

Nicolas Barraclough  
10/18/2020  
CS373 - Defense Against the Dark Arts

## Lab 1 - GDB Bomb!

After finally getting everything updated, downloaded, installed, updated again, and up & running, I was able to begin the lab. To start, I lit the fuse on the bomb to see first-hand how it would blow up. I ran it a few times to see if I could find anything out - to no conclusion. So I tried cat'ing it out to see if I could notice anything from the text version of the bomb. Not much could come out of that except for all the mid-run messages that the program spits out.

Next, I ran objdump with the disassemble (-d) flag to view the bomb executables as assembly code and wrote the output to a txt file. The assembly code helped me identify the different phases of the bomb and gave a little bit of insight as to what the input was being compared to in each phase. I will describe how objdump helped me in each phase when I walk through my process of finding each phase's key.

The 'strings' command was another useful tool in looking for the phase keys. I output this command into a file, too, and took out any string that didn't appear useful. I was left with quite a few strings but only a handful that look out of place. The printed messages that the bomb gives the user and the names of functions within the assembly code are apparent, but other strings stood out as possible keys like these ones:

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Public speaking is very easy.
%ld %c %d
giants
Wow! You've defused the secret stage!
So you think you can stop the bomb with ctrl-c, do you?
Well...
OK. :-)
Invalid phase%s
%ld %ld %ld %ld %ld %ld
Bad host (1).
Bad host (2).
Bad host (3).
GRADE_BOMB
nobody
defused
exploded
bomb-header:%s:%d:%s:%s:%d
bomb-string:%s:%d:%s:%d:%s
bomb
%s %s%s
BOOM!!!
The bomb has blown up.
%ld %s
austinpowers
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Congratulations! You've defused the bomb!
generic
isrveawhobnutfg
phase_2
phase_3
phase_0
phase_4
read_six_numbers
phase_5
secret_phase
phase_1
skip
node2
```

### PHASE\_1:

With these first few guesses, I will try to go through the phases. From the objdump file that I created, I can see that the first phase is looking at the string first by its length, and then it will run a “strings\_not\_equal” function 5 times. This was an indication to me of the number of words within the string being five words. And what do ya know, the first string that stood out as a possible key has 5 words in it. A match on the first one!! By looking at the strings within the file and using intuition with what the assembly code was doing, I was able to identify the key to the first phase as “Public speaking is very easy.”

```
08048b20 <phase_1>:
08048b20: 55                push    %ebp
08048b21: 89 e5            mov     %esp,%ebp
08048b23: 83 ec 08         sub     $0x8,%esp
08048b26: 8b 45 08         mov     0x8(%ebp),%eax
08048b29: 83 c4 f8         add     $0xfffffffff8,%esp
08048b2c: 68 c0 97 04 08   push    $0x80497c0
08048b31: 50              push    %eax
08048b32: e8 f9 04 00 00   call    8049030 <strings_not_equal>
08048b37: 83 c4 10         add     $0x10,%esp
08048b3a: 85 c0            test    %eax,%eax
08048b3c: 74 05           je      8048b43 <phase_1+0x23>
08048b3e: e8 b9 09 00 00   call    80494fc <explode_bomb>
08048b43: 89 ec           mov     %ebp,%esp
08048b45: 5d              pop     %ebp
08048b46: c3              ret
08048b47: 90              nop
```

```
So you got that one. Try this one.
Good work! On to the next...
Public speaking is very easy.
%d %c %d
giants
Wow! You've defused the secret stage!
```

### PHASE\_2:

Looking at the objdump file again, I can see that this phase is going to need six numbers as the key. There is a function that is called in phase\_2 of the assembly code named “read\_six\_numbers”. Looking at this function in gdb by setting a breakpoint at the function ((gdb) break read\_six\_numbers) and inputting a dummy key of “1 2 3 4 5 6”, I can see where in the code the program is comparing the values I entered to the ones it is expecting. I can see that when it compares, it is looking at what is in the eax register, so using the command “i r” at this cmp instruction, I will be able to see the value that is in the eax register. The value is 6. I know this is not the first number in the sequence, because if I do the same process with just an input of “6” instead of “1 2 3 4 5 6”, I can see that it is being compared to a value of 1 in the eax register. I must have guessed the first number correctly as 1. Continuing this pattern, I see that I have guessed the second number, ‘2’, correctly as well. This means that 6 is the third number in the sequence. This is where my method falls apart. The value in the eax register is still 6 when I get to the cmp instruction again. Iterating through the instructions until just before the bomb explodes, the value in the eax register is 0x18 or 24. Following the pattern, “1 2 6 24 ...” I think I may recognize the sequence.  $1 \times 2 = 2$ ,  $2 \times 3 = 6$ ,  $6 \times 4 = 24$ ... so the numbers are incrementally multiplied to find the next number in the sequence. Trying out “1 2 6 24 120 720” I see that my thinking is correct! Phase 2 passed.

### PHASE\_3:

The objdump file didn't have the same clues as the first two phases, there were no other functions to look at except for the one that explodes the bomb. By setting a breakpoint at the phase\_3 function and looking at the assembly code at this point, I can see that a value from memory at 0xx80497de was pushed onto the stack. Pulling this value in gdb, I can see that that

```
Breakpoint 1, 0x08048b9f in phase_3 ()
(gdb) x/s 0x80497de
0x80497de: "%d %c %d"
(gdb)
```

place in memory holds the string "%d %c %d" which tells me that the key might be a decimal number, then a character, then another decimal number. Rerunning the bomb with input "1 a 1" for phase 3's input, I was able to get past the first compare and the first instance that explode\_bomb would have been called, so I know I am on the right track. Now the bomb fails at instruction 0x08048c09, a jump if equal instruction. The compare instruction just before compares a hex value, 0xd6. In decimal, that value is 214. This may be my first decimal value. Trying out "214 a 1" as my next guess and seeing which instruction causes the failure, I can trace the instructions back a few to 0x08048bc9 which compares against a value of 7 and jumps if it is above 7. From this I can deduce that one of the digits must be no greater than 7. Since I have 1 as the second number, I am guessing that I had the digits in the wrong place. Trying "1 a 214", the jump if above instruction does not succeed, indicating that I have the correct arrangements of the decimals. Now, the input fails at instruction 0x08048c92, a jump if equal. The instructions just before the jump pointed me to a hex value 0x62 that was moved in the instruction just before. Since I know that the decimals are most likely correct, I can assume that this hex value represents a character which is 'b'. Trying out "1 b 214", I have found the key for phase 3!

#### PHASE\_4:

Phase 4 calls a function, func4, just after comparing eax to the value 1. This tells me that the input will be or include a digit. The jump instructions after this compare tells me that if the value in eax is equal to 1 or is less than 1, the bomb will explode. Trying "2" as my key, the bomb explodes due to a compare instruction that compares what appears to be my input, 2, against the value 0x37. 0x37h = 55d or char 7. Trying "55" as my key, I get caught hanging inside of func4.... I'm not too sure how I can determine the reason because gdb hangs here too. I'll try "7" instead. This time, the value in the eax register is set to 21.. I can't recognize a pattern yet. Perhaps eax is manipulated in func4. Iterating through func4 and checking the value of eax at each instruction, I can see that my input of 7 changes to 6, 5, 4, 3, 2, and then one before recursion of the function fails to provide a pattern. While iterating through the recurring function,

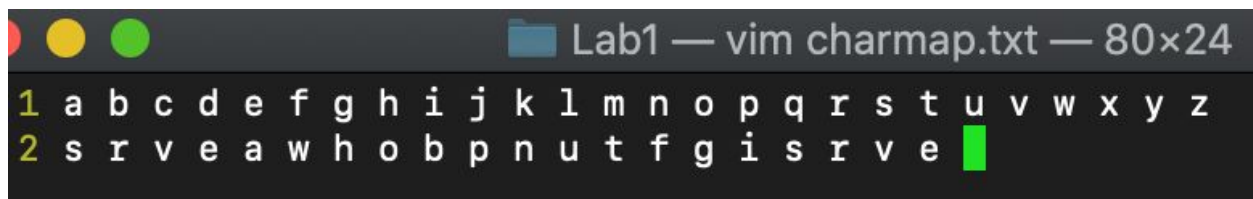
```
(gdb) ni
0x08048b60 in phase_2 ()
(gdb) i r
eax      0x6          6
ecx      0xffffcc63  -13213
edx      0x0          0
ebx      0xffffd2d4  -11564
esp      0xffffd1c0  0xffffd1c0
ebp      0xffffd1f8  0xffffd1f8
esi      0xf7fa9000  -134574080
edi      0xf7fa9000  -134574080
eip      0x8048b60   0x8048b60 <phase_2+24>
eflags   0x202       [ IF ]
cs       0x23        35
ss       0x2b        43
ds       0x2b        43
es       0x2b        43
fs       0x0          0
gs       0x63        99
(gdb)
```

```
0x08048c02 in phase_3 ()
(gdb)
0x08048c09 in phase_3 ()
(gdb)
0x08048c0f in phase_3 ()
(gdb)
BOOM!!!
The bomb has blown up. So I know I am on the right track!
[Inferior 1 (process 177893) exited with code 010]
(gdb)
```

I blew up the bomb before being able to check the `eax` register upon leaving `func4`. The next attempt I will try "8" and go slower with my iterations so that I can catch `eax` upon leaving `func4`. The `eax` register contained 34 upon explosion. I could not find a pattern to how the recursion manipulated my input. I started checking the value of `eax` just before the explosion for each value greater than 1. So far, I can see that the output of `func4` could be 2, 3, 5, 8, 13, 21, 34. I recognize this! It's the fibonacci sequence. Since I know that the accepted value could be 55, and that's the next number in the list of `eax` values that I built, I presume the input should be 9. I was right! The key for phase 4 is "9"!

#### PHASE\_5:

I see the `string_length` function is called again in phase 5, so the key must be a string. From the first phase, I know that the `string_length` function loads the length of the input string into `eax`. Just before the first instance where the bomb could explode, `eax` is compared to the value 6, so this tells me that the string should be 6 characters long. From my list of strings that could be possible keys, I can see that "giants" and "nobody" could be possible matches. Neither worked as the correct key and I can see that my input is being converted somehow. Instead of trying to understand the conversion, I first tried finding what my input was being compared to. I decided to check the value of `eax` before the `strings_not_equal` function and see that it is a jumbled string of 6 characters. This jumbled string changes with each input I give, so I know that this is my input. I need to find another string that is used in `string_not_equal`. I see that along with `eax` being pushed just before the function, a value in `0x804980b` is also pushed. This value is "giants". So my input must have to be converted into "giants". By testing many different inputs, I was able to build a mapping of each character to its respective conversion.



```
1 a b c d e f g h i j k l m n o p q r s t u v w x y z
2 s r v e a w h o b p n u t f g i s r v e
```

Using the map I created, I was able to decipher the correct input for phase 5 to be "opekmq".

#### PHASE\_6:

Reading six numbers again I see! I do not have enough time to look very far into this phase, but from what I can tell using the same methods as the previous phases, I can deduce that this phase uses a linked list of "nodes" to crack its key.