Assembling & Debugging

Assembly File Structure

In this we will finally begin crafting our first Assembly file, as well as assembling it and debugging it, we will start with the simple `Hello World!` template to get an understanding of the structure. Then we will assemble it and then debug it

```
        global _start

        section .data
message: db      "Hello World!"

        section .text
_start:
        mov     rax, 1
        mov     rdi, 1
        mov     rsi, message
        mov     rdx, 12
        syscall

        mov     rax, 60
        mov     rdi, 0
        syscall
```

Once this code is assembled and linked it should print `Hello World!` to discover how this works we will go through file structure to understand what is happening

Looking at this code it can be divided into a few parts like so

Labels	Instructions	Operands
	`global`	`_start`
	`section`	`.data`
`message:`	`db`	`"Hello World!"`
	`section`	`.text`
`_start:`		
	`mov`	`rax, 1`

Labels	Instructions	Operands
	`mov`	`rdi, 1`
	`mov`	`rsi, message`
	`mov`	`rdx, 12`
	`syscall`	
	`mov`	`rax, 60`
	`mov`	`rdi, 0`
	`syscall`	

On top of this if we look at this code line by line there are 3 main sections for us to look at :

Section	Description
`global _start`	This is a directive that says that the code should start executing at the `_start` label that is defined below
`section .data`	This is the data section which should contain all the variables
`section .text`	This is the text section which contains all the code to be executed

Directives

Assembly code is line based which means that the code file is processed line-by-line, executing the instruction of each line. This means that the first instruction to be read is the `global _start` instruction which tell our machine to start to process the instructions after the `_start` label, the machine then goes to the `_start` label and executes the instructions which will print the result to the screen

Variables

We then have the `.data` section which holds our variables to make it so we do not have to write them multiple times after defining them. When the program runs, all the variables will be loaded into the memory in the data segment.

When we run this program it will load any variables we have defined into memory so that they are ready for use when called upon, we will eventually notice that by the time we start executing instructions at the `_start` label that all of our variables will have already been loaded into memory

We can also define variables using `db` for a list of bytes, `dw` for a list of words, `dd` for a list of digits and so on. On top of this, we can also label any of our variables so we can call it or reference it later on, here are some examples of defining variables

Instruction	Description
`db 0x0a`	Defines the byte `0x0a` which is a new line
`message db 0x41, 0x42, 0x43, 0x0a`	Defines the label `message => abc\n`
`message db "Hello World!", 0x0a`	Defines the label `message => Hello World!\n`

Building on this we can use the `equ` instruction with the `$` token to evaluate an expression, like the length of a defined varable's string but, labels defined with `equ` instruction are constants which means that they cannot be changed later. Just as an example of this I will demonstrate setting a constant using `equ`

```
section .data
    message db "Hello World!", 0x0a
    length equ $-message
```

the `$` token indicates the current distance from the beginning of the current section. As the `message` variable is at the beginning of the `data` section, the current location, i.e,. value of `$`, equals the length of the string. For the scope of this module, we will only use this token to calculate lengths of strings, using the same line of code shown above.

Code

The second and most important section is the `.text` section which holds all the assembly instruction and loads them to the text memory segment. Once all instructions are loaded into the text section the processor starts executing them one after the other. The typical convention is to have the `_start` label at the beginning of the `.text` section which as a result of the `global _start` directive starts the main code that will be executed as the program runs.

The text memory segment is a read only area meaning that we cannot store any variables within it unlike the data section which is read/write. However the data section is a non-executable section so any code written in there is not executed. This is done as a mitigation against different types of binary exploitation

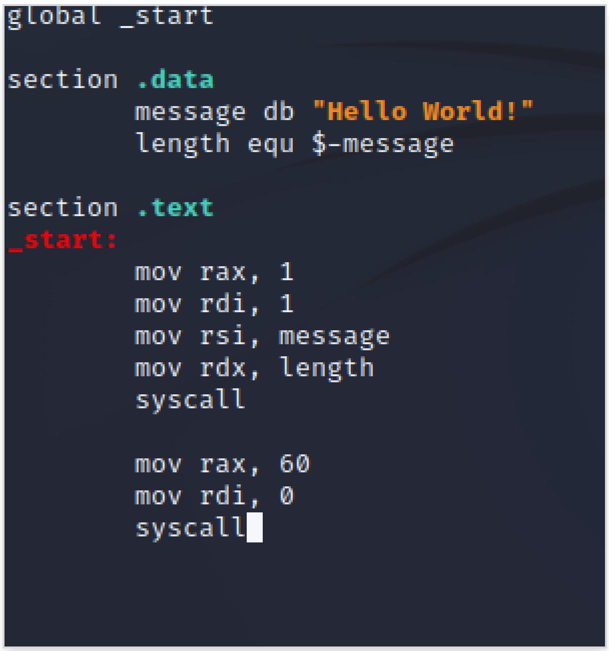
```
side note: we can create comments in assembly by using a semi colon on the line
```

Assembling and Disassembling

For the purposes of this we will be using the `nasm` tool which is the file structure that we have been using thus far, after assembling our code with `nasm` we can link it using `ld` to utilize OS features and libraries.

Assembling

First we will save our assembly code with the file extention `.s` or `.asm` , here I will show my code, I did this in kali linux as it is easier to do

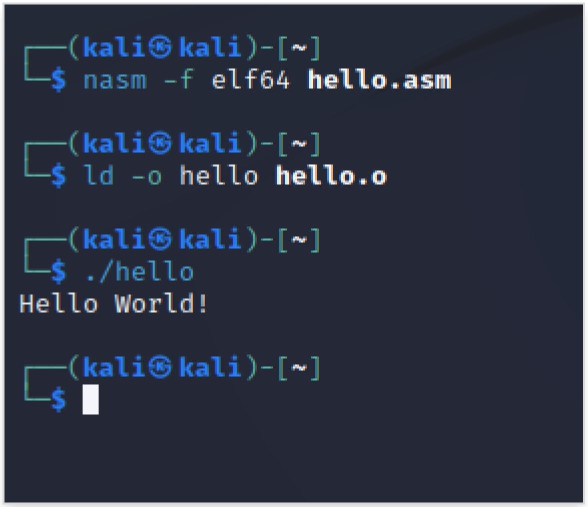


```
global _start

section .data
        message db "Hello World!"
        length equ $-message

section .text
_start:
        mov rax, 1
        mov rdi, 1
        mov rsi, message
        mov rdx, length
        syscall

        mov rax, 60
        mov rdi, 0
        syscall
```

After doing this I run the command `nasm -f elf64 [filename]` or `nasm -f elf [filename]` for 32-bit, which will then create an object file which is then assembled into machine code along with all the variables and sections, it is, however, not executable yet because we have to link it

Linking

The final step in the process is linking our file through using `ld`. This is because the object file by itself is not executable even though it is assembled, this is because many references and lablels used by nasm need to be resolved into actual addresses along with linking the file with various OS libraries that may be needed. This is why the Linux library is called ELF which stands for "Executable and Linkable Format". To link our file we would just issue the following command `ld -o hello hello.o` or `ld -o hello hello.o -m elf_i386` for 32-bit. After this we should have an executable file



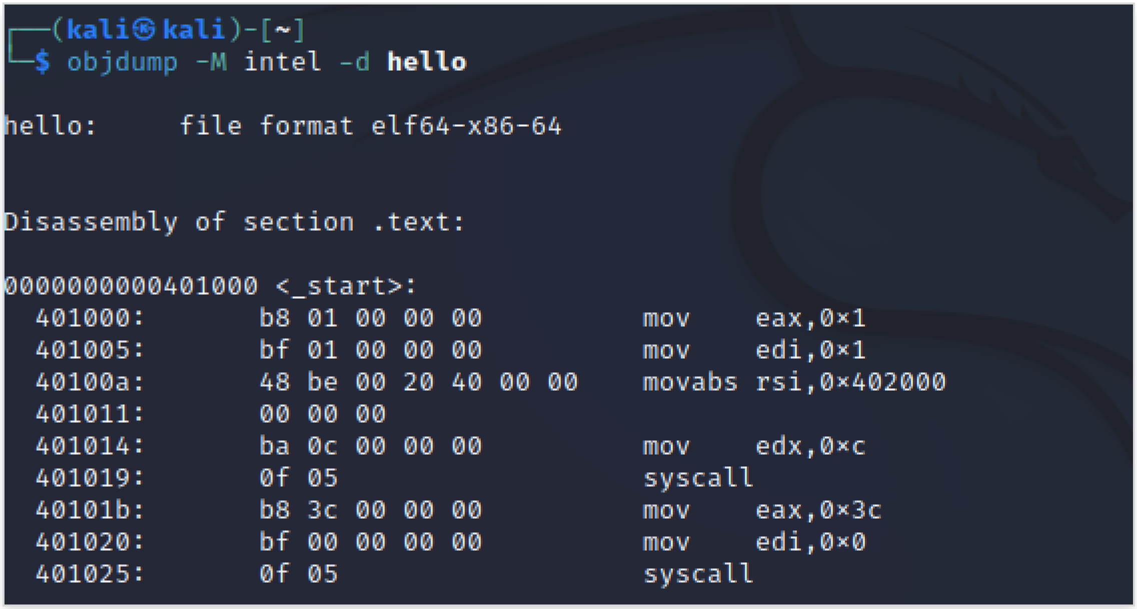
```
┌──(kali㉿kali)-[~]
└─$ nasm -f elf64 hello.asm

┌──(kali㉿kali)-[~]
└─$ ld -o hello hello.o

┌──(kali㉿kali)-[~]
└─$ ./hello
Hello World!

┌──(kali㉿kali)-[~]
└─$ 
```

We can also save ourselves some time by making a script to automate this process like so

```bash
#!/bin/bash

fileName="${1%%.*}"

nasm -f elf64 ${fileName}".asm"
ld ${fileName}".o" -o ${fileName}
[ "$2" == "-g" ] && gdb -q ${fileName} || ./${fileName}
```

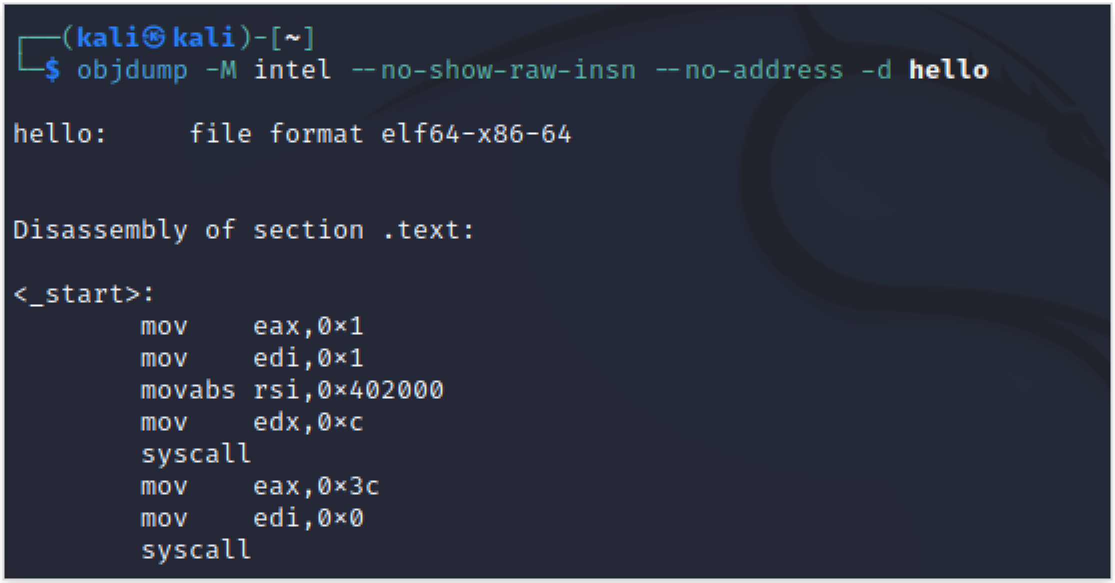
Disassembling

To disassemble a file we can use the `objdump` tool which dumps machine code from a file and interprets the assembly instruction of each hex code, we can disassemble a binary by using the `-d` flag. We will also be using the `-M intel` flag so it dumps in the Intel syntax. After running this we should see the following result



```
┌──(kali㉿kali)-[~]
└─$ objdump -M intel -d hello

hello:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       b8 01 00 00 00          mov    eax,0x1
  401005:       bf 01 00 00 00          mov    edi,0x1
  40100a:       48 be 00 20 40 00 00    movabs rsi,0x402000
  401011:       00 00 00
  401014:       ba 0c 00 00 00          mov    edx,0xc
  401019:       0f 05                   syscall
  40101b:       b8 3c 00 00 00          mov    eax,0x3c
  401020:       bf 00 00 00 00          mov    edi,0x0
  401025:       0f 05                   syscall
```

we can see that our code remained quite similar with the only major change being that items were converted to their hex values along with converting our sub-registers to the proper size to avoid using more memory than necessary. If we wanted to remove the machine code and/or addresses we can add the `--no-show-raw-insn` and `--no-address` flags



```
┌──(kali㉿kali)-[~]
└─$ objdump -M intel --no-show-raw-insn --no-address -d hello

hello:     file format elf64-x86-64


Disassembly of section .text:

<_start>:
        mov    eax,0x1
        mov    edi,0x1
        movabs rsi,0x402000
        mov    edx,0xc
        syscall
        mov    eax,0x3c
        mov    edi,0x0
        syscall
```

also another side note is that movabs is the same as mov so if we wanted to copy and paste we could just change it back to mov.

The `-d` flag will only disassemble the .text section of code, if we want to see the data section we can use `-j .data` and add `-s` to show strings, so the command would look like `objdump -sj .data hello`

GNU Debugger

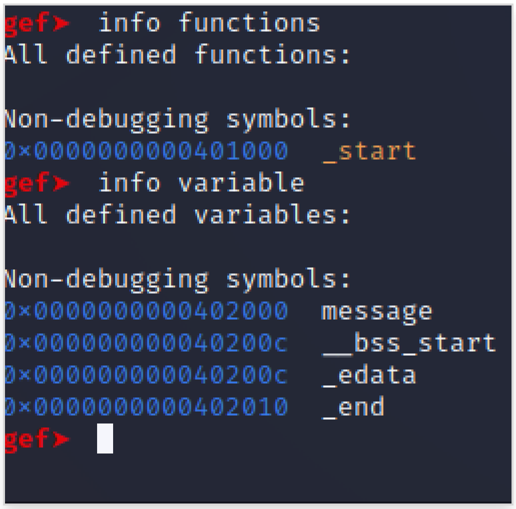
For this next step we will use GNU debugger to debug our programs, we are using this because it is released by GNU and therefore has better integration with linux systems. First we should make sure it is installed by issuing the following commands

```
sudo apt-get update
sudo apt-get install gdb
```

One of the best things about gdb is its support for 3rd party plugins and with that we will install GEF which is a free and open source gdb plugin with reverse engineering and binary exploitation in mind. To add it to gdb we need to issue the following commands

```
wget -O ~/.gdbinit-gef.py -q https://gef.blah.cat/py

echo source ~/.gdbinit-gef.py >> ~/.gdbinit
```

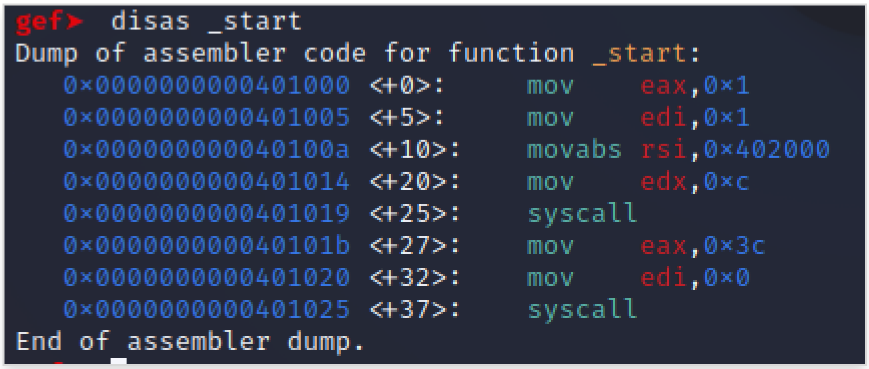
after doing this we can use the assemble script from earlier like so `./assemble.sh [filename] -g` or we can use the command `gdb -q [executable file name]` and we should see that gef is running. To this we can begin searching for info via the info command like so



```
gef➤  info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000  _start
gef➤  info variable
All defined variables:

Non-debugging symbols:
0x0000000000402000  message
0x000000000040200c  __bss_start
0x000000000040200c  _edata
0x0000000000402010  _end
gef➤
```

Disassemble

We can now begin to view the specific instructions within a specific function using the disas command along with the function name like so



```
gef➤  disas _start
Dump of assembler code for function _start:
   0x0000000000401000 <+0>:     mov    eax,0x1
   0x0000000000401005 <+5>:     mov    edi,0x1
   0x000000000040100a <+10>:    movabs rsi,0x402000
   0x0000000000401014 <+20>:    mov    edx,0xc
   0x0000000000401019 <+25>:    syscall
   0x000000000040101b <+27>:    mov    eax,0x3c
   0x0000000000401020 <+32>:    mov    edi,0x0
   0x0000000000401025 <+37>:    syscall
End of assembler dump.
```

This looks very similar to our objdump output but the primary difference is the memory addresses for each instruction and operands like arguments. This is important because having the memory addresses is critical to examining the variables and operands as well as setting breakpoints for certain instruction.

Another important note is that the memory addresses are in the form of `0x00000000004xxxxx` as opposed to their raw address in memory like `0xffffffffaa8a25ff`. This is because of `$rip-relative addressing` in Postition-Independent Executables(PIE), where memory addresses are used relative to their distance from the instruction pointer `$rip` within the program's own VRAM rather than using raw memory addresses, this feature might be disabled to reduce the risk of binary exploitation.

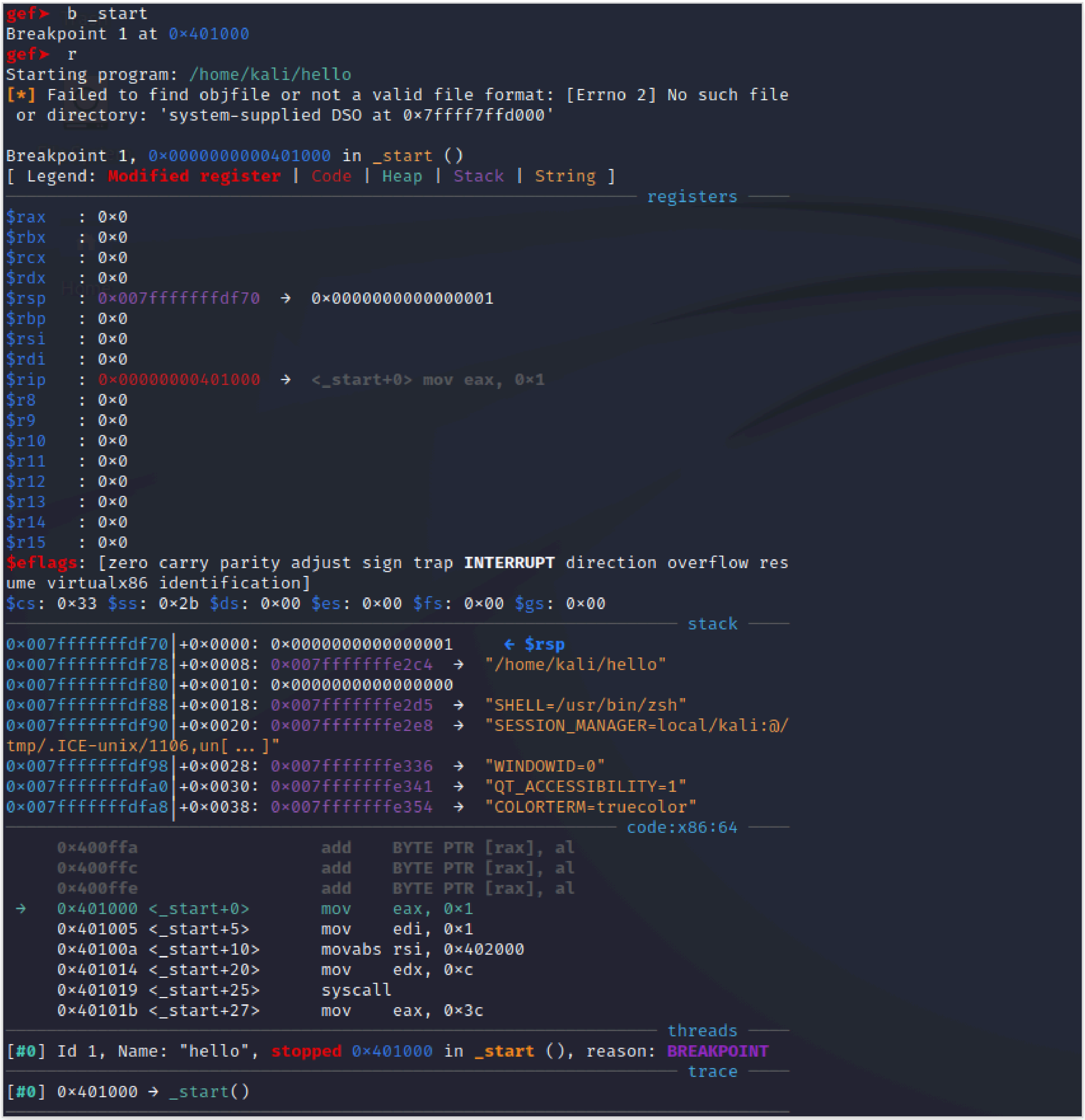
Debugging with GDB

Now that we know how the program runs we can now start running and debugging it, debugging mainly consists of 4 steps

step	description
Break	Setting breakpoints at points of interest
Examine	Running the program and looking at the state of the program at these points
Step	Moving through the program to see how it acts with each instruction and user input
Modify	Changing values in specific registers or addresses at specific breakpoints to determine changes in execution

Break

The first step of debugging is setting breakpoints to stop the program's execution at a certain point or when a condition has been met, this is to see the state of the program and the value of the registers at that point. Breakpoints also allow us to stop the execution at that point so we can step into each instruction and see how it changes the program and values. We can set breakpoints at a specific address or function which we can do in GEF with the break or `b` command along with the address or function name we want to break at. So if I wanted to break at `_start` I would use the command `b _start` in GEF. Then I would use the `r` command to run it, at the `_start` function it would then break and then return the values like so



```
gef➤  b _start
Breakpoint 1 at 0x401000
gef➤  r
Starting program: /home/kali/hello
[*] Failed to find objfile or not a valid file format: [Errno 2] No such file
 or directory: 'system-supplied DSO at 0x7ffff7ffd000'

Breakpoint 1, 0x0000000000401000 in _start ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x0
$rsp   : 0x007fffffffdf70  →  0x0000000000000001
$rbp   : 0x0
$rsi   : 0x0
$rdi   : 0x0
$rip   : 0x00000000401000  →  <_start+0> mov eax, 0x1
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────── stack ────
0x007fffffffdf70│+0x0000: 0x0000000000000001     ← $rsp
0x007fffffffdf78│+0x0008: 0x007fffffffe2c4  →  "/home/kali/hello"
0x007fffffffdf80│+0x0010: 0x0000000000000000
0x007fffffffdf88│+0x0018: 0x007fffffffe2d5  →  "SHELL=/usr/bin/zsh"
0x007fffffffdf90│+0x0020: 0x007fffffffe2e8  →  "SESSION_MANAGER=local/kali:@/
tmp/.ICE-unix/1106,un[...]"
0x007fffffffdf98│+0x0028: 0x007fffffffe336  →  "WINDOWID=0"
0x007fffffffdfa0│+0x0030: 0x007fffffffe341  →  "QT_ACCESSIBILITY=1"
0x007fffffffdfa8│+0x0038: 0x007fffffffe354  →  "COLORTERM=truecolor"
──────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400ffa                  add    BYTE PTR [rax], al
     0x400ffc                  add    BYTE PTR [rax], al
     0x400ffe                  add    BYTE PTR [rax], al
 →   0x401000 <_start+0>       mov    eax, 0x1
     0x401005 <_start+5>       mov    edi, 0x1
     0x40100a <_start+10>      movabs rsi, 0x402000
     0x401014 <_start+20>      mov    edx, 0xc
     0x401019 <_start+25>      syscall
     0x40101b <_start+27>      mov    eax, 0x3c
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "hello", stopped 0x401000 in _start (), reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401000 → _start()
─────────────────────────────────────────────────────────────────────────────────
```

If we wanted we could even set the breakpoint at a certain address like `_start+10` by using `b *_start+10` or `b *0x40100a` the * used in this tells gdb to break at the instruction stored in the address, and if we want to continue the program we can use the `c` command or we can rerun it with the `r` command. To disable, enable, or delete a breakpoint we can use `info breakpoint` and use `disable 1` or `enable 1` or `delete 1`

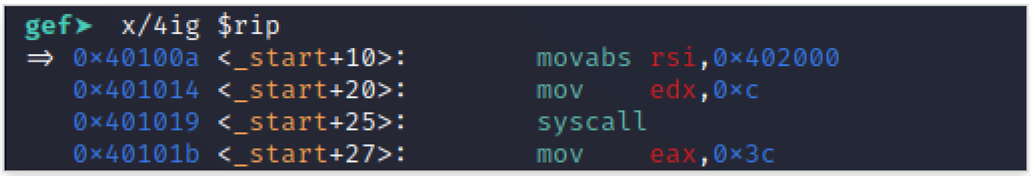
Examine

The next step is examining the values in addresses and registers, off the bat, we can see that GEF gave us tons of information that is helpful but if we want it to be more focused for the purposes of our program then we can use the `x` command which is used in the following format `x/FMT ADDRESS` where the address is the location we wish to examine and the FMT is the examine format. The examine format FMT can have 3 parts

Argument	Description	Example
Count	The number of times we want to repeat the examine	2,3,9
Format	The format we want the result represented in	x(hex),s(string),i(instruction)
Size	The size of the memory we want to examine	b(byte),h(halfword),w(word),g(giant, 8 bytes)

Instructions

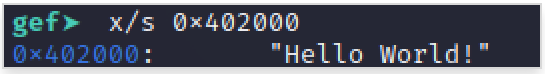
If we wanted to examine the next 4 instructions in line we will have to examine the `$rip` register and use 4 for the count, i for the format, and g for the size which would look like so



```
gef➤  x/4ig $rip
⇒ 0x40100a <_start+10>:        movabs rsi,0x402000
   0x401014 <_start+20>:        mov    edx,0xc
   0x401019 <_start+25>:        syscall
   0x40101b <_start+27>:        mov    eax,0x3c
```

Strings

We can also see stored variable data at a specific memory address, with us knowing that our message variable is stored in the .data section on address `0x402000` and also see the upcoming command `movabs rsi, 0x402000` , we might want to confirm what is being moved from this address which we can do like so

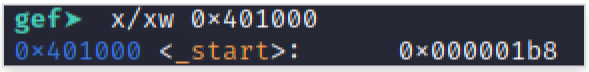


```
gef➤  x/s 0x402000
0x402000:       "Hello World!"
```

in this case we only needed to fetch the string at this exact address

Addresses

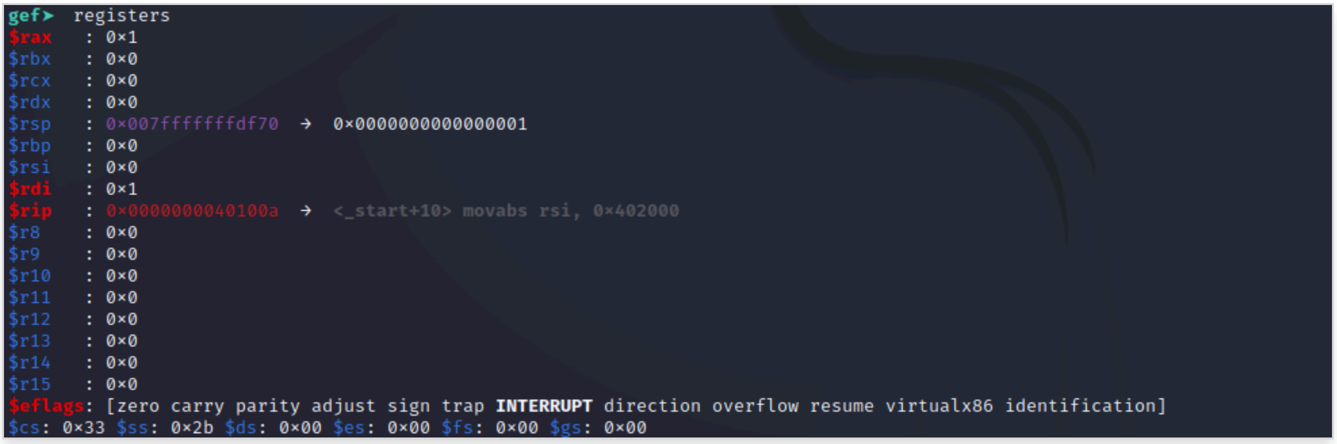
The most common form of examining that we will run into is hex which we often need to examine the addresses and registers containing hex data. These things include memory addresses, instructions, or binary data, in this example we will look at the first line of the `_start` function in hex format



```
gef➤  x/xw 0x401000
0x401000 <_start>:      0x000001b8
```

we see that instead of `mov eax, 0x1` that we now have `0x000001b8`, which is its hex representation machine code in little endian formatting which means that it is actually read as `b8 01 00 00`

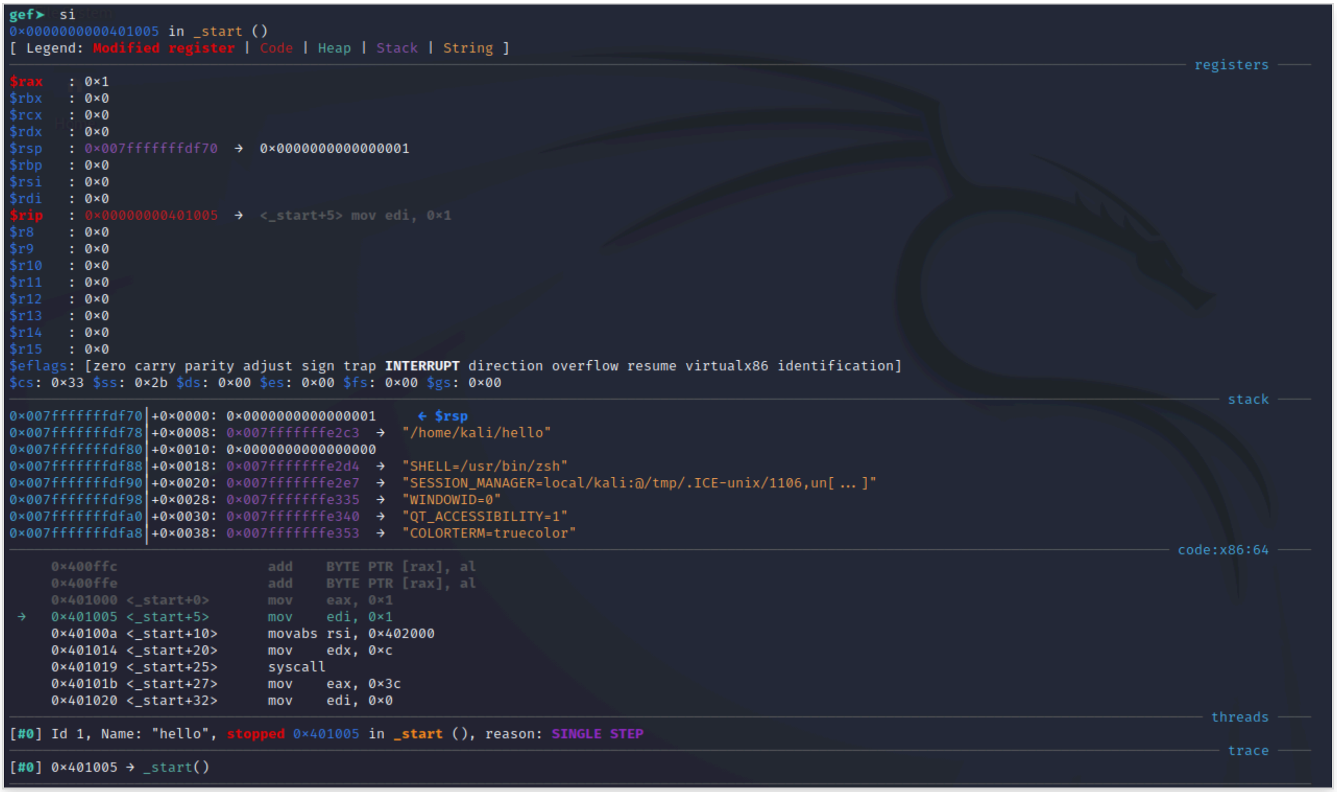
We can also repeat this on all the other addresses which we can easily obtain with the `registers` command



```
gef➤  registers
$rax   : 0x1
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x0
$rsp   : 0x007fffffffdf70  →  0x0000000000000001
$rbp   : 0x0
$rsi   : 0x0
$rdi   : 0x1
$rip   : 0x0000000040100a  →  <_start+10> movabs rsi, 0x402000
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
```

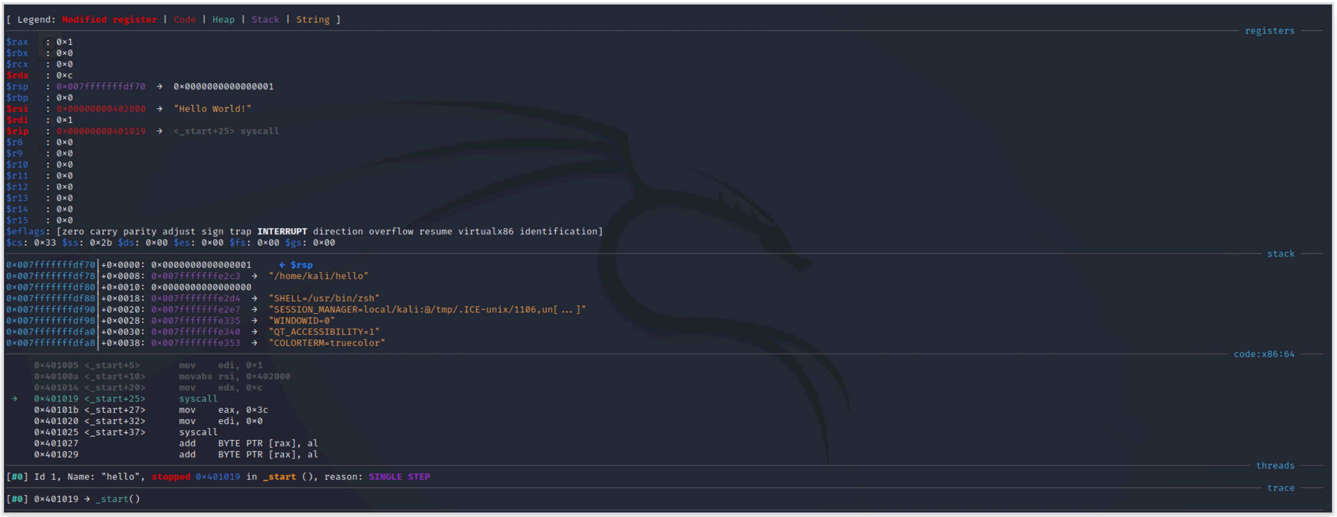
Step

The third step of debugging is stepping through the program on instruction/line at a time which can be done with the `si` command and if we add a number to the end of the command it will jump that many steps like so



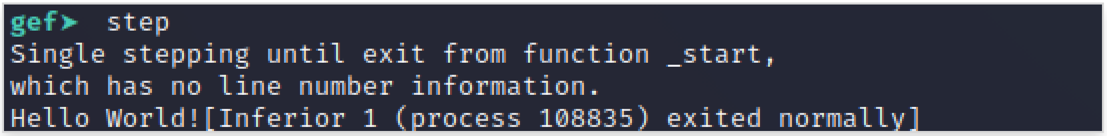
```
gef➤  si
0x0000000000401005 in _start ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x1
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x0
$rsp   : 0x007fffffffdf70  →  0x0000000000000001
$rbp   : 0x0
$rsi   : 0x0
$rdi   : 0x0
$rip   : 0x00000000401005  →  <_start+5> mov edi, 0x1
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────── stack ────
0x007fffffffdf70│+0x0000: 0x0000000000000001    ← $rsp
0x007fffffffdf78│+0x0008: 0x007fffffffe2c3  →  "/home/kali/hello"
0x007fffffffdf80│+0x0010: 0x0000000000000000
0x007fffffffdf88│+0x0018: 0x007fffffffe2d4  →  "SHELL=/usr/bin/zsh"
0x007fffffffdf90│+0x0020: 0x007fffffffe2e7  →  "SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1106,un[ ... ]"
0x007fffffffdf98│+0x0028: 0x007fffffffe335  →  "WINDOWID=0"
0x007fffffffdfa0│+0x0030: 0x007fffffffe340  →  "QT_ACCESSIBILITY=1"
0x007fffffffdfa8│+0x0038: 0x007fffffffe353  →  "COLORTERM=truecolor"
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400ffc                  add    BYTE PTR [rax], al
     0x400ffe                  add    BYTE PTR [rax], al
     0x401000 <_start+0>       mov    eax, 0x1
 →   0x401005 <_start+5>       mov    edi, 0x1
     0x40100a <_start+10>      movabs rsi, 0x402000
     0x401014 <_start+20>      mov    edx, 0xc
     0x401019 <_start+25>      syscall
     0x40101b <_start+27>      mov    eax, 0x3c
     0x401020 <_start+32>      mov    edi, 0x0
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "hello", stopped 0x401005 in _start (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401005 → _start()
```

and with adding a number to increase the steps:



```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x1
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0xc
$rsp   : 0x007fffffffdf70  →  0x0000000000000001
$rbp   : 0x0
$rsi   : 0x00000000402000  →  "Hello World!"
$rdi   : 0x1
$rip   : 0x00000000401019  →  <_start+25> syscall
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────── stack ────
0x007fffffffdf70│+0x0000: 0x0000000000000001    ← $rsp
0x007fffffffdf78│+0x0008: 0x007fffffffe2c3  →  "/home/kali/hello"
0x007fffffffdf80│+0x0010: 0x0000000000000000
0x007fffffffdf88│+0x0018: 0x007fffffffe2d4  →  "SHELL=/usr/bin/zsh"
0x007fffffffdf90│+0x0020: 0x007fffffffe2e7  →  "SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1106,un[ ... ]"
0x007fffffffdf98│+0x0028: 0x007fffffffe335  →  "WINDOWID=0"
0x007fffffffdfa0│+0x0030: 0x007fffffffe340  →  "QT_ACCESSIBILITY=1"
0x007fffffffdfa8│+0x0038: 0x007fffffffe353  →  "COLORTERM=truecolor"
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401005 <_start+5>       mov    edi, 0x1
     0x40100a <_start+10>      movabs rsi, 0x402000
     0x401014 <_start+20>      mov    edx, 0xc
 →   0x401019 <_start+25>      syscall
     0x40101b <_start+27>      mov    eax, 0x3c
     0x401020 <_start+32>      mov    edi, 0x0
     0x401025 <_start+37>      syscall
     0x401027                  add    BYTE PTR [rax], al
     0x401029                  add    BYTE PTR [rax], al
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "hello", stopped 0x401019 in _start (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401019 → _start()
```

now if we just use the step command or `s` it will continue until the following line of code is reached or it exits from the current function, if our code had another function called within the function then it would break at the beginning of the newly called function.



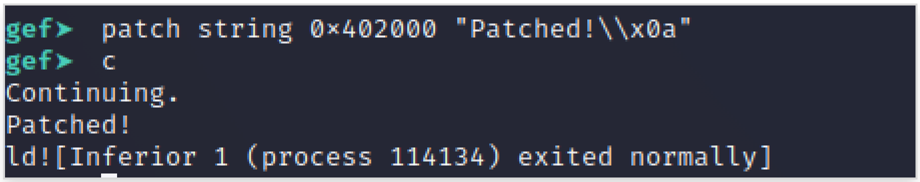
```
gef➤  step
Single stepping until exit from function _start,
which has no line number information.
Hello World![Inferior 1 (process 108835) exited normally]
```

Modify

The final step is modifying values in the registers and addresses at certain points which helps us locating how changes affect the program

Addresses

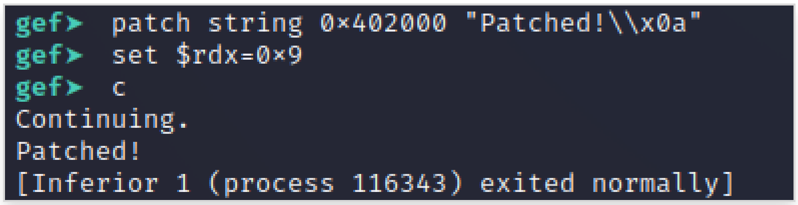
To modify values in GDB we can use the `set` command but, we will use the `patch` command in GEF to make this much easier, we have to provide the type/size of the new value, the location it will be stored, and the value. We should try changing the value of the string at `0x402000` to `"Patched!\n"` and then break at the first syscall at `0x401019`



```
gef➤  patch string 0x402000 "Patched!\\x0a"
gef➤  c
Continuing.
Patched!
ld![Inferior 1 (process 114134) exited normally]
```

Registers

If we want to ensure that an entire register is changed we can change the value of the register, which is useful because the last part of our "Hello World!"string was printed. With just using a couple commands like so we can ensure it works



```
gef➤  patch string 0x402000 "Patched!\\x0a"
gef➤  set $rdx=0x9
gef➤  c
Continuing.
Patched!
[Inferior 1 (process 116343) exited normally]
```

what we did here is set the length of the string equal to 9 instead of 12 which is why we had that weird "ld!" on the last attempt