Introduction to Reverse Engineering

By Alexander Wang

42 Silicon Valley^{TM}

Foreword

Warning: Out of all the resources for this curriculum, this will be by far the worst one because it is not possible to teach this effectively without a video of some kind or at least in person. Unlike the other ones, I will be providing a large number of links for resources as an alternative.

Welcome back to another exciting episode of our courses on reverse engineering. The main topic of the day and the thing that all of you have been waiting for is reverse engineering – probably one of the most glorified and poorly portrayed thing in all of film.

It takes a very special group of people to be able to do it. If you get frustrated easily or irritated then this is seriously not a good path to go down and I highly recommend you stop here.

However since you have gotten this far then you managed to go through the hell that was called Assembly. And you now should… in theory, be able to read it with some documentations needed on the more interesting functions.

Now, what is reverse engineering?

Pretty simple question with a fairly complex answer. It depends on what we are talking about. To reverse engineer is to understand the underlying functionality of a program (this document). However, it could also mean to gain illegal or legal access (if somebody gives you explicit permission to do so) to a program and do nasty things to them – think hacking videogames for the less nice area of reverse engineering.

This will *not be a document on doing illegal things*. This will be a really tiny taste of reverse engineering binaries. As such if you did the projects previously and did not half-ass your way through it. These should be *very easy* by comparison up until the next section.

Let’s begin.

Chapter 1: Numbers? Again? WHY?!

“Et tu, Brute?” – Julius Caesar as he is being stabbed, *Shakespeare*

Introduction:

Now I know what you’re thinking. Numbers? Again? This is torture, and insert some other complaint. But I have a good reason this time! You can apply this to your reverse engineering *and* it’s practical now too. However, you will be using **hexadecimals** from now on. **Use of decimal numbers is strictly forbidden** unless told otherwise. But, since your author is lazy, he will be breaking that rule almost immediately by having you look at something kind of interesting and you will see why using normal base-10 (decimal system) let alone binary is a really bad choice.

Relationship of Files, binary number, and hexadecimals

Let us look at this seemingly innocuous line of assembly:

```
movl $1, %eax
```

If you did the previous projects you should know that this sets the value of register EAX to 1 or equivalently:

EAX = 1

However, that is not how a computer looks at anything. Everything is compiled down to 0’s and 1’s (binary digits, remember those?). So how would this look like in binary? Something like the following:

1011100001000000

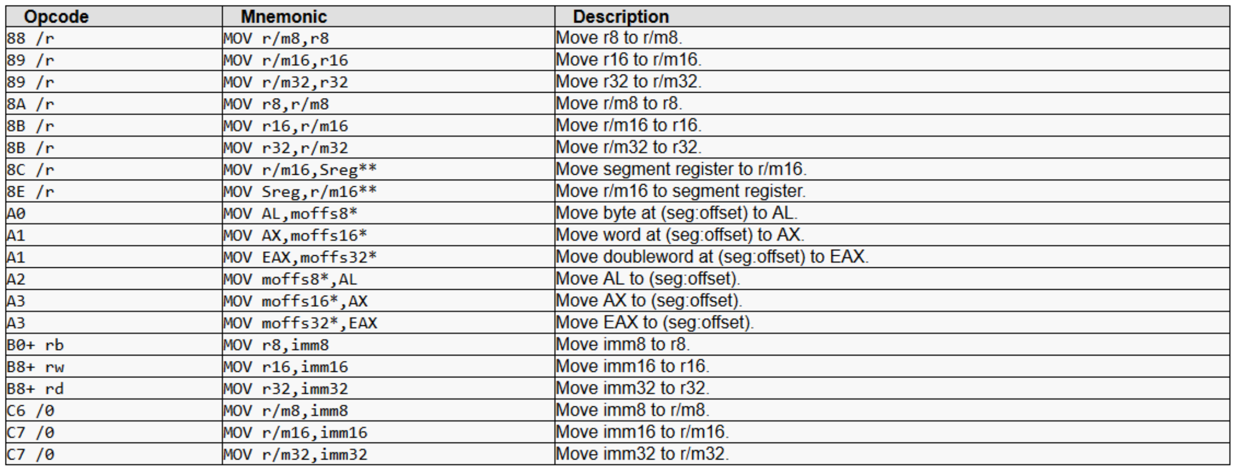
Alternatively, in slightly more human readable format [in hexadecimal separated by spaces every 2]:

BA 01 00 00 00

Those 10 digits is equivalent to that line of Assembly. How? Every single operation done on the CPU is called an *operation code* or opcode for short, and of course, the CPU can't tell you which operation you want to do before first telling it what you want to do. So, every *opcode* has an associated value.

One problem: the same opcode has multiple different possibilities and different values. Getting exactly the one you want is difficult. Could you imagine writing hexadecimals or binary and having to figure out which number to use for the correct operation?

This is what move looks like:



Opcode	Mnemonic	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8.
89 /r	MOV r/m16,r16	Move r16 to r/m16.
89 /r	MOV r/m32,r32	Move r32 to r/m32.
8A /r	MOV r8,r/m8	Move r/m8 to r8.
8B /r	MOV r16,r/m16	Move r/m16 to r16.
8B /r	MOV r32,r/m32	Move r/m32 to r32.
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16.
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register.
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL.
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX.
A2	MOV moffs8*,AL	Move AL to (seg:offset).
A3	MOV moffs16*,AX	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	Move EAX to (seg:offset).
B0+ rb	MOV r8,imm8	Move imm8 to r8.
B8+ rw	MOV r16,imm16	Move imm16 to r16.
B8+ rd	MOV r32,imm32	Move imm32 to r32.
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8.
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16.
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32.

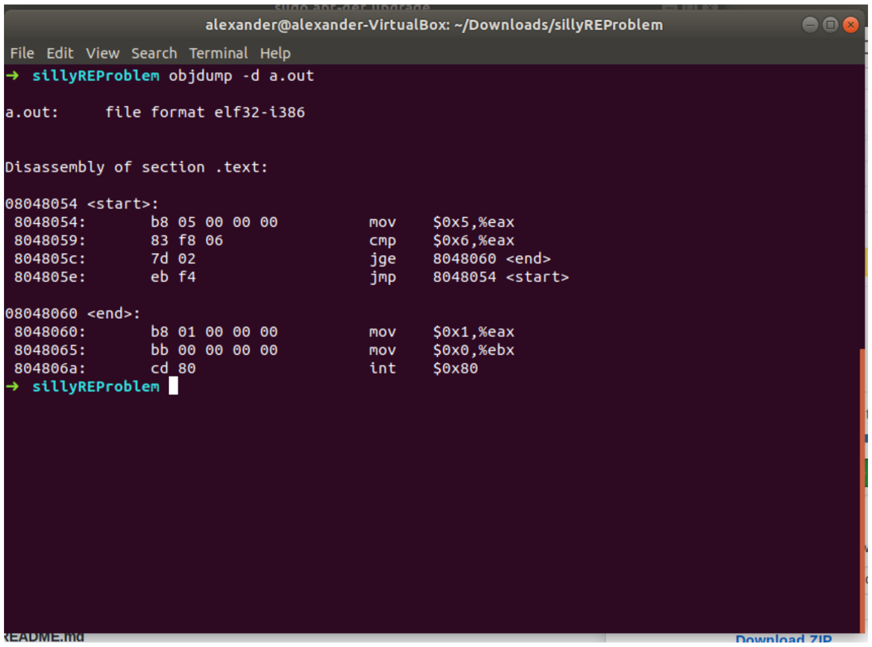
Figure 1 http://ref.x86asm.net/coder32.html

At a glance, there are 20 different numbers just to do move. So… why am I telling you this?

For your first trick: you will be editing a binary with *only a hex editor*; don't worry there is actually a very good reason for this: for when viruses outright don't let you use a debugger on it, but you can still manually edit the hex values by hand.

Now this is probably the most painful part of this introduction. On the bright side, you will only be modifying three operations to do something different in the singular exercise on this. Let's get started.

Example Project:


alexander@alexander-VirtualBox: ~/Downloads/sillyREProblem
File Edit View Search Terminal Help

```
➜ sillyREProblem objdump -d a.out

a.out:     file format elf32-i386


Disassembly of section .text:

08048054 <start>:
 8048054:       b8 05 00 00 00          mov    $0x5,%eax
 8048059:       83 f8 06                cmp    $0x6,%eax
 804805c:       7d 02                   jge    8048060 <end>
 804805e:       eb f4                   jmp    8048054 <start>

08048060 <end>:
 8048060:       b8 01 00 00 00          mov    $0x1,%eax
 8048065:       bb 00 00 00 00          mov    $0x0,%ebx
 804806a:       cd 80                   int    $0x80
➜ sillyREProblem
```

Figure 2

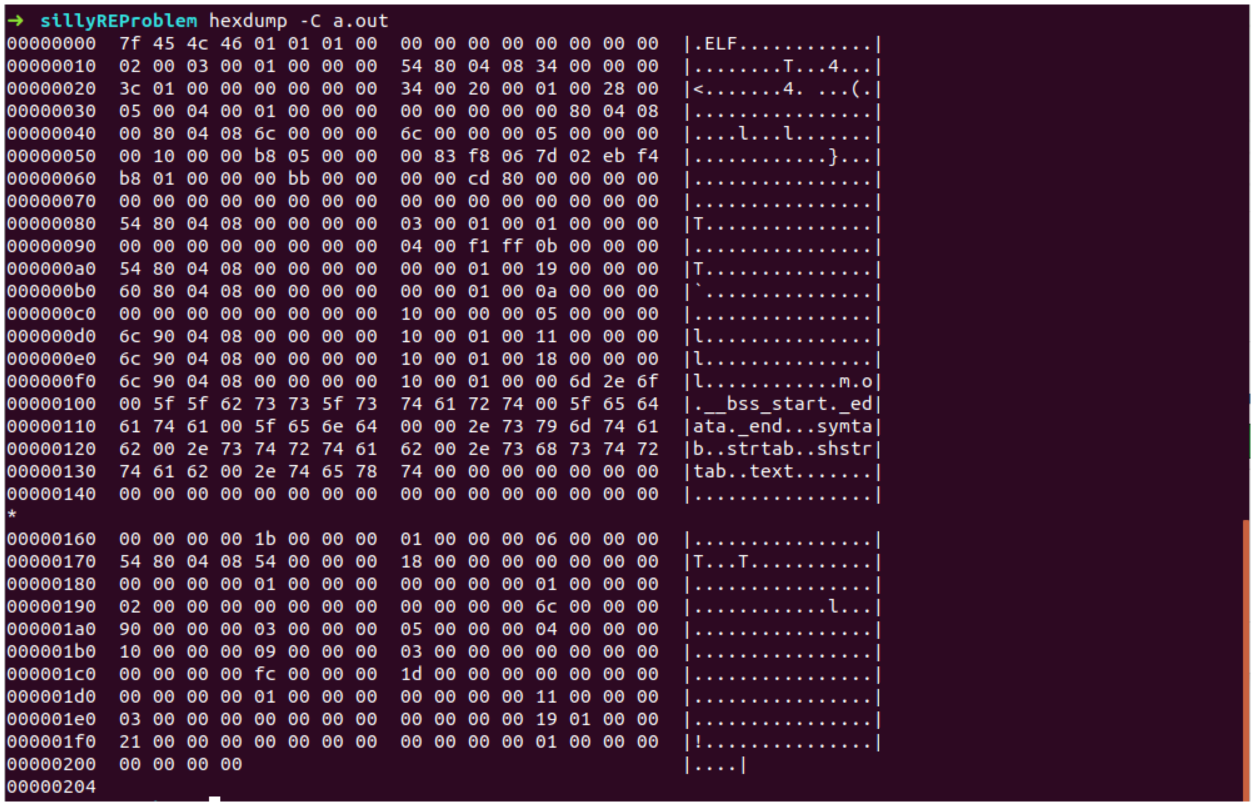
The above image is a disassembly of a ridiculously simple program. I will not bother explaining what it does because by this point you should be able to read Assembly especially if it is something this trivial.

Now let’s look at this carefully to understand what we need to do. We know it’s an infinite loop and we want to exit out of it. So we have two options immediately:

1. Change the “jge” to a “jmp”
2. Change the comparison

Pick your own adventure.

Change the JGE to a JMP:

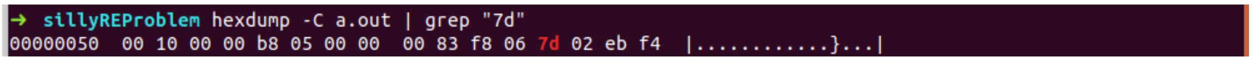


```
➜ sillyREProblem hexdump -C a.out
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |........T...4...|
00000020  3c 01 00 00 00 00 00 00  34 00 20 00 01 00 28 00  |<.......4. ...(.|
00000030  05 00 04 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
00000040  00 80 04 08 6c 00 00 00  6c 00 00 00 05 00 00 00  |....l...l.......|
00000050  00 10 00 00 b8 05 00 00  00 83 f8 06 7d 02 eb f4  |............}...|
00000060  b8 01 00 00 00 bb 00 00  00 00 cd 80 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  54 80 04 08 00 00 00 00  03 00 01 00 01 00 00 00  |T...............|
00000090  00 00 00 00 00 00 00 00  04 00 f1 ff 0b 00 00 00  |................|
000000a0  54 80 04 08 00 00 00 00  00 00 01 00 19 00 00 00  |T...............|
000000b0  60 80 04 08 00 00 00 00  00 00 01 00 0a 00 00 00  |`...............|
000000c0  00 00 00 00 00 00 00 00  10 00 00 00 05 00 00 00  |................|
000000d0  6c 90 04 08 00 00 00 00  10 00 01 00 11 00 00 00  |l...............|
000000e0  6c 90 04 08 00 00 00 00  10 00 01 00 18 00 00 00  |l...............|
000000f0  6c 90 04 08 00 00 00 00  10 00 01 00 00 6d 2e 6f  |l............m.o|
00000100  00 5f 5f 62 73 73 5f 73  74 61 72 74 00 5f 65 64  |.__bss_start._ed|
00000110  61 74 61 00 5f 65 6e 64  00 00 2e 73 79 6d 74 61  |ata._end...symta|
00000120  62 00 2e 73 74 72 74 61  62 00 2e 73 68 73 74 72  |b..strtab..shstr|
00000130  74 61 62 00 2e 74 65 78  74 00 00 00 00 00 00 00  |tab..text.......|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000160  00 00 00 00 1b 00 00 00  01 00 00 00 06 00 00 00  |................|
00000170  54 80 04 08 54 00 00 00  18 00 00 00 00 00 00 00  |T...T...........|
00000180  00 00 00 00 01 00 00 00  00 00 00 00 01 00 00 00  |................|
00000190  02 00 00 00 00 00 00 00  00 00 00 00 6c 00 00 00  |............l...|
000001a0  90 00 00 00 03 00 00 00  05 00 00 00 04 00 00 00  |................|
000001b0  10 00 00 00 09 00 00 00  03 00 00 00 00 00 00 00  |................|
000001c0  00 00 00 00 fc 00 00 00  1d 00 00 00 00 00 00 00  |................|
000001d0  00 00 00 00 01 00 00 00  00 00 00 00 11 00 00 00  |................|
000001e0  03 00 00 00 00 00 00 00  00 00 00 00 19 01 00 00  |................|
000001f0  21 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |!...............|
00000200  00 00 00 00                                       |....|
00000204
```

So, there are a number of ways you can go about this. I will do the simple way. I will parse the output using "grep" for the "jge end" except I can't do that directly and I must find the hexadecimal equivalent which is: "7d 02". So it will look like this:

hexdump –C a.out | grep "7d 02"

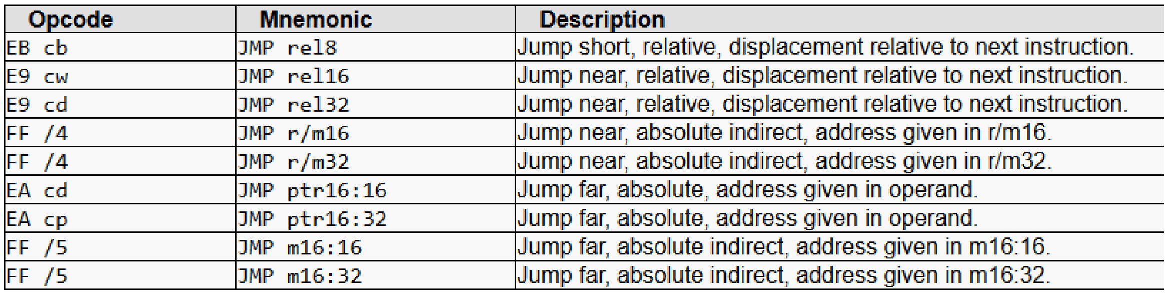
With the following output:



```
➜ sillyREProblem hexdump -C a.out | grep "7d"
00000050  00 10 00 00 b8 05 00 00  00 83 f8 06 7d 02 eb f4  |............}...|
```

Now from here we know the exact line we need to change. But wait, what is the opcode for jmp? Let's look at a trusty site:

Jump



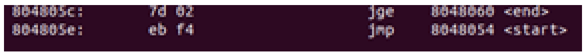
Opcode	Mnemonic	Description
`EB cb`	`JMP rel8`	Jump short, relative, displacement relative to next instruction.
`E9 cw`	`JMP rel16`	Jump near, relative, displacement relative to next instruction.
`E9 cd`	`JMP rel32`	Jump near, relative, displacement relative to next instruction.
`FF /4`	`JMP r/m16`	Jump near, absolute indirect, address given in r/m16.
`FF /4`	`JMP r/m32`	Jump near, absolute indirect, address given in r/m32.
`EA cd`	`JMP ptr16:16`	Jump far, absolute, address given in operand.
`EA cp`	`JMP ptr16:32`	Jump far, absolute, address given in operand.
`FF /5`	`JMP m16:16`	Jump far, absolute indirect, address given in m16:16.
`FF /5`	`JMP m16:32`	Jump far, absolute indirect, address given in m16:32.

Figure 3 https://c9x.me/x86/html/file_module_x86_id_147.html

But which one? The first one of course!

Why is that? The description says “displacement relative to the next instruction”.

Looking back at our disassembly we have the following two lines:



```
804805c:        7d 02                   jge     8048060 <end>
804805e:        eb f4                   jmp     8048054 <start>
```

Reading this we know the following:

1. “7d 02” (or jge) starts on the address 0x804805c
2. “eb f4” (or jmp) starts on address 0x804805e.
3. End is on address 0x8048060
4. The “02” looks like an offset from 0x804805e. This suspicion is confirmed from “jge 8048060 <end>” to the far right. We can confirm this by doing some addition: $0x804805e + 0x2 = 0x8048060$.
5. Since it’s a displacement we can narrow it down to one of the first 3 in the table above.

With the above facts which one could we choose?

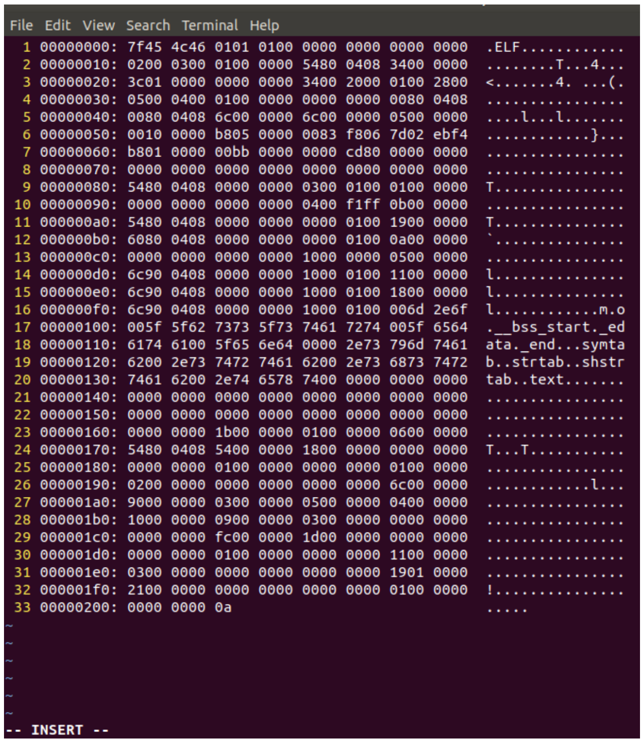
To be honest, any of them would work just fine. But, I’m going to decide on using the first one “eb”

Now how would we write in hexadecimal? With a hex editor or our wonderful vim editor in the following steps (alternatively:

https://vi.stackexchange.com/questions/2232/how-can-i-use-vim-as-a-hex-editor):

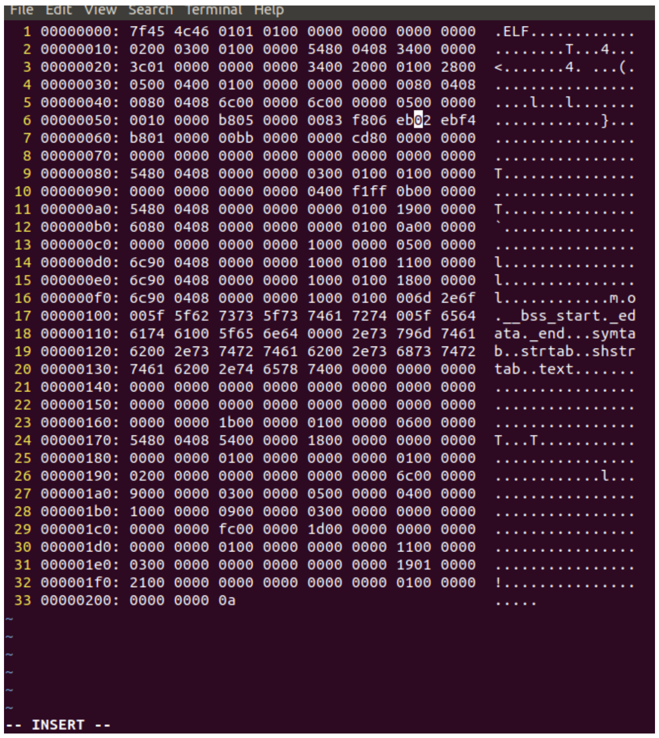
1. Open vim.
2. Run the following: ESC + “%!xxd”
    a. Should see the output in hexadecimal
    b. Edit the line
3. Run the following: ESC + “%!xxd –r”
4. Save the file
5. Run the file

Or in picture format:



```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 0300 0100 0000 5480 0408 3400 0000  ........T...4...
00000020: 3c01 0000 0000 0000 3400 2000 0100 2800  <.......4. ...(.
00000030: 0500 0400 0100 0000 0000 0000 0080 0408  ................
00000040: 0080 0408 6c00 0000 6c00 0000 0500 0000  ....l...l.......
00000050: 0010 0000 b805 0000 0083 f806 7d02 ebf4  ............}...
00000060: b801 0000 00bb 0000 0000 cd80 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 5480 0408 0000 0000 0300 0100 0100 0000  T...............
00000090: 0000 0000 0000 0000 0400 f1ff 0b00 0000  ................
000000a0: 5480 0408 0000 0000 0000 0100 1900 0000  T...............
000000b0: 6080 0408 0000 0000 0000 0100 0a00 0000  `...............
000000c0: 0000 0000 0000 0000 1000 0000 0500 0000  ................
000000d0: 6c90 0408 0000 0000 1000 0100 1100 0000  l...............
000000e0: 6c90 0408 0000 0000 1000 0100 1800 0000  l...............
000000f0: 6c90 0408 0000 0000 1000 0100 006d 2e6f  l............m.o
00000100: 005f 5f62 7373 5f73 7461 7274 005f 6564  .__bss_start._ed
00000110: 6174 6100 5f65 6e64 0000 2e73 796d 7461  ata._end...symta
00000120: 6200 2e73 7472 7461 6200 2e73 6873 7472  b..strtab..shstr
00000130: 7461 6200 2e74 6578 7400 0000 0000 0000  tab..text.......
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 1b00 0000 0100 0000 0600 0000  ................
00000170: 5480 0408 5400 0000 1800 0000 0000 0000  T...T...........
00000180: 0000 0000 0100 0000 0000 0000 0100 0000  ................
00000190: 0200 0000 0000 0000 0000 0000 6c00 0000  ............l...
000001a0: 9000 0000 0300 0000 0500 0000 0400 0000  ................
000001b0: 1000 0000 0900 0000 0300 0000 0000 0000  ................
000001c0: 0000 0000 fc00 0000 1d00 0000 0000 0000  ................
000001d0: 0000 0000 0100 0000 0000 0000 1100 0000  ................
000001e0: 0300 0000 0000 0000 0000 0000 1901 0000  ................
000001f0: 2100 0000 0000 0000 0000 0000 0100 0000  !...............
00000200: 0000 0000 0a                             .....
~
~
~
~
~
~
-- INSERT --
```

Figure 4 Before Editing

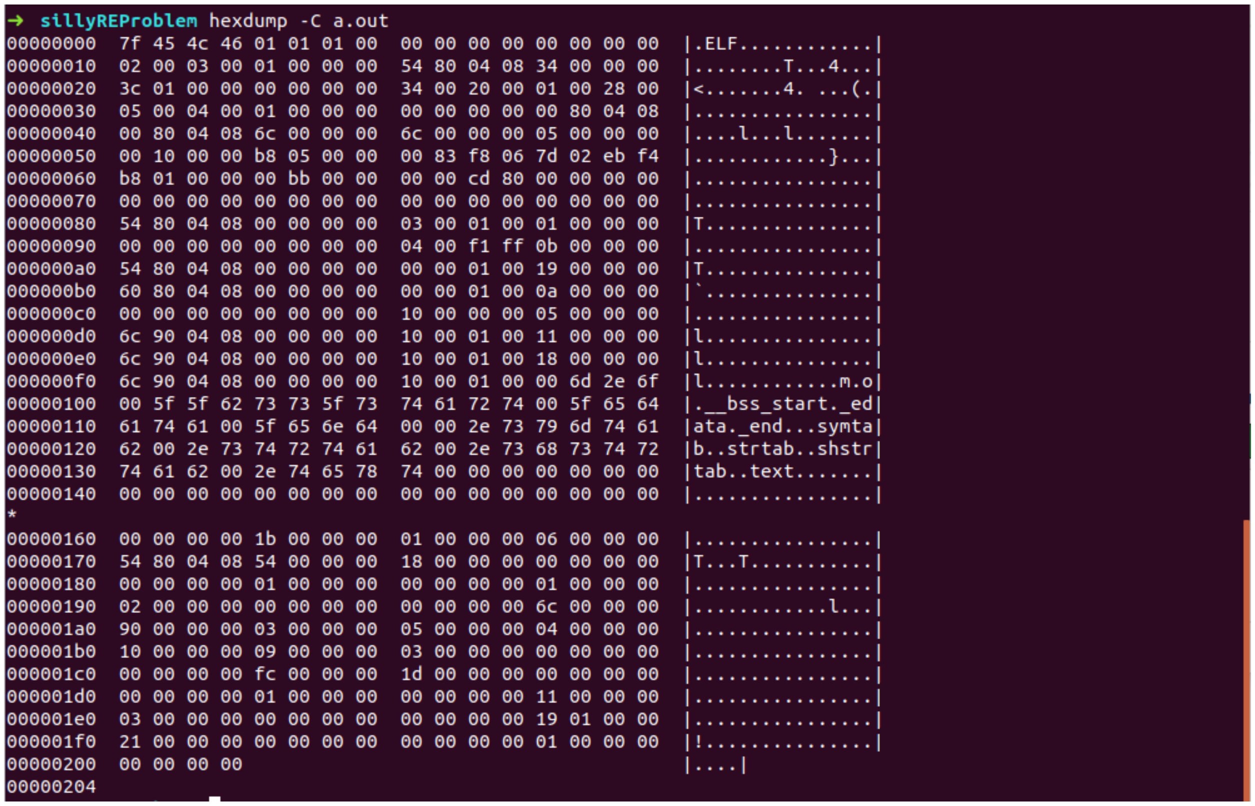


```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 0300 0100 0000 5480 0408 3400 0000  ........T...4...
00000020: 3c01 0000 0000 0000 3400 2000 0100 2800  <.......4. ...(.
00000030: 0500 0400 0100 0000 0000 0000 0080 0408  ................
00000040: 0080 0408 6c00 0000 6c00 0000 0500 0000  ....l...l.......
00000050: 0010 0000 b805 0000 0083 f806 eb02 ebf4  ............}...
00000060: b801 0000 00bb 0000 0000 cd80 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 5480 0408 0000 0000 0300 0100 0100 0000  T...............
00000090: 0000 0000 0000 0000 0400 f1ff 0b00 0000  ................
000000a0: 5480 0408 0000 0000 0000 0100 1900 0000  T...............
000000b0: 6080 0408 0000 0000 0000 0100 0a00 0000  `...............
000000c0: 0000 0000 0000 0000 1000 0000 0500 0000  ................
000000d0: 6c90 0408 0000 0000 1000 0100 1100 0000  l...............
000000e0: 6c90 0408 0000 0000 1000 0100 1800 0000  l...............
000000f0: 6c90 0408 0000 0000 1000 0100 006d 2e6f  l............m.o
00000100: 005f 5f62 7373 5f73 7461 7274 005f 6564  .__bss_start._ed
00000110: 6174 6100 5f65 6e64 0000 2e73 796d 7461  ata._end...symta
00000120: 6200 2e73 7472 7461 6200 2e73 6873 7472  b..strtab..shstr
00000130: 7461 6200 2e74 6578 7400 0000 0000 0000  tab..text.......
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 1b00 0000 0100 0000 0600 0000  ................
00000170: 5480 0408 5400 0000 1800 0000 0000 0000  T...T...........
00000180: 0000 0000 0100 0000 0000 0000 0100 0000  ................
00000190: 0200 0000 0000 0000 0000 0000 6c00 0000  ............l...
000001a0: 9000 0000 0300 0000 0500 0000 0400 0000  ................
000001b0: 1000 0000 0900 0000 0300 0000 0000 0000  ................
000001c0: 0000 0000 fc00 0000 1d00 0000 0000 0000  ................
000001d0: 0000 0000 0100 0000 0000 0000 1100 0000  ................
000001e0: 0300 0000 0000 0000 0000 0000 1901 0000  ................
000001f0: 2100 0000 0000 0000 0000 0000 0100 0000  !...............
00000200: 0000 0000 0a                             .....
~
~
~
~
~
~
-- INSERT --
```

Figure 5 After Editing

Then when you run it. It’ll run and immediately exit. Yay and that concludes this one.

Change the Comparison

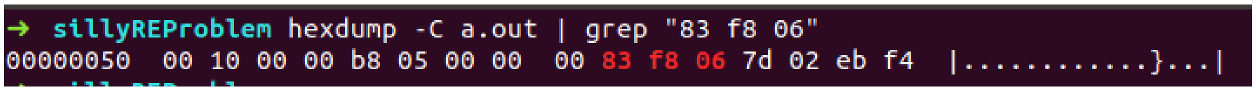


```
→ sillyREProblem hexdump -C a.out
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |........T...4...|
00000020  3c 01 00 00 00 00 00 00  34 00 20 00 01 00 28 00  |<.......4. ...(.|
00000030  05 00 04 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
00000040  00 80 04 08 6c 00 00 00  6c 00 00 00 05 00 00 00  |....l...l.......|
00000050  00 10 00 00 b8 05 00 00  00 83 f8 06 7d 02 eb f4  |............}...|
00000060  b8 01 00 00 00 bb 00 00  00 00 cd 80 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  54 80 04 08 00 00 00 00  03 00 01 00 01 00 00 00  |T...............|
00000090  00 00 00 00 00 00 00 00  04 00 f1 ff 0b 00 00 00  |................|
000000a0  54 80 04 08 00 00 00 00  00 00 01 00 19 00 00 00  |T...............|
000000b0  60 80 04 08 00 00 00 00  00 00 01 00 0a 00 00 00  |`...............|
000000c0  00 00 00 00 00 00 00 00  10 00 00 00 05 00 00 00  |................|
000000d0  6c 90 04 08 00 00 00 00  10 00 01 00 11 00 00 00  |l...............|
000000e0  6c 90 04 08 00 00 00 00  10 00 01 00 18 00 00 00  |l...............|
000000f0  6c 90 04 08 00 00 00 00  10 00 01 00 00 6d 2e 6f  |l............m.o|
00000100  00 5f 5f 62 73 73 5f 73  74 61 72 74 00 5f 65 64  |.__bss_start._ed|
00000110  61 74 61 00 5f 65 6e 64  00 00 2e 73 79 6d 74 61  |ata._end...symta|
00000120  62 00 2e 73 74 72 74 61  62 00 2e 73 68 73 74 72  |b..strtab..shstr|
00000130  74 61 62 00 2e 74 65 78  74 00 00 00 00 00 00 00  |tab..text.......|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000160  00 00 00 00 1b 00 00 00  01 00 00 00 06 00 00 00  |................|
00000170  54 80 04 08 54 00 00 00  18 00 00 00 00 00 00 00  |T...T...........|
00000180  00 00 00 00 01 00 00 00  00 00 00 00 01 00 00 00  |................|
00000190  02 00 00 00 00 00 00 00  00 00 00 00 6c 00 00 00  |............l...|
000001a0  90 00 00 00 03 00 00 00  05 00 00 00 04 00 00 00  |................|
000001b0  10 00 00 00 09 00 00 00  03 00 00 00 00 00 00 00  |................|
000001c0  00 00 00 00 fc 00 00 00  1d 00 00 00 00 00 00 00  |................|
000001d0  00 00 00 00 01 00 00 00  00 00 00 00 11 00 00 00  |................|
000001e0  03 00 00 00 00 00 00 00  00 00 00 00 19 01 00 00  |................|
000001f0  21 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |!...............|
00000200  00 00 00 00                                       |....|
00000204
```

So, there are a number of ways you can go about this. I will do the simple way. I will parse the output using "grep" for the "cmp $6, %eax" except I can't do that directly and I must find the hexadecimal equivalent which is: "83 f8 06". So it will look like this:

hexdump –C a.out | grep "83 f8 06"

And your output will be:



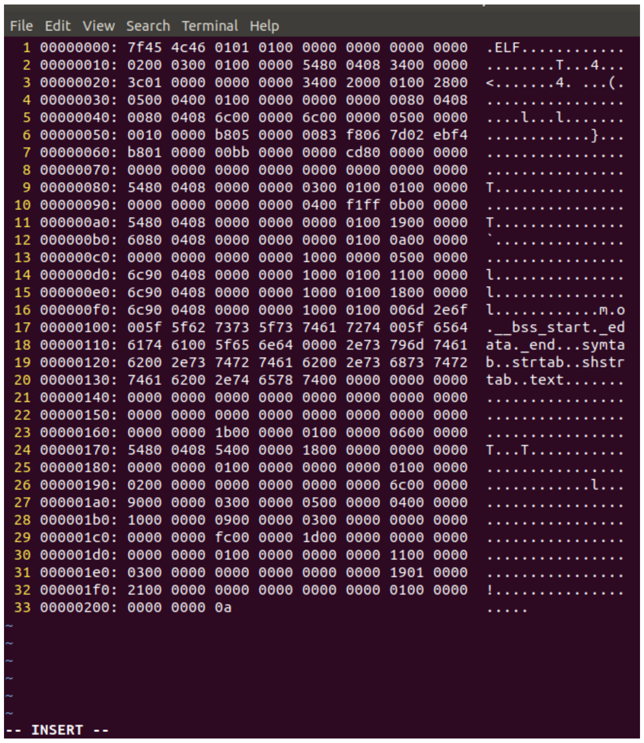
```
→ sillyREProblem hexdump -C a.out | grep "83 f8 06"
00000050  00 10 00 00 b8 05 00 00  00 83 f8 06 7d 02 eb f4  |............}...|
```

Now from here we know the exact line we need to change and where it would be located if we had a hexdump that we could easily modify. The simplest way is to just change that "06" to a "0" and that is the method I chose for this demonstration.

Now how would we write in hexadecimal? With a hex editor or our wonderful vim editor in the following steps [alternatively: https://vi.stackexchange.com/questions/2232/how-can-i-use-vim-as-a-hex-editor] :

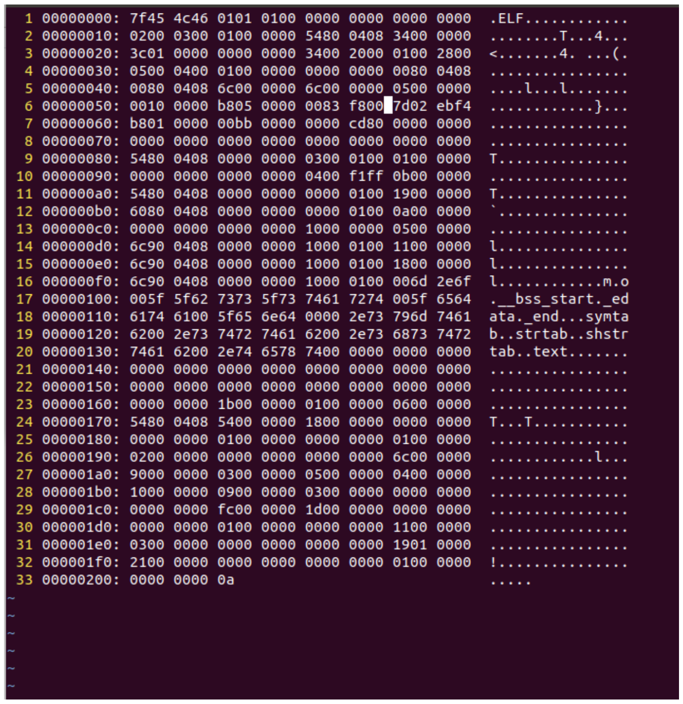
1. Open vim.
2. Run the following: ESC + “%!xxd”
    a. Should see the output in hexadecimal
    b. Edit the line
3. Run the following: ESC + “%!xxd –r”
4. Save the file
5. Run the file

In picture format:



```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 0300 0100 0000 5480 0408 3400 0000  ........T...4...
00000020: 3c01 0000 0000 0000 3400 2000 0100 2800  <.......4. ...(.
00000030: 0500 0400 0100 0000 0000 0000 0080 0408  ................
00000040: 0080 0408 6c00 0000 6c00 0000 0500 0000  ....l...l.......
00000050: 0010 0000 b805 0000 0083 f806 7d02 ebf4  ............}...
00000060: b801 0000 00bb 0000 0000 cd80 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 5480 0408 0000 0000 0300 0100 0100 0000  T...............
00000090: 0000 0000 0000 0000 0400 f1ff 0b00 0000  ................
000000a0: 5480 0408 0000 0000 0000 0100 1900 0000  T...............
000000b0: 6080 0408 0000 0000 0000 0100 0a00 0000  `...............
000000c0: 0000 0000 0000 0000 1000 0000 0500 0000  ................
000000d0: 6c90 0408 0000 0000 1000 0100 1100 0000  l...............
000000e0: 6c90 0408 0000 0000 1000 0100 1800 0000  l...............
000000f0: 6c90 0408 0000 0000 1000 0100 006d 2e6f  l............m.o
00000100: 005f 5f62 7373 5f73 7461 7274 005f 6564  .__bss_start._ed
00000110: 6174 6100 5f65 6e64 0000 2e73 796d 7461  ata._end...symta
00000120: 6200 2e73 7472 7461 6200 2e73 6873 7472  b..strtab..shstr
00000130: 7461 6200 2e74 6578 7400 0000 0000 0000  tab..text.......
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 1b00 0000 0100 0000 0600 0000  ................
00000170: 5480 0408 5400 0000 1800 0000 0000 0000  T...T...........
00000180: 0000 0000 0100 0000 0000 0000 0100 0000  ................
00000190: 0200 0000 0000 0000 0000 0000 6c00 0000  ............l...
000001a0: 9000 0000 0300 0000 0500 0000 0400 0000  ................
000001b0: 1000 0000 0900 0000 0300 0000 0000 0000  ................
000001c0: 0000 0000 fc00 0000 1d00 0000 0000 0000  ................
000001d0: 0000 0000 0100 0000 0000 0000 1100 0000  ................
000001e0: 0300 0000 0000 0000 0000 0000 1901 0000  ................
000001f0: 2100 0000 0000 0000 0000 0000 0100 0000  !...............
00000200: 0000 0000 0a                             .....
~
~
~
~
~
~
-- INSERT --
```

Figure 6 Before Editing



```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 0300 0100 0000 5480 0408 3400 0000  ........T...4...
00000020: 3c01 0000 0000 0000 3400 2000 0100 2800  <.......4. ...(.
00000030: 0500 0400 0100 0000 0000 0000 0080 0408  ................
00000040: 0080 0408 6c00 0000 6c00 0000 0500 0000  ....l...l.......
00000050: 0010 0000 b805 0000 0083 f800 7d02 ebf4  ............}...
00000060: b801 0000 00bb 0000 0000 cd80 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 5480 0408 0000 0000 0300 0100 0100 0000  T...............
00000090: 0000 0000 0000 0000 0400 f1ff 0b00 0000  ................
000000a0: 5480 0408 0000 0000 0000 0100 1900 0000  T...............
000000b0: 6080 0408 0000 0000 0000 0100 0a00 0000  `...............
000000c0: 0000 0000 0000 0000 1000 0000 0500 0000  ................
000000d0: 6c90 0408 0000 0000 1000 0100 1100 0000  l...............
000000e0: 6c90 0408 0000 0000 1000 0100 1800 0000  l...............
000000f0: 6c90 0408 0000 0000 1000 0100 006d 2e6f  l............m.o
00000100: 005f 5f62 7373 5f73 7461 7274 005f 6564  .__bss_start._ed
00000110: 6174 6100 5f65 6e64 0000 2e73 796d 7461  ata._end...symta
00000120: 6200 2e73 7472 7461 6200 2e73 6873 7472  b..strtab..shstr
00000130: 7461 6200 2e74 6578 7400 0000 0000 0000  tab..text.......
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 1b00 0000 0100 0000 0600 0000  ................
00000170: 5480 0408 5400 0000 1800 0000 0000 0000  T...T...........
00000180: 0000 0000 0100 0000 0000 0000 0100 0000  ................
00000190: 0200 0000 0000 0000 0000 0000 6c00 0000  ............l...
000001a0: 9000 0000 0300 0000 0500 0000 0400 0000  ................
000001b0: 1000 0000 0900 0000 0300 0000 0000 0000  ................
000001c0: 0000 0000 fc00 0000 1d00 0000 0000 0000  ................
000001d0: 0000 0000 0100 0000 0000 0000 1100 0000  ................
000001e0: 0300 0000 0000 0000 0000 0000 1901 0000  ................
000001f0: 2100 0000 0000 0000 0000 0000 0100 0000  !...............
00000200: 0000 0000 0a                             .....
~
~
~
~
~
~
```

Figure 7 After Editing

When you run it, it will immediately go to the exit because “6” is always greater than 0.

Detour – Other Option:

1. Change them to “nop” – no instruction or “00”. This you have to be careful about because the number of “nop” instructions must equal the number of bytes taken. Example:

   “cmp $6, %eax” is “83 f8 06”

   This is 3 bytes. Therefore, you would need to change all 3 bytes to “00” the reason being if you didn’t those remaining bytes could be “valid instructions” and do unwanted things.

OPTIONAL DETOUR:

You might be wondering then how the heck does “f4” equate to going backwards then? This is unique only to SHORT JUMP instructions. Remember how negative numbers are represented in binary? The last bit determines whether it is negative or not if it is set or not. In this case, a short = 1 byte or “0000 0000”. Therefore, “1000 0000” is equivalent to “-1” and “1111 1111” is equivalent to -127. Or in hexadecimal the range is 0x80 – 0xFF. **However**, we aren’t done yet because the CPU thinks of this as a “two’s complement” to solve a situation (this would require another detour which I’ll attach as an Appendix). Essentially, what we now need to do is:

1. Change all the bits that 1’s to 0’s and all the 0’s to 1s in the binary representation
2. Add 1 to the number.

In this case:

1. F4 is 1111 0100 in binary
2. Then it becomes: “0000 1011”
3. Add 1: “0000 1100”
4. Becomes 0xC when converted

Then:

$$0x804805e - 0xC = 0x8048054 < start >$$

Chapter 2 – Meet GDB – Godawful Destroyer of Binaries

"Each new user of a new system uncovers a new class of bugs" – Brian W. Kernighan

Introduction (A Rant on Command Line Debuggers):

Okay that was a joke. It actually stands for: **GN**U **De**bugger and it is a command line debugger – lets you "step through" your code and let you see the changes as they happen line-by-line. So, I will let you know right now. Debugging with command line *for Assembly* is the worst hell imaginable. Three hundred lines of commands to do something even remotely useful and memorize them. That is painful and not something you would normally want to do. Now there are benefits to using a command line tool:

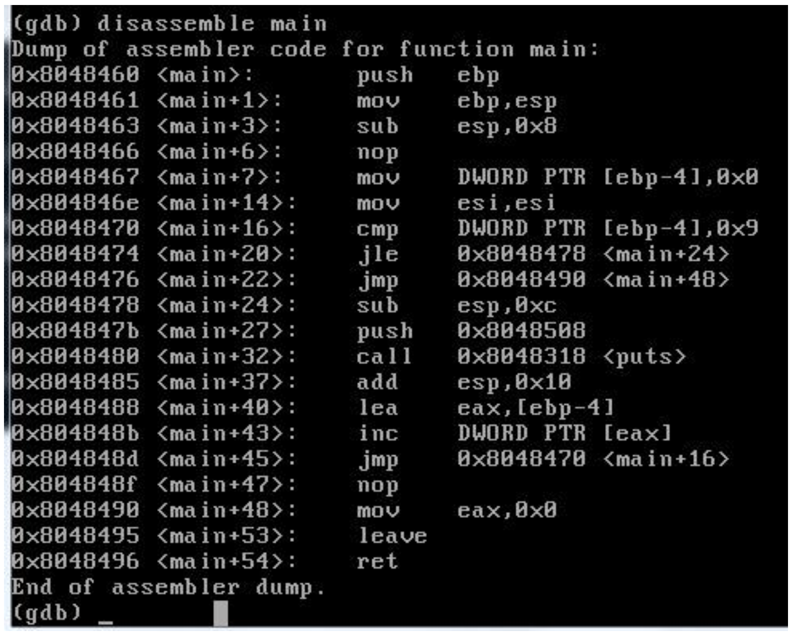
1. Doesn't take 300 years to start up
2. It's a command line tool if your one of those weirdoes.

Now the only real benefit is that it is quick to load up and also for those people who think that command line tools are gifts from God himself.

This is the author's personal opinion and apparently, something shared by other people who program and debug Assembly, command line debuggers are *awful for Assembly debugging*. Now is it a different story for other languages? *Absolutely*.

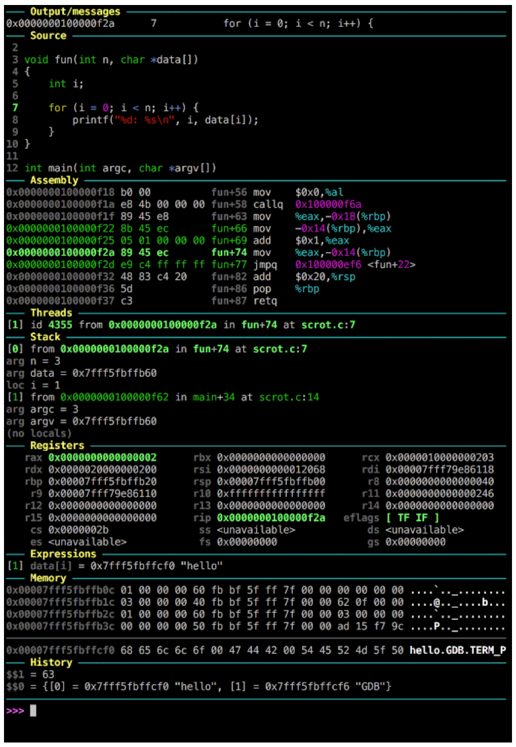
So, why would this author of yours who despises GDB make you learn GDB. The same reason that he forced you to learn a hex editor… he is a sadist. Nah just kidding, it is because sometimes a command line is faster than pretty GUI.

Incidentally this is what it looks like normally:



```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048460 <main>:        push   ebp
0x8048461 <main+1>:      mov    ebp,esp
0x8048463 <main+3>:      sub    esp,0x8
0x8048466 <main+6>:      nop
0x8048467 <main+7>:      mov    DWORD PTR [ebp-4],0x0
0x804846e <main+14>:     mov    esi,esi
0x8048470 <main+16>:     cmp    DWORD PTR [ebp-4],0x9
0x8048474 <main+20>:     jle    0x8048478 <main+24>
0x8048476 <main+22>:     jmp    0x8048490 <main+48>
0x8048478 <main+24>:     sub    esp,0xc
0x804847b <main+27>:     push   0x8048508
0x8048480 <main+32>:     call   0x8048318 <puts>
0x8048485 <main+37>:     add    esp,0x10
0x8048488 <main+40>:     lea    eax,[ebp-4]
0x804848b <main+43>:     inc    DWORD PTR [eax]
0x804848d <main+45>:     jmp    0x8048470 <main+16>
0x804848f <main+47>:     nop
0x8048490 <main+48>:     mov    eax,0x0
0x8048495 <main+53>:     leave
0x8048496 <main+54>:     ret
End of assembler dump.
(gdb) _
```

Now that command: “disassemble main”? Yeah, that gives you the entire list of commands. Command line debuggers usually do not do that without running an equivalent command to the above. They only give out one line at a time. So in an effort to make this section as painless as possible you will end up with something like the following instead after the next section:



```
── Output/messages ──────────────────────────────────────────────
0x0000000100000f2a      7           for (i = 0; i < n; i++) {
── Source ───────────────────────────────────────────────────────
 2
 3 void fun(int n, char *data[])
 4 {
 5     int i;
 6
 7     for (i = 0; i < n; i++) {
 8         printf("%d: %s\n", i, data[i]);
 9     }
10 }
11
12 int main(int argc, char *argv[])
── Assembly ─────────────────────────────────────────────────────
0x0000000100000f18 b0 00          fun+56 mov    $0x0,%al
0x0000000100000f1a e8 4b 00 00 00 fun+58 callq  0x100000f6a
0x0000000100000f1f 89 45 e8       fun+63 mov    %eax,-0x18(%rbp)
0x0000000100000f22 8b 45 ec       fun+66 mov    -0x14(%rbp),%eax
0x0000000100000f25 05 01 00 00 00 fun+69 add    $0x1,%eax
0x0000000100000f2a 89 45 ec       fun+74 mov    %eax,-0x14(%rbp)
0x0000000100000f2d e9 c4 ff ff ff fun+77 jmpq   0x100000ef6 <fun+22>
0x0000000100000f32 48 83 c4 20    fun+82 add    $0x20,%rsp
0x0000000100000f36 5d             fun+86 pop    %rbp
0x0000000100000f37 c3             fun+87 retq
── Threads ──────────────────────────────────────────────────────
[1] id 4355 from 0x0000000100000f2a in fun+74 at scrot.c:7
── Stack ────────────────────────────────────────────────────────
[0] from 0x0000000100000f2a in fun+74 at scrot.c:7
arg n = 3
arg data = 0x7fff5fbffb60
loc i = 1
[1] from 0x0000000100000f62 in main+34 at scrot.c:14
arg argc = 3
arg argv = 0x7fff5fbffb60
(no locals)
── Registers ────────────────────────────────────────────────────
   rax 0x0000000000000002     rbx 0x0000000000000000     rcx 0x0000010000000203
   rdx 0x0000020000000200     rsi 0x0000000000012068     rdi 0x00007fff79e86118
   rbp 0x00007fff5fbffb20     rsp 0x00007fff5fbffb00      r8 0x0000000000000040
    r9 0x00007fff79e86110     r10 0xffffffffffffffff     r11 0x0000000000000246
   r12 0x0000000000000000     r13 0x0000000000000000     r14 0x0000000000000000
   r15 0x0000000000000000     rip 0x0000000100000f2a  eflags [ TF IF ]
    cs 0x0000002b              ss <unavailable>           ds <unavailable>
    es <unavailable>           fs 0x00000000              gs 0x00000000
── Expressions ──────────────────────────────────────────────────
[1] data[i] = 0x7fff5fbffcf0 "hello"
── Memory ───────────────────────────────────────────────────────
0x00007fff5fbffb0c 01 00 00 00 60 fb bf 5f ff 7f 00 00 00 00 00 00 ....`.._........
0x00007fff5fbffb1c 03 00 00 00 40 fb bf 5f ff 7f 00 00 62 0f 00 00 ....@.._....b...
0x00007fff5fbffb2c 01 00 00 00 60 fb bf 5f ff 7f 00 00 03 00 00 00 ....`.._........
0x00007fff5fbffb3c 00 00 00 00 50 fb bf 5f ff 7f 00 00 ad 15 f7 9c ....P.._........
─────────────────────────────────────────────────────────────────
0x00007fff5fbffcf0 68 65 6c 6c 6f 00 47 44 42 00 54 45 52 4d 5f 50 hello.GDB.TERM_P
── History ──────────────────────────────────────────────────────
$$1 = 63
$$0 = {[0] = 0x7fff5fbffcf0 "hello", [1] = 0x7fff5fbffcf6 "GDB"}
─────────────────────────────────────────────────────────────────
>>> 
```

It has everything you could possibly want every time you want to debug/reverse engineer an Assembly project:

1. Shows you all the data section (remember? Globals, constants, etc?)
2. Shows you what's on the stack.
3. Gives you all the registers, and their values and even lets you know which flags in the flag registers are set
4. The assembly code being processed

So… let's install it!

Prettify GDB:

1. Download the following:
   https://github.com/cyrus-and/gdb-dashboard
2. If you didn't git clone it:
   a. Unrar/unzip it
   b. In a terminal:
      i. Go to the directory you unzipped it to
      ii. Run the following command: "cp .gdbinit ~/"
3. If you cloned it:
   a. In a terminal:
      i. Go to the directory you cloned it to
      ii. Run the following command: "cp .gdbinit ~/"

Example Execution:

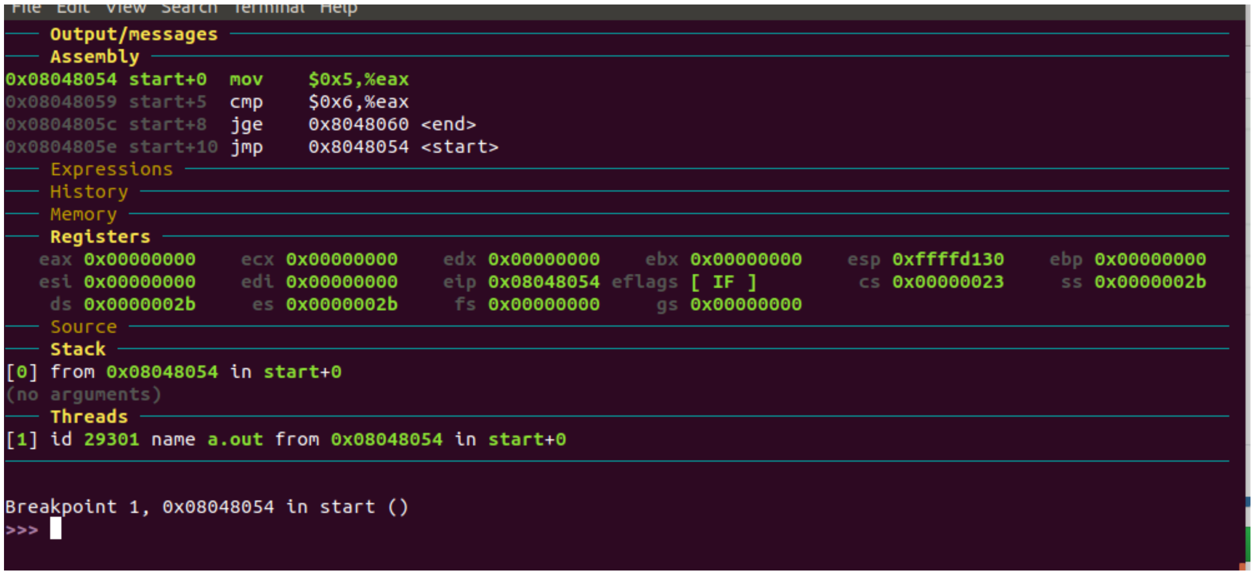
1. Run the following commands:

   "gdb a.out"

   "break start"

   "run"

   Should see something like the following now:



```
── Output/messages ───────────────────────────────────────────────────
── Assembly ──────────────────────────────────────────────────────────
0x08048054 start+0  mov    $0x5,%eax
0x08048059 start+5  cmp    $0x6,%eax
0x0804805c start+8  jge    0x8048060 <end>
0x0804805e start+10 jmp    0x8048054 <start>
── Expressions ───────────────────────────────────────────────────────
── History ───────────────────────────────────────────────────────────
── Memory ────────────────────────────────────────────────────────────
── Registers ─────────────────────────────────────────────────────────
   eax 0x00000000    ecx 0x00000000    edx 0x00000000    ebx 0x00000000    esp 0xffffd130    ebp 0x00000000
   esi 0x00000000    edi 0x00000000    eip 0x08048054 eflags [ IF ]         cs 0x00000023     ss 0x0000002b
    ds 0x0000002b     es 0x0000002b     fs 0x00000000     gs 0x00000000
── Source ────────────────────────────────────────────────────────────
── Stack ─────────────────────────────────────────────────────────────
[0] from 0x08048054 in start+0
(no arguments)
── Threads ───────────────────────────────────────────────────────────
[1] id 29301 name a.out from 0x08048054 in start+0
──────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x08048054 in start ()
>>> 
```

2. Now let's step through by running the following command:

   "stepi"

   You should see it highlight the next command in bright green and also a bunch of registers being modified.
3. Stepi again.
4. Ah… now we that the eflags change to the following:

   [CF PF AF SF IF]
5. But we know this is an infinite loop… so pick your adventure:
   a. If you choose to change the eflag (what do we need to do to take the "jge"? Sign Flag == Overflow flag:
      i. Pick your adventure:
         1. Make Carry Flag = 0:
            a. Run the following command:

               set $eflags &= (1 << 0)
         2. Make Sign Flag = 1

a. Run the following command:

```
set $eflags |= (1 << 7)
```

b. If you choose to directly change the operation:

1. Run the following command to change “jge” to a jmp:

```
set *(unsigned char *)0x0804805c = 0xeb
```

2. Run the following command to change “jmp” to a nop:

```
set *(unsigned char*) 0x0804805e = 0x90
set *(unsigned char*) 0x0804805f = 0x90
```

6. Then type:

   “continue”

7. Program will exit because you bypassed the loop.