

Step 2:

The input I found that causes a segmentation fault is a passwords that contains one hundred of A's.

```
input
main.c:13:3: warning: 'gets' is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.c:(.text+0x4b): warning: the 'gets' function is dangerous and should not be used.
Actual password set to: p@ssw0rd
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Failure: The user could not be authenticated.
User-provided password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Actual password: p@ssw0rd
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)

...Program finished with exit code 134
```

In the main function, the password is set to contain 10 characters at most. The length of the password I provide is long enough to cover all the space reserved for both username and password. The original content of the stored values of \$fp and \$ra now have been overwritten.

The stored value of \$ra contains the address of the next instruction to execute once the main function completes executing. The content of the actual register \$ra will be restored with the old value stored in the stack. But my very long password has overwritten the value stored in the stack. Then \$ra restored part of my password, AAAA. Then the program will try to execute the instruction at the address of 0x41414141. This will lead to core dump.

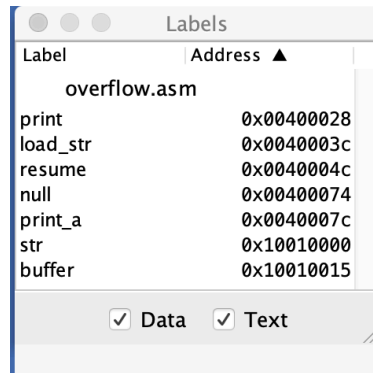
Step 3:

The input I found that won't crash the application but authorized me in is a password that contains twenty A's. In line 15, the `strncmp()` function compares the first ten bytes of `user_password` and `password`. So I want to take advantage of overflow and overwrite the 'Actual password' to be the last ten bytes of my input. So my password at least needs to be longer than 10 to start overwriting the 'Actual password' and actually needs to be longer than 20 to completely overwrite the 'Actual password'. So I choose to input twenty A's because it overwrites the 'Actual password' to be ten A's which is exactly equal to the last ten bytes of my input.

```
❏ clang-7 -pthread -lm -o main main.c
main.c:13:3: warning: implicit declaration of function 'gets' is
      invalid in C99 [-Wimplicit-function-declaration]
      gets(user_password); // get password from user
      ^
1 warning generated.
/tmp/main-08e5db.o: In function `main':
main.c:(.text+0x4c): warning: the `gets' function is dangerous a
nd should not be used.
❏ ./main
Actual password set to: p@ssw0rd
AAAAAAAAAAAAAAAAAAAA
Success! The user was authenticated successfully.
User-provided password: AAAAAAAAAAAAAAAAAAAAAA
Actual password: AAAAAAAAAA
❏
```

Step 4:**1)**

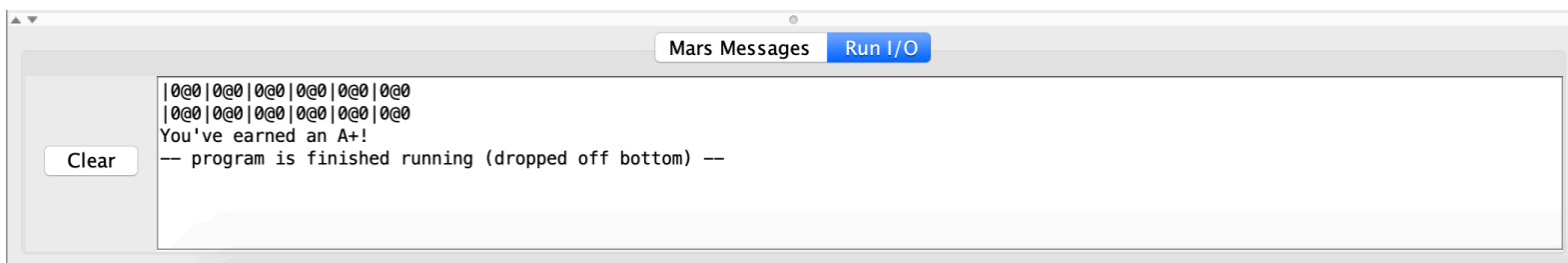
By looking at the Labels section, I notice the print_a function is stored at the address '0x0040007c'. So this will made up part of my malicious input.



By referring to AsciTable.com, I found the following interchange results:

Hex	Char	Name
40	@	At-sgin
7c		Vertical-bar

And by playing around, I found zero will be stored as '00' in address and I notice Data Segment uses little-endian order to store values. So part of my malicious input will look like 'vertical-bar zero at-sign zero'. I simply repeat this pattern six times to make sure I will overwrite address of \$ra. And I was able to trigger print_a function.



2)

