Lab 1 - GDB Bomb!
Alex Young
7/9/2021

Write-Up

Phase 1

I started with phase 1, and followed the example steps shown in the video under week 2 overview where the lecturer went over how to solve step 1 using gdb gef.
(Note that text output shown is from the original kali cmd prompt while the screenshots are from the zsh and gdb-gef because text output instructions did not work as described)

What this entailed was setting a breakpoint at phase_1 and then stepping through the code to see that there was a comparison between 0x80497c0 and eax, which contained the input.

```
8048b2c:    68 c0 97 04 08      push   $0x80497c0
8048b31:    50                  push   %eax
8048b32:    e8 f9 04 00 00      call   8049030 <strings_not_equal>
```

With this above assembly code from the objdump, I can confirm the same information that I saw in GDB. Using the examine in gef, this value is easily found to be Public speaking is very easy.

```
gef➤  x/s 0x80497c0
0x80497c0:      "Public speaking is very easy."
```

This is the phrase for phase 1, and can be also found using the strings command.

```
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.
```

Phase 2

This next phase was a lot more difficult for me to solve. I started by stepping through gdb instruction by instruction to understand what was going on. I was able to quickly find a function called <read_six_numbers>, but I was very careful here because I was not sure if the function name was supposed to be truthful or was a trick. Eventually I stepped past the instructions that are pasted below, which essentially ensured that there are more than 5 inputs.

```
8049007:    83 f8 05            cmp    $0x5,%eax
804900a:    7f 05               jg     8049011 <read_six_numbers+0x39>
804900c:    e8 eb 04 00 00      call   80494fc <explode_bomb>
```

Furthermore, I saw a register in the gdb that was loading in values for the sscanf that was in the format %d %d %d %d %d %d, which further confirmed that the input for phase 2 was 6 integers.

In order to confirm what these integers were I stepped down further into the instructions in gdb. I found that these following assembly instructions were repeated multiple times, and seemed to be in a for loop that executed 5 times.

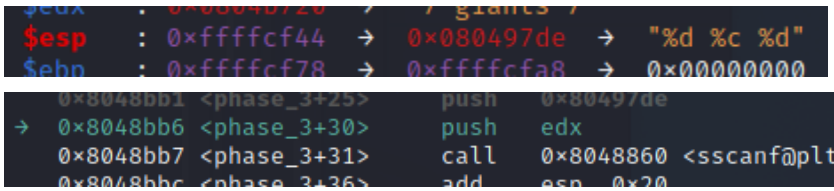
```
8048b76:    8d 43 01          lea    0x1(%ebx),%eax
8048b79:    0f af 44 9e fc    imul   -0x4(%esi,%ebx,4),%eax
8048b7e:    39 04 9e          cmp    %eax,(%esi,%ebx,4)
8048b81:    74 05             je     8048b88 <phase_2+0x40>
```

Essentially what I found was that every time I got the cmp, if the value of $eax (which was 0x1 0x2 0x6 0x18...) was not equal to the next input, the bomb would trigger. This value of $eax was shown in the gdb interface which was very useful.

When I looked at these values in decimal -- 1, 2, 6, 24… I realized a very obvious mathematical pattern and was able to guess the final values of 120 (x5) and 720 (x6).
This confirmed that the phase 2 solution is <u>1 2 6 24 120 720</u>

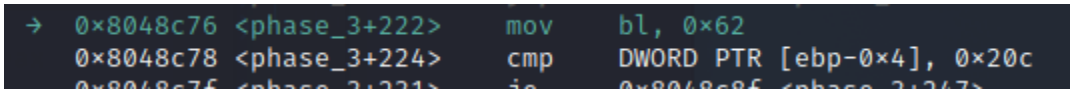
<u>Phase 3</u>
In this phase I started with a similar plan as to how I solved the second phase. I checked the objdump and slowly stepped through the gdb.

The first point of interest is that I found similar code that loaded in the input at the sscanf that confirmed the input was in the format “%d %c %d”.




Next, I checked the assembly code and found 9 cmps, which confused me because there were only 3 inputs. However, I was able to find out what was going on by testing a random input. My input was 7 h 96. I choose 7 as my first integer because the very first cmp checks that the input is less than or equal to 7 before continuing to the other 8 cmps. (As seen in the ss below, ja is jump above - which will jump to trigger a bomb)


However, I found that my code had skipped 7 of the cmps and ended up in the very last one (in order of the assembly code), which did a cmp to 0x20c.



```
8048c76:    b3 62                   mov    bl,0x62
8048c78:    81 7d fc 0c 02 00 00    cmp    DWORD PTR [ebp-0x4],0x20c
8048c7f:    74 0e                   je     8048c8f <phase_3+0xf7>
```

This was interesting because what this felt like to me was that the first integer value could correspond to 0-7 of the cmps in the assembly code, similar to a case statement. What this would mean is that perhaps if the first integer was a 7, then the other integer would be a 0x20c, which is 524.

However, this still left the character value in question. Here, I realized that each group of cmp had a line that updated a value of 'bl', and this was cmp to one last time at the end of phase_3.

```
8048c62:    eb 2b                   jmp    8048c8f <phase_3+0xf7>
8048c64:    b3 76                   mov    bl,0x76
8048c66:    81 7d fc 0c 03 00 00    cmp    DWORD PTR [ebp-0x4],0x30c
8048c6d:    74 20                   je     8048c8f <phase_3+0xf7>
8048c6f:    e8 88 08 00 00          call   80494fc <explode_bomb>
8048c74:    eb 19                   jmp    8048c8f <phase_3+0xf7>
8048c76:    b3 62                   mov    bl,0x62
8048c78:    81 7d fc 0c 02 00 00    cmp    DWORD PTR [ebp-0x4],0x20c
8048c7f:    74 0e                   je     8048c8f <phase_3+0xf7>
8048c81:    e8 76 08 00 00          call   80494fc <explode_bomb>
```

```
8048c8f:    3a 5d fb          cmp   -0x5(%ebp),%bl
8048c92:    74 05             je    8048c99 <phase_3+0x101>
8048c94:    e8 63 08 00 00    call  80494fc <explode_bomb>
```

Here the 'bl' value seems to be compared to the character value, which would indicate that for the 7, 524 set, the character value would be 0x62. This hex value corresponds to the character 'b'.

This would mean that the solution for phase 3 would be 7 b 524 - and when I entered this solution, it was correct. However, this would also mean that there are 7 other viable answers for phase 3, which is quite interesting.

Phase 4

The starting point for this phase seemed similar to the other phases, other than that it had an additional 'func4' in the assembly code. With this in mind, I used a trick that I found in phase 3 when I was stepping through the code.

```
8048cf0:    68 08 98 04 08          push   0x8049808
8048cf5:    52                      push   edx
8048cf6:    e8 65 fb ff ff          call   8048860 <sscanf@plt>
```

I had noticed that these lines of assembly indicated that on 0x8049808 the formatting for the sscanf was located. From this I examined this address in GEF and found that the format of this phrase is an integer:

```
gef➤  x/s 0x8049808
0x8049808:      "%d"
```

Finally, I found that after the input was loaded in and checked, there was only a single comparison to check the solution.

```
8048d14:       50                      push   eax
8048d15:       e8 86 ff ff ff          call   8048ca0 <func4>
8048d1a:       83 c4 10                add    esp,0x10
8048d1d:       83 f8 37                cmp    eax,0x37
8048d20:       74 05                   je     8048d27 <phase_4+0x47>
```

Here, on the line 8048d1d, the input value which is being updated in <func4> is being compared to 0x37, which is 55. I tried entering 55 as the solution, but the program seemed to stall out so I gave up. At this point, I did not really understand how func4 was interacting with the input so I decided to step through it in gdb.

Before I continue, one important thing to note here is that the input value that is being pushed into and out of func4 is stored in register $eax, as is shown in the above screenshot.

So, here I started with an input of 1 as phrase 4. I found that after func4 was called, the value of $eax was still 1, which triggered the bomb (as it was not 55).

```
   0x8048d14 <phase_4+52>     push   eax
   0x8048d15 <phase_4+53>     call   0x8048ca0 <func4>
→  0x8048d1a <phase_4+58>     add    esp, 0x10
```

```
$eax   : 0x1
```

Next I tried a value of 2 and found that after func4, $eax was still 2.

```
$eax   : 0x2
```

Next I tried 5, and found that after func4 was called, $eax changed to 8.

```
$eax   : 0x8
```

This was the first big change, so I decided to increment up a bit to 7 for the next test.

```
$eax   : 0x15
```

It appeared that I was getting close to the answer, so I decided to just run 8 and 9 to see if they would work. It turned out that 9 was the correct answer to phase 4. While I am not entirely sure what func4 was doing, it appeared to be increasing the value stored in $eap at an increasingly faster rate.

Phase 5

Once again, I started looking at phase 5 similarly to how I looked at the phases. From the start I noticed that in the phase_5 function itself there are no calls to sscanf, however there are calls to <string_length>, and <strings_not_equal>.

With this in mind, it appears that the solution is a string. Looking more closely at the objdump assembly code, it seems that there is a comparison between register $eax and 6, which due to the proximity to the string_length function made me feel that the string length should be 6.

```
8048d3b:       e8 d8 02 00 00          call   8049018 <string_length>
8048d40:       83 c4 10                add    esp,0x10
8048d43:       83 f8 06                cmp    eax,0x6
```

I was more assured of this when I checked the strings_not_equal. Similar to phase one, the code loads up a local string variable from memory to compare to the input value in register $eax.

```
8048d72:       68 0b 98 04 08          push   0x804980b
8048d77:       8d 45 f8                lea    eax,[ebp-0x8]
8048d7a:       50                      push   eax
8048d7b:       e8 b0 02 00 00          call   8049030 <strings_not_equal>
```

The address of the string is 0x804980b, which I examined using gdb.

```
gef➤  x/s 0x804980b
0x804980b:      "giants"
```

This came out to be “giants”, a 6 character string (confirming that the input is 6 characters long), that can also be found using the strings command. At this point I tried entering “giants” but the bomb triggered.

```
That's number 2.  Keep going!
7 b 524
Halfway there!
9
So you got that one.  Try this one.
giants

BOOM!!!
The bomb has blown up.
```

In order to understand how the code was changing the input into giants I loaded up gdb to step through phase_5, once again with the input “giants”.

At this point the very first thing I noticed was that in this section of the instructions, the code looped through each character.

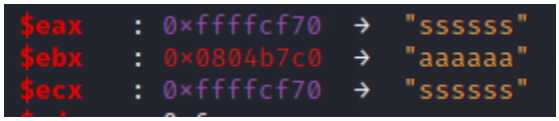
```
   0x8048d4f <phase_5+35>     lea    ecx, [ebp-0x8]
   0x8048d52 <phase_5+38>     mov    esi, 0x804b220
   0x8048d57 <phase_5+43>     mov    al, BYTE PTR [edx+ebx*1]
→  0x8048d5a <phase_5+46>     and    al, 0xf
   0x8048d5c <phase_5+48>     movsx  eax, al
   0x8048d5f <phase_5+51>     mov    al, BYTE PTR [eax+esi*1]
   0x8048d62 <phase_5+54>     mov    BYTE PTR [edx+ecx*1], al
   0x8048d65 <phase_5+57>     inc    edx
   0x8048d66 <phase_5+58>     cmp    edx, 0x5
```

After the loop was over, it appeared that the input had been changed from “giants” to “hbsfev”.

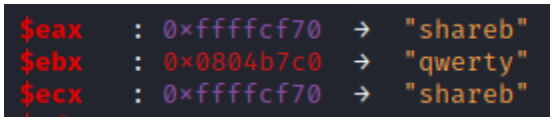
```
$ebx   : 0x0804b7c0  →  "giants"
$ecx   : 0xffffcf70  →  "hbsfev"
```

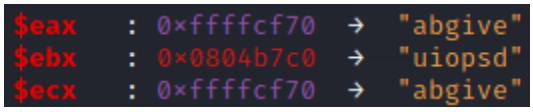
The value in register $ecx was later loaded into $eax to compare to “giants”. It appeared that the fifth phrase would need to be encrypted by this loop into “giants” in order to defuse the bomb.

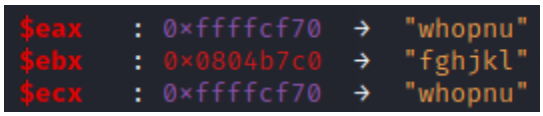
In order to attempt to understand how the encryption worked, I set up a breakpoint and tested out “aaaaaa”.

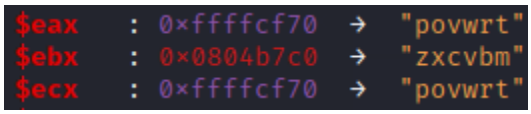



In this instance, the value of a seemed to be incrementing by a value of 18, however this method doesn’t seem foolproof because previously “g” incremented to “h” by a value of 1. However, what this does seem to show is that no matter which character ‘a’ appears at, it will always become an ‘s’.



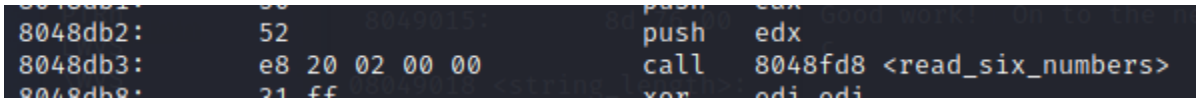




Here, I ran the test 4 more times and was able to figure out what each character was encrypted to . To decrypt “giants” → we would get: g → o, i → p, a → e (or a → u), n → k, t → m, s → a (or s → q).

This spells out opekma or opukmq, and there are likely other possible combinations that I did not find in my search. Both answers appear to work after testing them out.

Phase 6

Similarly to the previous phases, I start this phase out by searching through the objdump for noticeable clues about what is going on in the assembly code. The first thing that pops up is that <read_six_numbers> is called, which as I found in phase 2, means that the input will be 6 integers.



Next I checked for all cmp instructions, which in the past usually appear to indicate checks to values or loops. In order to find possible loops, I made sure to pay attention to the jump values to see if they went to previous lines.

```
8048dc6:      48                      dec    eax
8048dc7:      83 f8 05                cmp    eax,0x5
8048dca:      76 05                   jbe    8048dd1 <phase_6+0x39>
8048dcc:      e8 2b 07 00 00          call   80494fc <explode_bomb>
8048dd1:      8d 5f 01                lea    ebx,[edi+0x1]
```

```
8048de6:      8b 55 c8                mov    edx,DWORD PTR [ebp-0x38]
8048de9:      8b 04 32                mov    eax,DWORD PTR [edx+esi*1]
8048dec:      3b 04 9e                cmp    eax,DWORD PTR [esi+ebx*4]
8048def:      75 05                   jne    8048df6 <phase_6+0x5e>
8048df1:      e8 06 07 00 00          call   80494fc <explode_bomb>
8048df6:      43                      inc    ebx
8048df7:      83 fb 05                cmp    ebx,0x5
8048dfa:      7e ea                   jle    8048de6 <phase_6+0x4e>
8048dfc:      47                      inc    edi
8048dfd:      83 ff 05                cmp    edi,0x5
8048e00:      7e be                   jle    8048dc0 <phase_6+0x28>
```

The cmp here appears to be a loop that increments through all 6 characters in a loop from 0-5. It then jumps up to 8048dc0, which executes the cmp in the above code, which appears to compare the values in $eax to 5 and jumps to trigger a bomb if it is not below or equal to 5.

In order to confirm what values were being stored in $eax, I loaded into gdb with inputs of 1 1 5 5 7 7 and then 7 7 7 7 7 7. I was able to find that $eax would store values of the input minus 1, meaning values of 6 or below were allowed.

At this point I was quite confused because I would always trigger the bomb at line 8048df1 in the screenshot above. However, after a while I tested out 1 2 3 4 5 6, and it seemed to be the case that this was checking for unique inputs.

At this point I got even more confused because there were a whole lot of steps that I was able to step through without triggering a bomb until the very end of the phase. This was especially confusing because there didn't seem to be any large loops or for there to be any register that was storing 6 values.

At this point I set down a breakpoint because I felt that perhaps I could try to investigate what exactly was happening and what was being compared in the very end.

```
   0x8048e70 <phase_6+216>    mov    edx, DWORD PTR [esi+0x8]
   0x8048e73 <phase_6+219>    mov    eax, DWORD PTR [esi]
   0x8048e75 <phase_6+221>    cmp    eax, DWORD PTR [edx]
   0x8048e77 <phase_6+223>    jge    0x8048e7e <phase_6+230>   NOT taken [Reason: !(S==O)]
   0x8048e79 <phase_6+225>    call   0x80494fc <explode_bomb>
   0x8048e7e <phase_6+230>    mov    esi, DWORD PTR [esi+0x8]
```

After a while of searching, I eventually decided to examine what were in registers $eax and $edx.

Here I found something interesting, while $eax did not allow me to access what it stored, $edx appeared to be storing a value called <node2> which had a value of 0x2d5

```
gef➤  x/8x $edx
0x804b260 <node2>:      0xd5    0x02    0x00    0x00    0x02    0x00    0x00    0x00
```

I checked the other registers for anything similar, and the only other register that had anything was $esi. This makes sense because in the assembly in the gdb screenshot above, the value of $edx is being stored from the value of $esi+0x8.

```
gef➤  x/8x $esi
0x804b26c <node1>:      0xfd    0x00    0x00    0x00    0x01    0x00    0x00    0x00
```

$esi stored <node1> with a value of 0xfd.

However, while this was interesting, it didn't seem to serve any practical usage. At this time it took a while, but looking through the assembly objdump made some sense.

```
8048e70:    8b 56 08          mov   0x8(%esi),%edx
8048e73:    8b 06             mov   (%esi),%eax
8048e75:    3b 02             cmp   (%edx),%eax
8048e77:    7d 05             jge   8048e7e <phase_6+0xe6>
8048e79:    e8 7e 06 00 00    call  80494fc <explode_bomb>
```

It seemed that the important part was the cmp between $edx and $eax (which was $esi). If $eax <node1> was greater than $edx <node2>, then the bomb would be jumped over - because jge will jump if the previous compare sets a flag that indicates the value was greater than/equal to it.

Given that my input to the problem had been 1 2 3 4 5 6, I decided to try 2 1 3 4 5 6 to see if the two nodes switched locations.

```
gef➤  x/8x $esi
0x804b260 <node2>:      0xd5    0x02    0x00    0x00    0x02    0x00    0x00    0x00
gef➤  x/8x $edx
0x804b26c <node1>:      0xfd    0x00    0x00    0x00    0x01    0x00    0x00    0x00
gef➤
```

It worked as I hoped. However, it soon looped back around to check the nodes again at a new address. It seemed that there were likely more than just 2 nodes that needed to be ordered.

In order to find these nodes, I looked at the addresses of the previous nodes - <node1> was in is 0x804b26c and <node2> 0x804b260
Because the value of $edx is incrementing in assembly by 0x8, I examined 0x804b264.

```
gef➤  x/8x 0x804b264
0x804b264 <node2+4>:    0x02    0x00    0x00    0x00    0x54    0xb2    0x04    0x08
gef➤  x/8x 0x804b26b
0x804b26b <node2+11>:   0x08    0xfd    0x00    0x00    0x00    0x01    0x00    0x00
```

From this it seems like the size of a node is 12, which is the difference between nodes 1 and 2. Here I continued to subtract by 12 to see how many nodes there were.

```
gef➤  x/8x $esi
0x804b26c <node1>:      0xfd    0x00    0x00    0x00    0x01    0x00    0x00    0x00
gef➤  x/8x $esi-0xc
0x804b260 <node2>:      0xd5    0x02    0x00    0x00    0x02    0x00    0x00    0x00
gef➤  x/8x $esi-0xc-0xc
0x804b254 <node3>:      0x2d    0x01    0x00    0x00    0x03    0x00    0x00    0x00
gef➤  x/8x $esi-0xc-0xc-0xc
0x804b248 <node4>:      0xe5    0x03    0x00    0x00    0x04    0x00    0x00    0x00
gef➤  x/8x $esi-0xc-0xc-0xc-0xc
0x804b23c <node5>:      0xd4    0x00    0x00    0x00    0x05    0x00    0x00    0x00
gef➤  x/8x $esi-0xc-0xc-0xc-0xc-0xc
0x804b230 <node6>:      0xb0    0x01    0x00    0x00    0x06    0x00    0x00    0x00
gef➤  x/8x $esi-0xc-0xc-0xc-0xc-0xc-0xc
0x804b224:      0x65    0x61    0x77    0x68    0x6f    0x62    0x70    0x6e
gef➤
```

There were 6 nodes, which matched up perfectly with the number of values I had to find.
1: fd = 253  2: 2d5 = 725  3: 12d = 301  4: 3e5 = 997  5: 4d = 77  6: 1b0 = 432

It seemed as if all I had to do was order the nodes by their values from greatest to least. Using the values listed above, this gave me <nodes: 4 2 6 3 1 5 >

It turns out that this is the correct answer to the sixth phase, helping defuse the final bomb and complete the puzzle.

Conclusion

My final solutions to all the phrases are:
P1: Public speaking is very easy.
P2: 1 2 6 24 120 720
P3: 7 b 524
P4: 9
P5: opekma
P6: 4 2 6 3 1 5

```
┌──(kali㉿kali:pts/8)
└─(8:07:03:%)── ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
7 b 524
Halfway there!
9
So you got that one.  Try this one.
opekma
Good work!  On to the next...
4 2 6 3 1 5
Congratulations! You've defused the bomb!
```