

Tu Lam

CS 373 (Defense Against the Dark Arts)

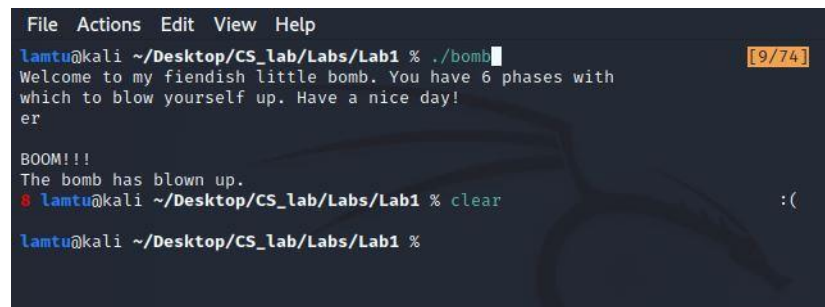
Dr. Bram Lewis

October 13, 2021

## **Lab #1 (GDB Bomb)**

This lab deal with the first lab of the class, GDB Bomb. In this lab, we learn about how to setup our own VM and run the bomb and learn to decode and figure out how to defuse the bomb through their six phases. Through this, we will explore it and report our finding through this first lab of the first bomb malware.

At the start, we were asked to setup the whole VM for ourselves, and it took a while to setup, but we manage to get it up and running. Then we install the necessary file that is needed for the lab before we even start the lab. Once we got the code, we run the file for lab. At the start of the lab, I run the program to see how it suppose to run. Once it starts up, it greeted me a message and waiting for the input from me. Then I enter something random and explode. Below is a figure of me running the program for the first time.



```
File Actions Edit View Help
lamtu@kali ~/Desktop/CS_lab/Labs/Lab1 % ./bomb [9/74]
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
er

BOOM!!!
The bomb has blown up.
lamtu@kali ~/Desktop/CS_lab/Labs/Lab1 % clear :(
lamtu@kali ~/Desktop/CS_lab/Labs/Lab1 %
```

**Figure #1:** Showing the process of just running the bomb

From there, I didn't know what the right way to defuse the bomb before it explodes on me after the attempt to guess it. Then I move onto GDB to see which where the code can help us reverse the order and help me decode the process. Below is an image of me trying to use GDB and run the program to see if it can detect it, but it wasn't giving me enough detail about it. So, I try to start adding breakpoint to help running the GDB hoping to find the answer.

```

This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
run
GEF for linux ready, type 'gef' to start, 'gef config' to configure
96 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
Reading symbols from ./bomb...
gef> run
Starting program: /home/lantu/Desktop/CS_lab/Labs/Lab1/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
red

BOOM!!!
The bomb has blown up.
[Inferior 1 (process 8956) exited with code 010]
gef>

```

**Figure #2:** First time running GDB with the bomb file

Then I move onto adding breakpoint in main hoping to go through the code one-by-one to see how the code would run. Below is another image of the GDB entering breakpoint mode and it shows the program starting position of where the breakpoint is located and the assembly language of it.

```

$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0006[0/146]
stack
0xffffcec0 +0x0000: 0x00000000 ← $esp
0xffffcec4 +0x0004: 0x00000000
0xffffcec8 +0x0008: 0x00000000
0xffffcecc +0x000c: 0xffffced8 → 0x00000000
0xffffced0 +0x0010: 0x0004870a → <_init+42> mov ebx, DWORD PTR [ebp-0x4]
0xffffced4 +0x0014: 0x00000000
0xffffced8 +0x0018: 0x00000000 ← $ebp
0xffffcedc +0x001c: 0xf7de0fd6 → <_libc_start_main+262> add esp, 0x10
code:x86:32
0x80489b1 <main+1> mov ebp, esp
0x80489b3 <main+3> sub esp, 0x14
0x80489b6 <main+6> push ebx
→ 0x80489b7 <main+7> mov eax, DWORD PTR [ebp+0x8]
0x80489ba <main+10> mov ebx, DWORD PTR [ebp+0xc]
0x80489bd <main+13> cmp eax, 0x1
0x80489c0 <main+16> jne 0x80489d0 <main+32>
0x80489c2 <main+18> mov eax, ds:0x804b664
0x80489c7 <main+23> mov ds:0x804b664, eax
threads
[#0] Id 1, Name: "bomb", stopped 0x80489b7 in main (), reason: BREAKPOINT
trace
[#0] 0x80489b7 → main(argc=0xf7faea28, argv=0x0)
gef>

```

**Figure #3:** GDB running with breakpoint

After setting the breakpoint at main, I was able to use GDB and find “phase\_1” code and set another breakpoint there. I then rerun the code and continue the code and hit that next breakpoint. Then using that new breakpoint, I was able to find my input string was sitting at address **0x804b680** which I will highlight in the figure below. I also found where the answer it through this reverse lookup and found the correct input is at the address of **0x80497c0**. Then I was trying different command to print out the solution and found that the message that was correct was “**Public speaking is very easy.**”. After I try this code out and it works, and I was able move to phase 2.

```

0xffffcea0 +0x0000: 0xf7fac000 → 0x001e9d6c ← $esp
0xffffcea4 +0x0004: 0xf7ffd9a0 → 0x00000000
0xffffcea8 +0x0008: 0xffffced8 → 0x00000000 ← $ebp
0xffffceac +0x000c: 0x08048a60 → <main+176> call 0x804952c <phase_defused>
0xffffceb0 +0x0010: 0x0804b680 → "string" ← my input
0xffffceb4 +0x0014: 0x0804b518 → 0x0804b598 → <_DYNAMIC+0> add DWORD PTR [eax], eax
0xffffceb8 +0x0018: 0xffffced8 → 0x00000000
0xffffcebc +0x001c: 0x08048a57 → <main+167> add esp, 0xfffffff4

0x8048b20 <phase_1+0> push ebp
0x8048b21 <phase_1+1> mov ebp, esp
0x8048b23 <phase_1+3> sub esp, 0x8
→ 0x8048b26 <phase_1+6> mov eax, DWORD PTR [ebp+0x8]
0x8048b29 <phase_1+9> add esp, 0xfffffff8
0x8048b2c <phase_1+12> push 0x80497c0 ← the answer
0x8048b31 <phase_1+17> push eax
0x8048b32 <phase_1+18> call 0x8049030 <strings_not_equal>
0x8048b37 <phase_1+23> add esp, 0x10

[#0] Id 1, Name: "bomb", stopped 0x8048b26 in phase_1(), reason: BREAKPOINT

[#0] 0x8048b26 → phase_1()
[#1] 0x8048a60 → main(argc=0x804b680, argv=0xffffcf84)

gef> p /25c 0x80497c0
Item count other than 1 is meaningless in "print" command.
gef> s /25c 0x80497c0
A syntax error in expression, near `/25c 0x80497c0'.
gef> x /25c 0x80497c0
0x80497c0: 0x50 0x75 0x62 0x6c 0x69 0x63 0x20 0x73
0x80497c8: 0x70 0x65 0x61 0x6b 0x69 0x6e 0x67 0x20
0x80497d0: 0x69 0x73 0x20 0x76 0x65 0x72 0x79 0x20
0x80497d8: 0x65

```

**Figure #4:** GDB detecting the string input and answer key

Now moving onto phase 2, I decide to put another breakpoint next to it and try to debug it. Going through the steps again, I look at the GDB and found out that the debugger for phase 2 is looking for a 6-digit numbers at address **0x8048fd8**. Then I use “disas” command to look at the assembly code before the bomb went off after guessing and using the “ni” command to go through the assembly language. Went through it and found out that the first number is 1, follow by the number 2, 6, 24, 120, 720. This number I got is that when I look at the assembly code at address 0x8048b79, it uses \$eax times it with \$ebx and every time \$ebx get incremented by 1, it multiplies that to \$eax number. So, 1 \* 1, then 2 \* 3, 6 \* 4, 24 \* 5, 120 \* 6. And that’s where we got “**1 2 6 24 120 720**” as our answer.

```

gef> disas
Dump of assembler code for function phase_2:
0x08048b45 <+0>: push ebp
0x08048b49 <+1>: mov ebp, esp
0x08048b4b <+3>: sub esp, 0x20
0x08048b4e <+6>: push esi
0x08048b4f <+7>: push ebx
0x08048b50 <+8>: mov edx, DWORD PTR [ebp+0x8]
0x08048b53 <+11>: add esp, 0xfffffff8
0x08048b56 <+14>: lea eax, [ebp-0x18]
0x08048b59 <+17>: push eax
0x08048b5a <+18>: push edx
0x08048b5b <+19>: call 0x8048fd8 <read_six_numbers>
0x08048b5e <+24>: add esp, 0x10
0x08048b63 <+27>: cmp DWORD PTR [ebp-0x18], 0x1
0x08048b67 <+31>: je 0x8048b6e <phase_2+38>
0x08048b69 <+33>: call 0x80494fc <explode_bomb>
0x08048b6e <+38>: mov ebx, 0x1
0x08048b73 <+43>: lea esi, [ebp-0x18]
0x08048b76 <+46>: lea eax, [ebx+0x1]
0x08048b79 <+49>: imul eax, DWORD PTR [esi+ebx*4-0x4]
→ 0x08048b7e <+56>: cmp DWORD PTR [esi+ebx*4], eax
0x08048b81 <+57>: je 0x8048b88 <phase_2+64>
0x08048b83 <+59>: call 0x80494fc <explode_bomb>
0x08048b88 <+64>: inc ebx
0x08048b89 <+65>: cmp ebx, 0x5
0x08048b8c <+68>: jle 0x8048b76 <phase_2+46>
0x08048b8e <+70>: lea esp, [ebp-0x28]
0x08048b91 <+73>: pop ebx
0x08048b92 <+74>: pop esi
0x08048b93 <+75>: mov esp, ebp
0x08048b95 <+77>: pop ebp
0x08048b96 <+78>: ret
End of assembler dump.

```

**Figure #5:** Using “disas” to break down the code

Moving onto phase 3, putting in another breakpoint as phase 3, we found out that at address **0x80497de** that it is looking for a “number char number”. From there back to the objdump to find out about the result using “disas”. We found that the first number is 1 and the last set of number 214 in the code as on the address **0x8048c02** with the hex of **0xd6** which translate to 214. And at

address 0x8048c8f where the char is being compared to "bl" which hold the hex of "62" and that convert to ASCII to 98 which is the letter 'b' in our case. So, for phase 3, the phrase is "**1 b 214**".

To the next phase, this phase looks like it is looking for a number represented by the "%d" symbol at address **0x8049808**. Using the objdump and guessing the number as 1 at first, I was able to trace to the assembly code and find out that it is comparing the hex 0x37 which 55 in decimal value and we need a number that will add to 55 when it reaches to that comparison. We also look inside the "func\_4" code and found out that it is doing the Fibonacci sequence. Looking up the Fibonacci sequence, we see that the 9<sup>th</sup> place in Fibonacci respond to 55 and the number we were looking for is **9**.

Moving to the next phase which 5, and adding in another breakpoint, we find that at address **0x8049018** called function "string\_length" which means that we are looking for a phase. Using objdump, we found out that the answer is at address **0x8048d72** where it is comparing the address **0x804980b** to see if it matches. I found that, the word was six characters long and it was "giants". I thought this was correct, but was wrong, when look at the code for those six characters, the result got cypher and I would have to decode it. Just putting in 'abcdef' as a testing string, I was able to pinpoint that it translates to '**opekmq**' as the answer.

Lastly the last phase, once again adding in another breakpoint to help identify with the next code. This one is dealing with a six number digits, and when looking at it through the "disas" through GDB, we found that the number is being compared to the number 6 and the six-digit must be below or equal to the number 6 and each can't be the same. Through looking at the comparison and trial and error, we find that the code is '4 2 6 3 1 5' to be the combination and we are done.

Through this lab, we were able to decode a bomb that ask us for input and looking through the GDB, we were able to crack it. Reverse GDB is a fun and good skill to know when look through this lab and I have a good grasp about the concept when it comes to the skill to use the GDB. Overall, this lab was a good way to introduce VM and the way to identify the threat of this bomb malware and crack it.