ECE455
Lab #0: Introduction to GDB
10/14/2020
Name: ____

This is an introduction to the GNU debugger “GDB” (see the GNU project’s site)

```
$ gdb --help
This is the GNU debugger.  Usage:

    gdb[options] [executable-file [core-file or process-id]]
    gdb[options] --args executable-file [inferior-arguments ...]
```

Overview

Goals

- Become familiar with GDB
- Gain practical experience with analyzing code

Grading & Submission

- This lab is not graded!

Lab Infrastructure

For your convenience we have configured a few VMs in the ee.cooper.edu subdomain. You may log into these with your ee username and password.

Hostname	IP
cybersec0	199.98.27.226
cybersec1	199.98.27.150
cybersec2	199.98.27.151
cybersec3	199.98.27.153
cybersec4	199.98.27.154

Introduction

This lab will give you a practical example of how to use GDB. This program was designed as a C debugger and has since extended support to C++ and many other modern languages. The debugger has many useful features that we will use to analyze C programs: view variables and registers, pausing execution, stepping through execution.

This is the example program we will use to learn about GDB:

```
#include <stdio.h>
// Example program to analyze with GDB

// Reads input and parses it as in integer
int read_int(void) {
    char buf[128];
    int i;
    gets(buf);
    i = atoi(buf);
    return i;
}

int main(int ac, char **av) {
    int x = read_int();
    printf("x=%d\n", x);
}
```

Let's start by compiling the program. We're adding a few flags here to make the x86 code easier to understand and adding debugging symbols.

```
gitzel@cybersec0:~/lab0$ gcc read_data.c -o read_data -m32 -static -g -Wno-deprecated-declarations
-fno-stack-protector

read_data.c: In function ‘read_int’:
read_data.c:9:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-
Wimplicit-function-declaration]
     gets(buf);
     ^~~~
     fgets
read_data.c:10:9: warning: implicit declaration of function ‘atoi’ [-Wimplicit-function-
declaration]
     i = atoi(buf);
         ^~~~
/tmp/ccyG4vR8.o: In function `read_int':
/afs/ee.cooper.edu/user/g/gitzel/lab0/read_data.c:9: warning: the `gets' function is dangerous and
should not be used.
```

Ok so we already get a few warnings, one of which is pointing out that “gets” is dangerous! Although it does not explain why…

Let's try and run the program to see what happens:

```
gitzel@cybersec0:~/lab0$ ./read_data
123
x=123
```

Seems normal enough. What about large input?

```
gitzel@cybersec0:~/lab0$ ./read_data
123123451823450128346501931346134613461346113461
x=2147483647
```

Surprising, but it makes sense since int32 can only store up to $2^{31} - 1$.

What about non-numerical input?

```
gitzel@cybersec0:~/lab0$ ./read_data
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
x=0
```

Ok, fair enough, it rejects our A's.

What if we paste in lots and lots of bytes?

```
gitzel@cybersec0:~/lab0$ ./read_data
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

Now we a get a crash. Looks like an illegal memory access occurred (segmentation fault).

Using the Debugger

Now that we've played around with the program, let's attach the debugger to see what we can understand about how it operates.

```
gitzel@cybersec0:~/lab0$ gdb ./read_data

GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./read_data...done.

(gdb) help     <------- This is the gdb console. “help” will show you some common commands
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous. <------- this means “b” and “r” are valid
commands
```

Let’s set a breakpoint and pause execution inside the “read_int” function.

```
(gdb) b read_int
Breakpoint 1 at 0x80488ba: file read_data.c, line 9.
```

And then run the program:

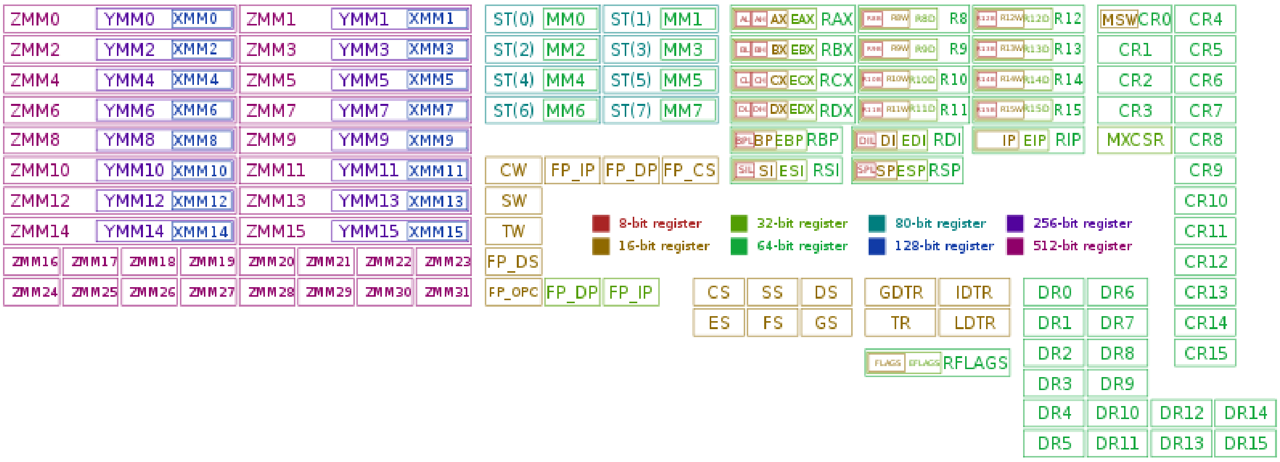
```
(gdb) r
Starting program: /afs/ee.cooper.edu/user/g/gitzel/lab0/read_data

Breakpoint 1, read_int () at read_data.c:9
9           gets(buf);
```

So we can see exactly where the program has stopped. But let’s take a closer look. If we want to understand the memory of the program, we’ll have to look at the registers.

```
(gdb) info reg
eax            0x80da8c0     135112896
ecx            0xffffdb30    -9424
edx            0xffffdb84    -9340
ebx            0x80d9000     135106560
esp            0xffffda60    0xffffda60       <--------- Stack pointer “SP” register
ebp            0xffffdaf8    0xffffdaf8
esi            0x80d9000     135106560
edi            0x80481a8     134513064
eip            0x80488ba     0x80488ba <read_int+21>
eflags         0x216  [ PF AF IF ]
cs             0x23   35
ss             0x2b   43
ds             0x2b   43
es             0x2b   43
fs             0x0    0
gs             0x63   99
```

History Lesson (x86 Architecture)



Recall from Computer Architecture, that x86 has multiple different types of registers. The earliest Intel 8086 chips used 16-bit registers with four general-purpose ones: AX, BX, CX, DX. Each of these was split into two bytes (the low byte “L” and the high byte “H”). So for example, register AX can be accessed in two pieces: AH and AL each is one byte (or 8 bits).

Two registers are special:

- SP (the stack pointer) points to the “top” of the stack
- BP (the base pointer) often points to a special place in the stack: right above the local variables
  - This is often used as a “frame pointer”, since stack frames may not all be the same size, popping a frame might not be as simple as decrementing SP by a fixed value

There are four address registers: SI, DI, BX, and BP (already mentioned) and a FLAGS register for things like the carry flag or zero flag (remember your DLD!).

We compiled with “-m32” so this is a 32-bit program. That is why we see the 32-bit registers in the GDB output. Now, 32-bit registers are prefixed with “E” for “extended” since they are twice as big as 16-bit registers. Just like AX had AL (for the lower byte), in 32-bit mode AX corresponds to the lowest two bytes of EAX. The “extension” is considered to be the upper two bytes (aka most-significant 16 bits). This is the same for other registers: SP is the lower half of ESP, etc.

64-bit registers are prefixed with “R”. For example, recompiling for x86-64 will result in a different set registers being displayed:

```
(gdb) info reg
rax            0x401bd8          4201432
rbx            0x400470          4195440
rcx            0x43f460          4453472
rdx            0x7fffffffe708    140737488348936
rsi            0x7fffffffe6f8    140737488348920
rdi            0x1               1
rbp            0x7fffffffe5a0    0x7fffffffe5a0
rsp            0x7fffffffe510    0x7fffffffe510
```

Disassembling Code

So what’s actually going on in x86 land? To understand how the processor is executing our C code we can disassemble the code pointed to by the Instruction Pointer register (EIP).

```
(gdb) disass $eip
Dump of assembler code for function read_int:
   0x080488a5 <+0>:    push   %ebp          <---- These instructions are saving the previous base
pointer EBP ans setting EBP to point to that save. This is called the “assembly function prologue”
   0x080488a6 <+1>:    mov    %esp,%ebp
   0x080488a8 <+3>:    push   %ebx
   0x080488a9 <+4>:    sub    $0x94,%esp          <---- So here we see a number subtracted from the
```

```
SP. The program is making space for the local variables.
   0x080488af <+10>:  call   0x8048780 <__x86.get_pc_thunk.bx>
   0x080488b4 <+15>:  add    $0x9074c,%ebx
=> 0x080488ba <+21>:  sub    $0xc,%esp        <---- Execution is paused at this instruction
   0x080488bd <+24>:  lea    -0x8c(%ebp),%eax
   0x080488c3 <+30>:  push   %eax
   0x080488c4 <+31>:  call   0x8050740 <gets>
   0x080488c9 <+36>:  add    $0x10,%esp
   0x080488cc <+39>:  sub    $0xc,%esp
   0x080488cf <+42>:  lea    -0x8c(%ebp),%eax
   0x080488d5 <+48>:  push   %eax
   0x080488d6 <+49>:  call   0x804e2e0 <atoi>
   0x080488db <+54>:  add    $0x10,%esp
   0x080488de <+57>:  mov    %eax,-0xc(%ebp)
   0x080488e1 <+60>:  mov    -0xc(%ebp),%eax
   0x080488e4 <+63>:  mov    -0x4(%ebp),%ebx
   0x080488e7 <+66>:  leave
   0x080488e8 <+67>:  ret
End of assembler dump.
```

Let’s try and reverse-engineer locations of the variables on the stack.

```
(gdb) print &buf[0]
$1 = 0xffffda6c "ga\t", <incomplete sequence \360>
(gdb) print &i
$2 = (int *) 0xffffdaec
```

We’ll draw a stack diagram to get our bearings in the stack. We’ve got three addresses ESP, &buf[0], and &i.

High Addresses	Value	Notes
????	????	Return Address to main()
	Other Stuff	
ffff daec		i
ffff da6c		buf[127] … buf[0] Note 0xEC – 0x6C is exactly 128 bytes
ffff da60		ESP
Low Addresses		

Let try and find the return address to the main function. By x86 calling convention, we know that the EBP pointer points to the Saved EBP point which is right before (below) the return address. If we examine the contents of that register we see:

```
(gdb) x $ebp
0xffffdaf8:    0xffffdb18
```

And adding the 4 byte offset (remember 32-bit integer memory addresses):

```
(gdb) x $ebp + 4
0xffffdafc:    0x0804890b
```

So now we found the return address! Let’s fill in our stack diagram:

High Addresses	Value	Notes
ffff dafc	0x0804890b	Return Address to main()
ffff daf8	Saved BP	Saved BP
	Other Stuff	
ffff daec	i	

`ffff da6c`	buf[127] ... buf[0]	Note 0xEC – 0x6C is exactly 128 bytes
`ffff da60`		ESP
Low Addresses		

But what is actually living at that address? We can use GDB to disassemble that address:

```
(gdb) disassemble 0x0804890b
Dump of assembler code for function main:
   0x080488e9 <+0>:   lea    0x4(%esp),%ecx
   0x080488ed <+4>:   and    $0xfffffff0,%esp
   0x080488f0 <+7>:   pushl  -0x4(%ecx)
   0x080488f3 <+10>:  push   %ebp
   0x080488f4 <+11>:  mov    %esp,%ebp
   0x080488f6 <+13>:  push   %ebx
   0x080488f7 <+14>:  push   %ecx
   0x080488f8 <+15>:  sub    $0x10,%esp
   0x080488fb <+18>:  call   0x8048780 <__x86.get_pc_thunk.bx>
   0x08048900 <+23>:  add    $0x90700,%ebx
   0x08048906 <+29>:  call   0x80488a5 <read_int>
   0x0804890b <+34>:  mov    %eax,-0xc(%ebp)       <---- Look at that! It's the instruction right
after the call to read_int. So this is where we will continue execution in main.
   0x0804890e <+37>:  sub    $0x8,%esp
   0x08048911 <+40>:  pushl  -0xc(%ebp)
   0x08048914 <+43>:  lea    -0x2d038(%ebx),%eax
   0x0804891a <+49>:  push   %eax
   0x0804891b <+50>:  call   0x804ff10 <printf>
   0x08048920 <+55>:  add    $0x10,%esp
   0x08048923 <+58>:  mov    $0x0,%eax
   0x08048928 <+63>:  lea    -0x8(%ebp),%esp
   0x0804892b <+66>:  pop    %ecx
   0x0804892c <+67>:  pop    %ebx
   0x0804892d <+68>:  pop    %ebp
   0x0804892e <+69>:  lea    -0x4(%ecx),%esp
   0x08048931 <+72>:  ret
End of assembler dump.
```

So now we have a good idea of where we are and where execution will go. Let's get back to our "read_int" function and execute the next instruction:

```
(gdb) next
        <---- GDB is waiting for us to input data. Let's try that nonsense "AAAAA...AAA" string
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
<Enter>
```

Let's see what happens next:

```
10          i = atoi(buf);
```

We see the call to atoi and everything seems OK, for now. So what will happen next? *Think about it for a bit then read on to see if your intuition is right.*

Our stack now looks something like this. This is a problem!

High Addresses	Value	Notes
`ffff dafc`	AAAAAAAA	Return Address to main()
`ffff daf8`	AAAAAAAA	Saved BP
	AAAAAAAA	Other Stuff
`ffff daec`	AAAAAAAA	i
`ffff da6c`	AAAAAAAA ...	buf

	AAAAAAAA	
ffff da60		ESP
Low Addresses		

We can confirm this by printing out the buffer variable:

```
(gdb) print &buf
$3 = (char (*)[128]) 0xffffda6c
(gdb) print &buf[0]
$4 = 0xffffda6c 'A' <repeats 180 times>
```

And we have 180 bytes of “A” which has definitely overrun the 128-byte buffer string. Not great… Let’s take a look at the EBP pointer again:

```
(gdb) x $ebp
0xffffdaf8:    0x41414141
```

Uh oh, looks like it got overwritten with 0x41! A quick look at an [ASCII chart] tells us that 0-x41 is “A”. So we’ve managed to overwrite the saved base pointer. What about the return address?

```
(gdb) x $ebp+4
0xffffdafc:     0x41414141
```

It’s toast. We’ve overwritten the return address as well. So if we continue to execute the program with “next” we will crash when trying to jump to that address.

```
(gdb) next
11          return i;
(gdb) next
12      }
```

Before we jump, let’s take a look at the buffer one last time.

```
(gdb) print &buf[0]
$1 = 0xffffda6c 'A' <repeats 128 times>
```

So what happened here? All of the sudden the string is 128 bytes… Well, remember that “atoi” actually handles non-integer input by return zero. So we now have null-terminated that string (inadvertently) by writing zero above “buf” on the stack.

High Addresses	Value	Notes
ffff dafc	AAAAAAAA	Return Address to main()
ffff daf8	AAAAAAAA	Saved BP
	AAAAAAAA	Other Stuff
ffff daec	0	i
ffff da6c	AAAAAAAA … AAAAAAAA	buf
ffff da60		ESP
Low Addresses		

And now if we execute another instruction we will jump to the 0x4141414 address.

```
(gdb) nexti
0x41414141 in ?? ()
(gdb) info reg
eax            0x0    0
ecx            0xa    10
edx            0x0    0
ebx            0x41414141     1094795585
```

```
esp            0xffffdb00     0xffffdb00
ebp            0x41414141     0x41414141
esi            0x80d9000      135106560
edi            0x80481a8      134513064
eip            0x41414141     0x41414141        <---- We've jumped to that 0x4141414 address
eflags         0x286  [ PF SF IF ]
cs             0x23   35
ss             0x2b   43
ds             0x2b   43
es             0x2b   43
fs             0x0    0
gs             0x63   99
(gdb) nexti
Program received signal SIGSEGV, Segmentation fault.        <---- We get a Segmentation Fault when
we try to execute code outside of the page table for this process (memory protection)
0x41414141 in ?? ()
```

Fun with Addresses

So now that we can overwrite the different memory locations we can jump around in memory to different locations of our choosing. For a simple example, we can manually change the return address to jump to "printf" in the program.

```
(gdb) disass main
Dump of assembler code for function main:
   0x080488e9 <+0>:   lea    0x4(%esp),%ecx
   0x080488ed <+4>:   and    $0xfffffff0,%esp
   0x080488f0 <+7>:   pushl  -0x4(%ecx)
   0x080488f3 <+10>:  push   %ebp
   0x080488f4 <+11>:  mov    %esp,%ebp
   0x080488f6 <+13>:  push   %ebx
   0x080488f7 <+14>:  push   %ecx
   0x080488f8 <+15>:  sub    $0x10,%esp
   0x080488fb <+18>:  call   0x8048780 <__x86.get_pc_thunk.bx>
   0x08048900 <+23>:  add    $0x90700,%ebx
   0x08048906 <+29>:  call   0x80488a5 <read_int>
   0x0804890b <+34>:  m ov    %eax,-0xc(%ebp)
   0x0804890e <+37>:  sub    $0x8,%esp
   0x08048911 <+40>:  pushl  -0xc(%ebp)
   0x08048914 <+43>:  lea    -0x2d038(%ebx),%eax        <---- Let's jump here right before printf
   0x0804891a <+49>:  push   %eax
   0x0804891b <+50>:  call   0x804ff10 <printf>
   0x08048920 <+55>:  add    $0x10,%esp
   0x08048923 <+58>:  mov    $0x0,%eax
   0x08048928 <+63>:  lea    -0x8(%ebp),%esp
   0x0804892b <+66>:  pop    %ecx
   0x0804892c <+67>:  pop    %ebx
   0x0804892d <+68>:  pop    %ebp
   0x0804892e <+69>:  lea    -0x4(%ecx),%esp
   0x08048931 <+72>:  ret
```

Let's manually set ESP and execute a different instruction:

```
(gdb) set {int}$esp=0x08048914
(gdb) c
Continuing.
x=1

Program received signal SIGSEGV, Segmentation fault.        <---- And we still crash?
```

So why did this happen? *Think about it for a bit then read on to see if your intuition is right.*
printf has to "ret" (return) at some point as well. But we didn't setup that return address. If we look at the registers when we're in printf:

```
(gdb) info reg
eax            0x413e7109     1094611209
ecx            0xa    10
edx            0x0    0
ebx            0x41414141     1094795585
esp            0xffffdaf8     0xffffdaf8
ebp            0x41414141     0x41414141

(gdb) x $ebp+4
```

```
0x41414145:    Cannot access memory at address 0x41414145
```

Our EBP+4 address is out of bounds and will cause a Segmentation Fault.

References

- Official GDB user manual from the GNU Project
- Internals guide to technical details of GDB
- Other documents on GDB resources site