LINGI2144: Secured System Engineering
Tutorial 3: Debugging with GDB

Academic year : 2020 - 2021

Teacher : Legay Axel
Course : LINGI2144
Collaborators :
Crochet Christophe
Duchene Fabien
Given-Wilson Thomas
Strebelle Sebastien

1 Prerequisite

Working directory: `~/SecurityClass/Tutorial-03`

Connection:

username	password
user	none

2 Exercise

2.1 Basic `gdb` with source code

Let's look at `basic.c` taken from the lectures. We can compile with the debugging flag set, we can run the program as usual to observe the output and we can open the executable in `gdb` (in quiet mode `"-q"` to reduce output) with

```
gcc -g -o basic basic.c
./basic
gdb -q basic
```

Now we can execute the "main" function in gdb with

```
run main
```

Observe that the behaviour is the same as when executed outside `gdb`. Now let's play a little. First we can set a **break point** at the function `"func"` we call from main:

```
b func //b for break
```

and if we execute main now the debugger will break execution here `run main`.

Now we're paused in the execution of the `"func"` function. Observe that `gdb` shows us the line of code we would execute next. We can continue the execution one step with

```
s
```

The next line of the code is the return, but before we continue we can **inspect** the local values will print the values of "a" and "b", respectively..

```
p a //p for print
p b
```

Now we can continue through the program by **stepping** repeatedly with `"s"`, however you may find this takes a long time to finish (pressing enter after you have previously executed many `gdb` commands well repeat that command, this includes step). Observe that this reveals a lot of information about how the `printf` call is handled (but is not very fun).

You can also **continue** the execution after a break with `"c"`. Let's run the program again and break at the same point `run main` again we stop at the beginning of the `"func"` function, and we can check the values of "a" and "b".

```
p a
p b
```

Observe that here gdb knows that "b" exists, but no value has been set yet (the value shown is whatever was in the register/stack space reserved for "b"). We can see that this is for "b" by:

```
p x
```

to observe that no symbol "x" is defined. we can also use gdb to skip past instructions. If we now enter

```
jump +1
```

we will see that the program continues without ever setting the value of "b". Let's start our program again `run main` and now when we reach the break point let's set another **(temporary) break point** and then jump to it with:

```
tbreak +1
jump +1
```

now if we check the values of our variables

```
p a
p b
```

we can see we've skipped the assignment of "b". Let's fix this by giving "b" a more reasonable value

```
set var b=100
p b
```

Now let's allow the program to continue as normal with `c`.

As we can see, we can control the execution, control flow, and values of our programs while running them. Finally we can exit gdb with `quit`.

2.2 gdb disassembly

Now let's look at the disassembly and assembly instructions for our `basic.c` program. Again we compile with debugging information.

Let's run the program once to observe it is the same, and also to initialise some address and behaviour in gdb `r main`. Now we can disassemble the

```
disass main
```

Like before we can set a break point, except now at the address instead of using the source code information. Here let's set a break point at the call to "`func`", something like:

```
b *0x004011d9
```

NOTE: your address may be different, so don't assume you can copy and paste from the tutorial here!

Now we run the code as before and we break at the function call. Here we can inspect the "`eip`" register which indicates which instruction to load next

```
i r eip
```

observe that this corresponds to the instruction to call the function "`func`". Now if we step forward one instruction with

```
si
```

and inspect the `eip` register again

```
i r eip
```

we can see that the next instruction to execute is now the address of the call. Let's disassemble the "`func`" function and check that the address matches:

```
disass func
```

Now we can step through this function until before the "leave" instruction.

NOTE: you may need to use `i r eip` to know where you are since the debugger will mostly show the C code information (or you can set a break point and continue to it).

Here we can inspect the return value (stored in `eax`)

```
i r eax
```

and observe that it is "15" as we expect. We can also change this value with

```
set $eax = 200
```

and check this has worked with

```
i r eax
```

Now we can let the program finish and see that we've changed the result and quit.

```
c
quit
```

2.3 gdb without source code

We can also `gdb` to debug programs we do not have the source code for. However, for our first exercise let's use the same code as before, **but compile without the debugging information**.

```
gcc -o basic basic.c
gdb -q basic
```

If we run the code we see the expected result`r`. We know what is in this program, but in general we could start our disassembly with

```
disass main
```

which shows the same code as before. Also by inspecting this code we can see which other functions are called and disassemble them (or at least "`func`").

Let's set a break point at the beginning of the main code:

```
b *0x004011d9
```

NOTE: your address may be different

Now if we start the execution with `r` we will be stopped at the beginning. Observe that if we now use `s` to step, we will go to the next "step" the has line number/debugging information... in the print calls. We can let the program finish with "`c`", this is to illustrate that we cannot step when we only have assembly code. Now use `gdb` to change the value printed by this program.

IMPORTANT NOTE For Sections 3.4 and 3.5 of the tutorial the binaries have been provided. For each one the command to build it is also included (at the end of the section), but this is only here in case `gdb` fails to work on the provided binary. (This could happen because of differences in architecture or OS, but should NOT be a problem if you use the image provided.)

If you do not have any problems with `gdb`, do not look at the source code unless you really need help. The goal of the rest of the tutorial is to work from the binary and exploit `gdb`.

2.4 Find a secret code

Let's look at the program `"code"` and see what we can find. By running it we have some idea of it's behaviour:

```
./code
```

However, it might take a long time to guess the code (even it it's only a single byte). Also without the source code this is going to be annoying to work with. Let's start by loading the program in `gdb`.

```
gdb -q code
```

and running the program to ensure it works.

```
r
```

As we can see, the behaviour is the same.

Let's disassemble the main function and see what we can see.

```
disass main
```

You should be able to see the structure of the program, and from this how to find the passcode.

- HINT: can you find somewhere where some values are compared?

From this you should be able to find the code, although some conversion between hexidecimal and decimal may be needed to give the right input to the program.

A slightly harder example is available as `"recode"`, can you find the right passcode now?

- HINT: the code has been compiled with debugging information included.
- CHEATING: To build the `"code"` binary run

```
gcc -g -o code ../Cheating/3.4/code.c
```

To build the `"recode"` binary run

```
gcc -g -o recode ../Cheating/3.4/recode.c
```

2.5 Simple Backdoor

Let's look at the program `"backdoor"` and see what we can find. By running it we have some idea of it's behaviour:

```
./backdoor
```

You suspect this code has some behaviour that only triggers sometimes, but you don't know what causes it. **Use `gdb` to exhibit the backdoor behaviour.**

- HINT: Recall you can use "jump" to go to specific locations in the assembly.
- BONUS: Work out what the trigger for the backdoor behaviour is.
- HINT: Trace back or inspect the values used in the comparison operations.
- CHEATING: To build the "backdoor" binary run

```
gcc -o backdoor ../Cheating/3.5/backdoor.c
```

2.6 Self Modifying Code

In complex.c you will find two examples of self modifying code. This can be compiled and executed with to see the output:

```
gcc -o complex complex.c
./complex
```

The main idea behind self modifying code is to change the behaviour of the program at runtime, which is done here in two different ways. Due to the way Linux handles memory safety and code execution and access, the source code is a little complex. The main points to look at are as follows.

1. By line 42 we have allocated a block of memory that we can execute. Now lines 42 to 47 load values into this memory that can be interpreted as instructions. These are then executed on line 49 and the result printed on line 50. Observe that the instructions here load the value "0xD" into EAX (the return value register) and then return immediately.
2. A similar modification of the memory is done from lines 53 to 61. The return value is different, however you can see that the call to the executable memory is the same (only out writing to the memory has changed the behaviour).
3. Lines 80 to 93 demonstrate a different kind of executable memory modification. Initially we execute the function "fun" with argument "4" and print the result. After this we use "ptr" and "offset" to over-write the function behaviour, then we call the same function with the same argument and see different output.

Trace through these examples in gdb and observe the modification to the memory to see how the code operates (not just from the print statements and explanation here). To help it might be good to build the executable with debugging information:

```
gcc -g -o debug complex.c
```

We can use commands like to **see the memory values** that are being addressed by testfun:

```
x/24xw testfun
```

Modify complex.c to write different behaviour into one of the executed code blocks to change the program behaviour. (For example, can you create an infinite loop?)

2.7 Breaking "Safe" Code from Tutorial-01

Here we have some of the "fixed" versions of the programs from Tutorial 01. Recall that these used to have various vulnerabilities that could be exploited. These can all be built (without debugging information) with:

```
for x in `ls *fixed* | awk -F'.' '{print $1}'`
    do echo Building $x;gcc -o $x $x.c;done
```

Use gdb to cause them to crash or behave badly with "safe" input.

NOTE: to pass command line arguments into gdb use:

```
gdb -q --args bettercat-fixed bettercat-fixed.c
```

- HINT: If this is difficult with pure assembly, try compiling with debugging information first.
- HINT: Recall that you can change the values of registers or change the control flow

2.8 Breaking "Safe" Code from Tutorial-02

Here are two fixed programs form Tutorial 2 that should have safe spacial memory bounds. These can be built (again without debugging information with):

```
for x in `ls *fixed* | awk -F'.' '{print $1}'`
    do echo Building $x;gcc -o $x $x.c;done
```

Use gdb to inspect their program flow and then read outside of memory safety. (You should be able to produce the same behaviour as the original unsafe versions from Tutorial 2.)

2.9 Free Play with gdb

If you get this far then play around with gdb and see what other things you can do. Some suggestions include:

1. Watch a variable or condition to catch a specific point in a program (e.g. inside a loop).
2. Attach to a running process.
3. Write a program with a hidden value (like 3.4) and see if a friend can find the value using gdb on the compiled program.