, etc.

6 Reverse execution

Gdb can actually go back in time! This is expensive, but can be convenient to track the last modification of something rather than the first.

— Start `gdb modif-rev`
— Set a breakpoint on `main`
— We would like to stop at the last modification of `a` Unfortunately there are thousands of modifications until then, we do not want to press `c` a thousand times (and we still assume that we don't know much about `f`, so we assume we cannot just set a breakpoint where we know `f` will modify `a` last)
— So first call `rec` (full command name `record`) to enable execution recording.
— In the `disas` of `main`, spot the addresse of the call to `g`, set a breakpoint just after it with `b * 0x12345`
— Run `c` , it takes a bit of time to record all the behavior of `f`.
— Print `a`, see that it has a completely modified value, we want to get the last modification.
— This time we do not want to let gdb use hardware watch, so use `set can-use-hw-watchpoints 0`
— Then use `watch a`
— Use `reverse-continue`
— gdb stops at the instruction that modified `a` last!

7 More details

Give
`https://dept-info.labri.fr/~thibault/gdb.html`
a read!