Binary Bomb Project Assignment

Overview

In this lab you are provided a binary compiled for a 32-bit IA32 Linux environment. You are *not* provided the C source code for the compiled binary. The binary bomb consists of five phases that you must *defuse* - or the bomb will explode. When you run the bomb it will wait for input from the command line.

The binary bomb can be defused by providing the proper input for each of the five phases. To defuse a single phase you must type in one or more single letters or positive decimal numbers, separated by spaces, and hit `enter`. If the input is acceptable for that phase the bomb *will not explode*. If you provide incorrect input the bomb *will explode* and the program will exit. You may type up to 5 letters (uppercase or lowercase) or numbers on each line to defuse each phase. Here is a successful defusing of a bomb:

```
phase-1: 4
OK
phase-2: x y z d
OK
phase-3: 9 45
OK
phase-4: v
OK
phase-5: y 3
OK
```

Input that you provide comes after `phase-n:` where `n` is the phase number. Each of the above input lines correspond to a sequence of characters or positive integers that can be used to defuse each phase. That is, `4` corresponds to phase 1, `x y z d` corresponds to phase 2, etc. Your goal is to successfully provide the proper sequence of characters or numbers for each phase to defuse the bomb and save the world!

Please download your custom binary bomb from **here** to start this assignment.

Tools

To be successful in this lab you will need to use tools that we have covered in class and the lab section. In particular, you will find the following tools the most helpful:

`objdump -d bomb`: Disassembles the binary and generates the assembly listing to standard output. You can capture the output with the following command:

`objdump -d bomb > bomb-asm.txt`. This is useful for looking over the assembly code to get an understanding of where things are located (i.e., what are the function names that you might want to use in `gdb` to set a breakpoint).

`gdb bomb`: Runs the bomb in the `gdb` debugger. This is useful for inspecting the values of memory and registers as well as control flow. You should do the former first and then use `gdb` to inspect the running program. Here are some of the `gdb` commands that you may find useful:

- `info registers`: This will show the current contents of the registers. Great when you need to determine the value of a result of executing an instruction. The abbreviation `i r` is convenient.
- `si`: This will step a single instruction (it is short for `stepi`). This is useful to go instruction by instruction looking at the contents of your registers and memory.
- `break <function>`: This will cause execution to break (stop) when you arrive in a particular function. In fact, you can also provide `break` an address and it will break just before the instruction at that address. You can abbreviate `break` with `b`.
- `disassemble <function>`: This will disassemble a function. This is useful to see where you are in a particular function. You will notice the arrow (`=>`) before the address of an instruction on the left-hand side - this is the instruction your program is about to execute. You can abbreviate with `disas`.

If you find that you need additional information when you are using `gdb` you should consult `gdb`'s online help system by typing in `help`, ask a question on the discussion forum, or Google your question. Google is great for finding out what a particular ia32 instructions does, for example. (It is also possible to download the Intel instruction set manuals, but they are big and take a bit of wandering around to see how they are organized.)

Bomb Details

Each bomb is customized for a person attempting this lab. That is, the input values to defuse the bomb will be different for everyone. However, the machinery (algorithms) in each bomb is the same. To begin the lab you should download your bomb from the web resource indicated as part of the assignment.

The bomb is implemented in 5 phases and each phase must be defused before proceeding to the next. If you accidentally "detonate" your bomb you will receive a message and the program will exit. It will also write out your current solution to a `solution.txt` file. You can use this file to reference what you typed in to defuse a phase. BE CAREFUL: the `solution.txt` file will be overwritten the next time you run the bomb - so if you want to save a correct

solution make sure you copy it to another file before executing the bomb again (or write the current solution down somewhere). When you are able to defuse all phases the generated `solution.txt` file is what you are to submit to Moodle. If you are unable to defuse all phases the `solution.txt` file will record your solution up to the phase you completed.

Grading

The phases will be graded according to their difficulty level:

- phase-1 : 1 points
- phase-2 : 2 points
- phase-3 : 3 points
- phase-4 : 4 points
- phase-5 : 3 points

Total number of points that you can score: 13

Ultimately, your score will be automatically determined by your `solutions.txt` file submission. Given the nature of this assignment, there is no peer assessment for it.

Questions

You should use the online discussion forum associated with this course to ask and answer questions about the bomb and its five phases. If you are stuck you should not hesitate to ask for help! This could be as simple as "Where should I start?" or as complicated as "Which `gdb` command should I use to view the contents of the register %eax?". **Please use the discussion forums to your advantage**!

You may not ask questions such as "What is the algorithm in phase 3?" However, you could ask a question such as "I understand phase 3, but can't seem to get anywhere. Help please". An acceptable response might be, "Pay attention to the %eax register".

HINTS

Start early! Start simple! Identify where the phases are in the binary. Try to vaguely sketch what the C code might look like (is there an if-else, a while-loop, a for-loop, etc.). Identify the number of inputs a phase requires (there is the easy way, and then there is the hard way). Run through a phase several times in `gdb` - find where your input values are - identify which registers and memory locations hold the letters and/or numbers you type in for a phase. Identify if a phase is expecting a number of a character (hint: remember characters are 1

byte in length and are encoded by ASCII). For convenience, here is the chart in hexadecimal and decimal:

```
   2 3 4 5 6 7
  ------------
0:   0 @ P ` p
1: ! 1 A Q a q
2: " 2 B R b r
3: # 3 C S c s
4: $ 4 D T d t
5: % 5 E U e u
6: & 6 F V f v
7: ´ 7 G W g w
8: ( 8 H X h x
9: ) 9 I Y i y
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

```
  30 40 50 60 70 80 90 100 110 120
 --------------------------------
0:    (  2  <  F  P  Z  d   n   x
1:    )  3  =  G  Q  [  e   o   y
2:    *  4  >  H  R  \  f   p   z
3: !  +  5  ?  I  S  ]  g   q   {
4: "  ,  6  @  J  T  ^  h   r   |
5: #  -  7  A  K  U  _  i   s   }
6: $  .  8  B  L  V  `  j   t   ~
7: %  /  9  C  M  W  a  k   u  DEL
8: &  0  :  D  N  X  b  l   v
9: ´  1  ;  E  O  Y  c  m   w
```

Submission Instructions

Please submit your `solution.txt` file to Moodle by the required due date. You should not modify your `solution.txt` file - it will be automatically generated by your bomb. You should submit your `solution.txt` file even if you did not complete all the phases.