CIS 450 -- Computer Organization and Architecture – Spring 2018

Programming Lab Assignment 2: Diffuse the Binary Bomb Lab

Lab Dates: Feb. 9, 12, 16, 23, Due Date: Mon., Feb. 26, 2018, 11:59 pm

(50 points)

1. Problem Statement

This assignment deals with using the GDB Debugger and friends to analyze code segments and diffuse a binary bomb. The bomb is a 32-bit binary ELF executable, **`kaboom`.** The bomb is also available as a 64-bit binary ELF executable, **`kaboom64`**. For this assignment, you should test your solution on a CS Linux machine – in particular, we will grade the lab using cslinux.cs.ksu.edu.

2. Logistics

The only file to be submitted via K-State OnLine will be your answers in a plain ASCII text file, lab2.txt. As you go through the exercises today, reference the following for help on GDB:

- Complete documentation on GDB is available online at http://sources.redhat.com/gdb/current/onlinedocs/gdb.html.
- While running GDB, type **`help`** or **`help <command>`** for more information.
- The GDB help manual (man) page: $ **`man gdb`**

We will also use the tool **`objdump`** in this lab. You can find help on `objdump` here:

- http://sourceware.org/binutils/docs/binutils/**objdump**.html
- And from `objdump` help output: **`objdump --help`**

3. Tutorial

To get started, follow the steps of this tutorial. You will use the following programs:

- A C program that represents a simple binary bomb, boom.c, and the resulting executable created using gcc with the command: **`gcc -fno-stack-protector -fno-inline -fno-omit-frame-pointer -g -O1 -o boom boom.c`**
- Binary executable files, **`kaboom`** and **`kaboom64`**.

You are encouraged to discuss GDB commands and the steps of this tutorial section with other students. You can find the files used in the lab in **`/pub/cis450/programs/Lab2.tgz`.**

Generating Assembly Code: The `–S` flag

Download the code from /pub/cis450/programs/Lab2:

```
$cd ~/cis450; cp /pub/cis450/programs/Lab2.tgz .; tar xvzf Lab2.tgz; cd Lab2
```

Copy the gzipped tar file from the public directory to our local directory and uncompress (z) and extract (x) files (f). Generate the output by invoking the **`make`** command.

```
$make
gcc -fno-stack-protector -fno-inline -fno-omit-frame-pointer -S -O1 boom.c
gcc -fno-stack-protector -fno-inline -fno-omit-frame-pointer -g -O1 -o boom
```

Note that you don't get the source, kaboom.c, for the binary bomb we will be diffusing for this lab, you already have the executable, **kaboom**. If you get a "Permission denied" error when you try to debug kaboom below, just **add execute (x)** permission using the command: **chmod +x kaboom**

Compiling for GDB: The `-g` flag

In order to create a symbol table for source-level debugging, you must compile with the '`-g`' option. This option **generates** additional debugging information, such as the symbol table, and stores this extra information inside the executable. The –O1 flag tells the compiler to perform Level-1 optimization. The –S flag tells the compiler to generate assembly code in the output file boom.s.

By looking at the source code, **`boom.c`**, determine what valid input could be entered by a user to diffuse the bomb. Execute the program (without GDB), and observe the results. Even if you blow up this test bomb, you will not lose any points. Then, take a look at the assembly code generated for boom.c, and complete the mapping between the function operations and the corresponding assembly code.

```
#define MAGIC_NUMBER 13

..

void phase_1_of_1 () {
   int args, x, y;
   int sum = 0;

   args = fscanf (stdin, "%d %d", &x, &y);
   if (args != 2)
       explode();

   sum = x+y;

   if ((x<=0)||(y<=0))
       explode();

   if (sum != MAGIC_NUMBER)
       explode();
}
```

```
.LC1:
    .string "%d %d"
phase_1_of_1:
.LFB21:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    leaq    -8(%rbp), %rcx
    leaq    -4(%rbp), %rdx
    leaq    .LC1(%rip), %rsi
    movq    stdin(%rip), %rdi
    movl    $0, %eax
    call    __isoc99_fscanf@PLT
    cmpl    $2, %eax
    jne .L10
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    leal    (%rdx,%rax), %ecx
    testl   %edx, %edx
    jle .L8
    testl   %eax, %eax
    jle .L8
    cmpl    $13, %ecx
    jne .L11
    leave
    ret
.L10:
    movl    $0, %eax
    call    explode
.L8:
    movl    $0, %eax
    call    explode
.L11:
    movl    $0, %eax
    call    explode
```

Starting GDB

After you have compiled a C source file with **`-g`**, you can run this program with GDB and obtain a listing of the source code within the debugger; e.g., the source code is embedded inside the executable.

Start debugging boom using the command: $ **`gdb boom`**. The GDB prompt should appear: (gdb)

```
% gdb boom
```

Note that at any time, you can **exit GDB** with the quit command **`quit`**

Running a program in GDB: `run`

You can execute a program within GDB with the command **`run`**

- Execute boom. The program should run normally.
  `(gdb) run`
  Starting program: ..boom..when prompted for input, enter "3 10"

  3 10
  You safely defused the bomb. Well done.
  (gdb) **`quit`**

- **`run args`** sets the command-line arguments to be input to your process.

  Of course, just running the program isn't generally enough to diffuse a bomb. We now look at some of the GDB commands used to examine the program as it executes.

Setting a breakpoint: `break`

You can set a **breakpoint** so that GDB will stop the execution at a certain line in the source code.
Use the command `break`, or the shortcut `b`, as follows:

```
break <function name>
break <linenumber>
break <filename:function name>
break <filename:linenumber>
```

You can also set a breakpoint at the memory address of an instruction:

```
break *<address>
```

You can display all active breakpoints with the command: **`info break`**
To delete a breakpoint, use one of the following:

```
delete <breakpoint number>
```

Set a breakpoint in the first line of the main function in boom and execute this program:

```
$ gdb boom
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1.1) 7.12.50.20170314-git
(gdb) break main
Breakpoint 1 at 0x7fd: file boom.c, line 31.
(gdb) break phase_1_of_1
Breakpoint 2 at 0x79a: file boom.c, line 14.
(gdb) break explode
Breakpoint 3 at 0x780: file boom.c, line 9.
(gdb) run
Starting program: /net/files.cis.ksu.edu/exports/public/cis450/programs/Lab2/boom

Breakpoint 1, main () at boom.c:31
31      int main() {
(gdb) list
26
27         if (sum != MAGIC_NUMBER)
28             explode();
29      }
30
31      int main() {
32         printf("Welcome to the demo bomb.\n");
33         printf ("Phase 1\n");
34         phase_1_of_1();
35         printf("You safely defused the bomb. Well done.\n");
(gdb)cont
Continuing.
```

```
Welcome to the demo bomb.
Phase 1

Breakpoint 2, phase_1_of_1 () at boom.c:14
14      void phase_1_of_1 () {
(gdb) where
#0  phase_1_of_1 () at boom.c:14
#1  0x0000555555554823 in main () at boom.c:34
(gdb) list 14
9       void explode() {
10         printf("KABOOM!!!\n");
11         exit(1);
12      }
13
14      void phase_1_of_1 () {
15         int args, x, y;
16         int sum = 0;
17
18         args = fscanf (stdin, "%d %d", &x, &y);
(gdb) cont
Continuing.
3 10
You safely defused the bomb. Well done.
[Inferior 1 (process 8708) exited normally]
(gdb) quit
```

You can add a breakpoint at any point **before** you have called **`run`** or **after** you have called run and the program is paused.

Stepping through a function, by line: next, step, and continue to examine a program as it executes:

`next` executes one line of source code in the current function.
`step` is similar to `next`, but will **step *inside*** a function that is called on the current line.
`next <N>` and **`step <N>`** increment N lines, rather than just one line.
`cont``inue` executes the program until the next break point

The **`list`** command displays source code around your most recent listing; **`list <number>`** displays code starting at a particular line. At any time, you can use **`continue`** (or just **`cont`**) to proceed automatically to the next breakpoint or until the program terminates.

Displaying data: print and watch
You can use the **`print`** command, or its shortcut **`p`**, to display the value of a variable in the current function. `print` is very flexible; you can dereference pointers, display addresses of variables, and index into arrays with the `*`, `&`, and `[]` operators. A watchpoint is a breakpoint that stops execution whenever the value of a variable or expression changes and displays the old and new values of that expression. Set a watchpoint with the command **`watch <expr>`**

- Start boom from the beginning; **`gdb boom`**.
- Set a breakpoint at the **`phase_1_of_1`** function, and run the program: **`run.`**
- Use the **`print`** command to display the values of `some variables; e.g,` **`x`**, note that it is initially 0.
- Set another breakpoint after the fscanf( ) call, say line 19: **`break 19.`** Continue execution, and enter **3 10**, after entering the data, the breakpoint should be hit and print the value of x again. It should be 3.
- Set a watchpoint on a local variable; e.g., **`sum`**, and **`continue`** execution until the program terminates.

```
(gdb) cont
Continuing.
3 10

Breakpoint 2, phase_1_of_1 () at boom.c:19
19         if (args != 2)
(gdb) print x
$2 = 3
(gdb) watch sum
Watchpoint 3: sum
(gdb) cont
Continuing.
```

```
Watchpoint 3: sum

Old value = 0
New value = 13
phase_1_of_1 () at boom.c:24
24          if ((x<=0)||(y<=0))
(gdb)
```

Examining the call stack

- The **backtrace** command can be called at any time in GDB, and shows which function is associated with the current point of execution. You can call `backtrace` to see how the function got to where it is, that is, through what function calls. `backtrace` will display the current *call stack*, with one line per *frame*, where each frame effectively corresponds to a function call.

  The commands **up** and **down** move "up" and "down" the call stack, that is, they will reset the current frame that is visible to you in GDB. The commands **where** and **info stack** are equivalent to **backtrace**. With no arguments, **frame** displays information about the current stack frame.

Displaying machine instructions: disas

The command `disas`, or `disas <function>`, disassemble - displays assembly instructions for a given function (default is the current frame).

```
(gdb) disas phase_1_of_1
Dump of assembler code for function phase_1_of_1:
   0x000000000000079a <+0>:     push   %rbp
   0x000000000000079b <+1>:     mov    %rsp,%rbp
   0x000000000000079e <+4>:     sub    $0x10,%rsp
   0x00000000000007a2 <+8>:     lea    -0x8(%rbp),%rcx
..
---Type <return> to continue, or q <return> to quit---q
Quit
```

Use the '&' operator to find the address of a function or variable; e.g.,

```
(gdb) print &phase_1_of_1
$1 = (void (*)()) 0x79a <phase_1_of_1>
(gdb)
```

Examining memory and registers

You can use the command **x** to examine the contents of a **memory address**. For example, to display instructions at address 0x400635, the first four lines of phase_1_of_1( ), you can use:

```
(gdb) x/4i 0x079a
   0x79a <phase_1_of_1>:        push   %rbp
   0x79b <phase_1_of_1+1>:      mov    %rsp,%rbp
   0x79e <phase_1_of_1+4>:      sub    $0x10,%rsp
   0x7a2 <phase_1_of_1+8>:      lea    -0x8(%rbp),%rcx
(gdb)
```

Here, `/4i` is an optional parameter, indicating that GDB should display 4 lines of instructions. Other formats are available, such as `x/d` for decimal, `x/x` for hex, and `x/s` for strings. You can also use the command `print/x $reg` to display the value of **register** `$reg`. Again, `/x` indicates that GDB should display the output in hex format. To display the contents of every registers, use the command: **info reg**`isters`.

Stepping through and into a function, by instruction: nexti and stepi

We saw above how we can use `next` and `step` to step through our program, line by line. The commands **nexti** and **stepi** are similar, but execute **one instruction** rather than one line our source code. These commands can be useful for lines that contain several operations or function calls.

Using objdump to examine binary files.

In addition to GDB, you can also use the program `objdump` to display a range of info about a binary file.

`objdump -d` disassembles a file (similar to **`disas`** in gdb).

`objdump -t` displays symbol table information

- Disassemble the executable boom. What information do you see that you did not get from GDB's `disas` command?
- Use `objdump` and `grep` to find symbol table entry for the function `phase_1_of_1`; e.g., **`objdump –t boom | grep phase_1_of_1`**

Problems to turn on for Lab #2:

Problem Statement

The bomb is a 32-bit binary ELF executable, **kaboom.** For this assignment, you should use a CS Linux machine for the assignment. No source code is provided. Determine what input strings should be passed to the program **kaboom** via standard input so that the built-in bomb doesn't explode. Each time the bomb explodes you will lose 0.5 points up to a maximum of 20 points lost.

Background

This bomb program is compiled using: **gcc –Og –fno-stack-protector –fno-inline -fno-omit-frame-pointer –m32 -o kaboom kaboom.c; it can be found at /pub/cis450/programs/Lab2/kaboom**, the 64-bit version is also available as **kaboom64**.

Since the –Og switch is present and the -g switch is not present, the binary is optimized for debugging (no optimizations that mangle the code are included), but the binary contains a limited amount of debugging information. Usually adding the option –g works best for debugging with source code, but we don't want you to be able to see the original source code. In this lab, you will be working with the assembly code directly.

Examining the Bomb

The symbol table is sometimes useful to identify calls to standard library functions, (e.g., printf), as well as the bomb's own functions. Note that the symbol table is always present in the executable, even if the executable was compiled without the -g switch. You can look at all the bomb's symbol table by using nm: **nm kaboom**

```
...
080486bf T explode
…
080486da T phase_1_of_5
08048734 T phase_2_of_5
080487b8 T phase_3_of_5
0804885b T phase_4_of_5
08048958 T phase_5_of_5
080486bd T report_explosion
08048600 T report_solution
```

Examine the symbols marked with a T (capital t), and ignore the ones that start with an _ (underscore), b, B, U, or R; e.g., **nm kaboom | grep T.** These are names of functions from the C program that was used to compile the bomb. Notice that there is a function called **explode()**; can you guess what this function does? Next, take a look at the **printable strings** from the executable file using the command: **strings kaboom | less**

In this way, you may find clues that will help you defuse some of the phases of your bomb. You can use objdump to disassemble the bomb: **objdump -d kaboom | less**, or dump it to a file: **objdump -d kaboom > kaboom.dump** to view with any text editor. In addition, the assembly version of kaboom is available as **kaboom.s.**

Part of the object dump (assembly code) for our example bomb:

kaboom: file format elf32-i386

```
.LC4:
    .string "KABOOOOM, THE BOMB EXPLODED!!!"
..
explode:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $16, %esp
    call    __x86.get_pc_thunk.bx
    addl    $_GLOBAL_OFFSET_TABLE_, %ebx
    leal    .LC4@GOTOFF(%ebx), %eax
    pushl   %eax
    call    puts@PLT
    movl    $1, (%esp)
    call    exit@PLT
phase_1_of_6:                       (ignoring tag lines starting with '.')
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $20, %esp
    call    __x86.get_pc_thunk.bx
    addl    $_GLOBAL_OFFSET_TABLE_, %ebx
    leal    -16(%ebp), %eax
    pushl   %eax
    leal    -12(%ebp), %eax
    pushl   %eax
    leal    .LC5@GOTOFF(%ebx), %eax
    pushl   %eax
    movl    stdin@GOT(%ebx), %eax
    pushl   (%eax)
    call    __isoc99_fscanf@PLT
    addl    $16, %esp
    cmpl    $2, %eax
    jne .L16
    movl    -12(%ebp), %edx
    testl   %edx, %edx
    jle .L17
    leal    0(,%edx,8), %eax
    subl    %edx, %eax
    cmpl    -16(%ebp), %eax
    jne .L18
    movl    -4(%ebp), %ebx
    leave
    ret
.L16:
    subl    $12, %esp
    pushl   $1
    call    explode
.L17:
    subl    $12, %esp
    pushl   $1
    call    explode
.L18:
    subl    $12, %esp
    pushl   $1
    call    explode
```

Look at the code of explode(); try to figure out what it does.
Note, the string at `.LC4` is **"KABOOOOM, THE BOMB EXPLODED!!!":**

Running the bomb
The bomb can be invoked by using: **./kaboom** or inside the debugger using **gdb kaboom**, and **run – WARNING, WARNING, WARNING, be ready to stop the execution at an appropriate spot to prevent an explosion -- set the breakpoint before you execute the "run" command.**

The program waits for you to enter a string. You can enter the input from the keyboard, or redirect input to come from a text file: **./kaboom < lab2.txt**
The bomb then examines the string, and either explodes, or not.

GDB (GNU DeBugger)

Now all we need to do is completely understand the assembly code, and then we can defuse the bomb. In this problem, we will be dealing with a lot of code, which can be difficult to understand. Even if we do a good job, we might make a mistake and accidentally detonate the bomb. This is where the debugger, gdb, comes in. It lets us step through the assembly code as it runs, and examine the contents of registers and memory. We can also set breakpoints at arbitrary positions in the program. Breakpoints are points in the code where program execution is instructed to stop. In this way, we can let the debugger run without interruption over large portions of code, such as code that we already understand or believe is error-free.

Starting gdb
Start gdb by specifying what executable to debug: **gdb kaboom**

We can run the bomb in the debugger just as we would outside the debugger, except that we can instruct the program to stop at certain locations and inspect current values of memory and registers. As a last resort, we can use (Ctrl-C) to stop the program and panic out. But this is not recommended and is usually not necessary, as long as we positioned our breakpoints appropriately.

To start a program inside gdb: (gdb) **run**

To start a program inside gdb, with certain input parameters:
(gdb) **run [parameters]**

Examples:
(gdb) **run < lab2.txt**
(equivalent to ./kaboom < lab2.txt , just this time inside gdb)
(gdb) **run -d 1**
(equivalent to ./kaboom -d 1; this is a made-up example in the specific case of the bomb program, as 'kaboom' supports no such parameters; this example is meant to demonstrate how things would work in general)

Exiting gdb
To exit gdb and return to the shell prompt:
(gdb) **quit**

Note that exiting gdb means you lose all of your breakpoints that you set in this gdb session. When you re-run gdb, you need to re-specify any breakpoints that you want to re-use. A common mistake is to forget this and then let the debugging proceed straight into the explode() routine.

Breakpoints
We wouldn't be using gdb if all we did was run the program without any interruptions. We need to stop program execution at certain key positions in the code, and then examine program behavior around those positions. How do we pick a good location for a breakpoint?

First, we can always set a breakpoint at 'main', since every C program has a function called 'main'. Dr. Evil accidentally gave us 'boom.c' in the previous lab. By examining this code and output above, we see that we might want a breakpoint at 'phase_1_of_6', as this is where our input is tested.

(gdb) **break phase_1_of_6**

Note: if you mis-type the name of the routine, gdb will print a warning and not set any breakpoints. Also, note that program execution will always stop just BEFORE executing the instruction you set the breakpoint on.

Another essential breakpoint to set is on the explode routine. For inputs that don't solve the puzzle, this breakpoint will be your last safeguard before explosion. I recommend ALWAYS setting this breakpoint. In addition to that, I recommend setting another breakpoint inside explode(), positioned after the call to the routine that prints "KABOOM..", but before the call to the routine that notifies the GTA of the explosion. This can be useful if you accidentally enter explode(), but don't notice that you hit the safeguard breakpoint. After several hours of debugging, when concentration drops down in a moment of weakness, it can happen that you accidentally instruct the program to keep on going. Then, the second breakpoint will save you.

Terminating program execution within gdb
We can terminate the program at any time using kill:
(gdb) **kill**
Note that this doesn't exit gdb, and all your breakpoints remain active. You can re-run the program using the run command, and all breakpoints will still apply.

Stepping through the code
To execute a single machine instruction, use:
(gdb) **stepi**

Note that if you use 'stepi' on a callq instruction, the debugger will proceed inside the called function. Also note that pressing <ENTER> re-executes the last gdb command. To execute several 'stepi' instructions one after another, type 'stepi' once, and then press <ENTER> several times in a row.

Sometimes we want to execute a single machine instruction, but if that instruction is a call to a function, we want the debugger to execute the function without our intervention. This is achieved using 'nexti':
(gdb) **nexti**

Program will be stopped as soon as control returns from the function; e.g., at the instruction immediately after the call in the caller function.

If you accidentally use stepi to enter a function call, and you really don't want to debug that function, you can use 'finish' to resume execution until the current function returns. Execution will stop at the machine instruction immediately after the 'callq' instruction in the caller function, just as if we had called 'nexti' in the first place:
(gdb) **finish**

Note: make sure the current function can really be run safely without your intervention. You don't want it to call explode(). To instruct the program to execute (without your intervention) until the next breakpoint is hit, use :
(gdb) **cont**

Note that the same warning as in the case of 'finish' applies. If program contains debugging information (-g switch to gcc; not the case for this assignment, but otherwise usually the case), we can also step a single C statement:
(gdb) **step**

Or, if next instruction is a function call, we can use 'next' to execute the function without our intervention. This is just like nexti, except that it operates with C code as opposed to machine instructions:
(gdb) **next**

Examining registers
To inspect the current values of registers:
(gdb) **info reg**
This prints out the current values of all registers.

To inspect the current values of a specific register:
(gdb) **p $edx**
To print the value in hex notation:
(gdb) **p/x $edx**
To see the address of the next machine instruction to be exectued:
(gdb) **frame**
or, equivalently, you can inspect the instruction pointer register:
(gdb) **p/x $eip**

Normally, when debugging a C/C++ program for which the source code is available (but not for this assignment), you can also inspect the call-stack (a list of all nested function calls that led to the current function being executed):
(gdb) **where**

Examining memory
To inspect the value of memory at location <address>:
(gdb) x/NFU <address>
where: N = number of units to display, F = output format (hex=h, signed decimal=d, unsigned decimal=u, string=s, char=c), U = defines what constitutes a unit: b=1 byte, h=2 bytes, w=4 bytes, g=8 bytes. Note that output format and unit definition characters are mutually distinct from each other.

Examples:
To use hex notation, and print two consecutive 64-bit words, starting from the address 0x400746 and higher:
(gdb) **x/2xg 0x5655579a**
To use hex notation, and print five consecutive 32-bit words, starting from the address 0x400746:
(gdb) **x/5xw 0x5655579a**
To print a single 32-bit word, in decimal notation, at the address 0x400746:
(gdb) **x/1dw 0x5655579a**

To Turn In
Upload a copy of the input strings used to diffuse the bomb, **kaboom**; e.g., lab2.txt. Note that the number of explosions will be reported automatically via e-mail. If you successfully diffuse the bomb, that will also be reported via e-mail.

Prizes: Prizes will be given to students who solve the most phases in the least amount of time. An e-mail message is also sent when you solve all phases, so let that one go.