Christian Znidarsic
3/17/21

Computer Organization
Module 8 Assignment

Step 2 Documentation

This step of the assignment was the easiest to complete. All that had to be done was an input needed to be entered to overflow and crash overflow.c. This proved to be simple because the only criterion that the input needed to meet was it had to be greater than or equal to 11 characters long. Any input of length 11 characters or more will cause the program to crash. The example simply uses an input of 11 lower-case a's as input to cause the error.

The reason that an 11+ character long input crashes the program is that the function gets(user_password) takes the user input and copies it, one character at a time, into user_password. This would be fine, except in line 8 of overflow.c, user_password is initialized as a char list of size 10. This means that only ten bytes are allocated on the stack to hold user_password. Thus, when the user enters a password longer than 10 bytes, there is a buffer overflow on the stack, and preceeding items on the stack are overwritten. Ordinarily this would manifest itself as a segmentation fault and a core dump, but onlinegdp.com handles this error by showing the error message: "*** stack smashing detected ***: terminated".

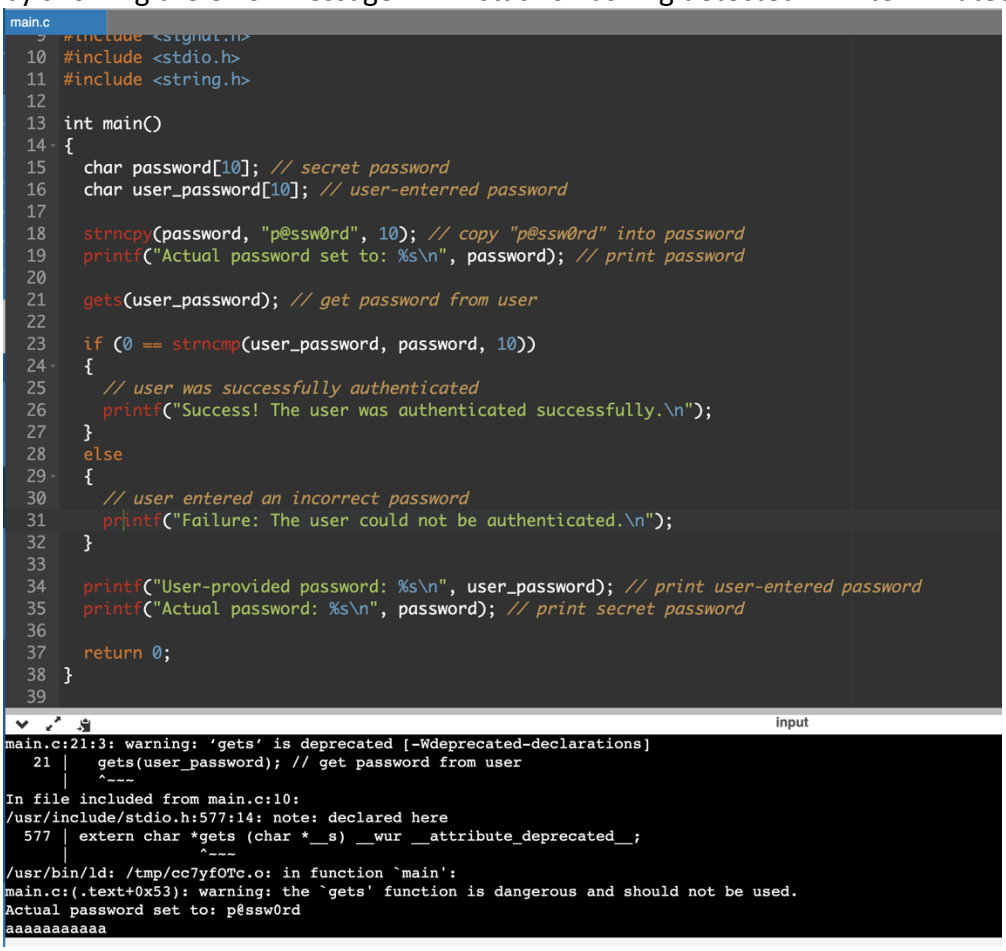


Figure 1: The overflow.c program with the input "aaaaaaaaaaa" (11 lower-case a's) before being run. The program warns the user that gets() is a dangerous function.

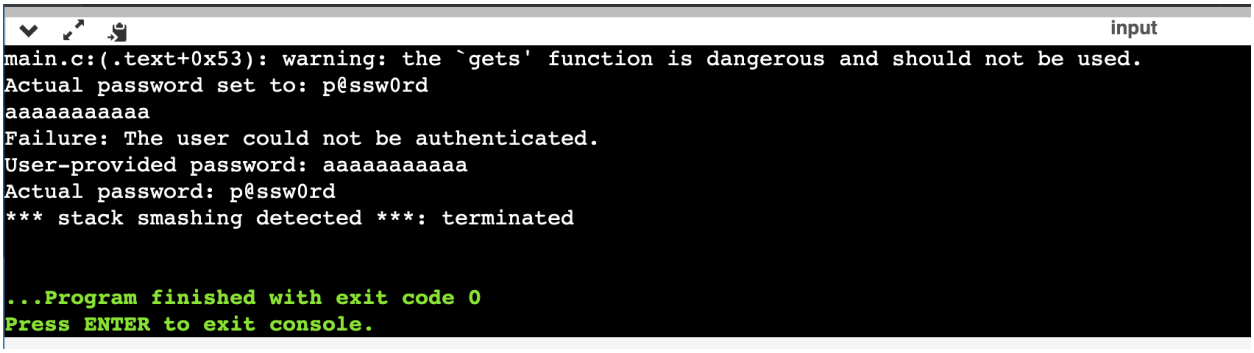


Figure 2: The output to the console by overflow.c after running. The error message is shown.

Step 3 Documentation

This step of the assignment proved very similar to Step 2. A buffer overflow was exploited in order to trick the program into allowing access to the program using an incorrect password. In order to do this, first the addresses of both the char array password[] and user_password[] were located. This was done using the following commands:

printf("User-provided password address: %p\n", (void *)user_password);
printf("Actual password: %p\n", (void *)password);

Together these two commands print the hex addresses of the char lists password and user_password. The results were as follows:

User-provided password address: 0x7ffff4ecce68
Actual password address: 0x7ffff4ecce72

Converted into decimal form, these two addresses are only ten bytes away from each other. This is what we would expect, because 10 bytes are allocated to both user_password and password. In memory, it looks something like this:

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [p] [@] [s] [s] [w] [0] [r] [d] [ ] [ ]

Where the left 10 bytes are the user-entered password, and the right ten bytes are the actual required password. They are adjacent to each other in memory, with the actual password stored directly after the user-input password. Thus, when the user-input password overflows, it overwrites the actual password. If the user enters 13 a's, then memory looks like this:

[a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [\n] [w] [0] [r] [d] [ ] [ ]

As you can see, four characters in the system password have been overwritten, three of them by a's, and one of them by a newline character. When the user-input is printed using printf, it

prints a's until it reaches a newline. When the system password is printed, it also prints until it hits the newline. Thus, the output of printing user-input and system password in this case would be:

user-input password: aaaaaaaaaaaa
system password: aaa

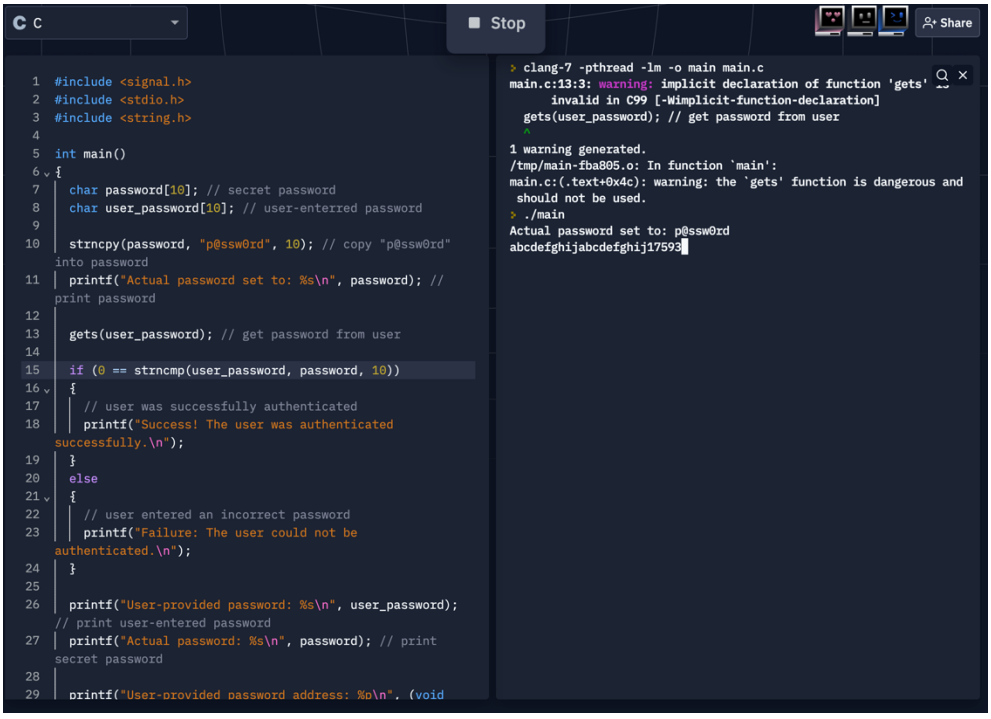
Because the condition that grants access simply compares whether the first 10 characters of user-input is the same as the first 10 characters of the system password, all the user needs to enter are 20 or more of the same character in order to gain access. In that case, memory would look like this:

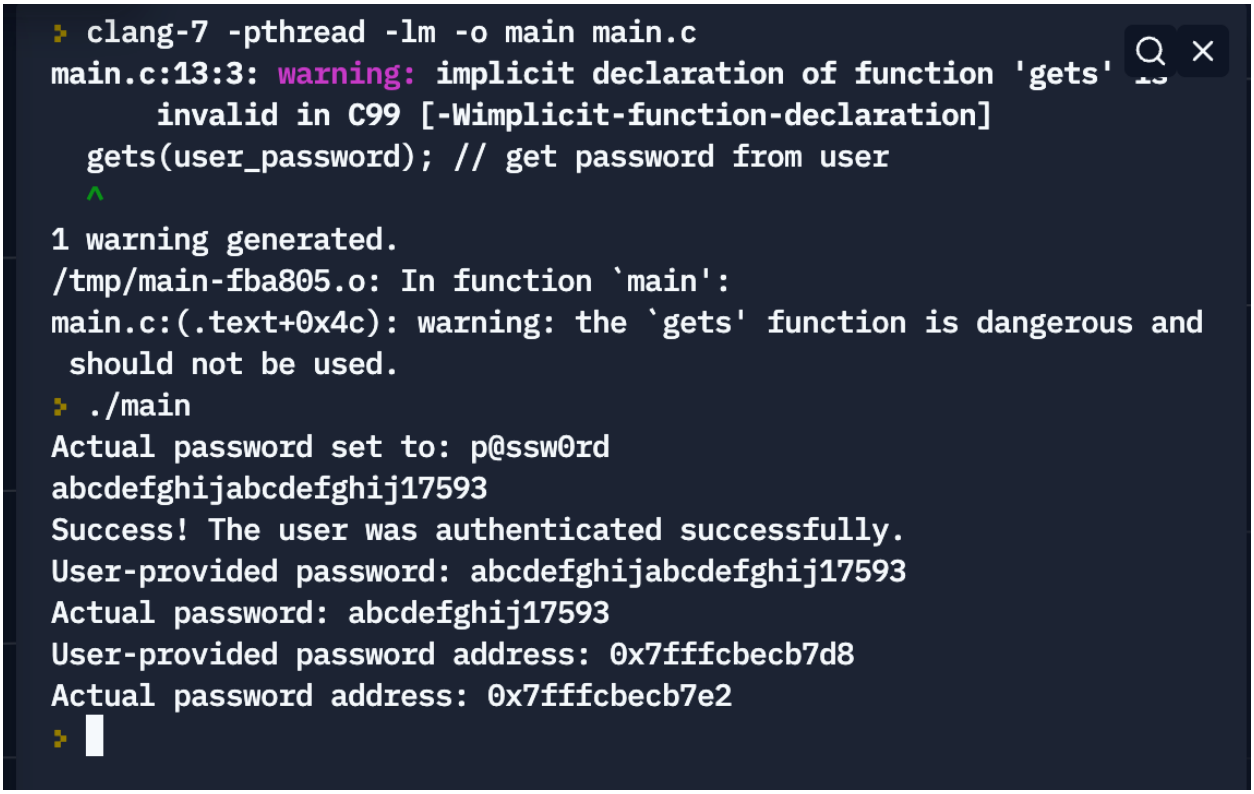
[a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] [a] … [\n]

Regardless of what is input after the second block of a's and the newline character, the first ten characters of user-input and the first ten characters of the system password are equal, and thus the program grants us access. Any input that follows this pattern will grant the user access. For example:

[a] [b] [c] [d] [e] [f] [g] [h] [i] [j] [a] [b] [c] [d] [e] [f] [g] [h] [i] [j] [1] [7] [5] [9] [3] [\n]

works just as well. The only other condition that the input needs to meet is it cannot exceed 32 characters in length or else it will cause a segmentation fault and core dump. This is likely because beyond 32 bytes, it begins to overwrite a return address or something else in the stack that is critical to the program at runtime.




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-fba805.o: In function `main':
main.c:(.text+0x4c): warning: the `gets' function is dangerous and
should not be used.
./main
Actual password set to: p@ssw0rd
abcdefghijabcdefghij17593
Success! The user was authenticated successfully.
User-provided password: abcdefghijabcdefghij17593
Actual password: abcdefghij17593
User-provided password address: 0x7fffcbecb7d8
Actual password address: 0x7fffcbecb7e2

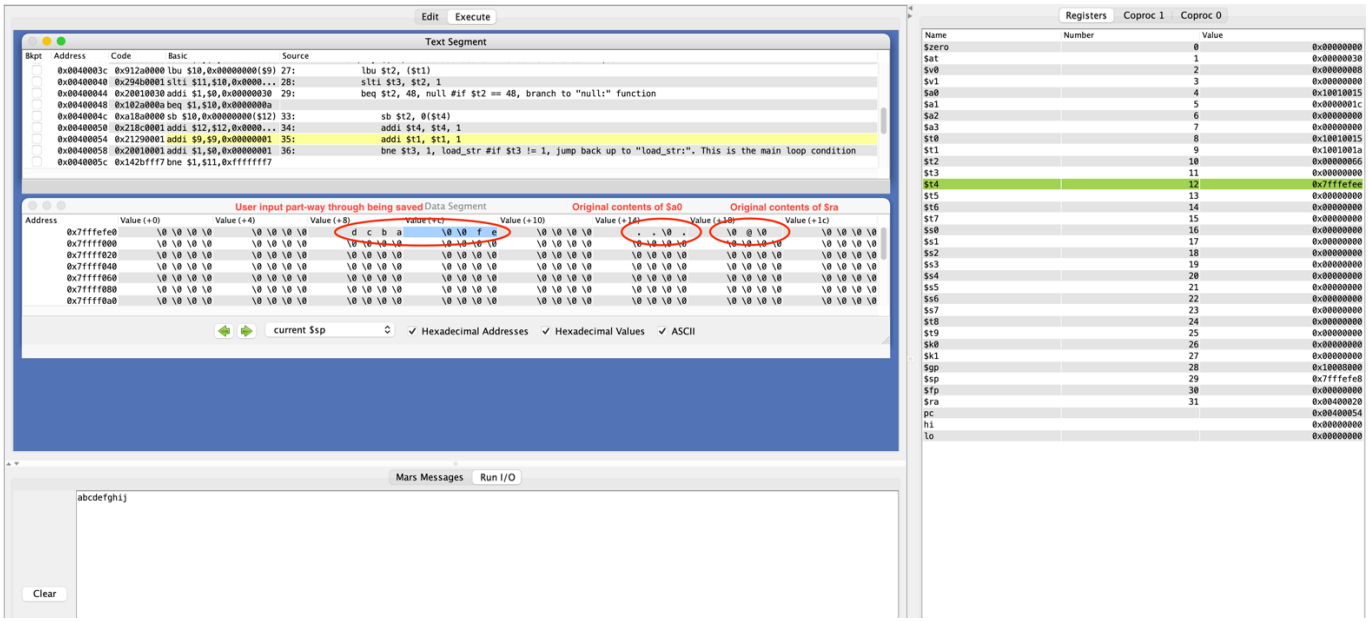
Christian Znidarsic
3/17/21

Step 4 Documentation

Part 1:

In order to jump to the “print_a:” function, the return address stored on the stack by the “print:” function must be overwritten with the address to “print_a:”. This address to “print_a:” can be found by looking at the Text Segment under the Execute tab. This shows all the MIPS commands and their addresses. The address for “print_a:” is 0x0040007c. This is the address that we must use to overwrite the return address that “print:” saves in memory. Just like in Step 3, we will provide a user input that exceeds the input limits so that it overwrites the return address.

In order to accomplish this result, we must know where the user input is stored on the stack, and where the old return address is stored on the stack. This way we will know precisely how long our input needs to be in order to fully overwrite the old return address. This can be seen clearly using the Execute tab, under the Data Segment section in Mars. Once selecting “current $sp” from the drop down menu, the stack and its contents are shown. After running the program we can see that the contents of $a0 have been saved at address 0x7fffefe0 + 14, and the old return address was saved at address 0x7fffefe0 + 18. This is what we expect because the stack pointer is at 0x7fffefe8, as seen in the Registers column to the right in Mars. In the program, “Print:” saves the contents of $a0 and $s0 to $sp + 12 and +16 respectively. When we enter an input it is saved, one character at a time, to the stack. The program uses the comand sb (store byte) to store each character of the user input in the stack. We can watch this happen by placing a breakpoint at line 20 (where “Print:” saves the return address to the stack) and then iterating step by step through the program using the “Run one step at a time” button at the top in Mars. If we check the “ASCII” box in the “Data Segment” tab, we can view the user input saved one character at a time to the stack. This process is shown in the following screenshot:

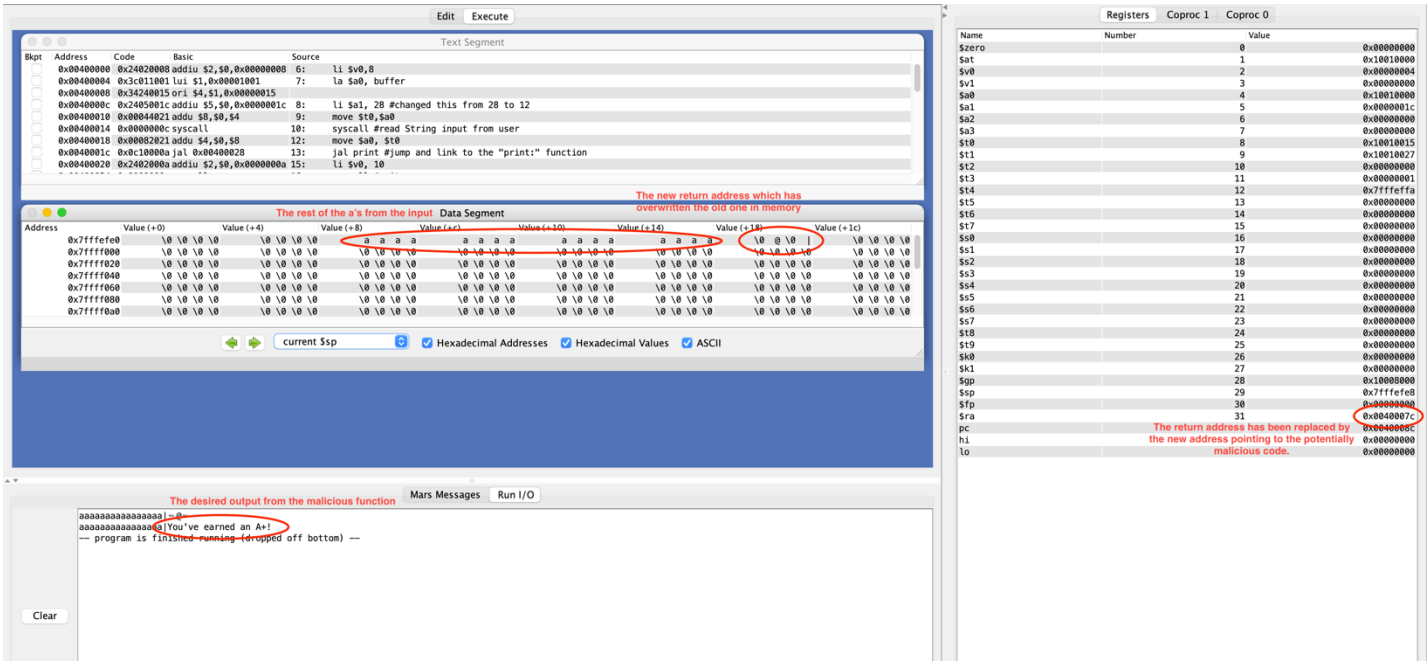


Christian Znidarsic
3/17/21

By viewing the stack contents in real-time, we can see that, in order to overwrite the original contents of $ra, we must write an input of length 16 + 4, where the first 16 bytes are all filler (can be anything) and the last 4 bytes are the new address we wish to sub in for the old return address (in little endian). The address we want to sub in is the 0x0040007c that we found earlier. This hex address converted into ASCII characters is "␌@␌|" according to the ASCII table. We flip this and append it to the end of a 16-character input of our choosing, let's say 16 a's, to get an input of:

aaaaaaaaaaaaaaaa|␌@␌

When we input this to our program, $a0 is overwritten in memory by a's, but that doesn't change the outcome of the program.



Christian Znidarsic
3/17/21

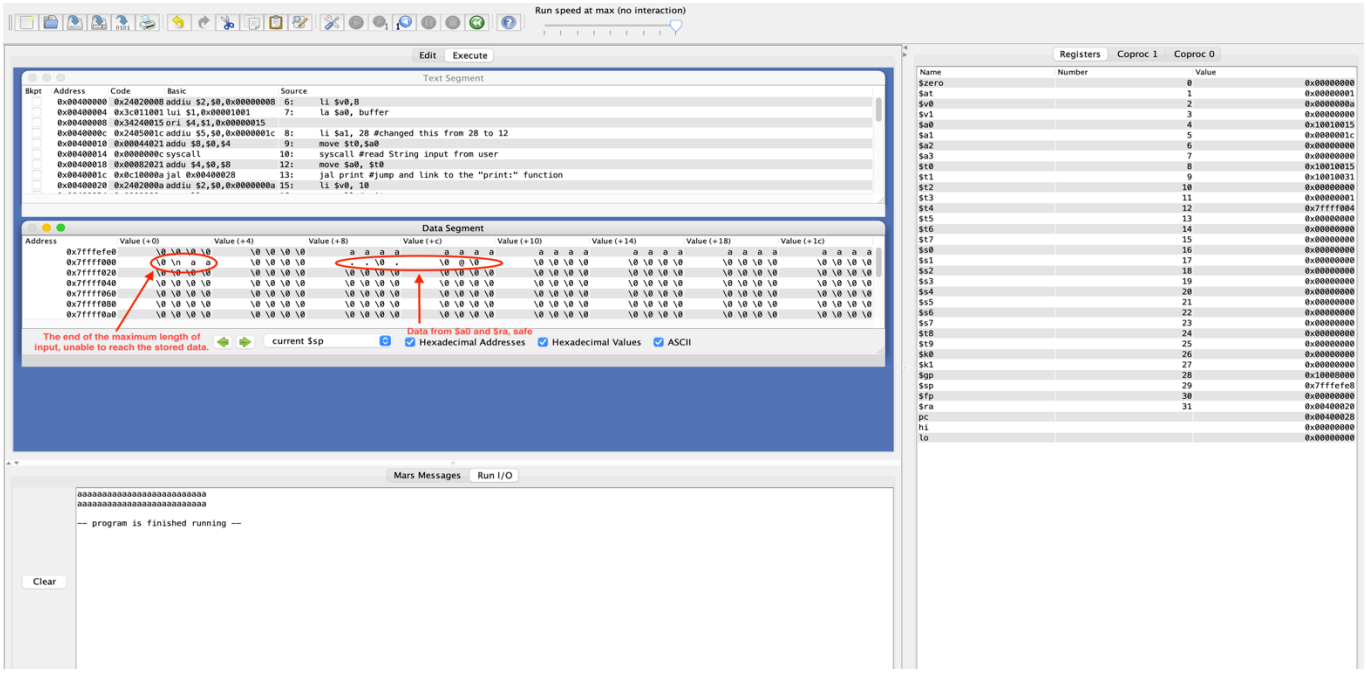
Part 2 (patch):

In order to patch the program, we must figure out the issue it has. The problem this program has is that it allows up to 28 character inputs from the user, while at the same time, $a0 and $sp are stored less than 28 bytes away from the user input in memory. This means that if the user enters any input larger than 16 bytes, saved register data will be overwritten. This can be solved in at least three different ways:

1. Decrease the length of allowed input to 12 bytes
2. Store $a0 and $sp more than 28 bytes away from the user input in memory
3. Store the user input *after* $a0 and $sp in memory

The easiest fix is option 1. The length of allowed input can be reduced to 12 bytes, which would mean that the program wouldn't get the chance to accept an input long enough to overwrite saved data. However, this greatly reduces the number unique inputs that can be entered.

A better fix is option 2. One can simply store the data from $a0 and $ra further away from the user input in the stack, so that it is out of reach of a malicious user input. In order to make enough room for the 28 byte user input, $a0 is stored 32 bytes away from the stack pointer, and $ra is stored 36 bytes away. The program will not accept inputs larger than 28 bytes, so the data cannot be overwritten.



The maximum input size ends up being 26 characters from the user (plus a newline and a null character which are automatically generated when using syscall of 4

to read the input String, for a total of 28 characters). In the example above, an input of 26 a's are given, and they are not able to reach the saved data in the stack.