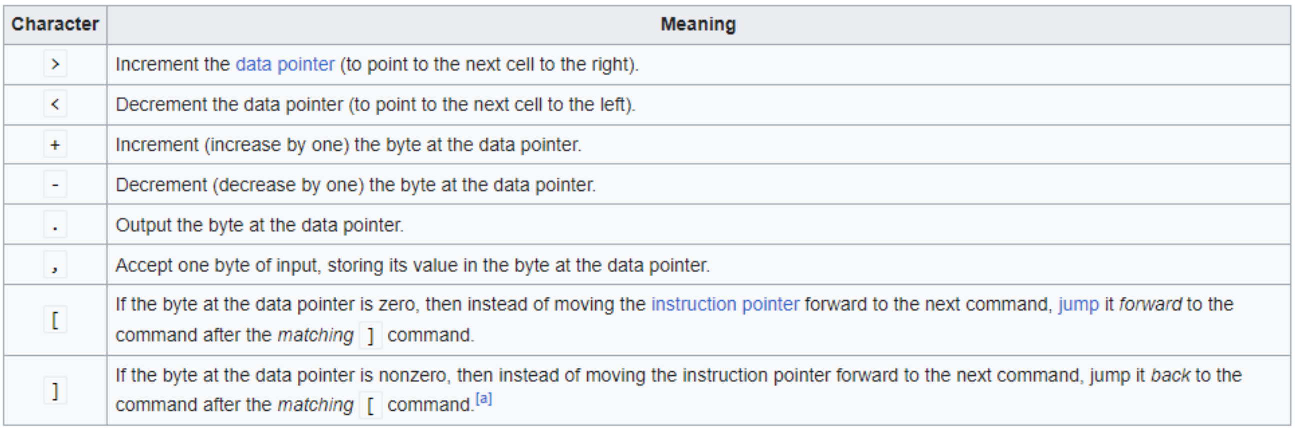
Brainfuck

Introduction

We are given a simple brain-fuck language emulation program written in C. The [ ] commands are not implemented yet. We should find a bug and exploit it to get a shell.

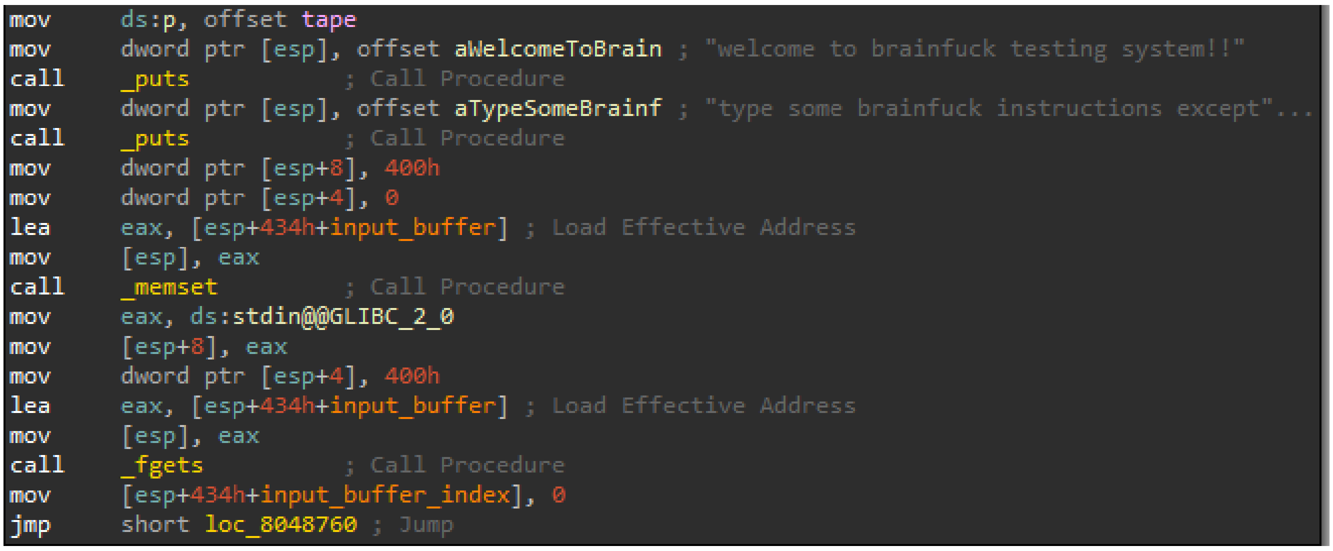
Static Analysis

I started by learning about brainfuck. It's a simple programming language, that maintains a data pointer and allows the following operation on it:



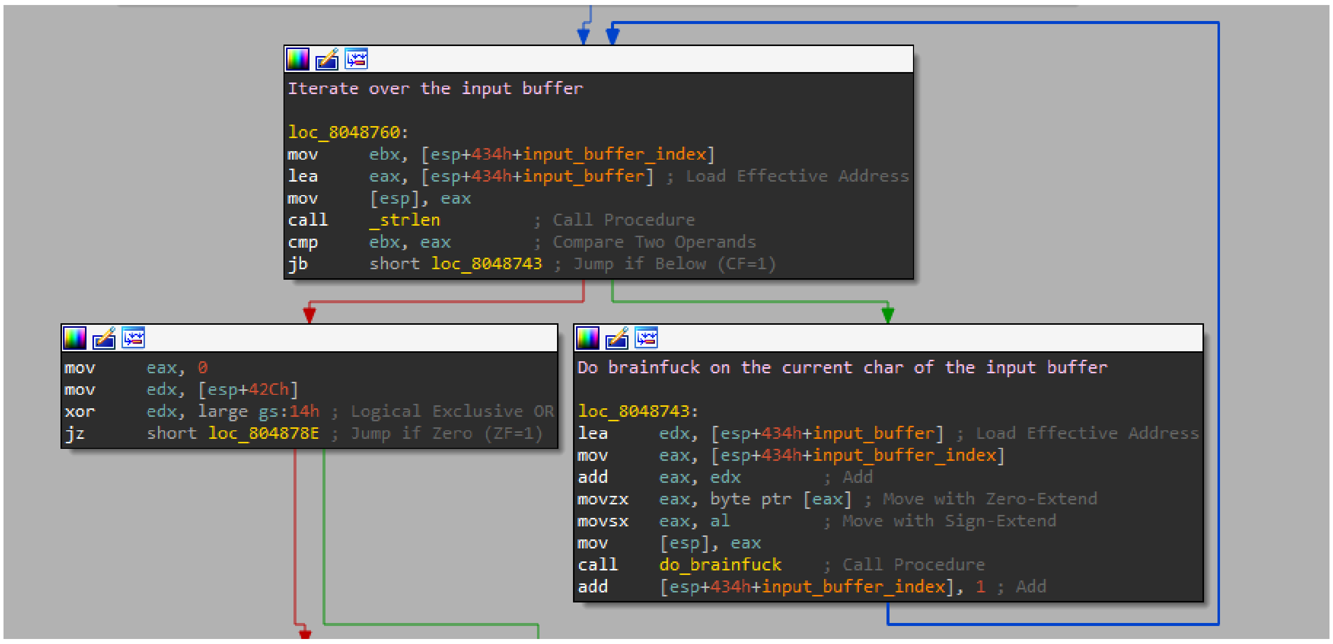
Character	Meaning
`>`	Increment the data pointer (to point to the next cell to the right).
`<`	Decrement the data pointer (to point to the next cell to the left).
`+`	Increment (increase by one) the byte at the data pointer.
`-`	Decrement (decrease by one) the byte at the data pointer.
`.`	Output the byte at the data pointer.
`,`	Accept one byte of input, storing its value in the byte at the data pointer.
`[`	If the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it *forward* to the command after the *matching* `]` command.
`]`	If the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it *back* to the command after the *matching* `[` command.[a]

Now let's open the binary in IDA and do some static analysis. The program starts by printing a welcome message to the user and then reading the user input, which is some brainfuck code, into a buffer:



```
mov     ds:p, offset tape
mov     dword ptr [esp], offset aWelcomeToBrain ; "welcome to brainfuck testing system!!"
call    _puts           ; Call Procedure
mov     dword ptr [esp], offset aTypeSomeBrainf ; "type some brainfuck instructions except"...
call    _puts           ; Call Procedure
mov     dword ptr [esp+8], 400h
mov     dword ptr [esp+4], 0
lea     eax, [esp+434h+input_buffer] ; Load Effective Address
mov     [esp], eax
call    _memset         ; Call Procedure
mov     eax, ds:stdin@@GLIBC_2_0
mov     [esp+8], eax
mov     dword ptr [esp+4], 400h
lea     eax, [esp+434h+input_buffer] ; Load Effective Address
mov     [esp], eax
call    _fgets          ; Call Procedure
mov     [esp+434h+input_buffer_index], 0
jmp     short loc_8048760 ; Jump
```

Then, the emulator evaluates the brainfuck code character by character, until it's done:

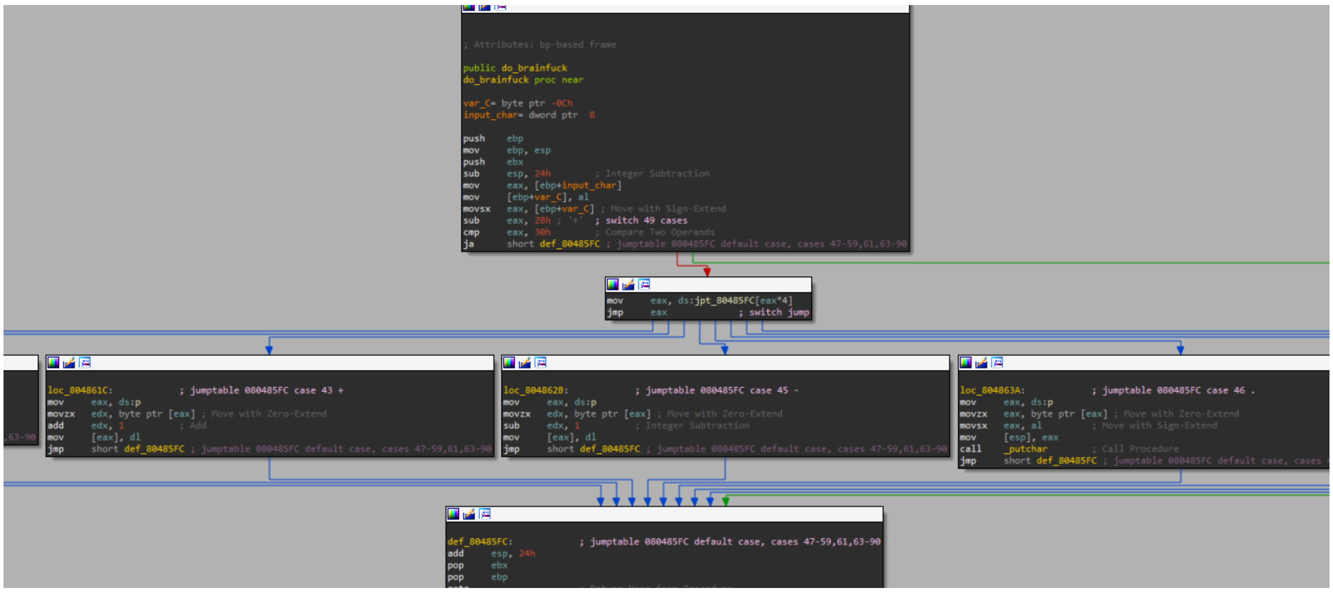


```
Iterate over the input buffer

loc_8048760:
mov     ebx, [esp+434h+input_buffer_index]
lea     eax, [esp+434h+input_buffer] ; Load Effective Address
mov     [esp], eax
call    _strlen         ; Call Procedure
cmp     ebx, eax        ; Compare Two Operands
jb      short loc_8048743 ; Jump if Below (CF=1)
```

```
mov     eax, 0
mov     edx, [esp+42Ch]
xor     edx, large gs:14h ; Logical Exclusive OR
jz      short loc_804878E ; Jump if Zero (ZF=1)
```

```
Do brainfuck on the current char of the input buffer

loc_8048743:
lea     edx, [esp+434h+input_buffer] ; Load Effective Address
mov     eax, [esp+434h+input_buffer_index]
add     eax, edx        ; Add
movzx   eax, byte ptr [eax] ; Move with Zero-Extend
movsx   eax, al         ; Move with Sign-Extend
mov     [esp], eax
call    do_brainfuck    ; Call Procedure
add     [esp+434h+input_buffer_index], 1 ; Add
```

Then, I continued looking at do_brainfuck and saw that it's a simple switch-case based on the evaluated character:



```
; Attributes: bp-based frame

public do_brainfuck
do_brainfuck proc near

var_C= byte ptr -0Ch
input_char= dword ptr  8

push    ebp
mov     ebp, esp
push    ebx
sub     esp, 24h        ; Integer Subtraction
mov     eax, [ebp+input_char]
mov     [ebp+var_C], al
movsx   eax, [ebp+var_C] ; Move with Sign-Extend
sub     eax, 2Bh ; '+'  ; switch 49 cases
cmp     eax, 30h        ; Compare Two Operands
ja      short def_80485FC ; jumptable 080485FC default case, cases 47-59,61,63-90
```

```
mov     eax, ds:jpt_80485FC[eax*4]
jmp     eax                     ; switch jump
```

```
loc_804861C:            ; jumptable 080485FC case 43 +
mov     eax, ds:p
movzx   edx, byte ptr [eax] ; Move with Zero-Extend
add     edx, 1          ; Add
mov     [eax], dl
jmp     short def_80485FC ; jumptable 080485FC default case, cases 47-59,61,63-90
```

```
loc_804862B:            ; jumptable 080485FC case 45 -
mov     eax, ds:p
movzx   edx, byte ptr [eax] ; Move with Zero-Extend
sub     edx, 1          ; Integer Subtraction
mov     [eax], dl
jmp     short def_80485FC ; jumptable 080485FC default case, cases 47-59,61,63-90
```

```
loc_804863A:            ; jumptable 080485FC case 46 .
mov     eax, ds:p
movzx   eax, byte ptr [eax] ; Move with Zero-Extend
movsx   eax, al         ; Move with Sign-Extend
mov     [esp], eax
call    _putchar        ; Call Procedure
jmp     short def_80485FC ; jumptable 080485FC default case, cases
```

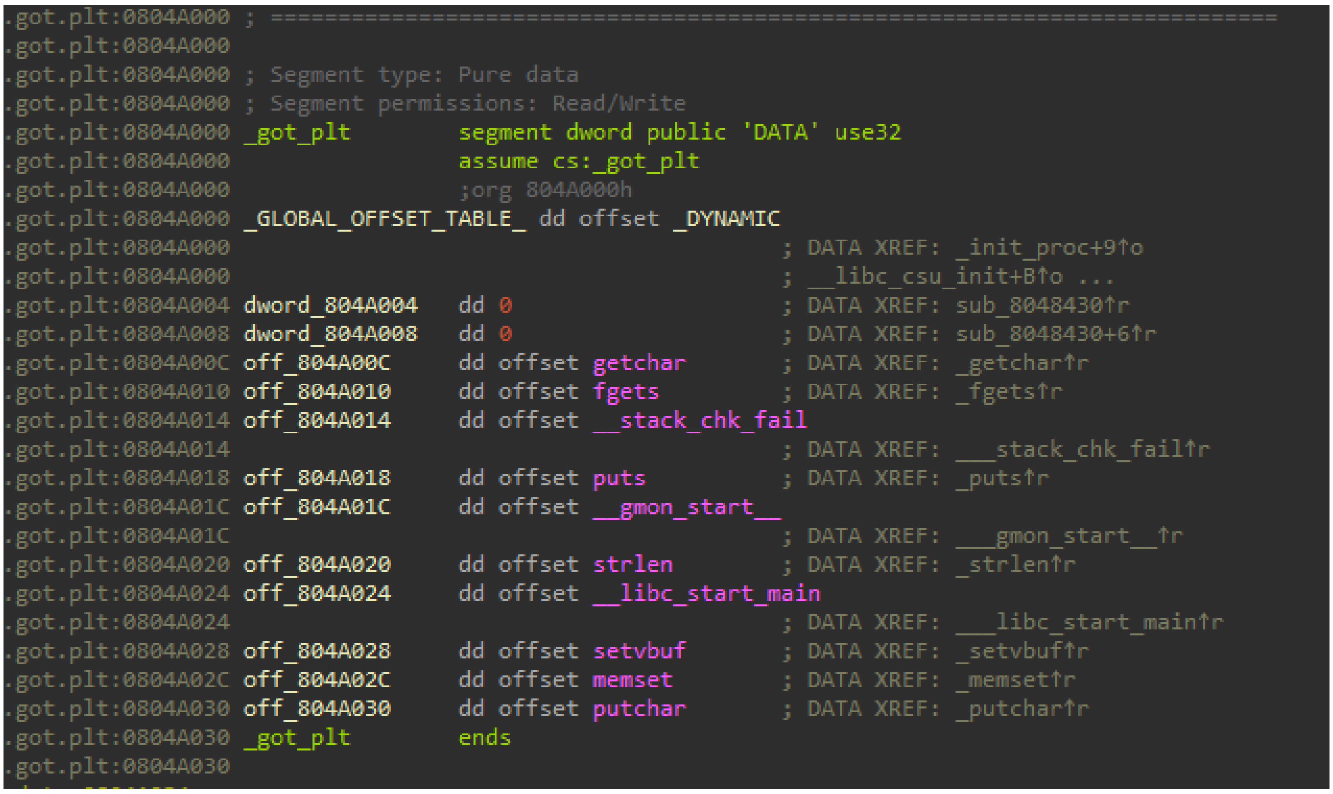
```
def_80485FC:            ; jumptable 080485FC default case, cases 47-59,61,63-90
add     esp, 24h
pop     ebx
pop     ebp
```

Vulnerability

The code is pretty straight-forward, and so is the vulnerability: Using the brainfuck language, we can traverse the program's memory (using brainfuck's "<>"), read the memory (".") and write arbitrary values to the memory (","), as there isn't any validation about the address of the data pointer, which resides in the BSS section.

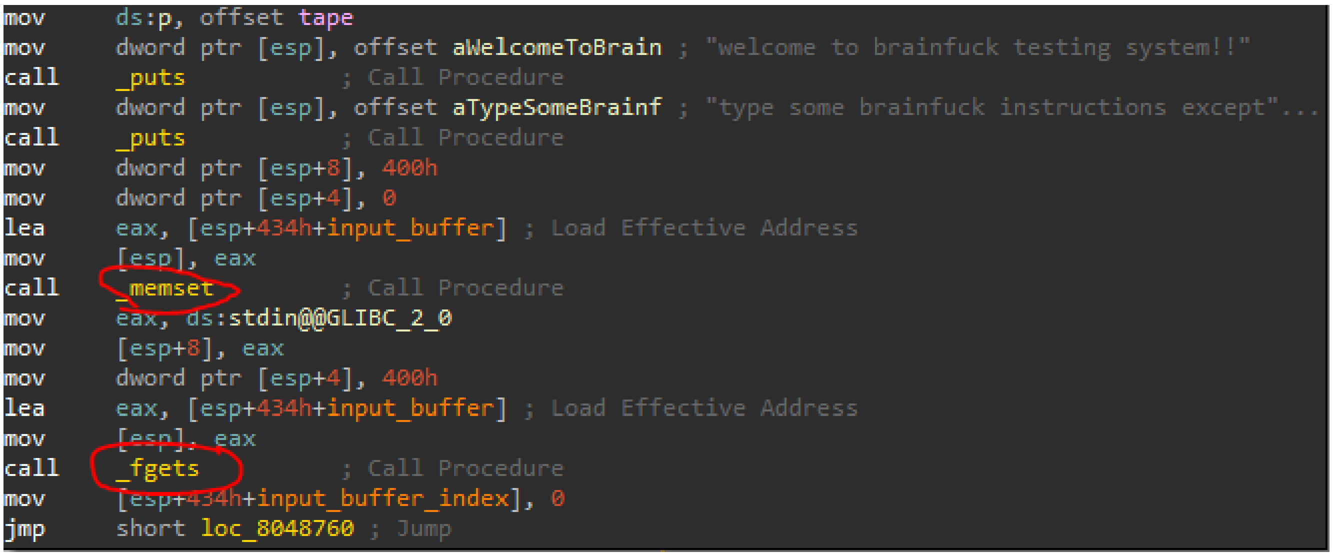
GOT.PLT Overrides

So we can write arbitrary values into memory, but eventually we should run `system("/bin/sh")` , how can we do so? I noticed the `got.plt` section:



```
.got.plt:0804A000 ; ===========================================================================
.got.plt:0804A000
.got.plt:0804A000 ; Segment type: Pure data
.got.plt:0804A000 ; Segment permissions: Read/Write
.got.plt:0804A000 _got_plt        segment dword public 'DATA' use32
.got.plt:0804A000                 assume cs:_got_plt
.got.plt:0804A000                 ;org 804A000h
.got.plt:0804A000 _GLOBAL_OFFSET_TABLE_ dd offset _DYNAMIC
.got.plt:0804A000                                         ; DATA XREF: _init_proc+9↑o
.got.plt:0804A000                                         ; __libc_csu_init+B↑o ...
.got.plt:0804A004 dword_804A004   dd 0                    ; DATA XREF: sub_8048430↑r
.got.plt:0804A008 dword_804A008   dd 0                    ; DATA XREF: sub_8048430+6↑r
.got.plt:0804A00C off_804A00C     dd offset getchar       ; DATA XREF: _getchar↑r
.got.plt:0804A010 off_804A010     dd offset fgets         ; DATA XREF: _fgets↑r
.got.plt:0804A014 off_804A014     dd offset __stack_chk_fail
.got.plt:0804A014                                         ; DATA XREF: ___stack_chk_fail↑r
.got.plt:0804A018 off_804A018     dd offset puts          ; DATA XREF: _puts↑r
.got.plt:0804A01C off_804A01C     dd offset __gmon_start__
.got.plt:0804A01C                                         ; DATA XREF: ___gmon_start__↑r
.got.plt:0804A020 off_804A020     dd offset strlen        ; DATA XREF: _strlen↑r
.got.plt:0804A024 off_804A024     dd offset __libc_start_main
.got.plt:0804A024                                         ; DATA XREF: ___libc_start_main↑r
.got.plt:0804A028 off_804A028     dd offset setvbuf       ; DATA XREF: _setvbuf↑r
.got.plt:0804A02C off_804A02C     dd offset memset        ; DATA XREF: _memset↑r
.got.plt:0804A030 off_804A030     dd offset putchar       ; DATA XREF: _putchar↑r
.got.plt:0804A030 _got_plt        ends
.got.plt:0804A030
```

Potentially, we can overwrite values in `got.plt` so that they will point to other functions instead of the intended functions. Then, we can control the flow of the program to run our "malicious" functions so that we'll eventually run `system("/bin/sh")`. Our potential functions to override are the functions that are called during the emulation of the brainfuck code: `putchar` (".") and `getchar` (",") cannot serve our purpose because we `system` receives a string argument, and these functions do not. `puts` ("[") could be interesting, but it's called with a read-only argument, so we cannot modify the argument to be `"/bin/sh"`. If we overwrite one of these functions with `main` (let's say `putchar`), we can extend the scope of our potential functions. Let's look back at `main`:



```
mov     ds:p, offset tape
mov     dword ptr [esp], offset aWelcomeToBrain ; "welcome to brainfuck testing system!!"
call    _puts           ; Call Procedure
mov     dword ptr [esp], offset aTypeSomeBrainf ; "type some brainfuck instructions except"...
call    _puts           ; Call Procedure
mov     dword ptr [esp+8], 400h
mov     dword ptr [esp+4], 0
lea     eax, [esp+434h+input_buffer] ; Load Effective Address
mov     [esp], eax
call    _memset         ; Call Procedure
mov     eax, ds:stdin@@GLIBC_2_0
mov     [esp+8], eax
mov     dword ptr [esp+4], 400h
lea     eax, [esp+434h+input_buffer] ; Load Effective Address
mov     [esp], eax
call    _fgets          ; Call Procedure
mov     [esp+434h+input_buffer_index], 0
jmp     short loc_8048760 ; Jump
```

We can overwrite `memset` to be `gets` so that we can input `"/bin/sh"` into the input buffer, and then modify `fgets` to be `system`, and basically that's it.

LIBC Address Resolution

Just one missing piece before summarizing the exploit. `libc` is dynamically linked to the binary, which means that we cannot know the addresses of `libc` functions in advance. However, by leaking an address in `libc`, and knowing its offset from the base of `libc`, we can infer the base of `libc`, and then infer any other function in `libc`.

Exploit summary

1. Traverse the memory using brainfuck to the offset of `fgets` in `got.plt`.
2. Leak `fgets` address using brainfuck.
3. Calculate `libc` base according to the leaked address.
4. Write the address of `system` to `fgets`.
5. Traverse to `memset` in `got.plt` and write the address of `gets`.
6. Traverse to `putchar` in `got.plt` and write the address of `main`.
7. Activate main again using "." (result of section 6).
8. input `"/bin/sh"`
9. Profit.