Lab 1

Observing assembly in GDB

Goal of this lab is to familiarize you with Gnu Debugger, interacting with binary and reading some basic assembly.

Brief Intro

A debugger is a tool used to control execution of a program down to single machine instructions with goal of gaining insight into its' inner workings.

In general, debuggers can be used in two ways:

- they can start the binaries themselves to have a control from the very beginning
- they can *attach* to running processes

These are very privileged actions and many rules apply what can be debugged and what not, but the general rule of thumb is – you can debug your own processes.

If these processes are SUID/SGID or elevate your permissions in some other way, you generally won't be able to look into them without administrative privs as that would be an easy privesc vector.

In this and any following lab we will use the following convention when it comes to typing commands:

- If code is executed in normal bash shell, it will be prepended with $>

```
$> cat flag
```

- If code is executed inside GDB's shell, we'll add (gdb)

```
$(gdb)> attach 420
```

- In very unlikely case we use python's shell

```
$(py)> print('A'*12)
```

Lab

You can start GDB by just typing

```
$> gdb
```

This will launch debugger with nothing attached and wait for your commands. You can now, for example, attach to some running process by providing it's PID (process id) to *attach* cmd:

```
$(gdb)> attach <pid>
```

The other and probably more useful for us way is to start GDB with our target binary loaded:

```
$> gdb <binary>
```

GDB has comprehensive help module. Just typing help or help <command> will provide you with lots of knowledge. We are explaining bare minimum to get through labs here, reading up on commands you use is very much advised for greater understanding

```
$(gdb)> help
```

```
$(gdb)> help attach
```

Before looking through assembly or running code, let's set the disassembly flavor to intel. GDB defaults to AT&T which is not covered in our course

```
$(gdb)> set disassembly-flavor intel
```

Luckily this can be automated and you don't have to actually do this every time. Our environment comes with configuration provided in `~/.gdbinit`, feel free to explore its contents.

In our lab, all binaries are compiled with debugging symbols and source code is provided. This makes analysis orders of magnitude easier by allowing us to reference variable or function names from source code directly. Just keep in mind that in real life you'll rarely be given everything for a proper white-box.

To explore functions and source, the following commands can help:

```
$(gdb)> info functions
$(gdb)> list <function_name>
$(gdb)> disass <function_name>
$(gdb)> disass /m <function_name>
```

To start running our program, two options present themselves.

`Run` will start the program and let it… well, run.

```
$(gdb)> run
```

`Start` will stop execution of the program on the first line of main's code. Very useful.

```
$(gdb)> start
```

Actually, what `start` is doing is setting a *temporary breakpoint* that is deleted once triggered. Breakpoints are exactly what they sound like, a points in code, where the execution will stop and yield the control back to the debugger. Learning to place and utilize breakpoints is absolutely critical.

You can place breakpoint with:

```
$(gdb)> break *<address or name>
```

Asterisk instructs gdb to treat argument as address or try dereferencing symbol if it exists. Without asterisk you can only use symbols, like a function name, without any additional math. This has very limited usability, so just make a habit of prepending asterisk.

Most of the time we'll want to set breakpoints as offsets from function start. For example, consider the following disassembly:

```
(gdb) disass hello_val
Dump of assembler code for function hello_val:
   0x0000555555555149 <+0>:     push   rbp
   0x000055555555514a <+1>:     mov    rbp,rsp
   0x000055555555514d <+4>:     sub    rsp,0x10
   0x0000555555555151 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x0000555555555154 <+11>:    mov    eax,DWORD PTR [rbp-0x4]
   0x0000555555555157 <+14>:    mov    esi,eax
```

We want to set a breakpoint at `SUB RSP, 0x10` instruction. The much more robust (and quicker to type) way is to use an offset of <+4> from the function's start.

```
$(gdb)> break *hello_val+4
```

All breakpoints can be listed through

```
$(gdb)> info breakpoints
```

You can manage breakpoints through the following commands. If you don't provide argument, these will act on all breakpoints.

```
$(gdb)> delete <bp_number>
$(gdb)> disable <bp_number>
$(gdb)> enable <bp_number>
```

Once the breakpoint is hit, gdb's console will become interactive again and let you inspect live state of the processor.

Checking registers:

```
$(gdb)> info registers
$(gdb)> i r
$(gdb)> info registers <reg_name>
```

To interact with program's memory and make any use of data found in registers, `print` and `examine` commands are used.

When passing registers by name to commands, $ has to be prepended to their name so that gdb understands we are not looking for a program symbol.

`Print` or just `p` is used to just return value of what is provided. It is used to read values of registers, performing math and conversions of data.

```
$(gdb)> print $rsp
```

`Examine` or just `x` is used to dereference addresses and inspect what data they point at. It can interpret the result in different formats and read a variable amount of data.

```
$(gdb)> examine $rsp
```

The biggest difference between those two is that `print` takes the value and returns it, while `examine` treats anything passed to it as an address to be dereferenced.

These commands are very powerful but may seem a little complicated at first. The general structure is:

```
$(gdb)> p/F <address/symbol>
$(gdb)> x/NUF <address/symbol>
```

Where:

- N – number of units we want to examine
- U – units size, possible are:
  - b – byte
  - h – half word (2 bytes)
  - w – word
  - g - giant word (8 bytes)
- F – output data format:
  - x – hex
  - d – decimal
  - u – unsigned decimal
  - i – instruction
  - s – string
  - c – char

Some examples:

Get 32 bytes at address pointed by RSP:

```
$(gdb)> x/32b $rsp
```

Get four 8-byte long values as hex at address from RSI

```
$(gdb)> x/4gx $rsi
```

Print the value of RSI as hex

```
$(gdb)> p/x $rsi
```

Convert hex to decimal

```
$(gdb)> p/d 0x42424242
```

Finally, once done inspecting the state, there are three major ways we can proceed:

`Stepi` (also known as step-into) – advances the program by one instruction at a time.

`Nexti` (also known as step-over) – advances the program by one instruction but will fast-forward execution of any CALLs.

`Continue` – continues execution normally. Won't stop until next breakpoint or end of the program.

If program is misbehaving or you want to try again, use `kill`, `run` or `start`

Tasks

This lab is more about being comfortable with gdb than understanding the underlying assembly. There might be a lot of unknown instructions there. Do not force yourself to understand them all, rather try to get comfortable with moving around the program and understanding relationship between RIP and what's being executed.

1. There are few executables in "1. Assembly" directory. Pick any of them and practice running commands shown before.
    a. Practice means more than once
2. In particular try to:
    a. disass main or other functions
    b. set breakpoints manually and remove them
    c. set breakpoint on different offsets from main start
    d. stepping through the programs and inspecting how registers change with every instruction
    e. when stepping through, try both `stepi` and `nexti` when going over printf calls. Which one does take you into internals of printf and which fast-forwards you over that call?
3. Perform some conversions using print command:
    a. What is the decimal value of 0x420?
    b. What is the hex value of 6969?
    c. What is the decimal value of current RIP?
4. Examine memory using x command:
    a. Print out 32 quad words at RSP
    b. Can you interpret these quad words as instructions? How is that possible?
    c. Print out 32 assembly instructions starting from RIP