Bomb Defusing with Gdb

COMP2100 Revised 25 September 2020

Introduction

The Binary Bomb Lab is designed to help you understand machine code and how the machine executes your programs. In order to solve this lab, you will need to become familiar with the debugger `gdb`. You will disassemble code and step through your program one instruction at a time. You cannot get much closer to the machine than by watching it execute each instruction and seeing what happens!

The Binary Bomb Lab (also known as Bomb Lab) involves finding the input strings that a program requires in order to not call a procedure that "explodes" the bomb. The bomb is divided into six phases, and each phase processes a single text line. Some phases convert the text to numbers while other phases process the text as a string.

This document contains an outline of how to go about defusing the bomb, followed by techniques and tips for successfully using `gdb` to defuse the bomb.

`gdb` is the GNU debugger, a program that allows you to control the execution of another program (in this case, your bomb), and to examine and even modify the contents of memory and registers as you step through the program one instruction at a time. You can perform every task required for Bomb Lab inside `gdb`, provided that you know how to use it properly.

Before going into details about `gdb`, however, we will outline the processes that you might find helpful to defuse your bomb, one phase at a time.

Debugging Key Concepts

This section briefly describes some key ideas that are found in most debuggers, including `gdb`.

Breakpoint. A marker is attached to an instruction so that, when the code attempts to execute the instruction, execution is paused and the debugger takes control. You can then examine the contents of registers and memory, and decide to continue execution or not.

Disassemble. Converting machine code into assembly representation so that the user can see what instructions are in the program. You can disassemble the current instruction, the current function, a named function or any range of memory addresses.

Stepping. Execution can be stepped, taking one instruction at a time. When a procedure call is involved, there are three important variations of stepping.

1. You can **step into** a procedure, executing the `call` instruction and then pausing immediately. This is useful if you want to debug the called procedure. The `gdb` command to step into a procedure is `stepi`.
2. You can **step over** a procedure, executing the procedure call and waiting for the procedure to return before pausing execution. This is useful if you don't want to bother with the details of the called procedure. The `gdb` command to step over a procedure is `nexti`.
3. You can **step out** of the current procedure, pausing execution when it returns to the caller. The `gdb` command to step out of a procedure is `finish`. Use this command if you have

stepped into a procedure but you want to avoid stepping through the instructions inside the procedure.

Examining registers and memory. Contents of registers and memory can be displayed in various ways.

Modifying registers or memory. Values can be stored into registers or memory of the running program.

Watchpoint. The contents of a memory location or register is monitored by the debugger. Execution is paused whenever the contents changes.

Stack trace/backtrace. The procedure call stack is dumped, showing the names (and if possible the parameters) of each procedure call leading to the current instruction. Typically, you can choose to debug the contents of variables in any of the active procedures.

Function call. You can call any function within your running program at any point.

Overview of the Bomb Main Program

The bomb is divided into phases. For each phase, the bomb main program executes the following steps, as you can see by examining `bomb.c`, the source code file that is provided to you.

1. A line of text is read from the named input file or from `stdin` (see the lab specification).
2. For some phases, the line of text is parsed into numbers using `sscanf`.
3. The line of text or the parsed numbers are passed to the bomb phase function. If the phase is not satisfied with the input it will call a function to explode the bomb.
4. The return status of the bomb phase function is reported for marking by calling a procedure and a congratulatory message is printed on the screen.

You should read the source code file `bomb.c` and refer to it from time to time because it contains helpful information, both in the source code and in the comments.

Methods for Defusing a Bomb Phase

This section describe several useful methods:

1. Prevent the bomb from exploding
2. Examine the main program for the phase you are working on
3. Disassemble the phase you are working on
4. Step through execution of the phase
5. Reverse engineer the phase
6. Earn marks for your phase solution

1. Prevent the bomb from exploding

The bomb will “explode” whenever a phase detects unsatisfactory input. This is most likely to happen when you don’t know how to defuse the phase, but it can also happen if you accidentally mistype the input to a phase that you have solved, or if you ask the bomb to read the wrong input file. You should guard against such mistakes. One of the best ways to protect yourself is to only run the bomb inside `gdb`, and to always set breakpoints so that the bomb cannot explode unexpectedly. The section “Breakpoint” in “Gdb Features” (below) describes how to work with breakpoints, and the section “Executing a gdb command automatically” is an advanced section that describes how

you can initialise `gdb` so that one or more breakpoints or other commands will be executed every time you start `gdb`.

>  In order to effectively protect yourself, you really want to stop the bomb from exploding. You can set a breakpoint at any function in the program. A logical place to set the breakpoint is the function that the bomb calls to explode itself. You can find out what that function is by disassembling the bomb phase functions. Each bomb is different, so be sure to disassemble your own bomb.

2. Examine the main program for the phase you are working on

The main program has comments that may hint at what the phase is doing. Also, the main program code may indicate what the input should look like at a high level. For example, if the main program parses the input line using `sscanf` then you can easily deduce what kind of information should be provided in the input line.

3. Disassemble the phase you are working on

Use `gdb` to disassemble the phase that you are trying to solve. There will be some parts of the code that you can understand, using what you learned in lectures. Spending some time thinking about the phase will be important as you work through it. For now, focus on what you **do** understand, rather than what you don't. Look for familiar structures such as:

- The parameter registers %rdi, %rsi, %rdx, %rcx, %r8, %r9. Relate these registers to the parameters listed in `bomb.c` for the call to the phase you are working on.
- Arithmetic and bit-wise operations.
- Comparisons, and conditional branch, `set` or conditional move instructions.
- If-then-else and loop constructs.
- A switch statement.
- In the more advanced phases, you may encounter simple data structures or collections of variables stored in memory.

4. Step through execution of the phase

4.1. Preparation

You can achieve a lot by stepping through the execution of the bomb phase. In order to do this, you need to set a breakpoint at the start of the phase and then start the bomb running. When execution reaches the breakpoint, `gdb` will pause execution and you can take over and step through the procedure one instruction at a time. More information about working with breakpoints can be found in the "Gdb Features" section below.

>  Remember to protect yourself by also setting a breakpoint to prevent the bomb from exploding. Gdb allows you to have as many breakpoints as you wish.

If you have already solved earlier phases, put your solutions in a text file and run the bomb with that text file as the parameter. For example, if you are working on phase 2, you could put the solution of phase 1 in a file called "`sol.txt`" and then start the bomb running with the command:

```
(gdb) run sol.txt
```

The bomb will read input from `sol.txt` until it reaches the end of the file, then it will read from `stdin`.

When the bomb prompts you and pauses waiting for your input for the stage you are working on, you won't know what the solution is. However, you can type a *guess*. If the phase requires numbers as input, then type some numbers and press Enter. If the phase requires text, then type some characters and press Enter. The bomb will then attempt to process your input.

You can make a boring guess such as "0 0 0" for three numbers, or "aaaaaaaaa" for some characters, but such a guess is not going to be particularly helpful because later on you won't be able to tell the difference between the first input number, the second or the third. So choose numbers or characters that are different from each other[1].

Choose input numbers or characters that you can recognise when you are stepping through the program.

After you enter your input, the bomb will continue running until it encounters the breakpoint that you set. At this time, it will be helpful to disassemble the phase so that you can see what instruction is about to be executed.

As you step through the phase, you can disassemble it and see where you are up to. It might also help to have a printout of the disassembly of your phase.

4.2. Examine the contents of registers and memory

When the program is paused, you can examine the contents of registers and memory. The relevant `gdb` commands are as follows. For more information, see the section "Gdb Features".

`(gdb) info reg`

Prints out the contents of all the general-purpose registers.

When you first enter a function, use `info reg` to check the parameters.

`(gdb) x/`*`FMT ADDRESS`*

Examine the contents of memory starting at *ADDRESS*. The optional *FMT* specifies the format for displaying the information, and how much information is to be displayed.

`(gdb) p/`*`FMT EXPRESSION`*

Print the value of an expression *EXPRESSION*, with an optional format specifier. The individual registers are known to `gdb`, although their names begin with a $ sign: `$rax, $rcx`, etc.

4.3. Step through the phase

Before executing an instruction of the phase, you want to be sure that it is not going to explode the bomb, so always make sure that you know what instruction will be executed next.

[1] For numeric input, it makes sense to choose numbers that are not simply related to each other. Some prime numbers would probably be more useful than "1 2 3".

You can tell `gdb` to always show you the next instruction to be executed by using the following command.

```
(gdb) display/i $pc
```

This command instructs `gdb` to print the instruction at the program counter whenever execution is paused.

To execute a single instruction, use the `gdb` command `stepi` which can be abbreviated to `si`.

As you step through the program, your purpose is to understand what is happening. When the program executes an instruction that affects a register, you should print the new value of the register and make notes about what you see happening. Use `info reg` regularly to be sure that you are tracking the changes to the registers.

4.4. Conditional instructions – the important decision points

At some point you will encounter a conditional branch, `set` or conditional move instruction. These are the places where the bomb program is making decisions about what to do. If you are stepping through the phase function, these decisions typically relate to whether the bomb will explode or not, or what mark your solution earns for the phase. However, loops and if-then-else constructs in the phase computation may also generate conditional branches, `set` or conditional move instructions.

When you encounter a conditional branch, look at the code to see whether taking the branch or *not* taking the branch is likely to cause the bomb to explode. For example, consider the following code snippet from a possible bomb:

```
0x00000000004012ff <+15>:    cmpl   $0xf,%edi
0x0000000000401303 <+19>:    je     0x401313 <bombphase_1+35>
0x0000000000401305 <+21>:    callq  0x4016ac <boom_boom>
```

The conditional branch here is `je` – jump equal. If the branch is *not* taken, execution will continue to the next instruction which, it appears, would explode the bomb. This means that the jump needs to be taken in order to avoid exploding the bomb. Looking at the compare instruction that sets the condition codes for the branch, the value in `%edi` needs to be `0xf` in order for the jump to be taken.

Sometimes, stepping through the code in this way will enable you to work out what input data you need to provide. In the above example, if `%edi` contained a number that was parsed from the input line, then entering the number as decimal 15 (corresponding to `0xf`) would avoid exploding the bomb at this point. You can determine whether the register `%edi` is directly or indirectly dependent on your input by examining the registers and by looking back over your notes of what has happened to the registers in the code leading up to this point.

5. Reverse engineer the phase

Reverse engineering software is the process of reproducing a software product by studying the available executable program and its behaviour. To reverse engineer the bomb phase you need to develop a detailed understanding of what it does. Reverse engineering may not be necessary, or there may be parts of the phase that do not need to be reverse engineered for you to achieve your goal, but if you do need to obtain a deeper understanding then you will want to employ reverse engineering techniques. For the binary bomb, the most useful technique is to convert the disassembly progressively into pseudo-C code.

Techniques for this type of reverse engineering have been discussed in lectures. Here we present a simple example: reverse engineering a simple function. The assembly code is as follows:

```
Dump of assembler code for function mixed:
   0x00000000004004d2 <+0>:     sub    $0x8,%rsp
   0x00000000004004d6 <+4>:     add    $0x1,%rsi
   0x00000000004004da <+8>:     shl    $0x4,%rsi
   0x00000000004004de <+12>:    and    $0xf,%edi
   0x00000000004004e1 <+15>:    or     %rsi,%rdi
   0x00000000004004e4 <+18>:    mov    $0x0,%eax
   0x00000000004004e9 <+23>:    cmp    $0x41,%rdi
   0x00000000004004ed <+27>:    je     0x4004fb <mixed+41>
   0x00000000004004ef <+29>:    callq  0x4004c4 <boom_boom>
   0x00000000004004f4 <+34>:    mov    $0xffffffffffffffff,%rax
   0x00000000004004fb <+41>:    add    $0x8,%rsp
   0x00000000004004ff <+45>:    retq
End of assembler dump.
```

The first step is to annotate each line of assembly with pseudo-C, using the register names as variables. Stack accesses also correspond to pseudo-C variables when stack space is used as temporary variables or procedure parameters. Use assembly labels as pseudo-C labels, and use `goto` statements to represent assembly branches. For example:

```
mixed:
   sub    $0x8,%rsp                  # Reserve 8 bytes on stack
   add    $0x1,%rsi                  # rsi += 1
   shl    $0x4,%rsi                  # rsi <<= 4
   and    $0xf,%edi                  # edi &= 0xf
   or     %rsi,%rdi                  # rdi |= rsi
   mov    $0x0,%eax                  # rax = 0
   cmp    $0x41,%rdi                 # flags = compare rdi, 65
   je     L1                         # if rdi == 65 goto L1
   callq  0x4004c4 <boom_boom>       # boom_boom()
   mov    $0xffffffffffffffff,%rax   # rax = -1
L1:
   add    $0x8,%rsp                  # Release stack space
   retq                              # Return rax
```

At this point, if the function prototype is available then you can find out the types of the parameters. Otherwise, you can guess the parameter types based on how they are used. In this case, the computation does not use the parameters as pointers, but does perform integer arithmetic and bit-wise operations so it is reasonable to assume that the parameters are integers. The 64-bit registers suggest that the parameters are `long int` or `unsigned long int` – there is no particular reason to prefer one interpretation over the other. The use of a 32-bit `and` instruction (`andl`) with register `%edi` seems strange until we recall that writing a 32-bit register on x64 sets the high bits of the 64-bit register to zero. Since the bit mask `0xf` has no high bits set, the 32-bit `andl` operation is exactly equivalent to a 64-bit `andq` in this case.

In fact, the C function prototype is

```
long mixed(long a, long b)
```

Knowing the parameter names, we can use them in place of register names in our pseudo code. Also, we can replace references to registers with temporary variables, as follows. Using a new name for each new assignment makes it clear that each time the register is used it can have a different purpose, or even a different data type[2], but you do need to be careful not to become confused in the renaming process.

```
long mixed(long a, long b) {
  long bplus1 = 1 + b;           // rsi += 1
  long shift4 = bplus1 << 4;     // rsi <<= 4
  long anded = a & 0xf;          // edi &= 0xf
  long value = shift4 | anded;   // rdi |= rsi
  long result = 0;               // rax = 0
  if (value == 65) goto L1;      // if rdi == 65 goto L1
    boom_boom();                 // boom_boom()
    result = -1;                 // rax = -1
L1:
  return result;                 // return rax
}
```

From here on, it is a matter of simplifying the code by substituting the computation of temporary variables into the corresponding expressions. For example, where `bplus1` appears in the expression for `shift4`, replace `bplus1` with the expression `1 + b`. Also, "if-goto" can be simplified to an ordinary "if-then" block by negating the condition.

```
long mixed(long a, long b) {
  long shift4 = (1 + b) << 4;
  long value = shift4 | (a & 0xf);
  long result = 0;
  if (value != 65) {
    boom_boom();
    result = -1;
  }
  return result;
}
```

Further substitution yields

```
long mixed(long a, long b) {
  long value = ((1 + b) << 4) | (a & 0xf);
  if (value != 65) {
    boom_boom();
    return -1;
  }
  return 0;
}
```

[2] This approach is similar to the concept of "Single Assignment" used in compiler technology.

The goal of this exercise is to produce a clear and concise representation of the program as C code, so it is up to you when to stop substituting variables into their definitions. However, you should probably avoid substituting the same variable in more than one place. For example:

```
long sum = (a + b) + c;
long square = sum * sum;
```

is probably easier to understand than

```
long square = ((a + b) + c) * ((a + b) + c);
```

5.1. Check your reverse engineering

You can check your reverse engineering by writing a proper C function and compiling it. The compiled result will only be the same as the original if you use the same version of the same C compiler in the same Operating System, and the same compilation options.

The 2019 binary bombs are compiled under Linux using gcc 5.4.0 with the options:

```
gcc –m64 -Wall –O1 –fno-stack-protector –c file.c
```

Taking the above example of reverse engineering "mixed", we compile the reverse engineered source code.

```
$ gcc -O1 –fno-stack-protector -S reversed.c
$ cat reversed.s
        .file   "reversed.c"
        .text
.globl mixed
        .type   mixed, @function
mixed:
.LFB0:
        .cfi_startproc
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        addq    $1, %rsi
        salq    $4, %rsi
        andl    $15, %edi
        orq     %rsi, %rdi
        movl    $0, %eax
        cmpq    $65, %rdi
        je      .L3
        call    boom_boom
        movq    $-1, %rax
.L3:
        addq    $8, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   mixed, .-mixed
        .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-17)"
        .section        .note.GNU-stack,"",@progbits
```

Ignoring the directives for the debugger, the assembly code is a perfect match for the original disassembly, although you might think that there was a difference between the two if you forgot that logical and arithmetic left shifts (`shl` and `sal`) are the same.

Unfortunately, ash/iceberg runs a newer version of gcc than the automarking system. Due to the differences between the gcc versions, there may be some assembly code differences if you compile your reverse engineered code. It will require some skill to be sure that the two pieces of assembly code actually do the same thing.

5.2. Solve the phase puzzle

When you have reverse engineered the phase, you need to find a solution to the phase puzzle – how to avoid exploding the bomb or how to achieve the return value that you want to achieve. In the example above, this means finding values of a and b such that

```
(((1 + b) << 4) | (a & 0xf)) == 65
```

This requires some thought and understanding of the operators involved. Since "`1 + b`" is bit shifted to the left by 4 bit positions, it only contributes to the top 60 bits of the 64-bit value, and the bottom four bits are derived exclusively from "`a & 0xf`" which can only contribute to the bottom four bits because of the anding mask which zeroes all but the bottom four bits from a.

Converting 65 to binary we get 0100 0001. Therefore,

- `a & 0xf == 1` so `a == 1` is a solution
- `((1 + b) << 4) == 64` so `(1 + b) == 4` so `b == 3` is a solution

Exercise: Try it for yourself! Write a short C program that prints the value calculated by the following expression with a assigned to 1 and b assigned to 3:

```
value = ((1 + b) << 4) | (a & 0xf);
```

6. Earn marks for your phase solution

When you have a solution, provide it as input to the running bomb and carefully let it run through the stage. The bomb automatically reports both explosions and successful defusions. However, this reporting only happens if you permit the relevant code to execute. You don't want to explode the bomb, but you do want to report successes. This means that when you have successfully defused a phase you need to let the bomb program run at least to the point in `bomb.c` where it has called the procedure to report the successful defusion.

You can set a breakpoint wherever you like to stop execution after that point, but you won't get the mark for defusing the stage if `phase_defused` is not permitted to run to completion. To be sure, let the main program print the congratulatory message.

6.1. Invalid solutions

Gdb is so powerful that you can fool the program into thinking that you have solved the bomb without actually entering the solution. For example, you can change the values in registers so that conditional instructions are taken or not taken as you wish. This won't be effective. We have automatic ways of validating your solution string, and fooling you bomb won't end up earning marks for you even if it appears successful at first.

6.2. Good luck!

With a combination of stepping through the phases and reverse engineering it should be possible to fully solve every phase of the bomb. Sometimes it helps to make a lucky guess, so good luck!

Gdb Features

This section details useful features of `gdb`. For more information, see the `info gdb` page, `man` page and built-in `help`. Also, there are numerous `gdb` tutorials and discussions available on the Internet.

Starting gdb

`gdb` is a powerful command-line debugger that can run your program, stop it at any point, and allow you to examine the contents of memory, the stack and registers. If you run your bomb under the control of `gdb` then you can stop the bomb from exploding. To start `gdb` with your bomb, issue the following system command:

```
$ gdb bomb
```

This does not run your bomb, it simply puts `gdb` in charge of the bomb. You can then use commands inside `gdb` to examine your bomb, set breakpoints, start it running, step through it one instruction at a time, etc.

Help

`gdb` includes its own help facility. Within `gdb`, type the help command as follows. Note that `(gdb)` is the `gdb` command prompt – you should only type the word "`help`" and press the Enter key.

```
(gdb) help
```

The `help` command without any parameters prints a list of the different classes (groups) of commands supported by `gdb`. You can then select one of those command classes to obtain a list of all the command related to that aspect of `gdb`. For example, all the commands related to *breakpoints* are listed by

```
(gdb) help breakpoints
```

Finally, for help on a specific command, type the command name as the parameter to `help`. For example, to obtain help on the `break` command that sets a breakpoint:

```
(gdb) help break
```

Breakpoint and stepping

A breakpoint is a debugging feature that pauses execution when a certain place is reached in the program. You can then use debugging commands to examine the state of registers and memory, and you can choose to continue execution or not. You can use a breakpoint to run your program and pause it at a point where you want to find out what is happening, or to pause it before executing code that you don't want to be executed.

The breakpoint command is:

```
(gdb) break [LOCATION]
```

The optional parameter *LOCATION* specifies where you want execution to pause. Two particularly useful breakpoint options are to pause execution at the start of a procedure, and to pause execution at the address of a particular instruction.

To pause execution at the start of a procedure, name the procedure as the parameter of the break command. For example, to pause at the start of the procedure `bombphase_1`, use the following command:

```
(gdb) break bombphase_1
```

The bomb program calls a procedure to explode the bomb. If you can find out what that procedure is called, you can use the break command to protect yourself against accidents by pausing execution before the code that explodes the bomb gets executed.

To pause execution at a particular instruction address, use the following notation:

```
(gdb) break *address
```

For example, to break at address 0x401c3, the command would be

```
(gdb) break *0x401c3
```

When a breakpoint is reached, execution pauses. You can then examine registers and memory, and you can single step through instructions, or set another breakpoint, and you can continue execution.

To continue execution:

```
(gdb) cont
```

This command can be abbreviated to 'c':

```
(gdb) c
```

This command continues execution after the breakpoint. The program will resume executing until it either hits a break point or exits.

Warning: If you continue execution after a breakpoint at the start of the function that explodes the bomb then that function will execute and the bomb will explode. Only continue execution when you know that the code that follows is safe, at least up to the next breakpoint.

To single step – execute exactly one instruction and then pause:

```
(gdb) stepi
```

This command can be abbreviated to '`si`':

```
(gdb) si
```

The `stepi` (or `si`) command steps *into* a procedure call.

Step over procedure call. If the instruction is a procedure call and you want the procedure to be called and return before pausing at the next instruction, use the `nexti` command. This command steps *over* a procedure call.

```
(gdb) nexti
```

This command can be abbreviated to '`ni`':

```
(gdb) ni
```

Step out of a procedure. The command `finish` continues execution until the current procedure returns. This is useful if you step into a procedure that you don't really want to try to understand, and you know that it won't explode the bomb. Use this command to continue execution until the program returns to the calling procedure, then you can single step through that procedure.

```
(gdb) finish
```

Run the program

The `gdb` command `run` commences execution of the program under control of the debugger. You can specify command line parameters to the program as parameters of the `run` command. For example, to run the bomb with a partial solution file `sol.txt`, use the command

```
(gdb) run sol.txt
```

If the program is already running, the `run` command will terminate it without permitting it to execute any more instructions, and then start it again.

Disassembly

The `gdb` command `disassemble` (which can be abbreviated to `disass`) will display the assembly code corresponding to the instructions of a function that you name. For example, here is part of the disassembly of `bombphase_1` in a particular bomb. To disassemble an entire function, simply type "`disass` " and the name of the function:

```
(gdb) disass bombphase_1
Dump of assembler code for function bombphase_1:
=> 0x00000000004012f0 <+0>:     sub    $0x8,%rsp
   0x00000000004012f4 <+4>:     cmpb   $0x4a,0x1(%rdi)
...
   0x000000000040132a <+58>:    retq
End of assembler dump.
```

The `disassemble` command has options to disassemble code around the current program counter, around a specific address or for a specific range of addresses. Disassembling an entire function, as shown above, is one of the most useful features.

Display the next instruction to be executed

The `x` (examine) command can display instructions using the format option `/i`. To display the next instruction that will be executed, examine the instruction at the program counter `$pc` as follows:

```
(gdb) x/i $pc
```

The `display` command instructs `gdb` to automatically print an expression value or examine memory every time execution is paused. A particularly useful form of this command instructs `gdb` to display the next instruction each time it pauses execution, as follows.

```
(gdb) display/i $pc
```

For example

```
(gdb) display/i $pc
1: x/i $pc
=> 0x4012f4 <bombphase_1+4>:   cmpb  $0x4a,0x1(%rdi)
(gdb) si
0x00000000004012f8 in bombphase_1 ()
1: x/i $pc
=> 0x4012f8 <bombphase_1+8>:   jne   0x401305 <bombphase_1+21>
(gdb)
```

Note how the next instruction to be executed is displayed after each step is taken with the `si` command.

Examining registers and memory

Examining the registers

The `info reg` command displays all the register values, in hexadecimal and decimal.

```
(gdb) info reg
rax            0x3ab3b90f80     252123352960
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffe4d8   140737488348376
rsi            0x7fffffffe4c8   140737488348360
rdi            0x1      1
rbp            0x0      0x0
rsp            0x7fffffffe3e8   0x7fffffffe3e8
r8             0x3ab3b8fba0     252123347872
r9             0x3ab340eba0     252115479456
r10            0x7fffffffe230   140737488347696
r11            0x3ab381ec20     252119739424
r12            0x4003e0 4195296
r13            0x7fffffffe4c0   140737488348352
r14            0x0      0
r15            0x0      0
rip            0x400500 0x400500 <main>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

In addition to the 16 general-purpose registers, the command shows the program counter/instruction pointer (`rip`) and the condition codes (`eflags`). It lists which flag bits are set – you should recognise ZF, CF, SF and OF when they appear. There are other flags that we do not consider in COMP202. Also, the register dump includes some special purpose registers that we do not consider in COMP202.

Print an individual register value

The `gdb` command `print` can print out the value of any register. The register name is similar to the hardware name, except that it is prefixed by $ instead of %. For example, here is one way to print the value of `%eax`, as decimal, hex (`/x` option), octal (`/o` option) and binary (`/t` option):

```
(gdb) p $eax
$3 = -1279717504
(gdb) p/x $eax
$4 = 0xb3b90f80
(gdb) p/o $eax
$5 = 026356207600
(gdb) p/t $eax
$6 = 10110011101110010000111110000000
```

Print the program counter (`rip`)

```
(gdb) p $rip
$7 = (void (*)()) 0x400500 <main>
```

Print the condition codes (`eflags`)

```
(gdb) p $eflags
$9 = [ PF ZF IF ]
```

In this example, the zero flag `ZF` is set and the other familiar condition codes are not set.

Examine memory

The `x` command examines data in memory. Specify the starting address and the format for dumping the data. The format consists of the number of items to be examined, the print format (hexadecimal, etc), and the size of the objects. The size letters are: '`b`' for byte, '`h`' for halfword (i.e. 2 bytes), '`w`' for word (i.e. 4 bytes) and '`g`' for giant word (i.e. 8 bytes). The following example shows the first 8 bytes of `main` examined as bytes, halfwords, words and a single giant word, all in hexadecimal.

```
(gdb) x/8xb 0x400500
0x400500 <main>:        0x48    0x83    0xec    0x08    0x8d
0x77    0x03    0x48
(gdb) x/4xh 0x400500
0x400500 <main>:        0x8348  0x08ec  0x778d  0x4803
(gdb) x/2xw 0x400500
0x400500 <main>:        0x08ec8348      0x4803778d
(gdb) x/1xg 0x400500
0x400500 <main>:        0x4803778d08ec8348
```

Here are some other formats for the same memory: as an instruction (`/i`) which is what it actually is; as decimal giant word (`/dg`), as binary byte (`/tb`) and as a string (`/s`).

```
(gdb) x/i 0x400500
=> 0x400500 <main>:     sub    $0x8,%rsp
(gdb) x/dg 0x400500
0x400500 <main>:        5189122643284755272
(gdb) x/tb 0x400500
0x400500 <main>:        01001000
(gdb) x/s 0x400500
0x400500 <main>:
"H\203\354\b\215w\003Hc\366Hc\377\350\300\377\377\377H\203\304
\bÐ\220\220\220\220\220\220\220\220\363\303fffff.\017\037\204"
```

You can also examine memory at a named symbol. For example, to print a character array with the symbol name "`wise_saying`" we can use the following command. Note that the `&` operator is being used to get a pointer to the character string.

```
(gdb) x/s &wise_saying
0x603900 <wise_saying>:  "Education is what remains after one
has forgotten what one has learned in school. - Albert
Einstein"
```

Examine symbols

The `gdb` command `info variables` lists all the compiled variables that `gdb` knows about. You can then examine the data at those locations. For example:

```
(gdb) info variables
All defined variables:

File bomb.c:
FILE *infile;

Non-debugging symbols:
0x0000000000402318  _IO_stdin_used
0x0000000000402320  __dso_handle
0x00000000004031f8  __FRAME_END__
0x00000000006031fc  __init_array_end
0x00000000006031fc  __init_array_start
0x0000000000603200  __CTOR_LIST__
0x0000000000603208  __CTOR_END__
0x0000000000603210  __DTOR_LIST__
0x0000000000603218  __DTOR_END__
0x0000000000603220  __JCR_END__
0x0000000000603220  __JCR_LIST__
0x0000000000603228  _DYNAMIC
0x00000000006033c0  _GLOBAL_OFFSET_TABLE_
0x00000000006034e0  __data_start
0x00000000006034e0  data_start
0x0000000000603500  cow
0x0000000000603900  wise_saying
...
```

The gdb command `info symbol` can be used to inquire about a procedure or function symbol (name). E.g.

```
(gdb) info symbol main
main in section .text
```

Interrupt

The control-C keyboard interrupt can be used to interrupt the running program and return control to the debugger. You can then find out where execution has been paused in the program.

Quit gdb

The gdb command `quit` exits the debugger. The program that was being debugged is terminated without being permitted to execute any more instructions.

```
(gdb) quit
```

Executing gdb commands automatically

It is possible to get gdb to execute one or more commands automatically every time it starts. This can be used to set one or more breakpoints every time you run gdb.

There are several ways to have gdb execute some command or commands automatically each time it starts. The simplest way is to create a `.gdbinit` file in your home directory.

Create a `.gdbinit` file in your home directory to have gdb automatically perform commands that you want done every time it starts.

File names that begin with a dot ('.') are commonly used in Unix for configuration files. The `ls` command does not normally show files whose names begin with a dot. To see them, use the option `-a` on the `ls` command:

```
$ ls -a
```

The content of the `.gdbinit` file can be any sequence of commands that you want executed. However, the commands are executed before your program is loaded, so they cannot display information from within the program. As a special feature, you can set breakpoints in the `.gdbinit` file, but you have to tell gdb that you want those breakpoints to take effect later on when gdb has loaded your program. The command `set breakpoint pending on` tells gdb to remember breakpoints and apply them when it loads the program.

Here is an example `.gdbinit` file that sets a breakpoint at the start of bomb phases 1 and 2 every time gdb is started to debug the bomb.

```
set breakpoint pending on
break bombphase_1
break bombphase_2
```

If you want to execute other commands every time you start `gdb`, such as examining data in the program, then you cannot use `.gdbinit` for the commands because the executable program is not loaded when `gdb` processes `.gdbinit`. The `display` command also does not work in `.gdbinit`. Gdb has a command line option `-x` which can be used to execute commands from a file, and `-ex` can be used to execute a `gdb` command specified on the command line.

Index