CS 304, Fall 2007
Lab Assignment L2: Defusing a Binary Bomb

Assigned: Wednesday, September 19, 2007

Due: Wednesday, October 3, **8:50AM**

1 Introduction

The nefarious *Dr. Evil* has created a slew of "binary bombs" for our class. A binary bomb is a program that consists of a sequence of six phases. Each phase expects you to type a particular string on *stdin*. If you type the correct string, then the phase is *defused* and the bomb proceeds to the next phase. Otherwise, the bomb *explodes* by printing `"BOOM!!!"` and then terminating. The bomb is defused when every phase has been defused.

Each person in the class will be supplied with a bomb to defuse. The assignment is to defuse your bomb before the due date. Good luck, and welcome to the bomb squad!

Step 1: Getting Your Bomb

Sometime during the afternoon or evening of Wednesday, September 19, 2007, a bomb will be emailed to each class member's account on the CS server. The bombs will be numbered from `bomb1` to `bomb`n, where n is the number of students in the class. Each bomb is constructed with a random choice of phases, and the bomb numbers are assigned in a random order relative to the class roll (that is, the alphabetically first student on the roll will most probably **not** be assigned `bomb1`).

Step 2: Defuse Your Bomb

Once you have received your bomb, save it in a secure directory. Your job is to defuse the bomb.

You can use many tools to help you with this; please look at the **hints** section for some tips and ideas. The best way is to use your favorite debugger to step through the disassembled binary.

Each time your bomb explodes it notifies the project logfile, and you lose 1/2 point (up to a max of 20 points) in the final score for the lab. So there are consequences to exploding the bomb. However, as you learn about setting breakpoints in `gdb`, you will realize that you can protect yourself by setting a breakpoint in `explode_bomb`.

Phase one is worth 10 points, and phases 2 through 6 are worth 14 points each, for a lab total of 80 points.

There is also a challenging, extra-credit (4 points) "secret" phase that only appears if a student appends a certain string to the solution of one of the earlier phases.

The phases get progressively harder to defuse, but the expertise you gain as you move from phase to phase should offset this difficulty. However, the last phase will challenge even the best students, so please don't wait until the last minute to start.

The bomb ignores blank input lines. If you run the bomb with a command line argument such as `psol.txt` (a name chosen arbitrarily), the bomb will read the input lines from `psol.txt` until it reaches EOF, and then switch over to `stdin`. In a moment of weakness, Dr. Evil added this feature so you don't have to keep retyping the solutions to phases you have already defused.

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints. You will also need to learn how to inspect both the registers and the memory states. One of the nice side-effects of doing the lab is that you will get very good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career.

Logistics

Any clarifications and revisions to the assignment will be posted on the Web page for this lab.

You must complete the assignment on the machines in M-S 121. In fact, there is a rumor that Dr. Evil really is evil, and the bomb will always blow up if run elsewhere. There are several other tamper-proofing devices built into the bomb as well, or so they say.

Hand-In

There is no explicit hand-in. The bomb will notify your instructor automatically after you have successfully defused it. You can keep track of how you (and the other students) are doing by following the link on the lab web page to the page summarizing the current status of the bomb tests. This web page is updated continuously to show the progress of being made on each bomb. Results are displayed by bomb number. **Note:** If the update timestamp stops changing, then please send email to `lowekamp@cs.wm.edu` to let me know. This means that the autograder daemon has died and needs to be restarted. Successfully debugged phases won't show up until the autograder daemon is running.

Hints *(Please read this!)*

There are many ways of defusing your bomb. You can examine a disassembly of it in great detail without ever running the program, and figure out exactly what it does. This is a useful technique, but it may not give you enough information about what the bomb is doing.

You can also run the bomb with a debugger, inspect its memory locations, watch what it does step by step, and use this information to defuse it. This is probably the fastest way of defusing the bomb. Combining this technique with the hard copy of the disassembly of the bomb is probably the strongest combination you can use.

We do make one request, *please do not use brute force!* You could write a program that will try every possible key to find the right one. But this is no good for two reasons:

- You lose 1/2 point (up to a max of 20 points) every time you guess incorrectly, the bomb explodes, and a message is written to a log file for the class and to your terminal screen.
- We haven't told you how long the strings are, nor have we told you what characters are in them. Even if you made the (wrong) assumptions that they all are less than 80 characters long and only contain lower case letters, then you will have 26^{80} guesses for each phase. This will take a very long time to run, and you will not get the answer before the assignment is due.

There are many tools that are designed to help you figure out both how programs work, and what is wrong when they don't work. Here is a list of some of the tools you may find useful in analyzing your bomb, and hints on how to use them.

- **gdb**
  The GNU debugger, this is a command line debugger tool available on every platform in M-S 121. As you know from class lectures, you can trace through a program line by line, examine memory and registers, look at both the source code and assembly code (we are not giving you the source code for most of your bomb), set breakpoints, set memory watch points, and write scripts. Here are some tips for using `gdb`.

  – There is a `gdb` command summary at the end of this specification. Both the course Web page and lab Web page contain a link to an online version of it You will find it helpful to look through the command summary.

  – To keep the bomb from blowing up every time you type in a wrong input, you'll want to learn how to set breakpoints (refer to the `gdb` command summary or to your class notes).

  – For other documentation, type "info gdb" or "man gdb" at a Unix prompt, or type "help" at the gdb command prompt. Some people like to run `gdb` under `gdb-mode` in `emacs`.

  – If you type `run bomb` inside `gdb`, then you will explode the bomb, no matter WHAT `gdb` breakpoints you have set (try to figure out why – see pages 1 and 4 for a hint).

  There is a graphical front-end to `gdb` called `ddd`. Type "info ddd" for more information.

- **objdump -t**
  This will print out the bomb's symbol table. The symbol table includes the names of all functions and global variables in the bomb, the names of all the functions the bomb calls, and their addresses. You may learn something by looking at the function names!

- **objdump -d**
  Use this to disassemble all of the code in the bomb. You can also just look at individual functions. Reading the assembler code can tell you how the bomb works.

  You can also use the `disas` command in `gdb` to disassemble a given subroutine, but it gives you slightly different information than `objdump` does. For example, here is the output produced by the `disas` command for the `read_six_numbers` routine:

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
0x08048f0d <read_six_numbers+0>:        push   %ebp
0x08048f0e <read_six_numbers+1>:        mov    %esp,%ebp
0x08048f10 <read_six_numbers+3>:        sub    $0x8,%esp
0x08048f13 <read_six_numbers+6>:        mov    0xc(%ebp),%edx
0x08048f16 <read_six_numbers+9>:        lea    0x14(%edx),%eax
0x08048f19 <read_six_numbers+12>:       push   %eax
0x08048f1a <read_six_numbers+13>:       lea    0x10(%edx),%eax
0x08048f1d <read_six_numbers+16>:       push   %eax
0x08048f1e <read_six_numbers+17>:       lea    0xc(%edx),%eax
0x08048f21 <read_six_numbers+20>:       push   %eax
0x08048f22 <read_six_numbers+21>:       lea    0x8(%edx),%eax
0x08048f25 <read_six_numbers+24>:       push   %eax
0x08048f26 <read_six_numbers+25>:       lea    0x4(%edx),%eax
0x08048f29 <read_six_numbers+28>:       push   %eax
0x08048f2a <read_six_numbers+29>:       push   %edx
0x08048f2b <read_six_numbers+30>:       push   $0x80498c2
0x08048f30 <read_six_numbers+35>:       pushl  0x8(%ebp)
0x08048f33 <read_six_numbers+38>:       call   0x8048900 <_init+392>
0x08048f38 <read_six_numbers+43>:       add    $0x20,%esp
0x08048f3b <read_six_numbers+46>:       cmp    $0x5,%eax
0x08048f3e <read_six_numbers+49>:       jg     0x8048f45 <read_six_numbers+56>
0x08048f40 <read_six_numbers+51>:       call   0x8049447 <explode_bomb>
0x08048f45 <read_six_numbers+56>:       leave
```

```
0x08048f46 <read_six_numbers+57>:       ret
End of assembler dump.
```

and here is the corresponding output from the `objdump` command:

```
08048f0d <read_six_numbers>:
 8048f0d:       55                          push   %ebp
 8048f0e:       89 e5                       mov    %esp,%ebp
 8048f10:       83 ec 08                    sub    $0x8,%esp
 8048f13:       8b 55 0c                    mov    0xc(%ebp),%edx
 8048f16:       8d 42 14                    lea    0x14(%edx),%eax
 8048f19:       50                          push   %eax
 8048f1a:       8d 42 10                    lea    0x10(%edx),%eax
 8048f1d:       50                          push   %eax
 8048f1e:       8d 42 0c                    lea    0xc(%edx),%eax
 8048f21:       50                          push   %eax
 8048f22:       8d 42 08                    lea    0x8(%edx),%eax
 8048f25:       50                          push   %eax
 8048f26:       8d 42 04                    lea    0x4(%edx),%eax
 8048f29:       50                          push   %eax
 8048f2a:       52                          push   %edx
 8048f2b:       68 c2 98 04 08              push   $0x80498c2
 8048f30:       ff 75 08                    pushl  0x8(%ebp)
 8048f33:       e8 c8 f9 ff ff              call   8048900 <sscanf@plt>
 8048f38:       83 c4 20                    add    $0x20,%esp
 8048f3b:       83 f8 05                    cmp    $0x5,%eax
 8048f3e:       7f 05                       jg     8048f45 <read_six_numbers+0x38>
 8048f40:       e8 02 05 00 00              call   8049447 <explode_bomb>
 8048f45:       c9                          leave
 8048f46:       c3                          ret
```

Notice that `objdump` correctly identifies the `sscanf` C library call, but the `disas` command of `gdb` simply lists the call location as `_init+392` in the `bomb` executable. The `disas` command of `gdb` is more convenient to use if you're already in `gdb`, but you may occasionally find it necessary to refer to the `objdump` output to track down any mysterious C library calls.

I don't mean to over-emphasize the `read_six_numbers` routine here. I am simply pointing out the differences between `objdump -d` and the `disas` command of `gdb`. By the name of this procedure and by a glance at its construction, you can see that the procedure uses `sscanf` to parse the input string fed to it and place six `int` values in an array whose address is supplied as one of its parameters. That's about all you need to know about `read_six_numbers`.

- **strings**
  This utility will display the printable strings in your bomb.

Looking for a particular tool? How about documentation? Don't forget, the commands `apropos` and `man` are your friends. In particular, `man ascii` might come in useful. Also, the web may also be a treasure trove of information. If you get stumped, feel free to ask me for help.

Useful GDB Commands

```
Starting:
   gdb <file>
   gdb -h       (lists command line options)
   gdb          (you will have to use the "file <file>" command once in gdb
                 to load the executable <file>)

Using the .gdbinit File to Initialize gdb
```

```
   At startup, the gdb debugger will execute initialization files
(usually named .gdbinit) and process command line options
in the following order

   1. Reads the init file (if any) in your home (login) directory.

   2. Processes command line options and operands.

   3. Reads the init file (if any) in the current working directory.

   4. Reads command files specified by the ‘-x’ option.

   For more details, type "info gdb" at the UNIX command line,
then look for "Command Files".

   For example, for the bomblab, you will probably want to create
a .gdbinit file in the directory containing your bomb to be defused
that has at least the following three lines:

         b main
         b explode_bomb
         disp /i $eip

Exiting:
   quit
   q
   Ctrl-d
   Note: Ctrl-C does not exit from gdb, but halts the current
      gdb command


General gdb commands

   run      (start the loaded program with no arguments )
   r
   NOTE:  If you have started gdb with "gdb <file>",
          then you do NOT type "r <file>"!!!

   r arg1 arg2 arg3  (start the loaded program with 3 arguments)

   kill     (stop the loaded program)


Breakpoints

   break FUNCTION (set a breakpoint at the entry to the function)
   b FUNCTION

   break *ADDRESS (set a breakpoint at the specified address)
   b *ADDRESS

   disable <NUM>  (disable the breakpoint with that number)
   di <NUM>

   enable <NUM>   (enable the breakpoint with that number)
   en <NUM>

   clear FUNCTION (clear any breakpoints at the entry to the function)
   cl FUNCTION

   delete <NUM>   (deletes the breakpoint with that number)
   del <NUM>

   delete      (deletes all breakpoints)


Working at breakpoints
```

```
   stepi    (execute one machine code instruction)
   si

   stepi <NUM> (execute NUM machine instructions)
   si <NUM>

   step     (execute one C statement)
   s

   next (like step, but proceed through subroutine calls without stopping)
   n

   nexti    (like stepi, but proceed through subroutine calls without stopping)
   ni

   nexti <NUM> (execute NUM machine instructions like stepi, but treat
   ni <NUM>     subroutine calls as one instruction)

   until LOCATION (continue running until LOCATION is reached)
   unt LOCATION

   continue (resume execution)
   c

   continue <NUM> (continue, ignoring this breakpoint NUM times)
   c <NUM>

   finish      (run until the current function returns)

   backtrace   (print the current address and stack backtrace)
   where    (print the current address and stack backtrace)

Examining code

   print/a $pc (print the program counter)
   print $sp   (print the stack pointer)
   disas    (disassemble the function around the current line)
   disas ADDR  (display the function around the address)
   disas ADDR1 ADDR2 (display the function between the addresses)

Examining data

   print $eax  (print the contents of %eax)
   print/x $eax   (print the contents of %eax as hex)
   print/a $eax   (print the contents of %eax as an address)
   print/d $eax   (print the contents of %eax as decimal)
   print/t $eax   (print the contents of %eax as binary)
   print/c $eax   (print the contents of %eax as a character)

   print 0x100 (print decimal repr. of hex value)
   print/x 555 (print hex repr. of decimal value)

   x ADDR      (print the contents of ADDR in memory)
   x/NUF ADDR  (print the contents at ADDR in memory:
            N = number of units to display
            U = b (bytes), h (2 bytes), w (4 bytes))
            F = display format, x (hex), o (octal), d (decimal -- default)
                  t (binary), s (string), f (float), i (instruction), c (char)
   Examples:  x/20wx $esp     show the top 20 4-byte words of the stack
              x/128bx main    show first 128 bytes (in hex) of main pgm


Autodisplaying information

   display $eip   (print contents of %eip every time the program stops)
   disp $eip
   disp $pc
   display     (print the auto-displayed items)
   disp

   delete display <NUM> (stop displaying item NUM)
   und <NUM>

Useful information commands

   help info

   info program   (current status of the program)

   info functions (functions in program)
   i f

   info stack      (backtrace of the stack)
   i s

   info frame  (information about the current stack frame)
   i f

   info scope  (variables local to the scope)
   i s

   info variables (global and static variables)
   i v

   info registers  (registers and their contents)
   i r

   info breakpoints (status of user-settable breakpoints)
   i b

   info address SYMBOL  (use for looking up addresses of functions)
   i ad SYMBOL

Running gdb in emacs
   M-x gdb
   C-h m to see the features of GDB mode
```