CIS 450 – Computer Organization and Architecture
Lab 4

Problem Statement

The bomb is a 32-bit binary ELF executable, **kaboom**. For this lab, you should use a 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.

Note: Every team that successfully completes this will get 10 points extra credit. The first team to complete this will get a minor prize, along with global admiration.

Background

To get you started, there is a sample bomb in the public directory (/pub/cis450), **boom.c**, designed to illustrate the main points of this assignment. This bomb program is compiled using: **gcc -O1 boom.c -o boom**

Since the –O1 switch is present and the -g switch is not present, the binary is optimized for level 1 optimization, but contains a limited amount of debugging information. Usually using -g with no optimizations works best for debugging with source code. However, that is not the style to be used in this assignment because 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 boom**

```
...
08048440 T explode
...
080484c1 T main
08049750 d p.5692
0804845e T phase_1_of_1
```

Examine the symbols marked with a T (capital t), and ignore the ones that start with an _ (underscore). 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: **strings boom**

In this way, you may find clues that will help you defuse some of the phases of your bomb. Then, use objdump to disassemble the bomb: **objdump -d boom | less**

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

```
boom:     file format elf32-i386
...
08048440 <explode>:
 8048440:       55                      push   %ebp
 8048441:       89 e5                   mov    %esp,%ebp
 8048443:       83 ec 08                sub    $0x8,%esp
 8048446:       c7 04 24 e8 85 04 08    movl   $0x80485e8,(%esp)
 804844d:       e8 e2 fe ff ff          call   8048334 <puts@plt>
```

```
 8048452:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048459:       e8 06 ff ff ff          call   8048364 <exit@plt>
0804845e <phase_1_of_1>:
 804845e:       55                      push   %ebp
 804845f:       89 e5                   mov    %esp,%ebp
 8048461:       83 ec 28                sub    $0x28,%esp
 8048464:       8d 45 f8                lea    0xfffffff8(%ebp),%eax
 8048467:       89 44 24 0c             mov    %eax,0xc(%esp)
 804846b:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 804846e:       89 44 24 08             mov    %eax,0x8(%esp)
 8048472:       c7 44 24 04 f2 85 04    movl   $0x80485f2,0x4(%esp)
 8048479:       08
 804847a:       a1 54 97 04 08          mov    0x8049754,%eax
 804847f:       89 04 24                mov    %eax,(%esp)
 8048482:       e8 bd fe ff ff          call   8048344 <fscanf@plt>
 8048487:       83 f8 02                cmp    $0x2,%eax
 804848a:       74 05                   je     8048491 <phase_1_of_1+0x33>
 804848c:       e8 af ff ff ff          call   8048440 <explode>
 8048491:       8b 55 fc                mov    0xfffffffc(%ebp),%edx
 8048494:       b9 01 00 00 00          mov    $0x1,%ecx
 8048499:       85 d2                   test   %edx,%edx
 804849b:       7e 13                   jle    80484b0 <phase_1_of_1+0x52>
 804849d:       b9 01 00 00 00          mov    $0x1,%ecx
 80484a2:       b8 01 00 00 00          mov    $0x1,%eax
 80484a7:       42                      inc    %edx
 80484a8:       0f af c8                imul   %eax,%ecx
 80484ab:       40                      inc    %eax
 80484ac:       39 c2                   cmp    %eax,%edx
 80484ae:       75 f8                   jne    80484a8 <phase_1_of_1+0x4a>
 80484b0:       39 4d f8                cmp    %ecx,0xfffffff8(%ebp)
 80484b3:       74 05                   je     80484ba <phase_1_of_1+0x5c>
 80484b5:       e8 86 ff ff ff          call   8048440 <explode>
 80484ba:       c9                      leave
 80484bb:       90                      nop
 80484bc:       8d 74 26 00             lea    0x0(%esi),%esi
 80484c0:       c3                      ret
...
```

Look at the code of explode(); try to figure out what it does.
Hint: the string at 0x80485e8 is “KABOOM!!!”:

```
$ gdb boom
(gdb) break explode
(gdb) run
(gdb) x /s 0x80485e8
0x80485e8 <_IO_stdin_used+4>:    "KABOOM!!!"
(gdb)
```

Running the bomb

The bomb can be invoked by using: **./boom**
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: **./boom < solution.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 boom**

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 < solution.txt
(equivalent to ./boom < solution.txt , just this time inside gdb)
(gdb) run -d 1
(equivalent to ./boom -d 1; this is a made-up example in the speficic case of the bomb program, as 'boom' 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'. By examining this code, we see that we might want a breakpoint at 'phase_1_of_1', as this is where our input is tested.

(gdb) break phase_1_of_1
Note: if you mistype 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:
(gdb) break explode

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 grader 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.

To set a breakpoint at the machine instruction located at the address 0x401A23:
(gdb) break *0x401A23
Note: don't forget the '0x'. If you forget it, and if you are unlucky enough that the address doesn't contain any A,B,C,D,E,F characters, breakpoint address will be interpreted as if given in the decimal notation. This results in a completely different address to what was desired, and breakpoint won't work as expected.

To see what breakpoints are currently set:
(gdb) info break

To delete one or more breakpoints:
(gdb) delete <breakpoint number>
Example:
(gdb) delete 4 7
erases breakpoints 4 and 7.

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 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, 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 callq 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) continue
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

Disassembling code using gdb
You can use 'disassemble' to disassemble a function or a specified address range.

To disassemble function explode():
(gdb) disassemble explode

To disassemble the address range from 0x4005dc to 0x4005eb:
(gdb) disassemble 0x4005dc 0x4005eb
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 0x400746:
(gdb) x/NFU 0x400746
Here:
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 0x400746

To print a null-terminated string at location 0x400746:
(gdb) x/s 0x400746
To use hex notation, and print five consecutive 32-bit words, starting from the address 0x400746:
(gdb) x/5xw 0x400746
To print a single 32-bit word, in decimal notation, at the address 0x400746:
(gdb) x/1dw 0x400746

The source code for the example bomb:

```
// boom.c
#include <stdio.h>
#include <stdlib.h>

void explode() {
   printf("KABOOM!!!\n");
   exit(1);
}

void phase_1_of_1 () {
   int args, num, fact;
   int i = 0;
   int check_fact = 1;

   args = fscanf (stdin, "%d %d", &num, &fact);
   if (args != 2)
       explode_bomb();

   for (i = 1; i <= num; i++)
       check_fact = check_fact * i;

   if (fact != check_fact)
       explode_bomb();
}

int main() {
   printf("Welcome to the demo bomb.\n");
   printf ("Phase 1\n");
   phase_1_of_1();
   printf("You safely defused the bomb. Well done.\n");
   return 0;
}
```

To Turn In

Upload a copy of your input strings used to diffuse the bomb;e.g., solution.txt.
Note that the number of explosions will be reported automatically via e-mail.